Modeling and Simulating a Software Architecture Design Space

Charles W. Krueger

December 1997
CMU-CS-97-158

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
A. Nico Habermann, deceased, Co-Chair
David Garlan, Co-Chair
Jeannette Wing
Mahadev Satyanarayanan
Lori Clarke, University of Massachusetts at Amherst

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330; by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031; by the Defense Modeling and Simulation Office; by ZFE of Siemens Corporation; and by National Science Foundation Grant No. CCR 9357792.

Keywords: software engineering, software architecture, domain-specific software architecture, parameterized software architecture, software framework, software reuse, requirements engineering, object-oriented database, simulation

Table of Contents

Abstract

Frequently, a similar type of software system is used in the implementation of many different software applications. Databases are an example. Two software development approaches are common to fill the need for instances from a class of similar systems: (1) repeated custom development of similar instances, one for each different application, or (2) development of one or more general purpose off-the-shelf systems that are used many times in the different applications. Each approach has advantages and disadvantages. Custom development can closely match the requirements of an application, but has an associated high development cost. General purpose systems may have a lower cost when amortized across multiple applications, but may not closely match the requirements of all the different applications.

It can be difficult for application developers to determine which approach is best for their application. Do any of the existing off-the-shelf systems sufficiently satisfy the application requirements? If so, which ones provide the best match? Would a custom implementation be sufficiently better to justify the cost difference between an off-the-shelf solution? These difficult buy-versus-build decisions are extremely important in today's fast-paced, competitive, unforgiving software application market.

In this thesis we propose and study a software engineering approach for evaluating how well off-the-shelf and custom software architectures within the design space of a class of OODB systems satisfy the requirements for different applications. The approach is based on the ability to explicitly enumerate and represent the key dimensions of commonality and variability in the space of OODB designs. We demonstrate that modeling and simulation of OODB software architectures can be used to help software developers rapidly converge on OODB requirements for an application and identify OODB software architectures that satisfy those requirements.

The technical focus of this work is on the circular relationships between requirements, software architectures, and system properties such as OODB functionality, size, and performance. We capture these relationships in a parametrized OODB architectural model, together with an OODB simulation and modeling tool that allows software developers to refine application requirements on an OODB, identify corresponding custom and off-the-shelf OODB software architectures, evaluate how well the software architecture properties satisfy the application requirements, and identify potential refinements to requirements.

Acknowledgments

This work is dedicated to the spirit of two individuals who provided tremendous inspiration and motivation to me during its creation, but sadly were not able to celebrate with me in its culmination – Nico Habermann, my first thesis adviser, and Herbert Krueger, my father.

Nico was more than an adviser to me, he was a "mentor" in every sense of the word. Intellectually, professionally, and personally he became the most significant model in my life. His strength, vision, and kindness will live on through those who were fortunate enough to grow under his guidance.

My father was an inspiration to me in the way that he defined and achieved personal success using simple yet deep values. Selfless dedication, boundless optimism, and uncompromising ethics were the things for which he was widely admired and the things which I will always remember.

There are many people who shared, willingly or otherwise, in the sacrifices necessary to make my personal dream a reality. The greatest of these is Aaron, who until now has never had a father that didn't need to "take some time to work on the thesis". Your unconditional support and love, keen intellect, limitless enthusiasm, and best of all your friendship have provided me the perspective needed to persevere.

It has been said that the most important thing in the long-term success of a marriage is friendship. It is this special friendship that has kept our marriage strong through the perturbations offered over the years by this work. Thank you, Debbie, for always being there for me.

My mother showed me how to be a free-thinking individual. She always supported me and allowed me to "go against the grain" to pursue the things that were important to me. This spirit allowed me to believe in myself and not be deterred when so many others said it couldn't be done.

And finally I need to acknowledge the critical role that David Garlan played in accepting the role of thesis advisor after Nico died. While I was 1200 miles away from CMU, David was my advocate. I can truly say that this never could have happened without his hard technical and diplomatic work.

Chapter 1. Introduction

1.1. Overview

Frequently, a similar type of software system is used in the implementation of many different software applications. Databases are an example. Two approaches are common to fill the need for instances from a class of similar systems: (1) repeated custom development of similar instances, one for each different application, or (2) development of one or several general purpose off-the-shelf systems that are used many times in the different applications. Each approach has advantages and disadvantages. Custom development can closely match the requirements of an application, but has an associated high development cost. General purpose systems may have a lower cost when amortized across multiple applications, but may not closely match the requirements of all the different applications.

It can be difficult for application developers to determine which approach is best for their application. Do any of the existing off-the-shelf systems sufficiently satisfy the application requirements? If so, which ones provide the best match? Would a custom implementation be sufficiently better to justify the cost difference between an off-the-shelf solution? These difficult buy-versus-build decisions are extremely important in today's fast-paced, competitive, unforgiving software application market.

In this thesis we propose and study a software engineering approach for evaluating how well off-the-shelf and custom software architectures within the design space of a class of OODB systems satisfy the requirements for different applications. The approach is based on the ability to explicitly enumerate and represent the key dimensions of commonality and variability in the space of OODB designs. We demonstrate that modeling and simulation of OODB software architectures can be used to help software developers rapidly converge on OODB requirements for an application and identify OODB software architectures that satisfy those requirements.

Different OODBs typically serve a similar role, but generally each OODB is focused on a specialized market niche and has a corresponding architecture that is unique in one or more ways. No single software architecture for OODBs is generally applicable across the broad range of applications found in practice. We develop a modeling and simulation tool for OODB architectures and demonstrate how this tool can help software developers quickly identify the OODB requirements for an application, to find a custom OODB architecture that satisfies those requirements, and to determine which of the off-the-shelf OODB architectures most closely satisfy the requirements.

The technical focus of this work is on the circular relationships between requirements, software architectures, and system properties such as OODB functionality, size, and performance. We capture these relationships in a parametrized OODB architectural model, together with an OODB simulation and modeling tool that allows software developers to refine application requirements on an OODB, identify corresponding custom and off-the-shelf OODB software architectures, evaluate how well the software architecture properties satisfy the application requirements, and identify potential refinements to requirements.

1.2. The Problem

It is often the case in the software industry that many different instances from a class of similar software system are independently developed. Sometimes this duplication of effort is intended and desirable, such as in the case of competitive products. However, in other cases repeated development is a distinct disadvantage, such as independent development of similar software systems within a single large organization or repeated development of common software subsystems that do not contribute to the value-added in product development efforts. The cost of repeated development in these cases is unnecessary and wasteful.

One approach to avoiding the repetitive effort of building similar systems is to build a general purpose, reusable system that provides the union of functionality in that class of systems. Since a general purpose system can be used in many different products, its development cost is amortized across all the different uses.

Object-oriented databases (OODBs) are a good example of this type of general purpose system. OODB technology came about in response to the many custom-built object management systems being developed for applications with complex data management requirements. Developers needing object management functionality in their applications typically will find that it is less expensive to buy an OODB than it is to develop and maintain a custom OODB with the required functionality.

However, there are several problems with the general purpose system approach. First is excess functionality. General purpose systems are designed to support all of the functionality that might be needed in all possible deployments, but for a particular application much of the functionality may be extraneous, leading to excessive size or reduced performance. Second is missing functionality. A general purpose system might not support some functionality that is required for a particular application. Third is average case performance. Performance in general purpose systems is tuned to satisfy the average case. However, for a particular application, the average case performance may be inappropriate due to performance priorities on a specific system function.

General purpose OODBs exemplify these problems. For example, ITASCA is one of the most fully featured OODBs, aimed at very large-scale, distributed, multi-user applications needing object versioning, long transactions, security, and related advanced functionality. Because of the complex interactions of these functions, the system is relatively large and performance is relatively slow. For example, in our modeling we found that for the same application, ITASCA required nearly three times as much runtime space and 40% greater execution time as less ambitious OODBs. Some applications not requiring all of the ITASCA functionality may find the excess size and slower performance associated with the excess functionality to be unacceptable. In contrast, ObjectStore is aimed at large-scale, distributed, multi-user applications, but focuses on high performance rather than advanced functionality. Although it is fast, the footprint of ObjectStore is relatively large for some applications and may have missing functionality for other applications. ObjectStoreLite is an OODB for single-user, single-CPU applications, particularly those running on small portable computers. Without functionality for distribution and multiple users, the footprint for the OODB is relatively small and performance is good, but the limited functionality is a problem for many applications. The average case performance for all of these OODBs can be a problem for applications with specific performance profile requirements.

Developers attempting to identify and satisfy requirements for a particular application are faced with all of these price, performance, size, and functionality issues associated with off-the-shelf systems and with custom development. Will all of the available off-the-shelf systems work adequately? Will one work better than another? Are none of them sufficient, implying that custom development is needed?

For example, it is difficult to select the best commercial, share-ware, or free-ware OODB for a given application. First, it is not obvious what features of an OODB will determine a good or bad match for an application. Second, it is very difficult to compare existing literature on the various products and predict how well each will satisfy application requirements. Third, published benchmarks may not address the performance issues that are relevant for an application. And fourth, prototyping and benchmarking with evaluation software can be very expensive, both in terms of time and money, offsetting some of the advantage of using off-the-shelf products.

Further complicating the problem that developers face as they evaluate and select OODBs is the question of what their requirements really are. Without some application prototyping experience using a set of requirements, there is always concern that the requirements are incomplete, inaccurate, or have unexpected side-effects, leading to the selection of an OODB that does not conform well to the application.

1.3. Classes of Similar Systems

We have used the phrase "class of similar systems" several times, relying on the reader's intuition about what this might mean. This concept is central throughout the remainder of this study, so it deserves attention up front.

The notion of a *class of similar systems* is not new. According to Campbell[1], Dijkstra introduced the concept of *program families* in 1972[2] and Parnas next elaborated on the idea in 1976[3], both arguing that individual instances from a family of similar systems should not be developed as unique artifacts, or even as successive versions, but rather should all be derived from a common abstraction for the family. More recently, Lane characterizes a *design space* for user interface software and shows how to derive instances from the design space[4]. Garlan and Shaw discuss architectural styles that are common across many different software systems[5]. Campbell presents business and technical models that organize software development around *program families*[1]. Johnson presents a collection of common object-oriented design patterns as a basis for classes of software systems[6].

Similarly, we use the terms *class of similar systems*, or simply *class of systems*, to refer to a collection of software systems such as OODBs that share similar sets of requirements, similar architectures, and similar sets of system properties. We use the term *instance* when referring to an individual member from a class of similar systems. The notion of *similar* in this context depends on the intended use of the class of systems. In one case "similar" may have a very specific definition and encompass a small number of instances, while in another case "similar" may have a very general definition and encompass a large number of instances.

1.4. The Thesis

Our goal is to develop an effective software engineering technology to identify off-the-shelf and custom instances from the OODB class of software systems that match the requirements for a particular application. This technology would help software application developers refine OODB requirements, make informed evaluations of off-the-shelf OODBs, and make informed buy-versus-build decisions.

First we apply the notion of software architecture as a means of modeling the class of OODB systems and the instances within the class. Then we postulate that architectural modeling and simulation can be used as an effective mechanism to define and refine application requirements on OODBs and to identify OODB architecture instances that satisfies the requirements.

> **Hypothesis:** It is possible to apply an engineering approach, based on the modeling and simulation of OODB architectures, to efficiently guide developers in making principled choices within the architectural space for the class of OODB software systems, such that we improve our ability to identify OOBD architecture instances with high conformance to application requirements.

We evaluate this hypothesis in the context of custom and off-the-shelf OODBs and show that we can inexpensively and accurately match different OODB architectures to applications with different requirements, ranging from a simple single-user OODB for a laptop application to a sophisticated multi-user OODB in a networked workstation environment.

1.4.1. Overview of the Solution

1.4.1.1. The Foundation

Our solution exploits the relationships between *application requirements*, *software architectures*, and resulting *systems properties*. In this section we introduce these concepts and relationships in general software engineering terms and relate them to the specific case of OODBs.

Figure 1. illustrates how requirements, architectures, and system properties are related for today's typical application development. The application requirements specify constraints that must be satisfied in order for an application implementation to be considered acceptable. For example, requirements on an OODB might include a requirement to support multiple concurrent users and a size requirement to fit on a desktop PC. A software architecture is selected in an attempt to best satisfy the application requirements. The system properties are measurable characteristics of the implemented software architecture operating under typical application scenarios. System properties can then be used to determine whether or not the software architecture conforms to the application requirements and whether or not the requirements should be refined or modified.

Note that the *application requirements* in this model are the desired properties for a system instance, while the *system properties* are the measurable properties of an actual instance. Ideally, the application requirements are a subset of the system properties, though other forms of intersection are possible. Conformance is then informally defined as the degree to which the application requirements intersect with (i.e., are a subset of) the system properties. If conformance is too low, a different software architecture can be adopted so that the system properties intersect to a greater degree with the application requirements.

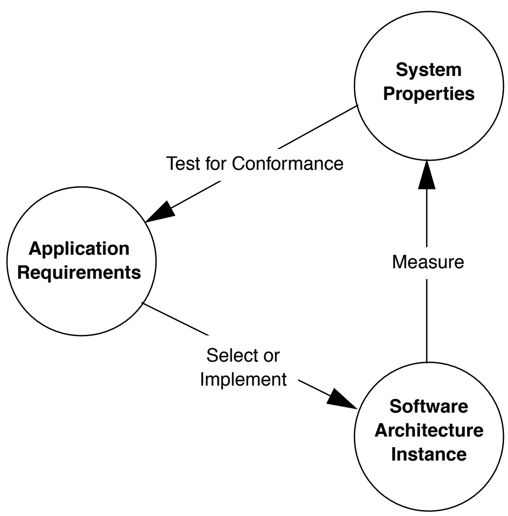


Figure 1. Application Requirements, Software Architecture, and System Properties

Figure 2. shows a similar diagram for multiple instances in a class of similar systems. Each of the instances has the triad of related requirements, architecture, and properties. Each triad is slightly different from the others, indicating the variations within that class of systems. For example, this diagram might represent a collection of applications using different OODBs with different architectures and with differing size, functionality, and performance properties.

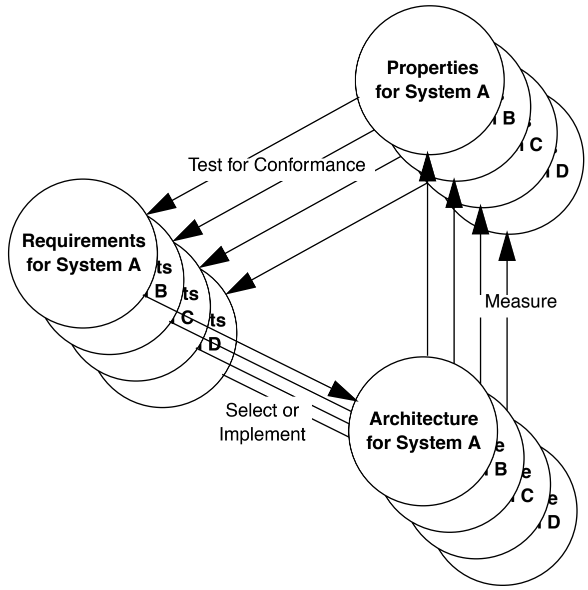


Figure 2. Requirements, Architectures, and Properties for a Class of Systems

Typically, the relationships between application requirements, system architecture, and system properties – and also the relationships between multiple triads – are dealt with in ad hoc ways. In particular, there is often no attempt to capitalize on the commonality among the different instances within the class of similar systems. Relationships among instances might only be found in marketing literature comparisons or limited amounts of published benchmarks.

1.4.1.2. The Approach

In contrast to this ad hoc treatment, we capitalize on the commonality among the different instances within the class of OODB systems by capturing in a software engineering tool the relationships between application requirements, system architecture, and system properties. This is illustrated in Figure 3.

- We use a *parameterized software architecture* to capture the commonality and variance among the instances for the class of OODB systems.
- We capture the relationship between application requirements and OODB architecture instances in a mapping that instantiates the parameterized software architecture based on a requirements definition.
- We capture the relationship between OODB architecture and system properties with OODB architecture modeling and simulation that profiles system properties.
- We capture the relationship between system properties and back to requirements with tests on how well system properties satisfy the requirements.

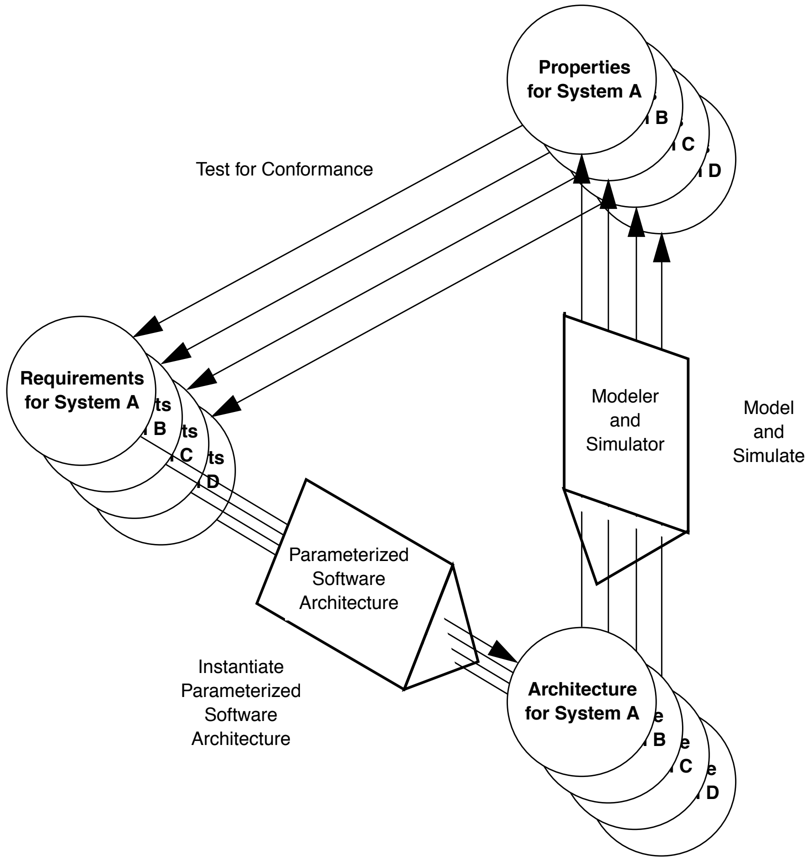


Figure 3. Engineering Tool Support for OODB Requirements, Architectures, and Properties

This approach provides the two key capabilities needed to help developers as they explore their requirements and architectural options in the OODB domain:

<u>Iteratively refine a set of OODB requirements</u>. This capability focuses developers on accurately defining requirements. Starting with an initial best estimate at the application requirements, developers iterate through the cycle of instantiating a software architecture, modeling and simulating the software architecture, testing for conformance, and refining requirements until the simulated system properties satisfy the application requirements. After the final iteration, the resulting set of requirements are referred to as the *baseline requirements* for the application.

<u>Identify an off-the-shelf architecture that best satisfies the baseline requirements</u>. This capability allows developers to select an off-the-shelf OODB system with system properties that best satisfy the baseline requirements. Starting with the baseline requirements (from the previous paragraph), developers select an off-the-shelf system to evaluate and then iterate through the cycle. During the instantiation of the parameterized architecture, inconsistencies between the baseline requirements and the off-the-shelf system are detected and reported to the developer. If there are major inconsistencies, then developers may choose to abandon the evaluation for that off-the-shelf system. If there are minor inconsistencies, developers can resolve the inconsistencies and continue with the architecture instantiation. Then the properties for the off-the-shelf system can be approximated by modeling and simulating the architecture instance. At this point, developers can compare the baseline requirements, architecture, and properties with the off-the-shelf system architecture and properties. The complete cycle can be repeated for other candidate off-the-shelf OODBs until a good match is found for the baseline requirements.

Figure 4. provides a more detailed view of how we implemented support for the approach in an engineering tool. The ovals in the diagram represent actions and tasks, while the rectangles represent output products from the tasks. This diagram has the same cycle of relationships among requirements, architectures, and properties as Figure 3., but there are some additional intermediate steps in these relationships that are relevant to the tool implementation. We illustrate these steps by reviewing the two key capabilities of the tool: iteratively refining a set of OODB requirements and identifying an off-the-shelf architecture that best satisfies requirements.

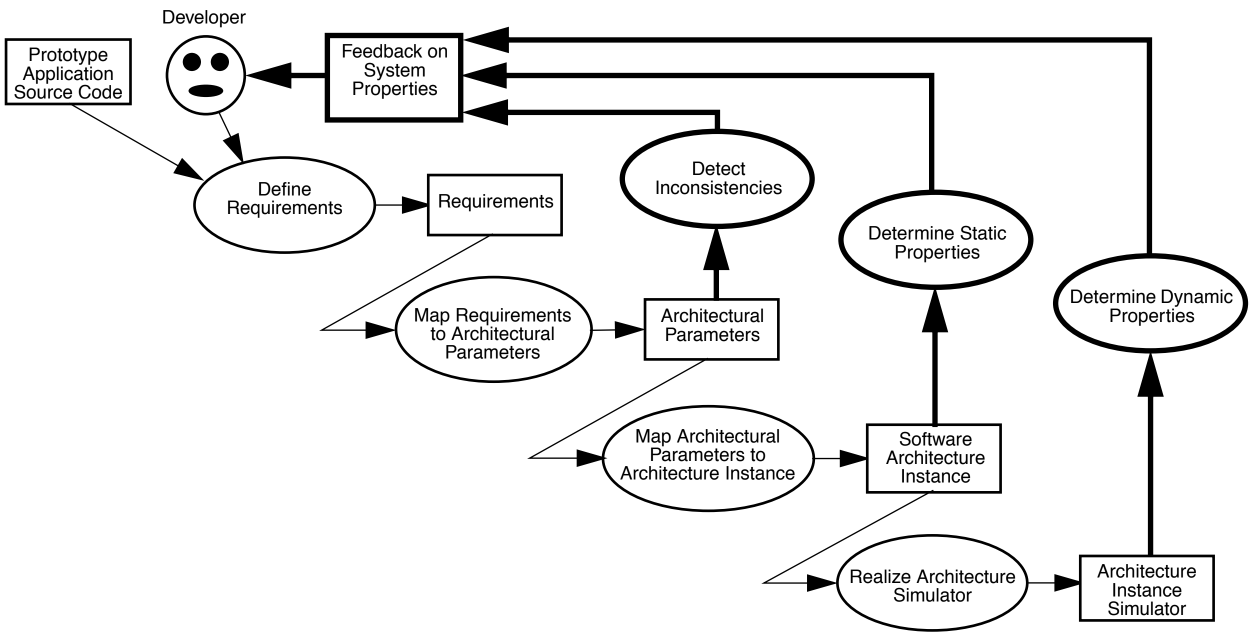


Figure 4. Selecting and Evaluating Software Architectures

To iteratively refine a set of OODB requirements, an initial best estimate of the application requirements is needed. As illustrated in the upper left of Figure 4., the requirements come from two sources, *prototype application source code* and the *developer*. The prototype code is a representative implementation of the application that requires an OODB. Our tool will extract some requirements by scanning the prototype source code. The remainder of the requirements are provided by the developer using a requirements definition editor.

With the initial requirements defined, the next objective is to instantiate a software architecture instance for modeling and simulation. This instantiation is relatively complex, so we partition it into three smaller tasks in the tool. The first task is to *map requirements to architectural parameters*. The next task, *map architectural parameters to architecture instance*, results in a *software architecture instance*, which is a model of a software architecture. The final task, *realize architecture instance*, configures a software architecture simulator for the instance.

The relationship between the software architecture instance and the system properties is provided by three tasks in Figure 4., *detect inconsistencies*, *determine static properties*, and *determine static properties*, all of which contribute to the *feedback on system properties*. The inconsistencies identify differences between baseline requirements and an off-the-shelf OODB. The static properties derived from the software architecture instance include functionality and approximate size. The dynamic properties derived from the simulation include execution times and runtime locality profiles.

The final relationship in the cycle between system properties and requirements is provided by comparing the *feedback on system properties* with the requirements. If this comparison indicates problems with the requirements definition, developers can refine the requirements and make another iteration through the cycle.

To identify an off-the-shelf system that best satisfies the baseline requirements, developers start with the baseline requirements in the *define requirements* task, select a particular off-the-shelf OODB for the tool to use in the evaluation, and then *map requirements to architectural parameters*. The resulting architectural parameters are used to *detect inconsistencies* between the off-the-shelf system and the baseline. The feedback to the developer is used to either modify the architectural parameters to be consistent with the off-the-shelf system (also illuminating the baseline requirements that cannot be fully supported) or abandon the evaluation of the off-the-shelf system. Once the inconsistencies are resolved, then the complete cycle is traversed to produce a software architecture instance, a simulation, and feedback about the static and dynamic properties about the off-the-shelf system.

Elaborating further on Figure 4., the requirements elicited by the tool in the *define requirements* task are those that discriminate among the instances in the class of systems. That is, the requirements that are common to all instances do not need to be specifically enumerated. In our OODB implementation we use 18 discriminating requirements. Examples include requirements for locality, concurrent access, and support of long transactions.

In the *map requirements to architectural parameters* task, architectural parameter values are derived from the requirements. Architectural parameters express the varying architectural features in the class of systems. They discriminate among the architecture instances. In our OODB implementation we use 26 architectural parameters. Examples, include alternatives for lock support, partitioning data among servers, distributed transaction support, and locality.

When the tool is used to evaluate off-the-shelf OODBs, inconsistencies in the architectural parameters can be detected by the tool (shown with a bold arrow pointing upward from the *architectural parameters*). The type of inconsistencies that can be detected in the architectural parameters are combinations of parameter values that are not supported by off-the-shelf architecture instances. These inconsistencies serve as early warnings to developers of a mismatch between the baseline requirements and the off-the-shelf system.

In the *map architectural parameters to architecture instance* task, a software architecture instance is derived from the architectural parameter values. The architecture instance is similar to a software design description. It denotes a configuration for a collection of software architecture components. In our implementation for OODBs there are over 250,000 possible architectural configurations. Some of these configurations represent off-the-shelf systems while others represent custom systems. Note that in the tool implementation for the thesis we model the architecture instances, but we do not implement the software architecture components.

Finally, in the *realize architecture simulator* task, an architecture simulation is configured based on the software architecture instance. In the implementation of our OODB simulator, the internal representation of the software architecture instance is augmented with simulation constants such as processor speed, disk access speed, inter-process communication costs, and so forth. An architecture-independent simulation engine uses the software architecture instance with the simulation constants to perform the simulation on the *prototype application source code*.

1.4.1.3. Roadmap for the Remainder of the Thesis

In Chapter 2, we consider related work in field. Chapter 3 provides a detailed overview of the key points in our modeling and simulation solution to the software architecture instantiation problem. Chapter 4 presents a practical example scenario of a software development team faced the problem of selecting an OODB that best meets their application requirements, and how this team might use our engineering tool to converge on the best choice. Chapter 5 is a high-level design description of the OODB parameterized architecture modeling and simulation tool. Chapter 6 provides the low-level implementation details for the tool. Chapter 7 describes experiments with the tool, applied to the problem of refining OODB requirements and selecting an off-the-shelf OODB for two different applications. Chapter 8 presents our conclusions, based on the experiments, about how well our parameterized architecture modeling and simulation techniques satisfy our hypothesis. In Chapter 9 we explore some of the lessons learned throughout our work. Chapter 10 discusses potential areas for continuing and extending our research.

1.5. Challenges

The fundamental challenge in this work is to capture the traditionally ad hoc relationships between application requirements, software architectures, and system properties for a class of similar software systems. Furthermore, these relationships must be expressed in an engineering mechanism that allows developers to make principled choices within the architectural space for that class of systems.

A self-imposed challenge is to devise an engineering mechanism that is practical and understandable to a broad range of software developers rather than a complex mechanism targeted for highly skilled and trained software engineers. The difficulty here is in controlling the inherent complexity of the relationships and mappings between requirements, architectures, and system properties. This applies both to the construction and use of the mappings and simulators for OODB architectures.

To address these challenges, we decompose the problem into modular sub-problems and then design extensive tool support for each sub-problem. Our tool helps to define requirements on OODBs, to automate the mapping between requirements and OODB architecture instance models and simulators, to simulate architecture instances in the context of a particular application, to collect static and dynamic system properties, and to support the comparative analysis of different OODB architecture instances relative to a particular application.

1.6. Contributions

The primary contribution of this work is the tool that demonstrates how principled engineering choices can be effectively and efficiently made within the design space for a class of OODB systems. We show how these choices can be based on concepts at the requirements level, such that application requirements are used as the input language for discriminating among system instances and then automated mappings can derive software architecture instances from the requirements. We demonstrate that by decomposing this mapping into multiple tasks, the independent concerns are isolated, thereby simplifying development of the tool. We also demonstrate how modeling and simulation play an important role in refining the requirements and in discriminating among instances within the class of OODB systems.

A secondary contribution of the work is a general software engineering approach that can be applied to other classes of software systems. We have defined, implemented, and described our tool and techniques so that they are generally applicable to software architectures rather than being tightly coupled to only the domain of OODB architectures.

Chapter 2. Related Work

In this chapter we discuss areas of research and previous results that relate to our work.

2.1. Software Architecture

This thesis fits within the broad context of research generally referred to as *software architecture*[26]. The study of software architecture deals with large-scale components within a software system, the composition of components, the relationships, constraints, and interfaces among components, the system properties attributed to the composition of components, scalability, ease of structural evolution, structural relationships within a class of similar of systems, rationale behind structural compositions, and so forth[5][25]. The objective of software architecture research is to enable the systematic reuse of large-scale software system structures and designs.

Areas of study within software architecture can be characterized along two dimensions, *domain* and *implementation*[17]. The *domain* dimension ranges from domain-independent to domain-specific. A domain-independent software architecture expresses architectural features for a class of similar systems useful in many application domains, while a domain-specific software architecture expresses architectural features for a class of similar systems in one application domain. The *implementation* dimension ranges from concrete to abstract. Concrete implementations are executable implementations while abstract implementations are analysis or design representations. Following are four major areas of study within software architecture research and where they lie along these two dimensions:

- Frameworks (domain-independent, concrete). Frameworks provide common, application-independent, executable software subsystems with build-time configurable architectures. For example, CORBA is a software architecture framework for distributed, object-oriented applications[12]. The CORBA architecture can be "bound" to different object-oriented languages such as Smalltalk and C++. Once configured, a framework provides an executable subsystem that is used directly in the implementation of a larger application. Frameworks are configured to best satisfy the requirements of a particular application.
- Architectural styles (domain-independent, abstract). Architectural styles capture high-level idioms for design structure of software applications. Examples of architectural styles include blackboard systems, pipes and filters, rule-based production systems, repository-based systems, and object-oriented systems[5][18][19][20]. Given a collection of architectural styles, application developers can evaluate which style most effectively supports the implementation of their application. Each architectural style is largely domain-independent and can be applied to multiple domains. Architectural styles are not expressed in an executable implementation language, but rather are expressed as either formal or informal design specifications for high-level system structure.
- Kits (domain-specific, concrete). Kits are like frameworks in that they capture build-time configurable, executable software systems. However, unlike frameworks, kits are specific to an application domain. Once configured, a kit provides an executable application. Configuration of the kit architecture is based on the requirements for a particular application instance. SEI produced a bibliography of different domain-specific software architecture publications, including papers describing software architecture kits for various domains such as space launch payload software, missile systems, and cartographic applications[24].

- Domain models (domain-specific, abstract). Domain models capture high-level software system structures for a specific application domain. Much of the early work in software architecture focused on domain-specific architecture analysis and modeling[10][21][22][23][24]. The intent of domain modeling is to capture knowledge about domain concepts and corresponding architectural constructs that can be disseminated and reused across different implementation efforts within a well-defined application domain. Examples of application domains for which domain analysis and modeling have been applied include aircraft navigation systems, databases, interactive video servers, industrial process control systems, and telecommunication servers.

Our work in this thesis lies within the general area of the last item in the preceding list – domain models. We explore requirements mappings, architecture models, designs, and simulators in the OODB domain. That is, our work is specific to the OODB domain and abstract in the sense that we deal more with designs, modeling, and simulation of OODBs rather than their executable implementations.

2.2. Domain-specific Software Architectures

Research on *domain-specific software architectures* focuses on large-scale software reuse in well understood application domains. The motivation behind this approach is that narrow, well understood domains have numerous existing software systems that can be studied in detail and the results captured, generalized, and disseminated for reuse[7][13][21].

Although we use the concept of *domain-specific software architectures* as a general area of study, an ARPA program also was given this name (DSSA)[27][28]. The DSSA program focused on software architectures in domains of military control systems such as avionics, command and control, and intelligent vehicle control[29][30].

The study of domain-specific software architectures addresses domain analysis, requirements, software architecture definition, and the implementation of executable components[29]. Because domain-specific software architectures deal with a collection of similar applications, their study focuses on the generalizations (or commonality) and the variants (or differences) within the domain. Generalizations and variants exist in the domain models, requirements, architectures, and executable components.

We share the central premise of domain-specific software architectures that high-leverage software reuse can result from the study of a narrow application domain. However, our work extends and differs from projects such as DSSA in the following ways. First, we focus on the *simulation* of software architectures to provide feedback for refining requirements. Our work, therefore, could be used as a front-end technology for providing accurate application requirements to a DSSA system instantiation technology. Second, we define modular *mappings* to assist in selecting the best software architecture candidates from a set of application requirements. These modular mappings clarify the separation of concerns in mapping domain-specific requirements to domain-specific architecture instances. And finally, we focus on the *evaluation and selection of off-the-shelf systems* rather than assembling and implementing custom systems.

2.3. OODB Toolkits and Extensible OODBs

There have been several efforts at developing flexible OODB systems that can be configured in different ways to satisfy the requirements of different applications. These types of systems are typically referred to as *OODB toolkits* or *extensible OODBs*[31][36]. OODB toolkits and extensible OODBs provide some or all of the architectural components needed to create a specific OODB instance. Developers select, specialize, extend, and integrate components in an attempt to create an OODB that best satisfies their application requirements.

One approach taken by systems like Exodus[32][35] and ObServer[41] is to provide major low-level architectural subsystems such as the persistent storage manager and let developers implement the other high-level architectural subsystems such as language compilers and language runtime systems. This approach provides a loose architectural framework for implementing OODBs, plus off-the-shelf implementations for one or more OODB major subsystems. Developers using these types of systems must have considerable knowledge about OODB requirements, architectures, implementations, and resulting system properties in order to effectively develop application-specific OODBs. Although, considerable development effort is required, it can be significantly less than developing a complete custom OODB from scratch.

Another more aggressive approach is to provide a complete collection of reusable architectural “building blocks” that can be selected, specialized, and configured to create an operational database for a particular application. Dadiasm[37] and Genesis[33][34] are early initiatives using this approach for relational DBMS systems. KALA[38] and TI's Open OODB[39][40] provide building block collections for OODBs. Since these types of systems provide a complete collection of architectural components, less implementation effort is required to create an executable database. However, developers still must have considerable knowledge about OODB requirements, architectures, implementations, and resulting system properties in order to effectively create databases that satisfy a specific set of application requirements.

While OODB toolkits and extensible OODBs focus primarily on executable architectural components, our work is more oriented towards the relationships between OODB requirements, architectures and properties. We focus on requirements definition, OODB modeling and simulation, and the conformance between a set of application requirements and a set of OODB properties.

OODB toolkits and extensible OODBs require knowledge about OODB analysis, design, and implementation in order to create an executable OODB that satisfies application requirements. In contrast, we focus on developers that may not be experts in OODB implementations. We assist developers in defining and refining requirements with a requirements definition tool, mappings from requirements to OODB models and simulations, and automated analysis of OODB modeling and simulation results. We also focus on selecting off-the-shelf OODBs that best satisfy application requirements. Our work, therefore, could help provide front-end requirements definition and architectural analysis prior to using OODB toolkits and extensible OODBs.

2.4. Requirements and Software Architecture

Lane did some early work on the relationship between application requirements and software architecture structures[4]. In that study he demonstrated that the relationship between an application’s functional requirements on the user interface subsystem and the software architecture structures that satisfy the requirements could be captured in a set of *rules*. Lane implemented a tool in which these rules were used to configure an executable user interface subsystem based on a stated set of functional requirements.

Our work supports Lane’s notion that the relationship between requirements and software architecture is an important one. Our work is similar to Lane’s in that we capitalize on the architectural commonality in a domain and that we encode expertise about the domain’s architectural designs in a software engineering support tool.

Our work differs from Lane’s in the following important ways:

1. Lane’s work assumes a well defined set of requirements prior to using his tool. In contrast, we address the problem of defining and validating requirements through the cycle of requirements definition, software architecture modeling and simulation, and conformance tests between system properties and requirements.

2. This cyclic prototyping of requirements is particularly important since, in addition to the functional requirements considered by Lane, we also consider performance requirements. The trade-offs and side-effects associated with performance requirements make them very difficult to accurately define without some type of prototyping and validation.

3. We study the domain of OODB architectures while Lane studied the domain of user interfaces.

4. Lane focused on implementing architectural components and assembling them into executable systems while we focus on implementing and configuring a modeling and simulation tool. This difference reflects our emphasis on prototyping and validating requirements.

5. Lane defined a software architecture generalization based on existing user interface systems, but his tool didn't relate the generalization back to the off-the-shelf systems. Our work relates a software architecture generalization of OODB systems back to the off-the-shelf systems from which the generalization is modeled. This allows us to evaluate how well different off-the-shelf systems satisfy the architectural requirements of a given application.

6. Lane used a rule-based system to implement the relationships between requirements and software architecture. We observed that there is a separation of concerns in these types of relationships, leading us to partition into four different mappings rather than a single mapping like Lane's. One of the contributions of our work is the decomposition of this mapping in four distinct sub-mappings with different concerns: concepts to requirement variables, requirement variables to architectural parameters, architectural parameters to software architecture instance models, and software architecture instance models to executable systems.

7. These four smaller, simpler sub-mappings from requirements to software architectures allow us to use simpler algorithmic mappings rather than encoding a single, complex mapping with a rule-based system similar to Lane's. This algorithmic approach simplified the implementation of our tool.

2.5. Simulation of Software Architectures

The *Simulation and Modeling for Software Acquisition* (SAMSA) project was a study in using simulation and modeling technology to aid in requirements definition, evaluation, purchase, scheduling, projection, development, deployment, and other software engineering related issues in acquiring major software systems[42][43]. The objective of this project was to improve the predictability of acquiring software systems by identify and promoting near-term and long-term research activities for the modeling and simulation of software development projects (costs, risks, elapsed time, staffing, and so forth) and simulating software systems during the early stages of analysis and design.

The SAMSA project consisted of two workshops. Participants were invited from academia, government, and industry. The end-product was a set of research recommendations and example descriptions of potential tools and scenarios based on modeling and simulation of various software engineering tools and tasks.

The final report from SAMSA had a broad scope, considerably larger than the modeling and simulation focus of this thesis. However, the notion of using modeling and simulation to clarify requirements and to evaluate and select off-the-shelf systems was identified as an important area of study. Therefore, this thesis can be viewed as advancing specific topics identified in the SAMSA recommendations.

2.6. Software Engineering Environments

Research in software engineering environments explores ways of providing an integrated collection of software engineering tools to support part of the software engineering life-cycle[11]. Although our work is not a software engineering environment study, our implementation is the most extensive software engineering environment implemented using the Gandalf software engineering environment generator[11]. Our implementation contains twelve different tools integrated around seven modular data collections. Example tools include:

- language-sensitive editors
- semantic analysis tools
- mappings from requirements to architectures
- architectural inconsistency detection tool
- language interpreter
- architecture simulator
- profile report generator
- profile analysis tool

Our application of the software engineering environment generator helps to validate its practicality. First, the Gandalf environment generator allowed us take on an aggressive implementation and validation effort relative to the limited scope of a thesis implementation. Second, our success in integrating a large number of tools into a single environment supports the claim that software engineering environments can effectively support a wide range of software engineering tasks within a uniform environment.

Chapter 3. Overview of Approach

In this chapter we detail our approach for modeling and simulating software architectures to refine requirements and identify off-the-shelf systems with architectures that satisfy those requirements. We also describe how we will validate the claim that this technology does indeed improve our ability to define and refine application requirements and to identify software systems with architectures that satisfy the requirements. In Chapter 4. we will illustrate the practical aspects of the concepts presented in this chapter through a case study.

3.1. Software Development Cycle using Software Architecture Modeling and Simulation

Two problems that application developers are faced with when they attempt to select an instance from a class of systems such as OODBs are (1) defining and refining an accurate set of requirements for the system instance and (2) selecting the system instance that best meets those requirements. We use the software development cycle shown in Figure 3. and Figure 4. to address both of these problems:

- *Defining and refining requirements*. Developers traversing this cycle one or more times to iteratively converge on an accurate set of OODB requirements, referred to as the *baseline requirements*.

- *Selecting among off-the-shelf systems*. Then, using these refined requirements, developers again traverse the cycle one or more times in order to evaluate and compare an off-the-shelf OODB system with the baseline requirements. Additional iterations can be made to evaluate additional off-the-shelf systems. The results from these comparisons help developers to determine the degree to which different off-the-shelf systems meet the baseline requirements and help developers to make an overall buy-versus-build decision.

The cycle from requirements to software architecture to system properties and back to requirements is important in the same sense that prototyping cycles are important in conventional software development – with complex software systems it is difficult to fully understand the requirements and architectural implications the first time through. Following are examples of where multiple cycles of requirements definition, architecture modeling and simulation, and profile analysis are valuable:

- There may be a difference between what developers specify as a requirement and their actual intent. It is only after an instance is simulated that developers recognize that one of the requirements is invalid and needs to be modified.

- There may be side-effects in the architecture when satisfying a requirement in off-the-shelf systems. For example, in a collection of OODB realizations, if all of the architectures that provide long transaction functionality also support multiple users, then an application with long transactions as a requirement would get multi-user support as a side-effect. This may result in a system that is too large or complex for the intended use, so the developer might want to re-think the relative importance of the long transaction requirement.

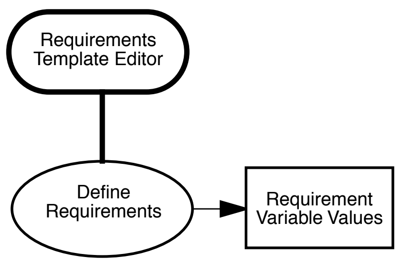
- Developers may find that a requirement was defined incorrectly. For example, a requirement for locality in a cluster of data objects may be contradicted in dynamic profiling data from simulation of a typical usage scenario.

Initially we attempted to implement tool support based on the simple software development cycle in Figure 3. However, the complexity of mapping from requirements directly to an architecture simulation (via the parameterized architecture) led us to explore ways to clarify and simplify this portion of the cycle. Closer examination showed that we could partition the mapping from requirements to architecture simulation into three simpler sub-mappings with separate concerns, (1) map requirements to architectural parameters, (2) map architectural parameters to architectural instance, and (3) realize architecture simulator. This led to the software development cycle in Figure 4. This cycle is equivalent to Figure 3., but it is composed of a larger number of simpler steps.

The following sections provide more detail on the different steps of the software development cycle illustrated in Figure 4. We orient this discussion around the four major efforts and contributions of this work. These are shown as the four tasks from the upper left to lower right in Figure 4.: *define requirements*, *map requirements to architectural parameters*, *map architectural parameters to architectural instance*, and *realize architecture simulator.*

3.2. Defining Requirements

We have mentioned before how the task of defining requirements arises in our overall software development cycle. The first time is when developers make their initial best estimate of the requirements for an OODB application. The second time is on subsequent iterations through the cycle when requirements are refined. In this section we describe more precisely what it means to define requirements in these contexts.



The above figure is an expanded version of the *define requirements* task in Figure 4. In addition to the items shown in Figure 4., we illustrate a tool called the *Requirements Template Editor* that is used to carry out the task, shown by the labeled rounded rectangle positioned above the task oval. The editor provides a template for defining values for a predefined set of requirements. For example, with the OODB requirements template editor there is a slot in the template for defining whether or not the OODB should support multiple concurrent users. The values for the requirements come from two sources, (1) a preprocessor that scans the prototype application source code to infer requirements, and (2) developers interactively define the remaining requirements to the best of their current understanding. Note that as developers iterate through the entire modeling and simulation cycle they may refine the requirements based on new knowledge gained in the previous cycle.

For the requirements definition task, we are interested in only those requirements that discriminate among the different instances that can be realized during modeling and simulation. We refer to these requirements as the *discriminating requirements*. Requirements that are common to all instances do not need to be re-specified each time requirements are defined for an instance. For example, following are two common requirements that do NOT have to be specified in the requirements template editor for OODB instances: (1) a persistence mechanism that allows objects to exist longer than the processes that create them and (2) an access mechanism that allows new processes to locate and use existing persistent objects. The common requirements can be thought of as an implicit part of the requirements definition phase since all OODBs must support them.

Typically there will be significantly more common requirements than discriminating requirements. Since the developers only specify the discriminating requirements, this represents a significant reduction in the complexity and level of effort that developers are faced with.

Discriminating requirements in our modeling and simulation tool are represented as a set of *requirement variables*. For example, a requirement variable that discriminates between instances that include or exclude a particular functionality could be a boolean requirement variable having the name of the functionality. During the requirements definition task, developers specify values for the requirement variables.

We selected the requirement variables in our implementation for OODBs to illustrate representative examples of the most significant dimensions of variability that we observed in the OODB domain. As we will discuss later in Section 9.5.2.1., this set of requirement variables can be extended to capture greater degrees of variance in the domain.

In our OODB architecture implementation we use a structured editing interface for the requirement variable template. The template contains the following requirement variables. The purpose of this list is simply to provide a sense of the number and type of requirement variables. We will define and describe the concepts addressed in these requirement variables in later chapters. Note that indentation in the following bullet items indicates hierarchical structure in requirement variables.

- whether to support long transactions (boolean)
- whether to support concurrent multi-user access (boolean)
- object locality clusters indicating access and locking locality (a set of graphs)
    - expected object utilization in the cluster (binary enumeration)
    - expected size of the cluster (binary enumeration)
    - level of primary persistence reliability (binary enumeration)
    - level of secondary (backup) persistence reliability (2-dimensional enumeration)
        - network separation of primary and backup device (binary enumeration)
        - frequency of backups (quintary enumeration)
- sets of low utilization or large object clusters (sparse cluster sets) with bounded limits on concurrent set access
    - global cluster access across all databases (binary enumeration)
- sets of high utilization and small object clusters (dense cluster sets)
- partitioning of sparse and dense cluster sets with common reliability requirements and bounded partition access per repository
    - expected global partition access across all databases (binary enumeration)
- overall transaction throughput across all databases (binary enumeration)
- reentrant partitions

These requirements address four broad issues in OODB architectures: appropriate functionality, data locality, adequate resources, and load balancing. Note that the requirement variables express those features that vary among OODBs and do not express requirements that are common to all such as persistence and object identity.

The requirement variables are defined to reflect the way that developers reason about application requirements on OODBs. In Sections 5.1.1. and 5.1.8. we discuss our rationale for choosing this particular collection requirement variables. It is based on capturing the most salient and practical discriminators in the OODB domain and also on emphasizing those discriminators that are relevant within the scope of this thesis.

Because of the pragmatic OODB domain focus, these requirement variables are not necessarily designed to possess mathematical properties such as orthogonality or completeness. For example, we will show later that the *reentrant partitions* requirement variable depends on the way that the variables for *object locality clusters* and *sparse* and *dense cluster sets* are defined. Although these requirement variables are not orthogonal, they do reflect the way that developers reason about applications in the OODB domain.

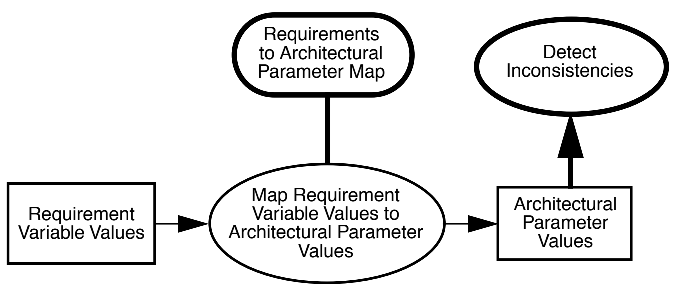
3.3. Mapping Requirement Variables to Architectural Parameters

Once the requirement variable values are defined, the next objective with our approach is to instantiate a software architecture for simulation. To automate the task, we have captured in our software engineering tool the relationship between OODB requirements and OODB architecture instances.

The first step is the mapping from requirement variables to *architectural parameters*. Similar to requirement variables, architectural parameters discriminate among the different architectural instances that can be realized during modeling and simulation. However, in contrast to requirement variables which are defined in terms of concepts from the domain of requirements on system properties such as functionality, size, and performance, the architectural parameters are defined in terms of concepts from the domain of software architectures such as component variants, variations in component integration, and alternate implementations. Therefore, the mapping from requirement variables to architectural parameters is from the domain of system requirements to the domain of system architectures.

An example of an architectural parameter is whether or not to implement read/write locks on objects in an OODB. This is an implementation decision that is determined by two requirements, (1) whether long transactions are needed and (2) whether multi-user access is supported. If either of these requirements are true, then the architectural parameter for objects locks is true. Note that object locking is not a concept in the requirements, but rather is an implementation alternative in the OODB architecture that is included to implement functional requirements such as long transactions and multi-user support.

The architectural features that are common to all instances are not addressed in the architectural parameters, but rather are implicit in the parameterized software architecture model. Since the discriminating parameters are typically few when compared to the fixed architectural features, this distinction reduces the level of complexity and effort involved when implementing and using the mapping from requirement variables to architectural parameters.



Above is an expanded figure for the *map requirement variable values to architectural values* task in Figure 4., augmented with the tool *Requirements to Architectural Parameter Map* that implements the mapping. This mapping isolates the developers that use it from architectural issues. The developers only interact with the requirements definition and do not have to interact with architectural parameters. As a result, they do not have to be experts in the architectural and implementation issues for the class of systems. They only need to understand how to specify what is required of a system instance. This represents another significant reduction in the complexity and level of effort that developers are faced with.

The figure also shows a task to *detect inconsistencies* in the architectural parameter values. Detection of architectural parameter inconsistencies plays an important role in our tool when developers are evaluating off-the-shelf OODBs. When a set of baseline requirements are mapped to architectural parameters, the tool checks to see if the parameter values are consistent with those supported by the off-the-shelf OODB under evaluation. If not, the tool reports the inconsistencies and suggests modifications to the architectural parameters that will resolve the inconsistencies. The tool also notifies developers of baseline requirements that cannot be fully supported due to the inconsistencies in the architectural parameters.

As with the requirement variables, we selected the architectural parameters in our implementation for OODBs to illustrate representative examples of the most significant dimensions of variability that we observed in the OODB domain. As we will discuss later in Section 9.5.2.1., this set of architectural parameters can be extended to capture greater degrees of variance in the domain.

In our OODB architecture implementation we use the following architectural parameters. The purpose of this list is simply to provide a sense of the number and type of architectural parameters. We will define and describe the concepts addressed in these architectural parameters in later chapters. Note that indentation in the following bullet items indicates hierarchical structure in architectural parameters.

- whether to support long transactions (boolean)
- whether object locking is needed (boolean)
- whether distributed transactions are needed (boolean)
- sets of object clusters, referred to as cells
- a set of computational object servers
    - allocation of cells (set of cells)
    - cache size (integer)
    - cache replacement policy (binary enumeration)
    - write-through policy (binary enumeration)
    - the control scheduling model (enumeration)
- OODB clients
    - allocation of cells (set of cells)
    - cache size (integer)
    - cache replacement policy (binary enumeration)
    - write-through policy (binary enumeration)
    - the control scheduling model (enumeration)
- a set of persistent cell managers
    - the allocation of object servers and clients (set of servers and clients)
    - the write-through policy (binary enumeration)
- whether the transaction manager is stand-alone dedicated per repository, stand-alone shared per repository, or embedded in the client.
- whether the object servers are stand-alone dedicated per repository or stand-alone shared per repository.
- whether the persistent cell managers are stand-alone dedicated per repository or stand-alone shared per repository.
- whether the repository manager is stand-alone shared per repository or embedded in the client.
- whether secondary backups are supported (boolean)
    - if true, whether the secondary backup is on LAN or WAN
    - if true, secondary backup frequency (enumeration)

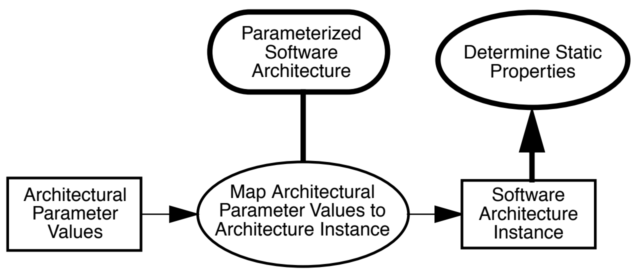
Note that, like requirement variables, the architectural parameters express those features that vary among OODBs and do not express architectural features that are common to all such as persistent storage.

In Sections 5.1.1. and 5.1.6. we discuss our rationale for choosing these architectural parameters. It is based on capturing the most salient and practical discriminators in OODB architectures and also on emphasizing those discriminators that are relevant within the scope of this thesis.

In our implementation we support the evaluation of three off-the-shelf OODB architectures, modeled off of ObjectStoreLite, Objectivity, and ITASCA. These three systems represent a spectrum of OODB architectures ranging from small size with simple features to large size with broad functionality. Overall there are about 50 kinds of inconsistencies that can be reported during the mapping from requirements to architectural parameters for these off-the-shelf systems. For example, for ObjectStoreLite, if the long transactions architectural parameter value is TRUE, then report that ObjectStoreLite does not support them and instruct developers to manually set this architectural parameter to FALSE.

3.4. Instantiating the Parameterized Software Architecture

Once the architectural parameter values are created by the mapping from requirement variable values, the next step towards an architecture simulator is to instantiate the parameterized software architecture using the architectural parameter values. The resulting software architecture instance is a static model of an OODB architecture instance. This architecture instance is used to model the size, structure, and functionality of an OODB instance, plus it is used as the input to our configurable OODB simulator. Note that this step of instantiating the parameterized software architecture is identical for defining baseline requirements via modeling and simulation and for evaluating off-the-shelf OODBs via modeling and simulation.



The above figure is an expanded version of the *map architectural parameters to architecture instance* task in Figure 4., extended to show the *Parameterized Software Architecture* tool that implements the mapping. The parameterized software architecture is an abstract architecture model that takes as input the set of architectural parameters that uniquely specify the instances that can be modeled. It has a framework for assembling an appropriate collection of architectural components into an architecture instance. The software architecture instance produced is a model much like a software design description. It is not source code, nor is it an executable. It describes a collection of software components (such as modules or classes) and how they are integrated to produce an OODB instance. We defined a grammar to describe the architectural parameters and architectural components, plus an imperative mapping between the architectural parameter and architectural component grammars.

The figure also shows a task to *determine static properties* from the software architecture instance. The type of properties that can be determined from the static model of the architecture instance include the processes that implement the OODB architecture, the approximate size of the processes and the overall architecture, the major functions implemented by the architecture, and the overall structural organization of the software architecture instance.

We defined the parameterized software architecture for our OODB implementation to illustrate the common and discriminating features of numerous custom OODBs, OODB research publications, and OODB implementations. As with the requirement variables and architectural parameters, we choose the most significant examples of architectural features that we observed in the OODB domain. As we will discuss later in Section 9.5.2.1., this set of architectural features can be extended to capture greater degrees of detail in the domain.

We first introduce the parameterized architecture in terms of a common framework of OODB architectural features. We refer to this as our *reference architecture*. The reference architecture is an abstraction for expressing the common OODB features in the parameterized software architecture. Following that, we discuss the variant characteristics in the parameterized architecture, how they specialize the reference architecture, and how OODB instances are created from the parameterized architecture.

Figure 5. shows the top-level diagram of the *reference architecture* for our parameterized OODB architecture. The central concept in this diagram is the "repository". Each repository holds the persistent data objects for a particular use of the application. For example, if a personal address book application used an OODB, then each person using the application would have their own repository to store their personal address objects in.

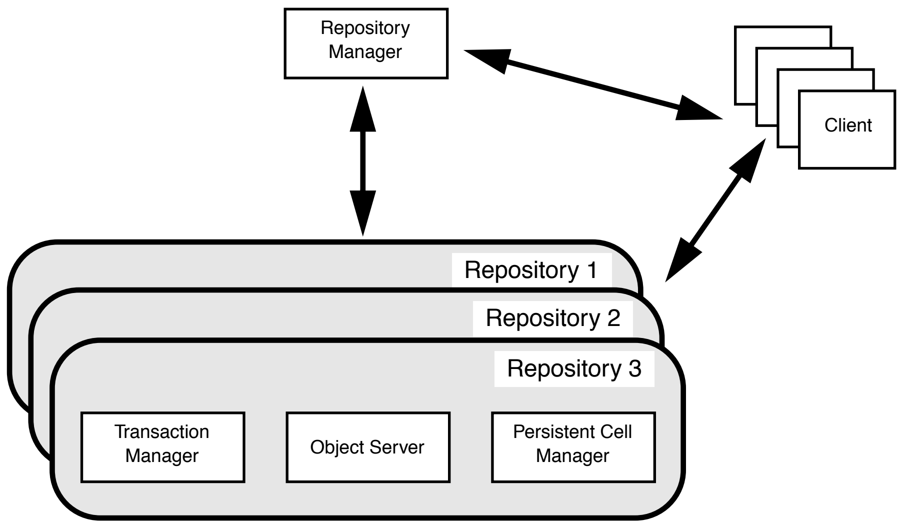


Figure 5. Common Features of the Parameterized Architecture

The Client component is the central site for application execution. It provides the user interface to applications. It also executes methods on application data objects.

The Repository Manager controls the creation and deletion of repositories. The Transaction Manager, Object Server, and Persistent Cell Manager provide operations on repositories and in some cases are independent processes. The Repository Manager creates and monitors these processes. The Repository Manager also helps to establish "bindings" between clients and repositories, so that clients can access the contents of a repository and communicate with the Transaction Manager, Object Server, and Persistent Cell Manager.

The Transaction Manager coordinates the transactions initiated by clients. Object Servers are servers that perform methods on application data objects. Computations for large and sparsely accessed data collections can be allocated to Object Servers rather than to clients in order to improve performance.

Persistent Cell Managers (PCMs) manage the persistent storage for data objects, similar to a file server. Data objects are stored in collections called *cells* on the PCMs. Cells are managed as indivisible units in the Persistent Cell Manager. PCMs provide cells to Clients and Object Servers when the data objects in the cells are needed, and conversely receive cells containing modified data objects from Clients and Object Servers that go back into persistent storage. Cells are "packed" and "unpacked" into data objects in the Clients and Object Servers.

The instantiation of an architectural instance is derived and expressed analogous to a conventional software system configuration. That is, a collection of abstract software components are selected, specialized, and integrated to create a particular instance of a system. In this case, the architectural parameters are used to specify an instance configuration.

The configuration model used in our parameterized OODB architecture is represented as a tree-structured composition of specializable components. Different *configuration nodes* are composed in different ways to form different configuration trees, where each tree represents a software architecture instance. Each configuration node denotes (1) the selection of a software component in the parameterized software architecture, (2) optional specializations (e.g., macro expansions) on the selected component, and (3) an optional set of children, where each child is another configuration node.

An architecture instance is configured by:

- selecting a root configuration node
- performing specializations on the node
- selecting one configuration node for each of the children
- for each of the child selections, recursively doing specializations and child selections

The following list contains some of the major configuration nodes in our parameterized OODB architecture. The purpose of this list is simply to provide a sense of the number and type of configuration nodes. We will define and describe the concepts addressed in these configuration nodes in later chapters.

- <u>Architecture Root</u>
    - *children*: client, object servers, PCM set, repository manager, transaction manager

- <u>Client</u>
    - *alternatives*: reentrant, nonreentrant
    - *children*: embedded repository manager, embedded transaction manager, object and cell services, cell allocations, root role

- <u>Object Server</u>
    - *alternatives*: reentrant, nonreentrant, threaded
    - *children*: object and cell services, cell allocations

- <u>Persistent Cell Manager</u>
    - *specializations*: lazy, eager
    - *children*: object server and client allocations, PCM long transactions, PCM distributed transactions, PCM locking, PCM backup

- <u>Repository Manager</u>
    - *alternatives*: embedded, shared
    - *specializations*: transaction manager management (shared, dedicated, embedded), object server management (shared, dedicated, embedded), PCM management (shared, dedicated)

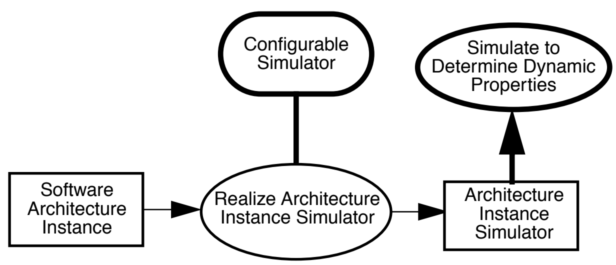
- <u>Transaction Manager</u>
    - *alternatives*: embedded, shared, dedicated
    - *children*: TM long transactions, TM distributed transactions

- <u>Object and Cell Services</u>
    - *alternatives*: single threaded, multi-threaded, locking, no locking, long transactions, no long transactions
    - *specializations*: cache size
    - *children*: replacement policy, OCS distributed transactions

From an architecture instance, our OODB implementation provides feedback to developers on static modeling properties. The type of feedback includes a summary of the configuration nodes, the relative number of processes used by the architecture, approximate sizes for each process, and size contributions by each major component in the architecture.

3.5. Realizing the Software Architecture Simulator

The final step in mapping from requirements to an architecture simulation is to use the software architecture instance to configure our OODB simulator. We refer to this step as *realizing the software architecture instance*. The resulting simulator is used to profile dynamic properties of the architecture instance such as execution times, operation counts, and locality measures. These properties correspond to those illustrated at the top of the development cycle in Figure 3. To complete the development cycle, the dynamic properties are compared to the requirements as part of the *test for conformance* relationship.



Above is an expanded figure for the *realize architecture simulator* task in Figure 4., augmented with the *Configurable Simulator* tool. The configurable simulator is a simulation engine that takes as an input a software architecture instance. This OODB architecture model is used to drive profiling in the simulation engine as it executes the prototype application code. The simulation profiles that are produced approximate the execution of the prototype application on the OODB architecture.

The simulation engine has two parts, the prototype application interpreter that executes the prototype application source code and the architecture instance simulator. As the interpreter executes the application code, it interacts with the architecture instance simulator to profile resource utilization in the architecture. For example, the first time that the application code accesses a persistent object, that object must be loaded from disk. The architecture simulator will increment the persistent storage utilization, data marshalling and unmarshalling costs, cache utilization, network traffic overhead, overall execution times, and so forth.

Following is a list of the dynamic properties collected in our OODB simulator. The purpose of this list is simply to provide a sense of the number and types of dynamic properties. We will define and describe the concepts in later chapters.

- number of object accesses that occur within cells (locality clusters of data objects), plus the associated execution time
- number of object accesses that occur across cells, but within the same process, plus the associated execution time
- number of object accesses that require a remote procedure call, plus the associated execution time
- number of object and cell creations, plus the associated execution time
- number of cell activations from persistent storage into active caches, plus the associated execution time
- number of cell de-activations from active caches to persistent storage, plus the associated execution time
- number of transaction commits and aborts, plus the associated execution time
- execution times by architectural component
- execution times by cell type
- execution times at the application source code statement level

The dynamic properties profiled by the simulator play an important role in our tool. When developers are using the tool to define and refine a set of baseline requirements, the comparison between the properties and the requirements will indicate whether or not the requirements need to be further refined. When developers are evaluating off-the-shelf OODBs, developers can compare the properties from the simulation of an off-the-shelf OODB with the baseline properties of an OODB architecture created from the baseline requirements in order to determine how well the off-the-shelf OODB matches the baseline OODB.

While similar information can be gained by prototyping with evaluation copies of off-the-shelf systems, software architecture simulation is much more cost and time effective since it focuses developers on those requirements that discriminate among the systems and the simulator provides efficient access to the profiling information needed to evaluate system properties. This point is discussed in detail in Section 8.2.

It is also possible to automate some of these comparisons between properties and requirements rather than having developers compare them manually in order to test for conformance. In our OODB implementation, the following two automated tests are simple demonstrations of this potential tool capability:

- *Test for inefficient data clusters*. If a cluster of data objects allocated to clients has a simulation profile indicating low utilization (i.e., that more time is spent activating the cluster than using objects in the cluster), developers are advised by the automated profile analysis to change the utilization requirement on that cluster to “low”. The cluster will then be allocated to a computational server in the architecture in an attempt to improve performance.
- *Test for oversized data clusters*. If a cluster of data objects allocated to a client grows too large, there will be a large latency when activating the cluster. When this is detected in a simulation profile, developers are advised by the automated profile analysis to change the size requirement on that cluster to “large”. The cluster will then be allocated to a computational server in the architecture in an attempt to improve performance.

3.6. Validation of the Technology

An ideal approach to validating the cost/conformance effectiveness of software architecture modeling and simulation technology would involve experimental and control groups developing system instances from a class of systems. The experimental groups would use the modeling and simulation technology to help define requirements, simulate system instances, select a general purpose system, produce a high-level system design, or produce an executable system instance. The control group would not use the modeling and simulation approach, but rather would use conventional software development techniques to analyze, prototype, design, select, and implement system instances. The development costs for both approaches would be measured, including the amortized cost for developing software architecture modeling and simulation technology. The instance conformance (functionality, size, performance profiles) for both approaches would be measured. The hypothesis would be evaluated according to the cost and conformance results.

The level of effort required for such experiments is beyond the scope of resources available to this study. The practical validation approach used in the context of this thesis avoids the labor cost of multiple experimental and control groups working over multiple months and relies on several practical small-scale scenarios using an OODB architecture simulator and intuitive arguments about the relative effectiveness of the technology when compared to conventional software development.

We conducted two experiments on two example applications. The first experiment is on an application that requires a light-weight, single-user, single-machine OODB for a graphical editing application. The second experiment is on an application that requires a more complex, high-performance OODB that supports a large number of concurrent users. In each experiment we explore the effectiveness of both (1) defining and refining OODB requirements for an application, and (2) evaluating and comparing how well different off-the-shelf OODB architectures satisfy the requirements for an application. We then argue in Chapter 8. that the cost/conformance results using this technology is better than with conventional software development techniques.

Chapter 4. Case Study

This chapter is a two-part case study to illustrate how our tool is used. The two parts of the case study demonstrate the two capabilities of the tool: (1) defining and refining OODB requirements for an application, and (2) evaluating and selecting among off-the-shelf OODBs. The tool described and used in this case study is called *UFO*, which stands for *Uniquely Fabricated OODBs*. All of the case study examples were run on the UFO tool.

This case study relies on concepts from the OODB architecture domain and a workflow editor application domain that have yet to be introduced. Rather than cloud the purpose of this chapter (which is to illustrate the capabilities of the UFO tool) with in-depth descriptions of OODB and workflow concepts that will be provided in later chapters, we only provide brief descriptions of the concepts as needed. We also include in this chapter screen dumps from the UFO tool that contain more detail than we will explain at this stage. The purpose of including these screen dumps is to illustrate the amount and type of information that users of the UFO tool work with.

4.1. Introduction to the Problem

Consider the following scenario. A software development team is in the initial stages of designing and building a commercial product: a graphical editor for workflow diagrams. They plan to license an OODB as the persistent storage technology for storing workflow diagrams created by their tool. The target market for this editor is project managers who make detailed diagrams of the tasks, resources, deliverables, and dependencies on their projects. The target platforms for the editor are networked PCs and workstations.

Since the company does not have previous experience using OODBs, they are concerned about accurately defining the system requirements for the OODB and also evaluating and selecting off-the-shelf OODBs. To aid in this effort, they have decided to use the UFO OODB modeling and simulation tool.

UFO allows developers to configure software architecture models and simulations for different custom and off-the-shelf OODBs. These configurations are derived from OODB requirements for an application. The tool helps developers to:

- *define a set of baseline OODB requirements for their application*. Developers use the tool to cycle through initial requirements definition, derivation of an OODB architecture instance, modeling and simulation of the architecture instance, comparing the profile properties with the requirements, and refining requirements. This cycle is repeated until the profile properties satisfy the requirements.

- *evaluate and select among off-the-shelf OODBs*. Developers use the tool to compare baseline requirements, baseline architecture, and baseline properties to off-the-shelf OODB architectures and properties. They can then decide which of the off-the-shelf OODBs most closely conforms to the baseline requirements.

Part 1 of the case study illustrates how the development team would use the tool to define and refine the set of baseline requirements for their OODB. Part 2 of the case study illustrates how different off-the-shelf OODBs are modeled, simulated, and evaluated with respect to the baseline requirements from Part 1.

4.2. Part 1: Prototyping OODB Requirements for the Graphical Editor

The first objective of the developers is to create an accurate set of baseline requirements for the OODB in their application. They use the UFO tool to define the discriminating requirements that distinguish among alternatives in the OODB design space.

The tool maps a set of requirements to an OODB model and an executable simulation that developers can use to test their requirements. Developers can iteratively refine their requirements based on modeling and simulation feedback until a baseline set of requirements is established for their application. These requirements will be the standard by which off-the-shelf OODBs will be evaluated for use in the graphical editor application.

Figure 4. illustrates the steps in using the tool. As shown in the upper-left of that diagram, the first item that developers need is the *prototype application source code* for their workflow editor application.

4.2.1. Prototype Application Source Code and Simulation Scripts

The prototype application source code is a representative implementation of the application that requires an OODB, which in this case study is a prototype of the workflow diagram editor. This prototype should make use of an OODB in a way that closely represents final product. The prototype could be a preliminary implementation of the product or a throw-away effort that is created only for UFO simulation and modeling.

Also needed are *simulation scripts*. The simulation scripts emulate the external system stimulus for typical application scenarios. For example, with the workflow editor application, the simulation scripts stimulate the editor to emulate typical user editing scenarios.

UFO provides a structure-oriented editor that developers use to write the prototype application source code. The editor guides developers in creating syntactically correct programs.

The programming language provided for writing the prototype code is a UFO-specific object-oriented language with OODB constructs for transactions, persistence, and so forth. UFO uses a single programming language that is common to all of the different OODB architecture instances supported by the UFO tool. In this way the prototype application code that drives the OODB simulations can be written once and used repeatedly as different OODB architectures are instantiated, modeled, and simulated. Details and rational for the UFO application programming language are presented in Section 5.1.2.

4.2.2. Defining the Requirement Variable Values

After the prototype code and simulation scripts are written, developers are ready to begin the cycle, shown in Figure 4., of defining requirements, instantiating OODB architecture instances, modeling and simulating the OODB architectures, comparing the modeled and simulated system properties with the requirements, refining the requirements as necessary, and repeating the cycle. The first step is to define the initial set of requirements on the OODB for the workflow editor application using the UFO tool. The concepts and terminology for this activity were introduced in Section 3.2.

UFO provides a requirements template editor for defining requirements. Figure 6. illustrates an empty template with the eight top-level requirements. Developers use this template to specify values for the requirement variables. The italicized terms (all beginning with a '$') are placeholders for requirement variable values. Two of these requirement variable values, the first and the last, are automatically derived by the tool scanning the prototype application source code, while the other requirement variable values are provided by developers. The template tool is a structure-oriented editor and therefore can assist users in constructing legal values. For example, if a developer moves the user cursor to the *Concurrent Users* requirement variable and enters a '?', the tool will display the legal values for that variable as *TRUE* or *FALSE*.

```
Long Transactions: $long_transactions_rv

Concurrent Users: $multi_user_rv

Locality Clusters:
   $locality_cluster_rv

Sparse Cluster Sets
   $cluster_set_rv

Dense $dense_cluster_set_rv

Cluster Set Partitions
   $cluster_set_partition_rv

Global Transaction Throughput: $global_transaction_throughput_rv

Reentrant Partitions: <Automatic Derivation Complete. Internal Representation Not Shown>
```

Figure 6. Requirement Variables Template

The first requirement variable, *long transactions*, is a boolean variable indicating whether or not support for long transactions is required for the OODB. Long transactions that are supported by the OODB simulator are persistent, nested transactions that can exist longer than the processes that create them. They are analogous to experimental workspaces in revision control systems. The tool scans the prototype code for use of long transaction constructs and then automatically sets the value of the long transaction requirement variable value. In this case, the developers did not need or use this functionality in their application, so the value is set to *FALSE*:

```
Long Transactions: FALSE
```

The *concurrent users* requirement variable is a boolean variable indicating whether or not the OODB must support multiple users concurrently editing the same workflow diagram. The developers choose *FALSE* for this requirement variable value:

```
Concurrent Users: FALSE
```

The next three requirement variables are for identifying and characterizing locality in the application's persistent data objects. Two or more data objects exhibit locality if there is a significant probability that when one of the objects is accessed that the other objects will also be accessed. A collection of data objects that exhibit locality is referred to as a *cluster*. Locality is important since OODB performance overhead can often be reduced by orders of magnitude when a cluster of objects is stored, loaded, transferred, and cached as a group rather than one object at time. UFO allows developers to explicitly express as requirements that collections of objects be clustered and managed by the OODB architecture in ways that improve application performance.

Figure 7. illustrates a completed definition for the *Locality Clusters* requirement variable with three locality clusters named *Root*, *Composite*, and *PContent*. *Root* is the initial object in a workflow diagram that all workflow editor processes access. The *Composite* cluster is for composite tasks in the workflow model and is the primary modular unit of editing with the workflow editor. Application users will typically use the workflow editor to edit the collection of objects in a single Composite cluster over a contiguous and extended period of time, which results in locality. The *PContent* cluster is for content of *Product* objects in a workflow diagram, such as structured documents, design descriptions, spreadsheets, or other work products. This is another unit of editing locality among objects.

```
Locality Clusters:

   Cluster: Root
      Roles:
         Role root_root is Class VPML_Root
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: Composite
      Roles:
         Role comp_root is Class CompositeTask
         Role prim is Class PrimitiveTask
         Role res is Class Resource
         Role prod is Class Product
         Role pset is Class ProductSet
         Role pmem is Class ProductMember
         Role pflo is Class ProductFlow
         Role fchn is Class FlowChain
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: PContent
      Roles:
         Role pcont_root is Class ProductContent
         Role lnode is Class Node
         Role rnode is Class Node
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE
```

Figure 7. Locality Clusters

For each cluster there are nested requirements for *Roles*, *Utilization*, *Size*, and *Reliability.* Roles specify the datatypes of the objects that belong to the cluster. Utilization indicates the relative number of objects in the cluster that are typically accessed whenever one of the objects in the cluster is accesses. The value for utilization can either be *HIGH* or *LOW*, where high means 50% or greater typical access of objects in the cluster. Size can have values of *LARGE* or *SMALL*, where large means 1 megabyte or larger typical runtime cluster size. The developers estimate that all three of their locality clusters are high utilization and small size. As we will later see, simulation provides more precise profile feedback on the utilization and size properties of clusters such that requirements can be refined as necessary for another iteration through requirements definition, architecture simulation, and property feedback.

Cluster reliability has two aspects, primary and secondary reliability. Primary cluster reliability allows developers to make trade-offs between requirements for higher performance and fault tolerance. Primary reliability can have values of *HIGH* or *LOW.* High means that committed modifications will survive system crashes because committed modifications are always written to persistent storage. Low reliability means that system crashes may causes recent commits to be lost since committed modifications are written “lazily” to disk in order to improve performance. By delaying writes to disk, several updates to an object can be collapsed into a single write.

Secondary reliability relates to automated backups for protecting against media failure. The possible values are *NONE* (indicating that persistent media is not automatically backed up by the OODB) or values that specify different automated backup frequencies and destinations. The trade-off associated with this requirement is a larger OODB footprint for automated backups versus the extra manual effort, less control, and potential risks without automated backups.

The developers in this case decide that workflow editing is likely to be an intellectually challenging activity and therefore decide that they want to protect users as much as possible from editing losses in the face of system crashes. They therefore declare the primary reliability to be HIGH. They also decide that the normal disk backups performed in the target environments of networked PCs and workstations are sufficient to protect against media failure, so they declare no additional secondary reliability requirements on the three clusters.

After the locality clusters are defined, developers move to the next two requirement variables, *Sparse Cluster Sets* and *Dense Cluster Set*. For these two requirements, developers assign each locality cluster to either a sparse cluster set or the dense cluster set. Locality clusters with *LOW* utilization or *LARGE* size (as defined above) will exhibit better performance when assigned to a sparse cluster set, while the remaining clusters will perform better when assigned to a dense cluster set. At runtime, clusters in the dense cluster set are operated on by the client process while clusters in the sparse cluster sets are operated on by server processes. Each sparse cluster set is allocated a separate server process.

The developers identify all three of their clusters as dense and assign them to a dense cluster set called *DCS*:

```
<No Sparse Cluster Sets>

Dense Cluster Set DCS
   Clusters:
       Root
       Composite
       PContent
```

For each cluster set there is a requirement variable called *Global Cluster Set Contention*. When large numbers of concurrent users are expected to be using multiple repositories with a high contention for cluster sets, then the value is *HIGH*, else *LOW*. At runtime, high contention cluster sets will have one server process for each repository rather than a single shared server process for all repositories. Since the workflow application will be single-user (implying no contention), developers set the value of this requirement variable to *LOW* on the dense cluster set, DCS, that holds their three locality clusters:

```
<No Sparse Cluster Sets>

Dense Cluster Set DCS
   Clusters:
       Root
       Composite
       PContent
   Global Cluster Set Contention: LOW
```

While the sparse and dense cluster sets determine the client and server processes that operate on clusters, the next requirement variable determines the persistent storage devices that store the clusters. Developers assign each sparse and dense cluster set to a *Cluster Set Partition*. A cluster set partition will be mapped to a persistent storage process in the OODB architecture. Developers in this case only have the single dense cluster set DSC (defined above), so they create a single cluster set partition, CSP, and assign the dense cluster set DCS:

```
Cluster Set Partitions
      Partition CSP
         Cluster Sets:
            DCS
```

For each cluster set partition there is a requirement variable called *Global Partition Contention*. When large numbers of concurrent users are expected to be using multiple repositories with a high contention for cluster set partitions, then the value is *HIGH*, else *LOW*. At runtime, high contention cluster set partitions will have one persistent storage process for each repository rather than a single shared persistent storage process for all repositories. Since the workflow application will be single user (implying no contention), developers set the value of this requirement variable to *LOW* on the cluster set partition CSP:

```
Cluster Set Partitions
    Partition CSP
        Cluster Sets:
            DCS
        Global Partition Contention: LOW
```

The next requirement variable indicates required transaction throughput rate for the OODB, *HIGH* or *LOW*. Applications that require high transaction throughput rates will have one transaction server process for each repository, while applications that require low transaction throughput rates will have a single shared transaction server process for all repositories. Since the workflow application will be single-user (implying a relatively small number of transactions), developers set the value of this requirement variable to *LOW*:

```
Global Transaction Throughput: LOW
```

The final requirement variable specifies which processes in the OODB architecture need to be reentrant and which ones can be non-reentrant. Non-reentrant processes are simpler and smaller at runtime since they don't require remote procedure call (RPC) call stacks and related reentrant mechanisms. Requirements for reentrant processes depend on (1) the call graph cycles among data objects in the prototype application source code, (2) the way that data objects are partitioned into locality clusters, and (3) the way that locality cluster are allocated to different processes via the sparse and dense cluster sets. The UFO tool has access to all of this information, so it can automatically derive those OODB process partitions that need to be reentrant. The representation of this information is an internal data structure in the tool and not presented as an external textual representation in the requirements template. This is indicated in the requirement variable as follows:

```
Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

4.2.3. Mapping Requirement Variables to Architectural Parameters

Once the developers have defined the requirement variable values, their next objective is to instantiate the UFO tool's parameterized OODB architecture using the requirement variables. The resulting OODB architecture instance will allow developers to model and simulate an OODB that satisfies the requirements and to determine from the modeling and simulation profiles if the properties of the architecture satisfy the intent of the requirements.

There are three steps in going from requirements to an OODB simulator (see Figure 4. on page 7). The first step is to map requirement variables to architectural parameters. This step was introduced in Section 3.3. The intent of this step is to map from concepts in the domain of discriminating requirements to concepts in the domain of architectural variations.

Developers invoke a command in the UFO tool that automatically derives architectural parameter values from the requirement variable values. The architectural parameter values that are produced by the tool from the requirement variable values above are shown in Figure 8. Following is a brief description of how each of these architectural parameter values is derived by the tool.

```
Long Transactions: FALSE
Object Locking: FALSE
Distributed Transactions: FALSE
Cell Declarations:
   Cell: Root

       Role Declarations:
          Role root_root is Class VPML_Root


   Cell: Composite

       Role Declarations:
          Role comp_root is Class CompositeTask
          Role prim is Class PrimitiveTask
          Role res is Class Resource
          Role prod is Class Product
          Role pset is Class ProductSet
          Role pmem is Class ProductMember
          Role pflo is Class ProductFlow
          Role fchn is Class FlowChain


   Cell: PContent

       Role Declarations:
          Role pcont_root is Class ProductContent
          Role lnode is Class Node
          Role rnode is Class Node



Client:
   Cell Allocations:
       Root
       Composite
       PContent

   Cache Size: 1000000
   Replacement Policy: LRU
   Write-through Policy: EAGER
   Control Scheduling: NON-REENTRANT


<No Object Servers>
Persistent Cell Managers:
   Persistent Cell Manager: CSP
       PCM Allocations:
          CLIENT

       Write-through Policy: EAGER



Transaction Manager Integration: EMBEDDED
Object Server Set Integration: EMPTY Object Server Set
Persistent Cell Manager Integration: SHARED
Repository Manager Integration: EMBEDDED
Backups: No Backups
```

Figure 8. Architectural Parameter Values

The value for the Long Transactions architectural parameter comes directly from the Long Transactions requirement variable, which in this case is false. The value for Object Locking is false when the requirement variable for concurrent users is false and the requirement variable for long transactions is false (performance is enhanced and system size is reduced when object locking is eliminated from the architecture). Distributed Transactions is false because the requirement values imply that only a single architectural process will participate in a transaction. The Cell Declarations are mapped directly from the Locality Clusters in the requirement variables.

The Client architectural parameter describes the architectural variants for the client processes. The Cell Allocations to the client are derived from the Dense Cluster Set in the requirement variables[1]. These cells have small size and high utilization, which makes it effective to copy the objects in these cells from persistent storage to the client and perform computations with them there. Cache size for clients is initially set to 1 megabyte until simulation data is available on cache performance. The Replacement Policy is always set to *LRU* (this feature is included for future tool enhancements). The Write-through Policy is derived from the Primary Reliability requirement variable values for the clusters allocated to the client. If any of the clusters has a high primary reliability requirement, then the write-through policy is *EAGER*, else *LAZY.* Control Scheduling is derived from the Reentrant Cluster Sets requirement variable. The client's Control Scheduling will be *REENTRANT* if the dense cluster set in the Reentrant Cluster Sets requirement variable is reentrant, otherwise *NON-REENTRANT.*

The next four architectural parameters determine the way the transaction manager, object servers, persistent cell managers, and repository manager are integrated into the overall architecture. Some of these may be *EMBEDDED* into the client process, some can be *SHARED* process for all repositories in the OODB, and others can be replicated and *DEDICATED* to each repository in the OODB. The actual values depend on whether or not the OODB has a multi-user requirement, the requirement variable values for partition and cluster set contention, and the transaction throughput requirement.

The final architectural parameter, Backups, is for automatic persistent media backup functionality and policy. In this case, none of the clusters had secondary reliability requirements, so no backup support is indicated in the architectural parameter value.

4.2.4. Mapping Architectural Parameters to Software Architecture Instances

Once the architectural parameters have been derived, the next step for developers is to invoke a command in the UFO tool that specializes the parameterized software architecture. This creates a software architecture instance using the architectural parameter values. The software architecture instance is a static OODB model that is used for two purposes, (1) to provide feedback on system properties such as size and functionality, and (2) to configure an OODB simulator for the architecture (see Figure 4. on page 7). This step was introduced in Section 3.4.

Recall that an OODB architecture instance is represented as a tree-structured composition of *configuration nodes* (Section 3.4.). Different configuration nodes are composed in different ways to represent different OODB architecture instances. Each configuration node denotes (1) the selection of a software component in the parameterized software architecture, (2) optional specializations on the selected component, and (3) an optional set of children, where each child is another configuration node.

A textual description of the OODB architecture instance, output by the UFO tool, is shown in Figure 9. Indentation is used to indicate nested composition. For example, the top few lines of the listing show that the *Embedded Repository Manager* is a nested sub-component of the *Non-reentrant Client*. The description of each architectural component includes the choices made for architectural variability in the component. For example, in the third architectural component from the top, *Embedded Transaction Manager*, the functionality to implement nested persistent transactions or distributed transactions is not included. The tool determined from the architectural parameter values that neither of these functions were required, so they were omitted to simplify the architecture.

The UFO tool can also use the static OODB architecture model that was derived in this step to project the number of OODB processes and their approximate sizes. Developers issue a command to the tool to request this information. The report is illustrated in Figure 10. It shows the number of processes and the runtime space demands for each process, plus the source of some of the space demands.

---

1. Note that when clusters are implemented in the architecture they are called *cells*.

```
Non-reentrant Client:

   Embedded Repository Manager:

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:
      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED

   Cell Allocations:
      Cell: Root

          Role Declarations:
             Proxy Role root_root is Class VPML_Root ...

      Cell: Composite

          Role Declarations:
             Proxy Role comp_root is Class CompositeTask ...
             Role prim is Class PrimitiveTask ...
             Role res is Class Resource ...
             Role prod is Class Product ...
             Role pset is Class ProductSet ...
             Role pmem is Class ProductMember ...
             Role pflo is Class ProductFlow ...
             Role fchn is Class FlowChain ...

      Cell: PContent

          Role Declarations:
             Proxy Role pcont_root is Class ProductContent ...
             Role lnode is Class Node ...
             Role rnode is Class Node ...

   Root Role: Root.root_root
   ...

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Processes allocated to PCM:
          CLIENT
      Write-through: EAGER
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS
      ...

Repository Manager.   Integration: EMBEDDED

   <Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED

   <Embedded in client.  See Client description.>
```

Figure 9. Textual Representation of Architecture Instance

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 1
Total size of shared processes: 2000000 Bytes

Non-reentrant Client:

Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:

   Total size: 50000 Bytes

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:

   Total size: 200000 Bytes

      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.

   Total size: 1100000 Bytes

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2000000 Bytes

   Persistent Cell Manager: CSP

   Write-through: EAGER
   Total size: 2000000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object Locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>
```

Figure 10. Architecture Properties

4.2.5. Mapping Software Architecture Instances to Executable Simulations

The developers are now ready to take the final step to realizing the OODB simulator for their application. They invoke a command in the UFO tool that configures an OODB simulation engine based on the software architecture instance derived in the previous step. The resulting OODB simulator can then be used to execute the prototype application code and provide profile developers with feedback on how well the OODB architecture satisfies the application requirements. The task of realizing the OODB simulator was introduced in Section 3.5.

The performance of a given OODB architecture will depend on underlying infrastructure such as computer processor speed, disk I/O rates, and network speed. Developers use the UFO tool to configure the simulator with *simulation constants* corresponding to the computing infrastructure that their application will run on. The UFO tool comes with examples of the simulation constants for various computers and networks so that developers can choose values that most closely correspond to their infrastructure. The values that the developers chose for the client machines and persistent cell manager machines are shown in Figure 11.

```
Non-reentrant Client:
...
   Processor Constants:
      Processor Speed:   33 MHz
      Intra-Cell Call Cost:    In: 50 cycles,   Out: 50 cycles
      Inter-Cell Call Cost:    In: 100 cycles,   Out: 50 cycles
      RPC Call Cost:           In: 550 cycles,   Out: 550 cycles
      Object Activation Cost:  100 cycles
      Object Passivation Cost: 100 cycles
      Object Creation Cost:    500 cycles
      Cell Activation Cost:    1000 cycles
      Cell Passivation Cost:   1000 cycles
      Cell Creation Cost:      600 cycles
      Cell Lock Check Cost:    50 cycles
...
Persistent Cell Manager. CSP
...
  Persistent Cell Manager Constants:
      Processor Speed:          66 MHz
      Object Activation Cost:   200 cycles
      Object Passivation Cost:  200 cycles
      Object Creation Cost:     1000 cycles
      Object Commit Cost:       200 cycles
      Object Abort Cost:        10 cycles
      Cell Activation Cost:     10000 cycles
      Cell Passivation Cost:    10000 cycles
      Cell Creation Cost:       15000 cycles
      Cell Commit Cost:         10000 cycles
      Cell Abort Cost:          5000 cycles
      Cell Lock/Unlock Cost:    5000 cycles
```

Figure 11. Processor Constants

4.2.6. Evaluating Requirements with Modeling and Simulation Feedback

The developers next use the prototype application code and the simulation scripts to drive a simulation. The purpose of the simulation is to help developers converge on set of baseline requirements. Profiles from the simulation may identify performance problems (such as poor locality), indicating that developers need to refine requirements and make another iteration through the architecture instantiation, modeling, and simulation cycle. When the simulation profile satisfies the intent of the requirements, the requirements serve as the baseline by which off-the-shelf OODBs are evaluated. The OODB baseline requirements, models, and simulation profiles created in this step are compared in the next section with requirements, models, and simulation profiles for off-the-shelf OODBs. Note that the requirements and architectures supported in the UFO tool are not constrained to model only off-the-shelf OODB implementations, but rather can be more finely tuned to match application requirements. This supports the option of producing a hand-tailored OODB if that becomes necessary.

During simulation the UFO tool collects profile data, both on the source code and on the OODB architecture model. We describe next how the developers interpret some of these results.

Figure 12. shows the top-level architecture profile, summarizing some of the key execution properties. The values for *Intra-cell*, *Inter-cell*, and *RPC* calls made and received relate to the cell locality performance[2]. Intra-cell is for access from one object to another within the same cell. Inter-cell is for object access between cells, but on the same processor. RPC is for object access that requires remote a procedure call, such as when two cells reside on different object servers. In this case study, all cells reside on the client, no RPCs are seen in the profile.

```
Note: All time measurements are expressed in micro-seconds.

Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 4747
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 146882
Activations:     Objects: 13441    Cells: 98
                 Time: 99277
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 19524
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1464740
```

Figure 12. Top-level Architectural Profile

If the locality cluster requirements have been specified well, then the intra-cell calls will dominate the inter-cell calls, and the intra-cell calls will dominate the RPCs. If developers find that this is not the case, then they can use more detailed information later in the profile to identify where locality is breaking down and subsequently redefine the associated requirements. However in this case, the developers note that the profile reflects good locality.

The profile values shown for *Activations*, *Passivations*, *Commits*, *Aborts*, and *Creations* all involve inter-process communication and persistent storage operations. (*Activation* is when objects are read from persistent storage into processor memory for computation. *Passivation* is the converse of activation; Objects are moved from processor memory to persistent storage.) These are relatively expensive operations that typically represent a significant portion of the overall execution time, and are therefore a key subject of performance analysis. One potential problem area is when a lot of time is spent activating and passivating cells that are rarely used after they are activated. The UFO tool provides some automated profile analysis to help identify this situation.

To test for this, a general heuristic used by the tool is that the ratio of profile value for activations should not be less than 50% of the profile values for intra-cell, inter-cell, and RPC calls received. In the profile from Figure 12., the ratio indicates a very high utilization of over 500% (i.e., each object activated is accessed on average 5 times before deactivation). If this ratio had been too low, the tool would have reported this to developers and they could use more detailed information later in the profile to identify the low utilization cells and then modify the utilization requirement value for the appropriate locality cluster to *LOW*. This would result in the cells being allocated to object servers that can keep large caches of low utilization cells, lowering overall activation costs without bloating client caches.

---

[2] Recall that cells correspond to the runtime implementation of locality clusters.

Figure 13. shows the client profile, summarizing key information that is specific to the client process. The same type of information is available as with the previous top-level summary, but only as related to the client. In this case there are no object servers in the architecture, so many of the overall profile figures can be accounted for in the client. However, note that total activation and passivation costs in the client are less than the total for the overall architecture. This is because the persistent cell managers also contribute a significant portion of the total activation and passivation costs. This illustrates how performance problems noted at higher levels in the architecture can be traced into the subcomponents that contribute to the performance profile.

```
Non-reentrant Client:

   Intra-Cell Calls Made:     46604    Time: 70607
   Intra-Cell Calls Received: 46604    Time: 70605
   Inter-Cell Calls Made:     1559    Time: 2361
   Inter-Cell Calls Received: 1567    Time: 4747
   RPCs Made:                 0    Time: 0
   RPCs Received:             0    Time: 0

   Object and Cell Services:

      Creations:        Objects: 4588    Cells: 32
                        Time: 70095
      Activations:      Objects: 13441    Cells: 98
                        Time: 43699
      Passivations:     Objects: 4593    Cells: 37
                        Time: 15039
      Object and Cell Services.

         Threads: SINGLE
         Object locking: NOT SUPPORTED
         Persistent nested transactions: NOT SUPPORTED
         Cache size: 1000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: NOT SUPPORTED
         Write-through policy: EAGER


   Total Execution Time: 779088

   Embedded Repository Manager:

      Number of repository creates:   1
      Number of repository deletes:   0
      Number of repository bindings:  7

   Embedded Transaction Manager:
      Nested persistent transactions: NOT SUPPORTED
         Number of persistent transaction creates:   0
         Number of persistent transaction commits:   0
         Number of persistent transaction aborts:    0
         Number of persistent transaction changes:   0
      Transient transactions:
         Number of transient transaction commits:   7
         Number of transient transaction aborts:
```

Figure 13. Client Profile

Figure 14. shows the profiles for the cells allocated to the client (which in this case are all of the cells). Of particular interest to the developers are the *Composite* and *PContent* cell since these are where editing locality is expected and where good performance is crucial to the success of their application. The locality profile (intra-cell versus inter-cell) and utilization profile (activation versus intra-cell and inter-cell) for these cells are good.

Figure 15. shows the persistent cell manager profile. These figures illustrate how the operations dealing with persistence contribute significantly to the overall execution costs and why accurately characterizing locality and utilization in order to minimize persistence operations is so important.

With the information and experience that the development team has gained in prototyping, they now feel confident that they clearly understand the OODB requirements for their graphical workflow editor application. They can now proceed to evaluate the off-the-shelf OODB options based on their baseline requirements.

```
Cell Allocations:
   Cell: Root
      Intra-Cell Calls Made:     939    Time: 1422
      Intra-Cell Calls Received: 939    Time: 1422
      Inter-Cell Calls Made:     1331    Time: 2016
      Inter-Cell Calls Received: 8    Time: 24
      RPCs Made:                 0    Time: 0
      RPCs Received:             0    Time: 0
      CREATIONS:        Objects: 1    Cells: 1
                        Time: 33
      ACTIVATIONS:      Objects: 6    Cells: 6
                        Time: 200
      PASSIVATIONS:     Objects: 6    Cells: 6
                        Time: 200
      Total Execution Time: 5317
      Maximum Cell Size: 160

   ...

   Cell: Composite
      Intra-Cell Calls Made:     3981    Time: 6029
      Intra-Cell Calls Received: 3981    Time: 6027
      Inter-Cell Calls Made:     228    Time: 345
      Inter-Cell Calls Received: 1337    Time: 4051
      RPCs Made:                 0    Time: 0
      RPCs Received:             0    Time: 0
      CREATIONS:        Objects: 195    Cells: 7
                        Time: 3081
      ACTIVATIONS:      Objects: 719    Cells: 24
                        Time: 2906
      PASSIVATIONS:     Objects: 195    Cells: 7
                        Time: 803
      Total Execution Time: 23242
      Maximum Cell Size: 9472

   ...

   Cell: PContent
      Intra-Cell Calls Made:     41684    Time: 63156
      Intra-Cell Calls Received: 41684    Time: 63156
      Inter-Cell Calls Made:     0    Time: 0
      Inter-Cell Calls Received: 222    Time: 672
      RPCs Made:                 0    Time: 0
      RPCs Received:             0    Time: 0
      CREATIONS:        Objects: 4392    Cells: 24
                        Time: 66981
      ACTIVATIONS:      Objects: 12716    Cells: 68
                        Time: 40593
      PASSIVATIONS:     Objects: 4392    Cells: 24
                        Time: 14036
      Total Execution Time: 248594
      Maximum Cell Size: 49024
```

Figure 14. Cell Profiles

```
Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Write-through: EAGER
      Creations:    Objects:  4588    Cells:  32
                    Time:  76787
      Activations:  Objects:  13441    Cells:  98
                    Time:  55578
      Passivations: Objects:  4593    Cells:  37
                    Time:  19524
      Commits:      Objects:  4593    Cells:  37
                    Time:  19524
      Aborts:       Objects:  0    Cells:  0
                    Time:  0
      Total Execution Time:   685652
```

Figure 15. Persistent Cell Manager Profile

4.3. Part 2: Selecting an Off-the-Shelf OODB for the Graphical Editor

The next objective of the developers is to find an off-the-shelf OODB that closely conforms to the baseline requirements. The developers use the UFO tool to compare the baseline requirements and the corresponding architecture instance and properties to those of different off-the-shelf OODBs.

To evaluate an off-the-shelf OODB, developers compare the architecture instance model and simulation profile for the off-the-shelf OODB to the baseline requirements, architecture instance model, and simulation profile in order to better understand how well the off-the-shelf OODB conforms to the baseline requirements. Developers follow a similar cycle of tasks as they did for defining and refining requirements (i.e., the cycle illustrated in Figure 4.). However, there are two primary differences. First is that additional operations are activated in the tool to detect inconsistencies between the baseline architectural parameters and a given off-the-shelf architecture (as introduced in Section 3.3.). This *Detect Inconsistencies* task is illustrated in Figure 4. Second is that the intent of developers is no longer to refine the best possible set of baseline requirements. Rather, the developers' intent is to determine how well the off-the-shelf OODB modeling and simulation properties satisfy the baseline requirements for the application.

After evaluating one or more off-the-shelf OODBs, developers can either choose the off-the-shelf OODB that most closely matches the baseline requirements for their application, or they can decide that none of the off-the-shelf candidates are suitable and choose to create a custom OODB implementation.

4.3.1. Objectivity Evaluation

The developers first enter the command in the UFO tool to activate the inconsistency detection for the off-the-shelf OODB that they want to explore, which in this case is Objectivity:

```
Choose an OODB target [ITASCA,ObjectStoreLite,Objectivity]: Objectivity
```

The developers' next goal is to instantiate from the parameterized OODB architecture an OODB instance that will be used to model static properties of Objectivity and simulate its dynamic properties.

4.3.1.1. Objectivity Modeling

The developers use the baseline requirements as the starting point for evaluating Objectivity. If it turns out that there are no inconsistencies detected between these requirements and Objectivity, then developers will know that Objectivity provides a good match for the baseline requirements. If inconsistencies are detected, then developers can observe how great the difference is between the baseline requirements and the requirements that Objectivity can support.

Developers invoke the mapping from requirement variables to architectural parameters. As the architectural parameter values are derived, the tool compares the values to the legal architectural parameter values for Objectivity. Inconsistencies are reported to the developers, along with suggested refinements to the architectural parameters that will resolve the inconsistencies. These inconsistency reports assist developers in understanding how well the baseline requirements are supported by Objectivity. They also guide developers in creating a set of architectural parameters that will map to a software architecture model and simulator consistent with the Objectivity architecture. In this case, three inconsistencies are reported:

```
Long Transactions: FALSE
** INCONSISTENCY: Objectivity always supports long transactions.
Set architectural parameter to TRUE.
FALSE Long Transactions requirement value not supported.

Object Locking: FALSE
** INCONSISTENCY: Objectivity always supports object locking.
Set architectural parameter to TRUE.
FALSE Concurrent Users requirement value not supported.

Transaction Manager Integration: EMBEDDED
** INCONSISTENCY: Objectivity does not support embedded transaction managers.
Remove embedded transaction manager and create a shared transaction manager.
FALSE Concurrent Users requirement value cannot be supported.
```

The first two inconsistencies indicate that Objectivity always supports two types of functionality not required by their application. This excess baggage relative to the baseline requirements may imply a larger system size and potentially poorer performance. The third inconsistency indicates an architectural feature that is not supported by Objectivity. In general, an embedded transaction manager is more efficient for single user applications, but Objectivity has a multi-user architecture with a stand-alone transaction manager process that is used for both multi-user and single-user applications. The UFO tool only supports stand-alone repository managers when the transaction managers are stand-alone, so the tool also requests this change be made.

At this point the developers can either decide that the differences between the baseline requirements and Objectivity are too extreme and abandon the evaluation of Objectivity, or they can modify the architectural parameter values as indicated and continue with the evaluation. In this case, the developers decide to modify the architectural parameter values according to the suggestions in the inconsistency messages. Based on these inconsistency messages, they also note which of the baseline requirements cannot be satisfied by Objectivity.

After the developers modify the architectural parameters they receive no inconsistency messages. The architectural parameter values that were produced are shown in Figure 16. These architectural parameter values most closely support the graphical editor application under the constraints of the Objectivity architecture. That is, the UFO tool guided developers in modifying the architectural parameters such that they are consistent with Objectivity, while at the same time trying to minimize the differences with the baseline architectural parameter values.

With the inconsistencies in the architectural parameters addressed, developers next invoke the mapping to produce the software architecture instance corresponding to Objectivity. The result is shown in Figure 17. The primary differences between this and the baseline instance shown in Figure 9. come from Objectivity's support for multiple concurrent users and long transactions. The embedded repository manager and embedded transaction manager are now shared. Objectivity's Object and Cell Services, Persistent Cell Manager, Repository Manager, and Transaction Manager all contain functionality to support long transactions (persistent nested transactions). The Object and Cell Services and the Persistent Cell Manager for Objectivity contain functionality to support object locking.

The developers next invoke a command to get the description of architectural properties for the Objectivity instance. This is shown in Figure 18. on page 47. The differences between the Objectivity model and the baseline architecture properties shown in Figure 10. come from the restructuring required to support multiple users and the additional functionality of multiple users and long transactions. The client size is estimated to be the same with Objectivity, but the number of shared processes has gone from 1 to 3, and the total size of the shared processes (PCMs, Transaction Manager, and Repository Manager) are nearly twice as large, going from 2 megabytes to nearly 4 megabytes. Following the directive of the last warning message that UFO issued (subtracting out the shared repository manager) results in 2 shared processes rather than 3.

```
Long Transactions: TRUE
Object Locking: TRUE
Distributed Transactions: FALSE
Cell Declarations:
   Cell: Root

      Role Declarations:
         Role root_root is Class VPML_Root


   Cell: Composite

      Role Declarations:
         Role comp_root is Class CompositeTask
         Role prim is Class PrimitiveTask
         Role res is Class Resource
         Role prod is Class Product
         Role pset is Class ProductSet
         Role pmem is Class ProductMember
         Role pflo is Class ProductFlow
         Role fchn is Class FlowChain


   Cell: PContent

      Role Declarations:
         Role pcont_root is Class ProductContent
         Role lnode is Class Node
         Role rnode is Class Node



Client:
   Cell Allocations:
      Root
      Composite
      PContent

   Cache Size: 1000000
   Replacement Policy: LRU
   Write-through Policy: EAGER
   Control Scheduling: NON-REENTRANT


<No Object Servers>
Persistent Cell Managers:
   Persistent Cell Manager: CSP
      PCM Allocations:
         CLIENT

      Write-through Policy: EAGER



Transaction Manager Integration: SHARED
Object Server Set Integration: EMPTY Object Server Set
Persistent Cell Manager Integration: SHARED
Repository Manager Integration: SHARED
Backups: No Backups
```

Figure 16. Architectural Parameter Values for Objectivity

4.3.1.2. Objectivity Simulation

The developers next run the prototype application code and simulation scripts on the Objectivity architecture simulator. The simulation profile for Objectivity will be compared with the baseline profile in order to determine how well Objectivity performs relative to the baseline requirements. The top-level results are shown in Figure 19. Comparing these results to the baseline profile in Figure 12., the developers note a 4% increase in execution time. Closer inspection at lower levels of detail in the profile shows that the increase comes from the object locking functionality in Objectivity.

```
Non-reentrant Client:

   Embedded Repository Manager:

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: SHARED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

    Object and Cell Services.
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root

      Cell: Composite
         Role Declarations:
            Proxy Role comp_root is Class CompositeTask
            Role prim is Class PrimitiveTask
            Role res is Class Resource
            Role prod is Class Product
            Role pset is Class ProductSet
            Role pmem is Class ProductMember
            Role pflo is Class ProductFlow
            Role fchn is Class FlowChain

      Cell: PContent
         Role Declarations:
            Proxy Role pcont_root is Class ProductContent
            Role lnode is Class Node
            Role rnode is Class Node

   Root Role: Root.root_root

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED
   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
      Write-through: EAGER
      Persistent nested transactions: SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object locking: SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
   <Embedded in client.  See Client description.>

Transaction Manager.   Integration: SHARED

   Nested persistent transactions: SUPPORTED
   Distributed transactions: NOT SUPPORTED
```

Figure 17. The Software Architecture Instance Description for Objectivity

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 2
Total size of shared processes: 3750000 Bytes

Non-reentrant Client:

Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:
   Total size: 50000 Bytes

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Object and Cell Services.

   Total size: 1300000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER


<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2750000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2750000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: SUPPORTED.    Size: 500000
      Distributed transactions: NOT SUPPORTED
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS



Repository Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>

Transaction Manager.   Integration: SHARED

Total size: 1000000 Bytes

   Nested persistent transactions: SUPPORTED.    Size: 500000
   Distributed transactions: NOT SUPPORTED
```

Figure 18. Architecture Properties for the Objectivity Model

```
Note: All time measurements are expressed in micro-seconds.

Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 7121
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 149307
Activations:     Objects: 13441    Cells: 98
                 Time: 106702
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 22327
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1972351
```

Figure 19. Top-level Objectivity Simulation Profile

This example demonstrates how individual applications can have different levels of conformance with OODBs. Although Objectivity has unnecessary object locking functionality for the developers' graphical editor application, the overhead doesn't cause a major increase in the performance profile because the application has a very high locality and utilization of the cells. An application with poor locality and low cell utilization, on the other hand, would show a major performance degradation by adding unnecessary object locking functionality in the OODB architecture.

4.3.2. ObjectStoreLite Evaluation

Next the developers evaluate another OODB that they are interested in, ObjectStoreLite. They go through the same series of steps as they did with Objectivity, starting with the baseline requirement variable values.

4.3.2.1. ObjectStoreLite Modeling

When the developers invoke the command to map from baseline requirement variables to architectural parameters, the UFO tool reports no inconsistencies between the architectural parameters and ObjectStoreLite. This indicates that the salient ObjectStoreLite features are a good match for the workflow editor application.

The developers next invoke mapping from architectural parameters to a software architecture instance, and then the mapping from software architecture instance to the OODB simulator realization for the instance. The resulting architecture summary, shown in Figure 20. on page 49, is the same as the baseline architecture. A summary of the architectural properties for this architecture is shown in Figure 21. on page 50. They are also the same as the baseline architecture.

4.3.2.2. ObjectStoreLite Simulation

Developers next run the simulation on the ObjectStoreLite model using the same prototype code and simulation scripts as before. The top-level simulation profile is shown in Figure 22. As expected, these results are the same as the baseline architecture since the architectural models are identical.

```
Non-reentrant Client:

       Embedded Repository Manager:
          Maximum number of concurrent clients on a repository: 1
          Maximum number of repositories per installation: 1000000000
          Transaction manager administration: EMBEDDED
          Object server set administration: NO OBJECT SERVERS
          Persistent cell manager set administration: SHARED

       Embedded Transaction Manager:
          Nested persistent transactions: NOT SUPPORTED
          Distributed transactions: NOT SUPPORTED

       Object and Cell Services.
          Threads: SINGLE
          Object locking: NOT SUPPORTED
          Persistent nested transactions: NOT SUPPORTED
          Cache size: 1000000 Bytes
          Cache replacement policy: LRU
          Distributed transactions: NOT SUPPORTED
          Write-through policy: EAGER

       Cell Allocations:
          Cell: Root
             Role Declarations:
                Proxy Role root_root is Class VPML_Root

          Cell: Composite
             Role Declarations:
                Proxy Role comp_root is Class CompositeTask
                Role prim is Class PrimitiveTask
                Role res is Class Resource
                Role prod is Class Product
                Role pset is Class ProductSet
                Role pmem is Class ProductMember
                Role pflo is Class ProductFlow
                Role fchn is Class FlowChain

          Cell: PContent
             Role Declarations:
                Proxy Role pcont_root is Class ProductContent
                Role lnode is Class Node
                Role rnode is Class Node

       Root Role: Root.root_root

    <No Object Servers>

    Persistent Cell Manager Set.   Integration: SHARED

       Persistent Cell Manager. CSP
          Processes allocated to PCM:
             CLIENT
          Write-through: EAGER
          Persistent nested transactions: NOT SUPPORTED
          Distributed transactions: NOT SUPPORTED
          Object locking: NOT SUPPORTED
          Backup policy: NO AUTOMATED BACKUPS

    Repository Manager.   Integration: EMBEDDED
       <Embedded in client.  See Client description.>

    Transaction Manager.   Integration: EMBEDDED
       <Embedded in client.  See Client description.>
```

Figure 20. The Software Architecture Instance Description for ObjectStoreLite

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 1
Total size of shared processes: 2000000 Bytes

Non-reentrant Client:
Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:
   Total size: 50000 Bytes
      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:
   Total size: 200000 Bytes
      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.
   Total size: 1100000 Bytes

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2000000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2000000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object Locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>
```

Figure 21. Architecture Properties for the ObjectStoreLite Model

```
Note: All time measurements are expressed in micro-seconds.

Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 4747
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 146882
Activations:     Objects: 13441    Cells: 98
                 Time: 99277
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 19524
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1464740
```

Figure 22. High-level ObjectStoreLite Simulation Profile

4.4. Wrapping up the OODB Evaluation

Although the developers could continue testing off-the-shelf OODB architectures trying to find other alternatives, they decide to terminate the search since ObjectStoreLite provides a good match for their application. The high-level modeling and simulation results from their tests are summarized in Table 1.

Table 1. Modeling and Simulation Summary for the Graphical Editor

OODB Model	Architectural Differences from the Baseline	Client Size	# Shared Processes & Total Size	Total Simulated Execution Time
Baseline	—	2.4 MB	1 2.0 MB	1.5 Seconds
Objectivity	Shared transaction manager. Support for concurrent users, including object locking. Support for long transactions.	2.4 MB	2 3.8 MB	2.0 Seconds
ObjectStoreLite	None	2.4 MB	1 2.0 MB	1.5 Seconds

In general, development projects can use modeling and simulation information from the UFO tool, plus other non-technical information such as licensing fees for off-the-shelf OODBs, time-to-market risks, and so forth, in order to make an informed buy-versus-build decision for OODBs. If the developers are not comfortable with the modeling and simulation figures for the off-the-shelf OODB alternatives, then they can consider a custom implementation of the baseline architecture.

Chapter 5. High-Level System Design Rationale

In this and the next chapter we describe our design and implementation of the UFO prototype tool. This chapter presents the insights and rationale behind the high-level design while the next chapter provides the low-level design and implementation issues.

The key technical components in the UFO tool are the parameterized software architecture and the OODB modeler and simulator. These are illustrated as the "wedges" in Figure 3. on page 6. The discussion in this chapter focuses on the important problems and alternatives that these two pieces of technology present, the key design decisions that we made in our solution, and the rationale behind these decisions.

The first section in this chapter, 5.1., addresses the problem of capturing the salient features of the OODB class of systems in the UFO parameterized software architecture. In Section 5.2. we address the OODB architecture modeling and simulation issues. Finally, in Section 5.3. we explore the issues of incorporating the UFO approach of defining and refining OODB requirements and evaluating off-the-shelf OODBs into conventional software development practice associated with application development using OODBs.

5.1. Capturing the Class of OODB Systems in the UFO Parameterized Software Architecture

As illustrated in Figure 3. on page 6, one of the key capabilities of our approach is to take a set of requirements that an application has on an OODB and identify an OODB architecture instance that satisfies these requirements. This capability requires a detailed understanding of the OODB domain. Furthermore, since we want to automate this capability with the UFO tool, the information about the OODB domain must be captured and represented in a tool. We refer to this representation as the *Parameterized Software Architecture*.

In this section we discuss the issues associated with creating the UFO parameterized software architecture. Section 5.1.1. deals with general issues in OODB domain analysis: salient features that we captured versus features we excluded, generalizations from the domain versus discriminators in the domain, language versus architectural issues, functionality issues versus performance issues. Section 5.1.2. deals with specific details in the UFO generalization of the OODB languages. Section 5.1.3. deals specifically with generalizing architecture commonality in the OODB domain. Section 5.1.4. through 5.1.9. deal with the specific issues of discriminating among instances in the domain – requirement variables, architectural parameters, configuration nodes, and the mappings between them.

5.1.1. OODB Domain Analysis

Domain analysis refers to the activity of studying and characterizing the features of a class of systems[10]. In our domain analysis for the OODB class of systems, we studied commercial off-the-shelf OODB and research OODB systems. In this study we were most interested in characterizing practical and proven features that developers use in application development, so we focused on OODB features that are commonly available in commercial off-the-shelf OODBs. The research systems provided additional valuable insight since they were often accompanied with better technical descriptions of architectural features.

As we present the results of our domain analysis in the following subsections, we consider four different characterizations of each domain feature. We used these feature characterizations to determine the potential impact of features on the UFO parameterized architecture.

- *Functionality versus performance*. From the point of view of an application executing on an OODB, does the feature provide functionality that impacts the execution semantics or is the feature included to enhance performance? OODB features typically fall into one or the other category.
- *Impact on OODB language*. Is the feature represented in the OODB language or is it only an architectural feature? This characterizes features considered for UFO's OODB language.
- *Generalization for all OODB instances versus discriminator among the OODB instances*. Is the feature a generalization that is common among all OODB instances or a discriminator that differentiates some instances from other instances. To construct the parameterized software architecture for the OODB domain we developed both generalized features and discriminating features.
- *Primary versus secondary features in the OODB domain*. In particular, we were looking for the most salient and representative features (i.e., the primary features) to characterize the spectrum of OODB functionality, performance, languages, architectures, generalizations, and discriminators. Since this thesis focuses on selecting among different OODB instances, we emphasize the discriminating features of the OODB domain.

Table 2. summarizes the features in the OODB domain that we identified and considered for the UFO tool. The table illustrates how we characterized each feature according to the four bullet items above. The first column is the name of the OODB feature. The second column indicates whether the feature is primarily for OODB functionality (F) or performance (P). The third column indicates whether we characterized the feature as a generalization (G) or a discriminator (D). The fourth column indicates whether the feature is reflected in the OODB language. The fifth column indicates whether (Y) or not (N) the feature was included in the UFO parameterized software architecture. The last column describes how discriminators are viewed from the perspective of an application developer try to select among different OODB instances.

For example, the first second entry in the table, *object identity*, is an OODB feature that provides functionality so that every object in an OODB can be uniquely referenced by its object identity. This is a generalized feature that is available in all OODBs, so we will not use this feature as a discriminator among OODBs. OODB languages provide a way to express object identifiers, so this feature impacts the OODB language.

From this list, our choice of features to include in UFO was driven by two primary objectives. Most importantly, we wanted to capture the most significant and practical generalizations and discriminators for modeling and simulating the functionality and performance of OODB architectures. Second, we wanted features that provided the biggest payoff from the investment we made to incorporate them in the tool. In other words, within the limited scope of this thesis we didn't want to spend time characterizing and implementing features that were relatively insignificant in the OODB domain or that were overly complex relative to the value that they provided to the tool. Third, we were more interested in features that represent important discriminations among OODB instances since UFO is a tool that helps developers select among OODBs.

Table 2. OODB Domain Features Considered for UFO

OODB Domain Features	Function-ality (F) or Perform-ance (P)	General-ization (G) or Discrim-ination (D)	Impacts OODB Lan-guage (L)	UFO	Related UFO Requirements (UFO discriminators only)
Persistence	F	G	L	Y	
Object identity	F	G	L	Y	
Short transactions (atomicity)	F	G	L	Y	
Long transactions (atomicity)	F	D	L	Y	Support for long transactions
Nested transactions (long and short)	F	D	L	Y	Support for long transactions
Object locking (implicit, explicit)	F	D	L	Y	Support of concurrent users
Object prefetch (clustering)	P	D		Y	Locality cluster declarations
Lock granularity (clustering)	P	D		Y	Locality cluster declarations
Computational object servers	P	D		Y	Cluster size & utilization
Events	F	D	L	N	
Schema evolution	F	D		N	
Query language	F	G	L	N	
Multimedia objects	F	D	L	N	
Security	F	D	L	N	
Inheritance	F	G	L	N	
Federated databases	F	D		N	
Garbage collection	F	D	L	N	
Explicit deletion	F	G	L	Y	
Multiple language bindings	F	D	L	N	
Distribution	P	D		Y	Cluster and cluster set utilization
Replication	P	D		Y	Backups
Deadlock detection	F	D		N	

Fortunately these three objectives typically did not conflict. The features that we determined to be most significant in the OODB domain, such as locality, distribution, and concurrency control, were also manageable within the scope of this thesis. The features that we determined to be most complex and difficult to implement, such as query languages and multiple OODB languages, were also less significant in discriminating among architectures that satisfy OODB requirements.

In the following sections we describe those OODB features that we chose to implement in the UFO tool and justify why they are included. First we discuss the generalizations from the domain. We use generalization for two purposes. The first is to generalize OODB languages to create a common OODB programming language that developers can use to prototype their OODB applications. The second is to generalize OODB implementation architectures to create an OODB *reference architecture*. The reference architecture is a conceptual aid for describing in the common, abstract framework of an OODB architecture and for abstracting away from the architectural details that are common to OODB instances. Following that we discuss the discriminators in the domain. Discriminators are central to the parameterized software architecture's ability to identify instances in the OODB domain that satisfy a set of requirements. From the discriminators we defined requirement variables, architectural parameters, configuration nodes, and the mappings among them.

5.1.2. Generalized OODB Concepts in UFO's Virtual OODB Language

Different off-the-shelf and custom OODBs use a variety of different APIs or programming language interfaces that serve a similar purpose. To simulate the different off-the-shelf OODB, we considered two alternatives. One alternative was to implement all of the APIs for the different OODBs. This approach has associated with it a significant overhead for us as developers of the UFO technology and also for UFO users. We would have to implement a multitude of different OODB programming languages that all served a similar purpose. UFO users would be required to repeatedly implement their application on different APIs in order to simulate on different OODBs. Furthermore, it would be difficult for developers to make meaningful comparisons of the architecture simulations since observed differences might be accounted for by architectural differences or application implementation differences on the different API.

The other alternative that we considered was to create an single interoperable OODB programming language that was a generalization of the APIs for the different off-the-shelf OODBs. This would allow us to implement a single language for the UFO tool and would allow developers to implement their application once for simulation on multiple OODB architectures. It would also allow developers to make meaningful comparisons of application performance on different OODB architectures. The disadvantages of this alternative are (1) we might not be able to include all of the special functionality provided by a individual off-the-shelf OODBs and (2) our language could not be used with any off-the-shelf OODB.

The field of relational databases has adopted an industry standard API called SQL to address the issue of interoperability among different relational database implementations. We took a similar approach and defined a single interoperable OODB language called the *UFO Virtual OODB Language*, or simply the UFO language. Note that there is a movement in the OODB community to define an interoperable OODB language[9].

In this section we use the UFO language as a means of introducing the OODB concepts supported by OODB instances in the UFO parameterized architecture. These concepts are generalizations from custom and off-the-shelf OODBs that we deemed to be the salient common features of OODBs. Our effort to identify these "salient" commonalities, of course, required us to make some judgement calls as to what was important and what was arbitrary. These choices are discussed in the following sections and from an architectural point of view in Section 5.1.3.

5.1.2.1. Applications, Repositories, and Clients

We found three general, top-level concepts associated with the OODBs we studied, although the terminology differed. We refer to these concepts as *applications*, *repositories*, and *clients*. An *application* refers to an end-user software system that uses an OODB. The persistent object space for an application is a set of named *repositories*. Each repository holds an independent collection of application data objects. For example, in the workflow management application described in Chapter 4., independent projects store their workflow diagrams in independent repositories. Each application user executes the application with a *client* process. A client *binds* to a repository in order to access the persistent data objects in the repository, analogous to opening a file. Clients also create and delete repositories.

We captured these three general concepts in the UFO language semantics. A UFO *application* is comprised of *repositories* and *clients*. Figure 23. illustrates a collection of clients and repositories for an application. Client 1 is bound to Repository A and is accessing the persistent data objects in the repository. Client 2 is not yet bound to a repository. Client 3 and Client 4 are concurrently accessing objects in Repository C. Client 5 is in the process of creating a new repository. Currently there are no clients bound to Repository B.

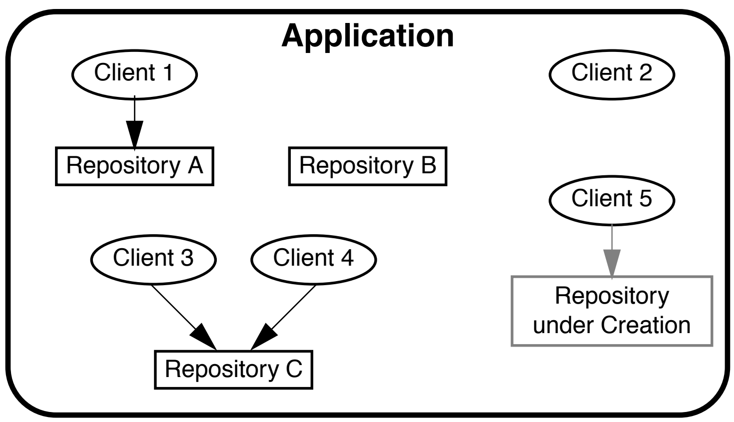


Figure 23. Repositories and Clients for an Application

5.1.2.2. Basic UFO Program Structure

In the OODBs that we studied, the execution of an application typically went through three phases: (1) an introductory *prelude* phase where the execution context is established within the address space of repositories, (2) a *bootstrap* phase where a persistent object in an OODB repository is identified for initiating the object-oriented execution, and (3) the execution of the application within the persistent object space.

We modeled these three phases in the basic program structure of the UFO language. In every UFO program there is a *Prelude*, followed by a *RootClass* declaration, followed by zero or more *Class* declarations. The Prelude is the initial program block where the execution context is established within the address space of repositories and long transactions. Although some OODB languages do not have an explicit language construct for this purpose, there is an implicit initial phase in execution that serves the purpose of the Prelude. This is discussed in the next section.

The RootClass and the Classes contain the type definitions, or *schema*, for the application program, while a repository contains data instantiated according to the schema. The methods for each type are also encapsulated in the RootClass and Class declarations.

Object-oriented execution is initiated from the Prelude by invoking a “bootstrap” method on a persistent RootClass object. From the RootClass object, other Class objects can be accessed, created, and deleted, methods can be invoked on objects, and so forth, similar to conventional object-oriented languages. All OODB languages have a way to bootstrap execution on an initial object, but not the notion of a special RootClass that supports bootstrap methods. We felt that this feature was worth emphasizing with a special kind of class that has distinct bootstrap methods.

For the interested reader, the following three sections discuss the design details of how we implemented the prelude, repository access, transactions, bootstrap, and the persistent object space. Readers less interested in these detail and more interested in the high-level concepts and rationale may skip ahead to Section 5.1.3.

5.1.2.3. The Prelude and Repository Commands

When a client begins executing the program, it first enters the Prelude block at the top of the program. The prelude is a program block within which a client process can bind to a particular repository and perform other repository operations such as repository creation, repository deletion, and status queries. The prelude block may also contain operations to control long-term concurrency. In conventional OODB languages, the concept of a prelude is implicit. That is, there is not an explicit scope for repository and long transaction operations. We chose to add this scope simply to mark a clear delineation between the preliminary phase of operations in the prelude and the more central OODB operations involving object-oriented execution with persistent objects.

5.1.2.4. Transaction Commands

We found two types of transactions in the OODBs we studied, *persistent transactions* (also referred to as *long transactions*) and *transient transactions* (also referred to as *short transactions*). Persistent transactions are useful for long-lived transactions that may span hours, days, or weeks. Transient transactions are conventional short-lived transactions. A persistent transaction may live longer than the client process that creates it, while a transient transaction, being like a conventional transaction, may not exist after the client process that creates it has either normally or abnormally terminated.

If persistent transactions are supported, then transient transactions are nested within persistent transactions. That is, an executing application first establishes its context within a persistent transaction and then begins executing transient transactions. In addition to this nesting, some OODBs support nested persistent transactions. Although none of the OODBs that we studied supported nested transient transactions, there appears to be no technical limitation to preclude this feature.

We choose to support the most general transaction model that we observed in OODBs: nested persistent transactions and single-level transient transactions. We chose this approach since we had a previous successful experience implementing this type of transaction model; the UFO transactions are based on a software engineering transaction design that we did for the Gandalf system OODB[11]. It is also analogous to transaction mechanisms in OODBs such as ITASCA and ObjectStore.

Figure 24. illustrates UFO transactions. Readers not interested in the details of the model can skip ahead to Section 5.1.2.5. As shown in the figure, both persistent and transient transactions are nested together in a tree-structured hierarchy. The figure shows that Client 1 through Client 6 are concurrently bound to Repository A. After a client binds to a repository it is always associated with a single transaction, referred to as the *current working transaction*. The current working transaction is illustrated in the figure by the bold arrows from each client. When a client initially binds to a repository it is automatically given access to the root persistent transaction of the transaction hierarchy, as is Client 4 in the illustration.

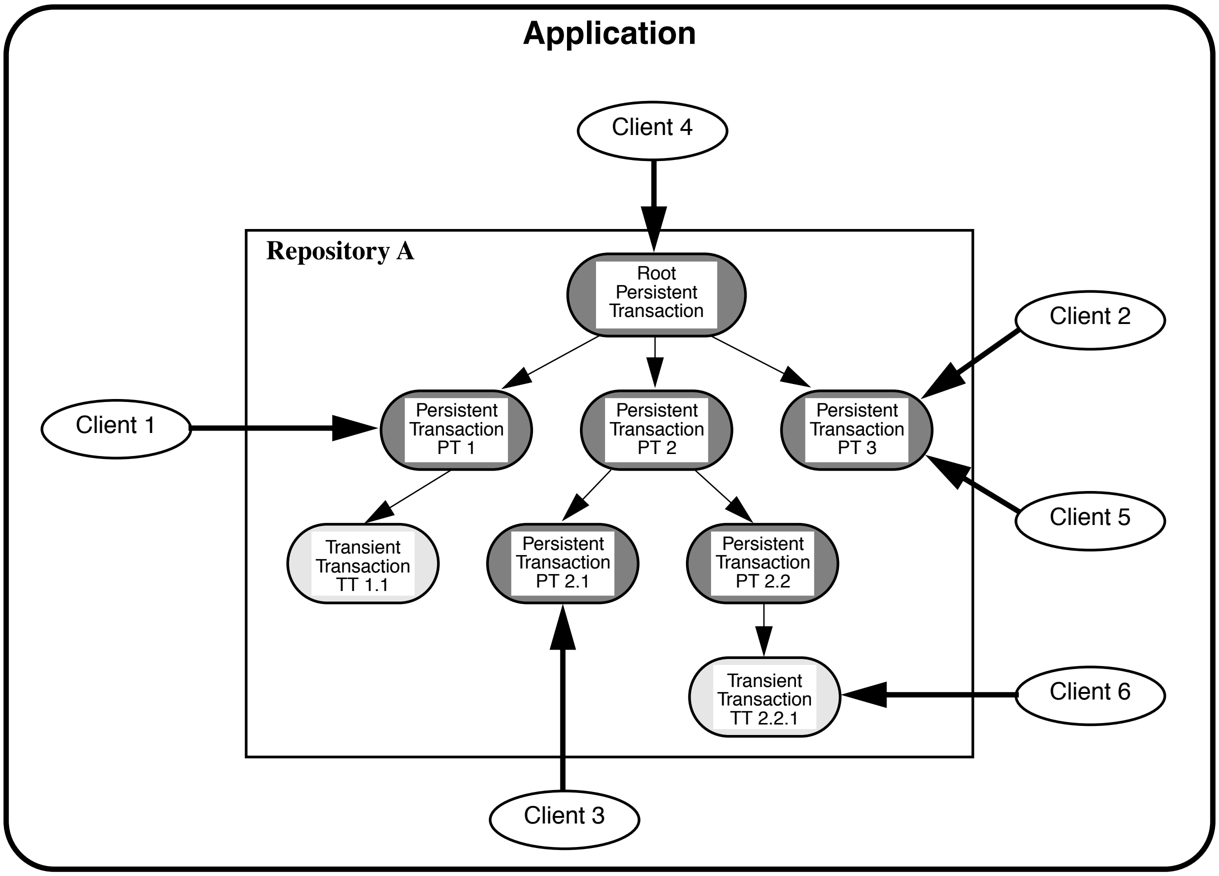


Figure 24. A Transaction Hierarchy in a UFO Repository

Clients can then traverse the hierarchy in order to access a particular persistent transaction to begin working in. This is analogous to moving about a UNIX directory hierarchy in order to access a particular directory to work in. For example, Client 1 has changed to persistent transaction PT 1 as its current working transaction.

In order for a client to access and modify the objects in a persistent transaction, the client must start a nested transient transaction off of a persistent transaction. For example, Client 6 has started the transient transaction TT 2.2.1 off of PT 2.2. The transaction TT 2.2.1 then becomes the current working transaction for Client 6 such that data objects can be accessed and modified.

When a client performs a commit operation on a transient transaction or persistent transaction, the modified objects and object locks are passed up to the parent persistent transaction. In the illustration, if Client 3 commits PT 2.1, the modifications and locks are passed up to transaction PT 2.

There are four UFO language statements relating to transactions inside of the repository binding block. CreatePersistentTransaction, ChangeTransaction, and CommitPersistentTransaction are operations on persistent transactions, while the TransientTransaction block is for transient transactions:

5.1.2.5. The Persistent Object Space

The object-oriented execution semantics were similar in all of the OODBs that we studied. We defined the UFO object-orient semantics to be similar to that of other object-oriented languages and OODBs. The runtime state of a UFO object is maintained in *component* values. A component can have a simple type (integer, boolean, character, string) or a *reference* type (a typed pointer to another object). The state of an object can only be examined or modified by *methods* that are encapsulated within the object's *class* declaration. A method is invoked by applying it to an object through a reference to the object. Control flow moves from one object to another when an object with a component that references another object applies a method via that reference.

One difference between UFO and some other OODBs is that all object in UFO are persistent. Most OODB languages allow for both persistent and transient data objects in an application program. However, since we are primarily focusing on the database aspects of OODB architectures in this thesis, we restricted our language to include only persistent data constructs that interact with the database architecture. While this simplification might not be suitable for a production OODB language, the inclusion or exclusion of transient data objects in the language does not significantly impact the issues studied in this thesis.

Following is a brief discussion of the object-orient language constructs and execution semantics in UFO. Readers not interested in this level of detail can skip ahead to Section 5.1.3.

Object-oriented execution in UFO can be bootstrapped only after a client gets access to a *root object* inside of a transient transaction block. A root object is the handle for a collection of persistent objects in a repository, providing a common entry point for clients. Once the client has access to the root object it invokes a *boot procedure* on the object. A boot procedure is similar to other procedures and functions in UFO, but the boot procedure is the only method that can be invoked from the prelude scope rather than within the scope of a class.

Every UFO program has a single RootClass declaration. The RootClass is a special class declaration for defining root objects. The RootClass differs from other Classes in that it contains *BootProcedures*.

There can be zero or more components declared in a Class or RootClass. Each component has a unique name in the RootClass. Components are typed, either with a UFO simple type (`INTEGER`, `BOOLEAN`, `CHARACTER`, or `STRING`) or with a reference type (the name of a RootClass or Class).

The BootProcedure declarations follow the component declarations. UFO allows for the definition of multiple BootProcedures so that different tools can be written to operate on the same repository. For example, in a retail sales database application, one BootProcedure might be a tool with commands used by a sales clerk while another BootProcedure would be a tool with commands used by an inventory manager. Code within the Prelude would determine which BootProcedure to invoke for each client.

The remaining method declarations are divided into *Exported Methods* and *Local Methods*. Exported methods available in the class interface, while local methods can only be invoked from within the class. These methods can either be functions having a return value or procedures with no return value.

Finally, the *Constructor* and *Destructor* are methods with predefined side effects and optional application-defined operation. After the object is created and initialized, control is passed to the application code in the constructor of the class so that the application can perform further initialization. Similarly, the destructor is used to deallocate an object. Before the storage deallocation is called, control is passed to the application code in the destructor for finalization and cleanup.

Note that UFO uses explicit object deletion rather than garbage collection. We chose this approach for two reasons: (1) it simplifies the prototype implementation, and (2) explicit deletion allows the application developer to better optimize the runtime architecture based on static information (i.e., garbage collection results in less predictable system performance profiles).

5.1.3. Generalizing OODB Implementations in the UFO Reference Architecture

In our search for ways to capture and express the complex features in the OODB architecture domain, we were faced with several related challenges. First, we needed a way to express, test, and evolve our understanding of the common "essence" of the OODB domain. With each custom and off-the-shelf OODB that we considered, we had to test our current knowledge of the domain to see if it adequately captured the salient architectural features of the OODB. If not we needed a way to appropriately extend or modify our general understanding.

The second challenge was to express, test, and evolve our understanding of the salient variations in the OODB domain. Similar to our search for commonality, we needed a way to identify meaningful differences among OODBs that served to address different application needs and a way to identify arbitrary implementation differences that we could abstract away from.

Also, after we complete our characterization of the OODB domain, we need our acquired knowledge of the commonality and variations in the OODB domain to be expressed in a form that would serve as our basis for the design of UFO tool. And finally, we need our acquired knowledge of the commonality and variations in the OODB domain to be expressed in a form that would serve as a conceptual basis for application developers using the UFO tool.

We could have chosen one mechanism to capture and express the commonality in the OODB domain and a different mechanism to express the variations. We could have also chosen different expressive notations for our tool design and for the tool users. However, this approach had a major drawback of having to constantly maintain consistency among the different mechanisms and notations as our knowledge of the domain evolved. Instead, we chose a single mechanism so that consistency would not be a problem. We refer to this mechanism as the *UFO Reference Architecture*.

We considered a formal notation for the reference architecture, but decided that this type of notation was not the most effective way to express OODB concepts to UFO tool users nor was the overhead and rigor associated with a formal notation necessary to validate the thesis. Instead we use informal design diagrams and textual descriptions for the reference architecture.

Figure 25. shows an entity-relationship diagram for the top-level runtime data entities in the UFO reference architecture. That is, the architecture manages and maintains these data entities during execution. An *application installation* consists of zero or more *repositories* and zero or more *execution instances*. A repository is the top-level entity for maintaining persistent database objects. An execution instance contains the execution state and the *application code* for an end-user using the application. The execution instance also provides access to *database objects* that are created, deleted, viewed, and modified. Execution instances are bound to a single repository at a time. Each repository contains a *transaction* hierarchy of persistent and transient transactions. An execution instance has a single transaction as its *current working transaction*. Each transaction provides access to one or more *cells*. A cell is a locality cluster of persistent *database objects*.

The UFO reference architecture is partitioned into functional units that operate on the runtime entities such as database objects, transactions, repositories, and execution instances. The functional units – clients, repository managers, transaction managers, persistent cell managers, and object servers – are common functional units in the OODBs that we studied. They are illustrated in the reference architecture diagram in Figure 26. (this figure is a copy of Figure 5. from page 23).

For the interested reader, the following paragraphs provide an overview of the functional units in the reference architecture. Readers less interested in these detail and more interested in the high-level concepts and rationale may skip ahead to Section 5.1.4.

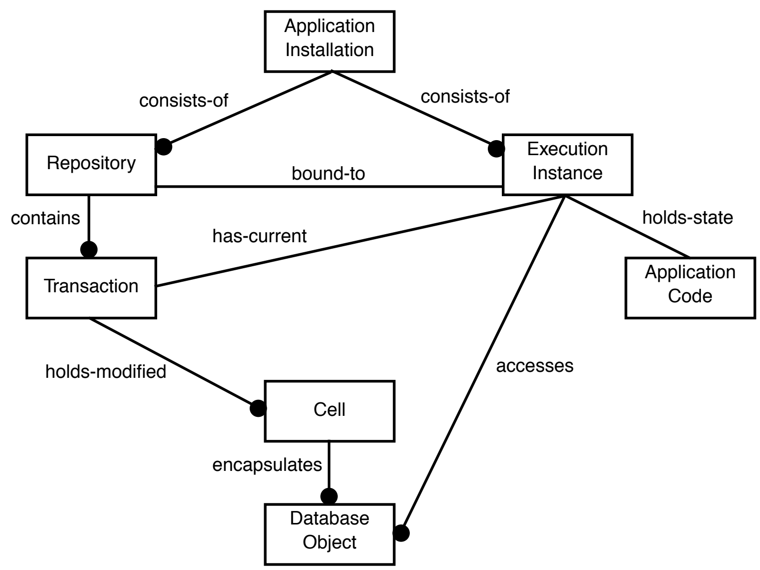


Figure 25. Entity-Relationship Diagram for UFO Architecture Entities

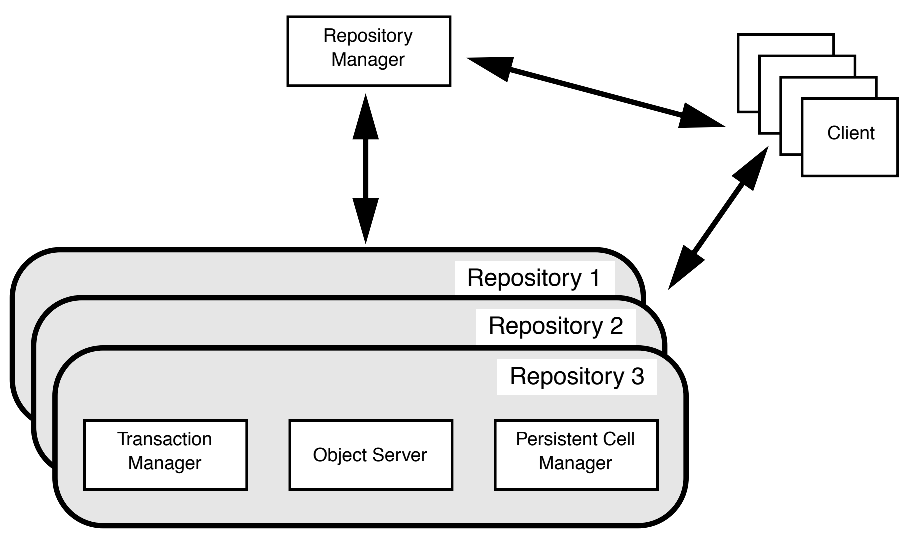


Figure 26. UFO Reference Architecture

The Client Functional Unit

The client executes application code and maintains state of the execution instance. During execution the client creates, deletes, accesses, and modifies database objects.

Figure 27. shows the high-level functional composition of a Client. The *Prelude and Method Execution* unit is responsible for executing the application code and maintaining state in the execution instance. The *Object and Cell Services* functional unit is responsible for (1) getting database cells from persistent storage and unpacking the cells into individual data objects in the client's cache memory, plus the inverse of packing active data objects into cells and returning them to persistent storage, and (2) participating in distributed transaction protocols. The *Remote Method Calls* unit coordinates the outgoing and incoming method invocations that cross the procedural boundaries of clients and object servers.

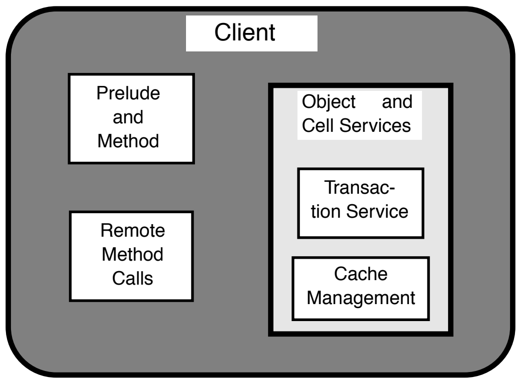


Figure 27. Client Functional Composition

The Repository Manager Functional Unit

The Repository Manager has several responsibilities: (1) it handles the creation and deletion of repositories, (2) it manages the binding between Clients and Repositories, (3) it manages the directory of Repositories, and (4) it monitors and manages the processes (e.g., Transaction Managers, Object Servers, Persistent Cell Managers) that are responsible for operations on repositories and repository contents.

The Repository Manager is illustrated in Figure 28. The *Binding Service* is responsible for handling requests from clients to bind to repositories. The *Repository Inventory Service* is responsible for creating and deleting repositories, maintaining a directory of repositories, and servicing queries on the repository directory. *Startup, Shutdown, and Recovery* manages the creation and monitoring of other system processes.

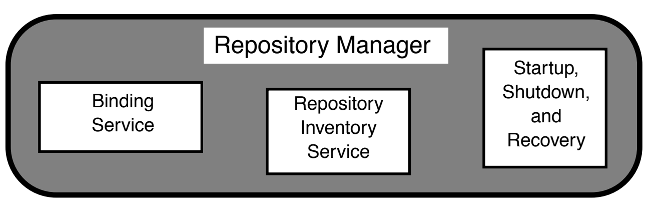


Figure 28. Repository Manager Functional Composition

The Transaction Manager Functional Unit

The transaction manager maintains the transaction hierarchy for repositories and coordinates transaction operations among multiple processes such as the clients, object servers, and persistent cell managers.

Figure 29. shows the Transaction Manager. The *Persistent Transaction Service* handles all of the persistent transaction operations in an application while the *Transient Transaction Service* handles the transient transaction operations. The *Distributed Transaction Service* manages the distributed transaction processing, such as two phase commit, in cases when an architectural configuration results in multiple processes participating in transactions.

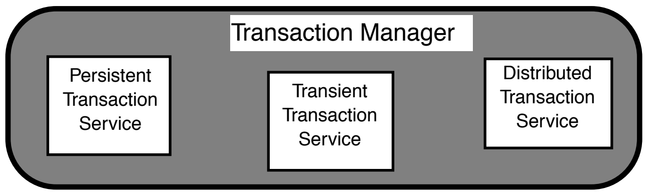


Figure 29. Transaction Manager Functional Composition

The Persistent Cell Manager Functional Unit

The persistent cell manager (PCM) is responsible for the persistent storage of cells. PCMs cooperate with clients in the activation and deactivation of cells. PCMs also maintain the persistent transaction cell tables and provide cache overflow storage for cells modified in transient transactions pending commit or abort. The automated backup of persistent cell storage to secondary persistent storage is also managed by PCMs.

Figure 30. shows the high-level functional composition of the Persistent Cell Manager. The *Persistent Transaction Service* manages the storage for persistent transaction structures. The *Transient Transaction Service* manages the backing storage for transient transactions. The *Distributed Transaction Service* handles the two phase commit protocol when distributed transactions are present in the architecture. The *Cell Activation and Deactivation Service* is responsible for reading persistent cells and sending them to clients and object servers (activation) and for receiving modified cells from clients and object servers and writing them back to persistent storage (deactivation). The *Cell Backup Service* manages requests to send backup copies of persistent cells to external storage devices.

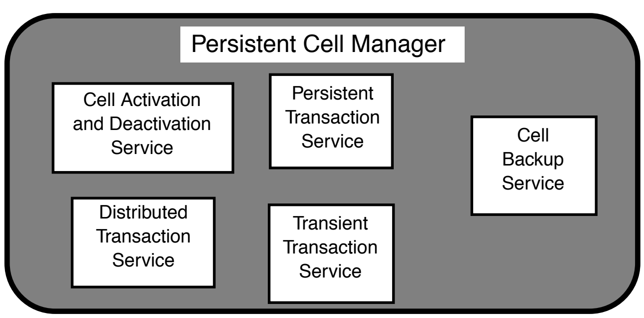


Figure 30. Persistent Cell Manager Functional Composition

The Object Server Functional Unit

In conjunction with clients, object servers execute application code and maintain state of the execution instance. Each object server resides on a separate processor, thus allowing for load sharing in a multi-user architecture. Object servers typically have very large object caches to minimize the number of cell activations during execution. This is particularly useful for cells with weak locality and large cells.

Figure 31. shows the high-level functional composition of the Object Server. It is essentially equivalent to the Client architecture, but the object server does not directly interact with end-users. It is a secondary source of computation for clients.

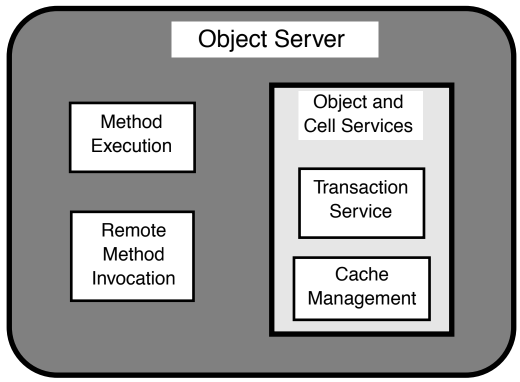


Figure 31. Object Server Functional Composition

5.1.4. Discriminating Among OODB Instances

Generalization helped us factor commonality out of the OODB domain in the form of a reference architecture. However, we still needed discrimination to help us structure the myriad of variations that we observed among the instances within the class of OODB systems. Our objective was to capture the salient architectural dimensions that varied from instance to instance in the off-the-shelf and custom OODBs that we studied and used.

We quickly found that OODBs differ in many ways at many different levels. The first problem that we encountered in our efforts at discrimination was determining which of the differences were salient for our work and which were arbitrary. There are many reasons why OODB developers implement their architectures in different ways, but the question for us was "which of these variations might be used to match OODB architectures to application requirements?"

Architectural differences are typically not documented or justified among OODBs. This meant that we had to do some reverse engineering in order to characterize architectural differences and the purpose that they served. We also took the inverse approach and looked at the different types of applications that use OODBs to further our understanding of the overall variability in requirements on OODBs and how these requirements related to architectural features. Although there appeared to be no innately clear way to distinguish between salient and arbitrary architectural variants, we were able to identify some guidelines that helped us in our characterization.

We found that the most significant dimensions of architectural variability in the OODB domain served one of the following three purposes for improving the conformance between applications and OODBs. We used these three categories as rationale for including (or excluding) architectural variability in the UFO parameterized architecture. For example, arbitrary architectural variations that appeared to serve none of these purposes were not captured in the parameterized architecture.

- *Parsimony*. Architectural choices for including *all and only* the functionality needed for an application. Parsimony means keeping unnecessary functionality out of an architecture instance in order to minimize system size and to minimize the length of code paths (e.g., improve performance).

- *Expected use profiles*. Architectural choices for optimizing system properties such as overall size and performance based on expectations on how the system will be used. Expected use profiles allow assumptions to be made in the architecture about data access locality in order to minimize the overall execution profiles.

- *User expectations*. Architectural choices for optimizing system properties such as localized size and performance based on expectations that end-users have on the system. User expectations lead to architectural features that optimize localized execution profiles to better satisfy performance desires of the end-user, even in cases where the overall performance is compromised in order to give better performance for a critical operation.

Another important problem that we were faced with was capturing the architectural variability and the rationale in our engineering tool. Since we were looking for a way to map application requirements to OODB architectures, it initially appeared that we should capture the discriminating relationships between OODB requirements and OODB architectures in a single mapping from requirement concepts to architecture realizations. We identified several implementation alternatives for this mapping, including production systems, data flow networks, and conventional imperative implementations.

Evaluating the production system option, we found that most of the productions for discriminating among OODB architectures were very simple and it appeared that a production system was more powerful than we needed for the OODB domain. An ad hoc imperative implementation, on the other hand, offered no explicit structure for the relationships and therefore appeared to be too informal. Data flow networks appeared to be an appropriate compromise between the two approaches.

We prototyped an initial implementation of the mapping from requirements to architectures as a data flow graph that we referred to as a *dependency graph*. Experience with the dependency graph indicated that it was relatively simple to implement but relatively complex and difficult to enhance and maintain. Closer examination indicated to us that several types of information were being represented in the dependency graph and that these different types of information could be segregated. Further prototyping showed that the dependency graph could be decomposed into four simpler pieces, each addressing a separate concern for discrimination in the OODB domain. These correspond to the four primary tasks illustrated in Figure 4. on page 7:

- *Define requirements*. Extraction of requirement variable values from the application source code and definition of requirement variable values by developers. The requirement variable values discriminate among the varying requirements that different applications can have on an OODB.

- *Map Requirements to Architectural Parameters*. Map from requirement variable values in the requirements domain to architectural parameter values in the architecture domain. The architectural parameter values discriminate at an abstract level along the different dimensions of architectural variability in the OODB domain.

- *Map Architectural Parameters to Architecture Instance*. Map from architectural parameter values to a software architecture instance. The software architecture instances discriminate among the concrete architecture configurations in the OODB domain.

- *Realize Architecture Simulator*. Map from a software architecture instance to a software architecture simulator. The simulator helps to discriminate among the runtime properties of the different instances in the OODB domain.

The decomposed representations and mappings simplified the encoding of discriminators in the OODB domain. The complex and convoluted dependency graph became a composition of simpler and straightforward representations and mappings, each with separate and clear concerns. These representations and mappings are easier to create, understand, modify, and maintain. We attribute this to the fact that the decomposed representations and mappings more accurately reflect the different tasks and relationships that exist in the conventional development of software architectures.

In the following sections we describe each of the representations and mappings that we used to capture the discriminating variants in the OODB domain.

5.1.5. Architectural Building Blocks: UFO Configuration Nodes

OODB architecture instances play a central role in parameterized software architecture. As such, one of the important problems we had for the parameterized software architecture was choosing an appropriate representation for OODB architecture instances. This representation needed to serve in the mapping from requirements to architecture, in modeling, and in simulation. Furthermore, we needed a flexible representation that could easily evolve as our understanding of OODB architectures and architectural variants evolved.

We seriously considered two alternatives for representing OODB architecture instances, one based on object-oriented analysis notations and one based on grammars. The OO analysis notations are used to model application domains during the analysis phase of software system development and seemed like a good candidate. However, we found that these languages were intended for capturing, expressing, and passing domain knowledge among humans in a human-readable format. Because of this, these notations were more expressive and complex than we needed. We considered a representation based on grammars because we were using a grammar notation to document the OODB reference architecture and architectural variants as we studied different OODBs. This simpler approach turned out to be appropriate for our needs.

We represent an architecture instance as a collection of building blocks call *configuration nodes*. A configuration node denotes an architectural component such as a software sub-system or software module. The collection of configuration nodes that define an architecture instance therefore represent the collection of software sub-systems that comprise an OODB architecture.

The UFO tool creates an architecture instance by selecting and assembling a collection of configuration nodes based on the requirements that developers define for an the OODB. We chose this “constructive” approach because it is analogous to conventional architecture development where architectural components are built, selected, assembled, and so forth in order to create an architecture that satisfies a set of requirements.

Configuration nodes are assembled in a tree structure. Each configuration node denotes:

- a software component in the software architecture
- zero or more specializations of the component
- zero or more children of the component, with each child being a predefined set of alternative configuration nodes

Specializations and child alternatives allow us to represent variations among the architecture instances. Specializations of a component correspond to limited and localized variability in a software component. An example of a simple specialization is setting the cache size for object servers in the architecture. A slightly more complex specialization is allocating classes to clients and object servers. When a class is allocated to a client or object server, the methods and data declarations for the class have to be made available to that client or object server at runtime. Specializations can typically be implemented using textual substitution in a compiler preprocessor or simple code generation.

Children of a component correspond to more extensive variability, such as might be associated with alternative implementations in a source code representation. An example of alternatives in a child component is whether the client component of the architecture is reentrant for remote method invocations or nonreentrant. In practice, this type of alternative is typically realized by different source code implementations.

The following sections describe the configuration nodes in the UFO parameterized architecture and the architectural variability that each represents. This collection of configuration nodes captures the key dimensions of architectural variability in the OODB domain. Readers less interested in these details and more interested in high-level concepts and rationale can skip ahead to Section 5.1.6.

5.1.5.1. Architecture Root Configuration Node

In the UFO reference architecture, every OODB architecture instance has clients, a set of object servers, a set of persistent cell managers, a repository manager, and a transaction manager. This common structure is seen in the root configuration node in the parameterized architecture, which simply referred to as the ARCHITECTURE configuration node. The variability at this level is in the alternate choices for the child configuration nodes: *client*, *object server set*, *persistent cell manager set*, *repository manager*, and *transaction manager.*

The *client* child of ARCHITECTURE represents the client processes. Clients for an architecture are either uniformly *reentrant* or uniformly *nonreentrant*. These two alternatives are discussed later in the client configuration node descriptions.

The *object server set* child of the ARCHITECTURE represents the object servers in the architecture (as defined in the reference architecture). There can be zero or more object servers defined for the object server set in an architecture. The alternatives for the object servers in an architecture are discussed in the object server configuration node descriptions.

The *persistent cell manager set* (*PCM set*) child of the ARCHITECTURE represents the persistent cell managers in the architecture (as defined in the reference architecture). There can be one or more persistent cell managers defined for the PCM set in an architecture. The alternatives for the persistent cell managers in an architecture are discussed in the persistent cell manager configuration node descriptions.

The *repository manager* child of ARCHITECTURE can either be *empty* or contain a *shared* repository manager. If it is empty, the repository manager will be embedded in the client process. Otherwise the shared repository manager node represents a single independent process in the architecture that is shared by all client processes. Shared repository managers are used in multi-user architectures.

The *transaction manager* child of ARCHITECTURE can either be *empty*, contain a *shared* transaction manager, or contain a *dedicated* transaction manager. If it is empty, the transaction manager will be embedded in the client process. A shared transaction manager node represents a single independent process in the architecture that is shared by all client processes. Shared transaction managers are used in multi-user architectures. A dedicated transaction manager node represents an independent process for each repository that is created. Clients bound to the same repository will share the dedicated transaction manager for that repository. This configuration is used when a single shared transaction manager is not sufficient to handle the transaction rate in multi-user, multi-repository applications.

5.1.5.2. Client Configuration Nodes

There are two types of client configuration nodes, reentrant and nonreentrant. The nonreentrant client represents a smaller software component with a subset of the functionality of the reentrant client. The nonreentrant client blocks while it is waiting for a return from one or more outgoing remote method invocations. An incoming remote method invocation while the client is blocked would lead to a deadlock. The nonreentrant client is provided for cases where control flow will never recursively reenter a client after going from the client to an object server. The reentrant client accepts incoming remote method invocations while it is waiting for a return from one or more outgoing remote method invocations. The implementation of the nonreentrant client will be smaller and simpler than the reentrant client since it doesn't need call reentrant stacks, and it will also have better performance due to shorter code paths.

The remaining architectural variance for the OODB clients is represented in the child configuration nodes. The child configuration nodes are the same for nonreentrant clients and reentrant clients: *object and cell services*, *embedded repository manager*, *embedded transaction manager*, and *cell allocations*.

There are three alternatives for *object and cell services* (refer to Figure 27. in the reference architecture), depending on whether or not object locking and persistent transactions are to be supported. The first option supports both object locking and persistent transactions. The second option supports locking, but not persistent transactions. The final option supports neither locking or persistent transactions. Additional details on the object and cell services configuration nodes are provided in a later section.

The *embedded repository manager* will be empty if the architecture contains a shared repository manager process. Likewise, the *embedded transaction manager* will be empty if the architecture contains either shared or dedicated transaction manager processes.

The *cell allocations* indicate the objects and classes that will be managed by the client. The overall allocation of cells to clients and object servers determines the call graph and whether or not reentrant calls to the client are possible (thereby determining whether the client should be reentrant or nonreentrant).

5.1.5.3. Object Server Configuration Nodes

Object servers are included in an architecture when locality is weak in a cluster of objects, in cases of large object clusters, or when objects in a cluster are not heavily utilized after they are activated. In all of these cases, overall performance may be improved by caching these clusters in a computational server and accessing the objects via remote method invocation rather than incurring the high amortized cost of activating these clusters into client processes.

Zero or more object servers to be specified to handle the these special computational needs. Initially a single object server might be used for large or low utilization clusters. Performance profiles from simulation will indicate whether this single object server is sufficient or whether it is overloaded and additional object servers are needed.

The set of object servers in the architecture can either be *dedicated* or *shared*. When dedicated, there will in a separate set of object server processes for each repository that is created (indicated for high contention repositories). When shared, there will be one set of object servers that is shared for all repositories (indicated for low contention repositories).

The set of object servers can be empty or consist of a heterogeneous collection of *reentrant*, *nonreentrant*, and *threaded* object server nodes. Each object server node represents an independent process in the architecture. The reentrant and non-reentrant are analogous to the client configuration nodes. The threaded configuration node is for supporting multiple execution threads from concurrent users. This is necessary in cases where the architecture supports multiple concurrent users or single user architectures with a shared object server set. Note that clients didn't need the threaded option since clients are always single user processes.

An object server configuration node has two child configuration nodes, *object and cell services* and *cell allocations*. These are both analogous to the child configuration nodes of the same name in the client.

There are three variants on the *object and cell services*, depending on whether or not object locking and persistent transactions are to be supported in the object server. The first option supports both object locking and persistent transactions. The second option supports locking, but not persistent transactions. The final option supports neither locking or persistent transactions. Additional details on the object and cell services configuration nodes are provided in the next section.

The *cell allocations* indicate the objects and classes that will be managed by the object server. The overall allocation of cells to clients and object servers determines the call graph and whether or not reentrant calls to an object server are possible (thereby determining whether the object server should be reentrant or nonreentrant).

5.1.5.4. Object and Cell Service Configuration Nodes

Object and Cell Services is a component in both clients and object servers, as illustrated in Figure 27. and Figure 31. in the reference architecture. To represent the variability in the parameterized architecture, there are six different object and cell services configuration nodes that correspond to the varying types of functionality supported by the object and cell services component. There are three dimensions that determine the six different object and cell servers: (1) single versus multiple execution threads, (2) object locking versus no object locking, and (3) persistent transaction support versus no persistent transaction support.

The following points in this three dimensional space are supported by corresponding configuration nodes:

- single threaded, object locking, persistent transactions
- single threaded, object locking, no persistent transactions
- single threaded, no object locking, no persistent transactions
- multi-threaded, object locking, persistent transactions
- multi-threaded, no object locking, persistent transactions
- multi-threaded, no object locking, no persistent transactions

The single execution thread alternative implies a more compact and efficient implementation than the multiple execution thread implementation. Having no support for object locking reduces the runtime overhead for lock checks and acquisition. Having no support for persistent transactions reduces the implementation size and code paths, implying a more efficient implementation. Therefore, the smallest and most efficient object and cell services alternative is “single threaded, no object locking, no persistent transactions” while the largest and least efficient (but most functional) is the “multi-threaded, object locking, persistent transactions”.

Each of the six types of object and cell services configuration nodes has one specialization and three child configuration nodes: *cache size*, *replacement policy*, *write-through policy*, and *distributed transactions*. The *cache size* specialization indicates the size of the object cache. The *replacement policy*, least recently used or most recently used, indicates the replacement policy for the cache. The *distributed transactions* child indicates whether or not distributed transactions are supported in the object and cell services node.

The *write-through policy*, eager or lazy, indicates the write-through policy of the cache during transaction commits. Eager write-through means that modified objects in the cache will be written to persistent memory during transaction commit (the conventional interpretation of “commit”). Lazy write-through means that the writes of some modified objects in cache may be delayed after a transaction commit in order to improve efficiency. This is suitable for applications that don’t need the strong persistence property in the conventional interpretation of “commit”.

5.1.5.5. Persistent Cell Manager Configuration Nodes

Persistent cell managers (PCMs) manage the persistent storage for clusters of objects. Depending on the performance load for an application, there can be one or more persistent cell managers running on different processors to distribute the load.

The set of persistent cell manager configuration nodes (the PCM set) in an architecture instance can either be *dedicated* or *shared*. Dedicated represents a separate set of PCM processes for each repository that is created (indicated for high contention repositories). Shared represents one set of PCMs that is shared for all repositories (indicated for low contention repositories).

The PCM set is a heterogeneous collection of one or more *eager* and *lazy* persistent cell manager configuration nodes. These correspond to the type of cache write-through policy defined for the object and cell services in the clients and object servers that use the PCM. An eager PCM serves as the persistent cell storage process for clients and object servers with eager write-through cache policies. Eager write-through is the conventional transaction approach of moving all modified objects from client or object server cache into persistent storage during a transaction commit. A lazy PCM serves as the persistent cell storage process for clients and object servers with lazy write-through cache policies. Lazy write-through differs from the eager write-through of the eager PCM in that there may be a delay in moving some modified objects from client or object server cache into persistent storage during a transaction commit. This delay increases the risk of loosing data as a trade-off for better performance.

Eager and lazy PCM configuration nodes have five child configuration nodes: *PCM allocations*, *persistent transactions*, *distributed transactions*, *locking*, and *backup*. The *PCM allocations* indicate which of the client and object server processes are assigned to use the PCM as their persistent storage server. The *persistent transactions* child is included when persistent transaction support is required throughout the architecture. PCMs store all persistent information related to persistent transactions. The *distributed transactions* child is included when the architecture has multiple PCMs and/or object servers contributing to transactions. The *locking* child is included when the architecture is supporting object locking.

The *backup* child configuration node is included when automated secondary storage backups are required in the architecture. This configuration node has two child configuration nodes, *separation* and *frequency*. The *separation* child indicates whether the destination of the backups is across a local area network or a wide area network. The *frequency* specialization indicates how often the automated backups are activated.

5.1.5.6. Repository Manager Configuration Nodes

The repository manager in the UFO parameterized architecture can be configured as a stand-alone process or an embedded component in the client process. There is a configuration node to represent each of these options.

The shared repository manager configuration node represents an independent process in the architecture that handles the repository operations such as create, delete, and binding. The repository manager is also responsible for creating other processes in the architecture that are associated with repositories such as transaction managers, object servers, and persistent cell managers.

The embedded repository manager serves a similar function as the shared repository manager, but rather than being an independent process it is embedded in the client process. This is possible when the architecture is single user and therefore will not require multiple concurrent repository management operations from different clients. This eliminates a process in the architecture, reduces the overall OODB architecture size, and eliminates RPC communication costs between the client and the repository manager.

There are three specializations for both the shared and embedded repository manager configuration nodes: *transaction manager integration*, *object server integration*, and *PCM integration*. These specializations indicate whether the transaction manager, object server set, and PCM set are shared, dedicated, embedded, or empty so that the repository manager can correctly create these processes and bind clients to these processes.

5.1.5.7. Transaction Manager Configuration Nodes

The transaction manager can be configured as a stand-alone process, multiple stand-alone processes with each dedicated to one repository, or an embedded component in the client process. There is a configuration node to represent each of these options.

The shared transaction manager configuration node represent an independent process in the architecture that serves as the central point of coordination for transactions.

The dedicated transaction manager configuration node represents one transaction manager dedicated to each repository rather than one transaction manager for all repositories. The dedicated transaction manager implementation implies a more compact implementation since it does not have to manage transactions in multiple repositories, but there will be more processes in the architecture. This is useful in applications where transaction throughput is very high.

The embedded transaction manager is embedded in the client process. This is possible when the architecture is single user and therefore will not require multiple concurrent transaction management operations from different clients. The eliminates remote procedure calls between the client and transaction manager and also implies a smaller size since it is not a stand-alone process.

Each of the transaction manager configuration nodes has two child configuration nodes, *persistent transactions* and *distributed transactions*. The *persistent transactions* child is included if persistent transaction functionality is required in the architecture. Likewise, the *distributed transaction* child is included if distributed transaction functionality is required.

5.1.6. Abstracting Architectural Variability: Architectural Parameters

Architectural parameters define how to select, specialize, and assemble a collection of configuration nodes for and OODB architecture. Although configuration nodes can capture the variations in the architectural building blocks for an OODB, remains a difficult task to select and compose a meaningful collection of configuration nodes that represent an OODB for a given application. For example, with the configuration nodes as defined in the previous section, over 250,000 different OODB architecture instances can be modeled.

To aid in this task, we needed abstractions for the dimensions of architectural variability and also automated guidance from these abstractions into meaningful collections of configuration nodes. Although we could have started with an “empty slate” in our search for these abstractions, there were some early clues as to what these abstractions should be.

In our initial attempt to implement the relationships between requirements and application instances with the dependency graph we observed an intermediate collection of nodes in the graph that served as an abstraction for architectural dimensions of variability. When we decomposed the dependency graph into four simpler tasks, a set of *architectural parameters* began to emerge from this intermediate layer of abstraction in the dependency graph.

The architectural parameters denote the dimensions of OODB architectural variability captured in the UFO parameterized architecture. Architectural parameters consolidate the implementation level variability in the configuration nodes into a high level expression of abstract variability. For example, sixteen of the different configuration nodes representing architectural building blocks have variability on whether or not long transactions are supported. However, at the architectural parameter level there is only one architectural parameter the specifies whether or not long transactions are supported throughout a particular instantiation of configuration nodes.

Following are descriptions of the twelve top-level architectural parameters. Note that some of these parameters are complex data structures containing multiple and nested values. We use these composite architectural parameter structures to group closely related architectural variants. Readers less interested in this level of detail can skip ahead to Section 5.1.7.

Long Transactions. A boolean parameter that indicates whether persistent transactions are supported. If long transaction functionality is not used by an application, then the architecture can be simplified by not including the functionality.

Object Locking. A boolean parameter that indicates whether object locking is supported. If the application does not use long transactions and is not multi-user, then object locking functionality is not needed in the architecture to control concurrent access to objects.

Distributed Transactions. A boolean parameter that indicates whether distributed transactions are supported. If the architecture does not have multiple client and object server processes that contribute to a transaction, then distributed transaction support need not be included.

Cell Declarations. A compound architectural parameter used to indicate locality in the data objects for an application. Cell declarations partition the runtime data structures for an application into groups of data objects called cells. This architectural parameter is included to improve performance based on expected use profiles and user expectations. The rationale behind defining data locality is discussed later in the section on requirement variables.

Clients and the Object Server Set. A compound parameter that expresses the configuration of each client and object server in an architecture instance. One parameter indicates non-reentrant, reentrant, or threaded. Others determine the *replacement policy* (least recently used or most recently used), *write-through policy* (lazy or eager), and *cache size* are for the object and cell services cache. A *cell allocations* parameter indicates the collection of cells – and therefore which objects – are managed by a client or object server. The cell allocations parameter will determine which class methods must be supported.

PCM Set. A compound parameter that expresses the configuration of each persistent cell manager. Includes a nested parameter value determining the *write-through policy* of the clients and object servers allocated to the PCM and therefore the type of commit write-through protocol to support. A *PCM allocations* parameter indicates the set of the client and object server processes that a PCM provides persistent cell storage for.

TM Integration, **OS Integration**, **PCM Integration**, and **RM Integration**. Expresses the way in which the transaction manager, the object servers, the persistent cell managers, and the repository manager are integrated into an architecture instance. TM Integration indicates whether the transaction manager for an architecture instance will be one stand-alone process dedicated per repository, one stand-alone process shared for all repositories, or embedded in the client process. OS Integration indicates whether the object server set for an architecture instance will be one set dedicated per repository or one set shared for all repositories. The value is EMPTY in the case where the Object_Server_Set_AP architectural parameter value is an empty set. PCM Integration indicates whether the persistent cell manager set for an architecture instance will be one set dedicated per repository or one set shared for all repositories. RM Integration indicates whether the repository manager for an architecture instance will be one stand-alone process shared for all repositories or embedded in the client process.

PCM Backups. Indicates whether or not automated backups of PCMs are supported by the architecture instance. If so, then there are two other nested parameter values. The *frequency* parameter indicates how often the automated backups should be performed. The *separation* parameter indicates whether the automated backup function supports backups across a local area network or a wide area network.

5.1.7. Mapping Architectural Parameters to Configuration Nodes

In this section we describe the UFO mapping in which a software architecture instance is configured from the configuration nodes, using the architectural parameters to drive the configuration process. This mapping is from the abstract dimensions of architectural variability defined by the architectural parameters to the configuration of architectural building blocks defined by the configuration nodes.

The relationship between abstract architectural variants, such as whether or not to support long transactions, and the implementation of these variants in an architecture instance can be complex. One architectural variable can impact multiple components in an OODB architecture instance. Also, one component in an OODB architecture can depend on multiple architectural variables. We needed a mapping from architectural parameters to configuration nodes that would capture these complex relationships from the OODB domain so that they could be reused each time an OODB architecture instance was configured in the UFO tool.

The architectural parameter to configuration node mapping is a top-down construction of configuration nodes, starting with the ARCHITECTURE configuration node at the root. We associate with each configuration node a *mapping description* that expresses how to specialize that configuration node and how to recursively select the appropriate child configuration nodes.

The mapping description for each configuration node varies based on the values of architectural parameters. For example, below is an operational mapping description in pseudo-code for the reentrant client configuration node, followed by a brief explanation. The mapping description addresses each specialization and child of the configuration node. The names of architectural parameters all end in “_AP”. Readers not interested in this level of detail can skip ahead to Section 5.1.8.

```
REENTRANT_CLIENT
    child object and cell services :=
        if (Object_Locking_AP)
            if (Long_Transactions_AP)
                construct SINGLE_THREADED_LOCKING_PERSISTENT_OCS
            else
                construct SINGLE_THREADED_LOCKING_OCS
            endif
        else
            construct SINGLE_THREADED_OCS
        endif
    child embedded repository manager :=
        case (RM_Integration_AP)
            when (EMBEDDED) construct EMBEDDED_RM
            when (SHARED) then skip
    child embedded transaction manager :=
        case (TM_Integration_AP)
            when (EMBEDDED) construct EMBEDDED_TM
            when (SHARED) then skip
            when (DEDICATED) then skip
    specialization cell allocations :=
        foreach CELLALLOC of Client_AP.<cell allocations>
            add <CELL_ALLOCATION> source CELLALLOC
```

The reentrant client configuration node construction consists of three child constructions and one specialization. The first child construction is for the *object and cell services* child of the reentrant client. The mapping description states that if the object locking architectural parameter value (*Object_Locking_AP*) is true and the long transactions architectural parameter value (*Long_Transactions_AP*) is true, then the object and cell services configuration node should be constructed with the single threaded variant that supports locking and persistent transactions (*SINGLE_THREADED_LOCKING_PERSISTENT_OCS*). If object locking is true and long transactions is false, then the *SINGLE_THREADED_LOCKING_OCS* is constructed. If object locking is false, then the *SINGLE_THREADED_OCS* is constructed. Note that all object and cell services configuration nodes for a client are single threaded. This is because no multi-processing is ever required in a client process in the OODB architecture. Also note that condition for object locking false and long transactions true is not supported. This is because long transactions always require object locking, so this combination of architectural parameter values will never exist.

The next child configuration node construction is for the *embedded repository manager* child of the client. The mapping shows that if the repository manager integration architectural parameter value (*RM_Integration_AP*) is *EMBEDDED*, then an *EMBEDDED_RM* configuration node is constructed, otherwise nothing is constructed in the client (the repository manager will be created elsewhere in the architecture in this case).

The next child configuration node construction is for the *embedded transaction manager* child of the client. This part of the mapping is analogous to that of the *embedded repository manager.*

The final entry in the mapping constructs the *cell allocations*. The cell allocations are used by the simulator to manage the cells allocated to the client. This is a specialization rather that a child construction since it requires only a simple code generation for data structures rather than a complete child configuration node that typically represents a significant architectural component. The specialization iterates through each of the *cell allocations* in the *Client_AP* architectural parameter to generate the cell allocations declarations for the reentrant client configuration node.

This pseudo-code example illustrates that the mapping this configuration node is relatively simple. We found that in general the mapping descriptions for configuration nodes have a similar small size and simplicity. There are no intricate computations involved, but rather straightforward mappings from architectural parameter constructs to related portions of the configuration nodes. By partitioning the overall mapping from architectural parameters to configuration nodes into smaller modular units (one per configuration node), we found that the complexity of writing and maintaining the mapping description is significantly reduced.

5.1.8. Abstracting Architecture Requirements Variability: Requirement Variables

Although the architectural parameters provide a layer of abstraction over the architectural building blocks, considerable knowledge about the OODB domain is needed to understand how to define architectural parameter values that satisfy the requirements for an application. For example, it is not obvious for which applications object locking is needed or how to allocate clients and object servers to different persistent cell managers in an OODB architecture. What we needed was a way to capture the knowledge about the variability in requirements on OODBs and then a mapping from requirements to architectural parameters.[1]

We first noticed the distinction between requirement variables and architectural parameters in the early implementation of our dependency graph. We observed that some of the entry points into the dependency graph were expressed very much like requirements while others dealt with implementation issues. Closer examination revealed that the entry points that dealt with implementation issues correspond to what we now call architectural parameters and that the dependency graph could be enhanced with an abstract layer of requirements over

1. Note that these requirements are for OODBs, not the end-user, application-level requirements.

all of the implementation concerns in the dependency graph. This eventually led to the current distinction between requirement variables, architectural parameters, and configuration nodes, each addressing separate concerns in the parameterized architecture.

The UFO parameterized architecture captures the knowledge about variability in OODB requirements in a set of *requirement variables,* plus a mapping from requirement variables values to architectural parameter values. Like architectural parameters, requirement variables denote the variability in the overall UFO architecture. However, the requirement variables are expressed in terms of OODB system *requirement* concepts rather than OODB system *implementation* concepts. This abstraction layer isolates developers from the OODB architectural issues and focuses them the OODB requirements domain.

Following are descriptions of the UFO requirement variables and what they specify in terms of requirements on an OODB architecture instance. The descriptions include the rationale and heuristics for setting requirement variable values for a particular application. Readers not interested in this level of detail can skip ahead to Section 5.1.9.

Long Transactions. This requirement variable is a boolean that indicates whether or not persistent transaction support is required in a UFO architecture instance. Static analysis of the application source code can determine this requirement variable value. In particular, if there are no calls in the application prelude to *CreatePersistentTransaction*, then no persistent transaction can ever exist below the root transaction and other persistent transaction operations such as *ChangeTransaction* and *CommitPersistentTransaction* will be illegal.

Multi-User. This requirement variable is a boolean that indicates whether or not a UFO architecture instance supports multiple concurrent users or only a single user at a time. This requirement variable value is set by the developers. The value cannot be extracted from the source code since the same source code can be used in a single user or multiple user OODB. The single user choice leads to a smaller and simpler architecture while multi-user provides the additional concurrent access functionality.

Locality Clusters. This is a compound requirement variable that indicates collections of persistent data objects called clusters that exhibit locality in locking and access a runtime. Two or more objects exhibit locality when there is a high probability that when one of the objects is accessed or locked that the other objects will also be accessed or locked in the same time frame. This requirement states an expectation for performance to be significantly enhanced when runtime locality occurs in a cluster.

The *role declarations* in a cluster identify the runtime objects that are members of the cluster and the class names of those objects. Roles have a name to indicate the “role” that they play in the cluster and a class to indicate the UFO Class or RootClass of the object. Objects from a single class can serve different roles in different clusters. The role declarations are used by the UFO runtime to distinguish which cluster that newly created objects are associated with.

Each cluster has a *utilization* variable that indicates the expected ratio of object accesses relative to the number of objects in the cluster for each activation of the cluster. Ideally this ratio is over 100%, indicating that objects are used repeatedly after they are activated. An expected utilization ratio value over 50% is specified with the value of HIGH, while lower than 50% is specified as LOW. The high and low utilization clusters are managed by different components in the OODB architecture instance.

The *size* variable for a cluster indicates the typical expected size for the cluster, where 1 megabyte or greater is considered LARGE and smaller is specified as SMALL. The large and small sized clusters are managed by different components in the OODB architecture instance. Profiling will provide the best information for the cluster utilization and cluster size values, though developers can make an initial guesses prior to profiling.

The *primary reliability* variable on a cluster indicates the reliability of transient transaction commits. HIGH corresponds to the traditional durability quality of all commits being persistent. LOW means that some recent commits may be lost in the face of a system crash. In either case, commits are atomic (i.e., have the “all or nothing” property). Primary reliability can be set to LOW in order to enhance system performance (commit operations may be delayed in order to batch process multiple commits). This only makes sense in cases where the durability property of the commit operation is not critical at the instant commit operations are called.

The *secondary reliability* variable indicates the automated backup support that is required on persistent storage media. *Frequency* indicates how often backups should be initiated and *separation* indicates the relative network location of the backup media (LAN or WAN). For very valuable data, the frequency should be high in order assure two copies of most data. For primary storage in high risk environments, the separation should be across a WAN to a low risk environment.

Sparse Cluster Sets. This requirement variable is used to specify requirements on how to best manage sets of low utilization or large sized clusters (as defined above for locality clusters). The intent is to identify clusters of objects that require special architectural support, such as keeping objects in large caches for long periods of time, in order to get good OODB performance. This is in contrast to the Dense Cluster Set requirement variable defined for clusters that exhibit characteristics that are most easily accommodated in the architecture for high OODB performance (small caches for very localized data access).

All of the clusters defined in the Locality Cluster requirement variable that were declared to have either low utilization or large size must go into a sparse cluster set. Each sparse cluster set will typically be allocated to an independent CPU and have a very large object and cell services cache, so there is a balance between too many clusters in a sparse cluster set over utilizing a processor and too many sparse cluster sets leading to an architecture with too many processors. Profiling can help to identify the appropriate balance, but developers can make an initial guesses prior to profiling.

Ideally, all of the clusters in a sparse cluster set have the same primary reliability (as defined above for locality clusters). If any cluster in a set has high primary reliability, then the entire set will be treated as high reliability.

Each sparse cluster set has a *global contention* variable. Without a multi-user requirement, the global cluster set contention value is always LOW. If there is a multi-user requirement, then global contention indicates the expected level of concurrent access across all repositories. It is set to HIGH if a single processor cannot adequately handle the computational requirements for all users of the OODB. Again, profiling provides useful feedback on establishing the appropriate value, although developers can make an initial guesses prior to profiling.

Dense Cluster Set. This requirement variable defines a single set (possibly empty) of high utilization and small sized clusters. The intent is to identify clusters of objects that exhibit characteristics that are most easily accommodated in the architecture to provide good OODB performance.

All of the clusters defined in the Locality Cluster requirement variable to have both high utilization and small size must go into the dense cluster set. The dense cluster set will be allocated to the client CPU since these clusters can be efficiently handled by a modestly sized processor and cache.

The dense cluster set has a *global contention* variable. Without a multi-user requirement, the global cluster set contention value is always LOW. If there is a multi-user requirement, then global contention indicates the expected concurrent access across all repositories. It is set to HIGH if a single persistent cell manager cannot adequately handle the computational requirements for all users of the OODB. Profiling provides useful feedback on establishing the appropriate value.

Cluster Set Partitions. In this requirement variable, the sparse and dense cluster sets are partitioned into sets with common primary reliability requirements (as defined above for locality clusters) and with computational requirement on persistent storage that can be handled by a single CPU. The cluster sets from the Dense Cluster

Set requirement variable and Sparse Cluster Sets requirement variable are partitioned into high and low primary reliability partitions. If the Multi-User requirement variable value is false, then at most one high reliability and/or one low reliability partition is sufficient.

If the multi-user requirement is TRUE, then the computational requirements on persistent storage must be reviewed for each partition so that the concurrent access to the clusters in the partition will not overload the computational resources of a persistent storage manager. If the computational requirements on persistent storage for a partition is more than a single CPU can support, then a *global partition contention* requirement value on the partition definition can be assigned a HIGH value. This will result in processors being allocated to handle the persistent storage requirements for each repository. If the computational requirements are still too high, or if there will be a large number of repositories with only a small fraction being utilized at a time, then the two partitions can be split into smaller partitions, each having a smaller computational requirement. Each of these smaller partitions will be allocated to an independent persistent storage processor. Again, global partition contention value can be set to HIGH or LOW, depending on whether the smaller partitions have excessive computational requirements. Profiling can provide feedback in establishing the partition requirements, but developers can make an initial guesses prior to profiling.

Global Transaction Throughput. This requirement variable specifies the expected overall transaction throughput rate across all repositories. Without a multi-user requirement, this value is always LOW. If there is a multi-user requirement, then the value should be set to HIGH if the computational facilities of a single processor is not capable of handling the transaction demands for all repositories. In this case, a separate processor is allocated to manage the transactions for each repository.

Reentrant Cluster Sets. This requirement variable specifies the inter-cluster call graph support required by the application. This information is used to determine when reentrant remote method invocation support is needed between clients and object servers and when it can be eliminated. The reentrant cluster sets are automatically derived by the UFO tool from the application source code and the cluster set requirement variables. By following the static inter-object method calls in the source code and by observing which classes of objects are in which cluster sets, the inter-cluster-set call graph is constructed. Cycles detected in the inter-cluster-set call graph indicate reentrant clusters.

5.1.9. Mapping Requirement Variables to Architectural Parameters

The final piece of technology in going from requirement variable values for an OODB to an architecture instance composed of configuration nodes is the mapping from requirement variables to architectural parameters. This is the mapping from the domain of requirements on OODBs to the domain of abstract architectural variability.

The mapping from requirement variables to architectural parameters is typically many-to-many; one requirement variable can impact multiple architectural parameters and multiple requirement variables can impact a single architectural parameter. For example, the requirement variable for single user versus multiple concurrent users impacts the architectural parameter for whether or not to implement object locking, the architectural parameter for control scheduling, and others. Conversely, the object locking architectural parameter is dependent on both the single versus multi-user requirement and the requirement variable for whether or not long transactions are required.

Below is an operational mapping description in pseudo-code for the client architectural parameters, followed by a brief explanation. The name of the architectural parameter in the pseudo-code ends in "_AP" and the names of requirement variables all end in "_RV". Readers less interested in this level of detail can skip ahead to Section 5.2.

Client_AP

```
    :=
    replacement policy := LRU
    write-through policy :=
        foreach Cluster of Dense_Cluster_Set_RV.<clusters>
            if ( Cluster.<primary reliability> == HIGH ) then EAGER ; EXIT
        LAZY
    cell allocations :=
        foreach ClusterName of Dense_Cluster_Set_RV.<clusters> insert with ClusterName
    control scheduling :=
        if (ReentrantCycles(Dense_Cluster_Set_RV) then
            REENTRANT
        else
            NON_REENTRANT
    cache size := 1000000
```

Client_AP is a composite architectural parameter with five component architectural parameters: *replacement policy*, *write-through policy*, *cell allocations*, *control scheduling*, and *cache size*. *Replacement policy* is always set to *LRU* (it is included for future tool enhancements). *Write-through policy* is set by scanning through all of the clusters in the *Dense_Cluster_Set_RV* to see if any of them have a *HIGH primary reliability* requirement. If so, then the *write-through policy* will be set to *EAGER* for the client to provide the highest reliability, else the value is set to *LAZY* to give higher performance at a lower reliability. Cell allocations are made to the client by scanning the cluster in the *Dense_Cluster_Set_RV* and inserting each cluster into the cell allocation set.

Determining the value for the *control scheduling* component requires searching for potential cycles in the call graph that go out of the client and back into the client. If such a cycle exists, then the client architecture must be *REENTRANT*, else the smaller and simpler *NON-REENTRANT* client architecture can be used. The *ReentrantCycles* function makes this determination from the *Reentrant Cluster Sets* requirement variable. If the dense cluster set, which is the cluster set managed by the client, has a method call cycle, then the *ReentrantCycles* function returns *TRUE* and the client will be *REENTRANT*.

This pseudo-code illustrates that the mapping for this architectural is relatively simple. As with the mappings from architectural parameters to configuration nodes, we found in general that the mappings from requirement variables to architectural parameters are relatively small and simple. We attribute this simplicity to the partitioning of the overall mapping from requirement variables to architectural parameters into appropriate modular units (one per architectural parameter).

5.2. Design of the OODB Architecture Modeler and Simulator

Referring back to Figure 3. on page 6, another key component in our approach, along with the parameterized software architecture, is the OODB architecture modeler and simulator. The feedback from modeling and simulation provides one of the three paths in our iterative cycle from requirements to architectures to system properties and back to requirements. The intent of modeling and simulation feedback is to provide developers with salient information about system properties for a particular OODB instance, relative to a particular application, while at the same time abstracting away from the myriad of details and system properties that are not significantly relevant in evaluating OODB architectures for a particular application.

In the initial phases of our studies, we focused on mapping from architecture abstractions to architecture instances. Feedback did not play an important role in these early efforts. We only included simulation in these studies as a way to implement the approach without having to incur to overhead of having to implement a configurable OODB architecture or of having to acquire and use multiple off-the-shelf OODBs.

Experience from our preliminary work taught us several things. First is that it is very difficult for developers to fully understand and accurately define OODB requirements for an application on the first attempt. This illustrated the importance of providing developers with feedback of system properties, helping developers to evaluate system properties in relation to their system requirements, and extending our approach with refinement of requirements followed by repeated iteration through the cycle. Second is that, just like us in our research, development projects can't afford the overhead of time and money to acquire multiple OODBs for the purpose of evaluation. Simulation provides an attractive alternative in a practical application of our approach. And finally, as we explored the importance of feedback, we identified static modeling and inconsistency detection as two other techniques in addition to simulation for providing developers with feedback about salient OODB properties.

The three types of feedback that we identified are illustrated as the three bold arrows pointing upward in Figure 4. on page 7: (1) inconsistencies between baseline and off-the-shelf architectural variants, (2) static architecture properties from modeling, and (3) dynamic architecture properties from simulation. The first type of feedback provides information to developers about inconsistencies between the architectural parameters for a baseline OODB instance for an application and the architectural parameters of an off-the-shelf OODB being considered for the application. The second type of feedback provides information to developers about static system properties such as architectural configuration and size. The third type of feedback provides developers with detailed profiling data on how an OODB architectural instance performs under simulated application scenarios. In the following sections we discuss the issue associated with these types of feedback from the UFO tool.

5.2.1. The Modeler

When we began to explore the feedback issues in our technology, one potential source we identified was the architecture instance create out of configuration nodes. Recall that an architecture instance is a static model of an OODB architecture, constructed prior to the simulation phase. We not only looked for system properties that we could extract from the architecture model and provide as feedback to developers, but also for ways that we could extend the representation to provide better feedback.

We identified two general types of system modeling properties that can be derived from architecture instance by the UFO tool, (1) architectural configuration data and (2) architectural size data. This information provides useful feedback to developers about resource utilization such multi-processing and memory utilization. This feedback can be used to identify resource utilization that may be excessive for the intended application execution environment. For example, an OODB instance with several large server processes may be inappropriate for a laptop computing application.

The *architectural configuration data* presents the collection of architectural components in the OODB architecture instance, the allocation of architectural components to processes, and the multiplicity of processes (such as one shared persistent cell manager versus multiple persistent cell managers). The architectural configuration data directly reflects the collection of configuration nodes for an architecture instance.

The *architectural size data* presents the estimated executable system size information for architectural components in the OODB architecture instance. This data is projected from static size attributes associated with configuration nodes.

Note that only static modeling properties can be modeled in this phase. For example, while we can present the size of the object cache allocated in the client architecture, we cannot project dynamic properties such as the ratio of cache hits to cache misses that are later exposed during simulation.

Developers can use this information to isolate architecture configurations or system size data that may be excessive or inconsistent with an application. If there are problems, developers can modify the requirement variable values to reflect adjustments to the architecture and then rederive the architecture instance.

The following two subsections provide more detail about the static modeling feedback supported by UFO. Readers not interested in this level of detail can skip ahead to Section 5.2.2.

5.2.1.1. Architectural Configuration Data

The architectural configuration data from UFO is a human readable representation of the internal configuration node composition for an architecture instance. For example, the architectural configuration data presented for the client configuration node will display the type of client (reentrant versus nonreentrant), recursively display each child configuration node (object and cell services, embedded repository manager option, embedded transaction manager option), plus display the specialization for the cell allocations.

The only implementation alternative worth noting is the format of the human readable display for the configuration nodes. For the UFO implementation, we use a textual representation with indentation to show composition. Other attractive alternatives for displaying the architectural configuration data include graphical display formats and the various analysis and design notations.

5.2.1.2. Architectural Size Data

The architectural size data in UFO comes from attributes we place on the configuration nodes. For the collection of configuration nodes in an architectural instance, we display the size for each configuration node plus the child configuration nodes in its composition. For example, for a non-reentrant client configuration node, the overall size is computed from:

- 1,000,000 bytes for the client configuration node
- plus 300,000 bytes for the child object and cell services configuration node (configured as single threaded, object locking, and long transactions)
- plus 50,000 bytes for an optional embedded repository manager child configuration node
- plus the object cache size
- plus code size for the application classes allocated to the client

For configuration nodes that represent dedicated architectural processes (i.e., one per repository), the size data is presented as "per repository".

The size data helps developers understand the runtime memory requirements for an OODB architecture instance. This information may be useful to developers in cases where an application is running on a machine with limited memory resources.

The overall number of processes in the architecture is also reported in the data. This information may be useful to developers in cases when an application is running on a machine with limited process resources.

5.2.2. The Simulator

Given the prototype application source code and an OODB architecture instance (represented as configuration nodes), UFO's OODB simulator will simulate the application running on the architecture. The simulator collects runtime data so that the execution properties of the application running on the OODB can be presented in a simulation profile.

An initial observation that we made is that interpretation of the application source code is distinct from simulation of the OODB architecture. Treating these as separate problems in our design and implementation offered several advantages:

- The application source code can be interpreted without running an OODB simulation. This allows application developers to prototype and debug their application source code before focusing on the OODB simulations and profiles.

- The separation in the design makes it easy to reconfigure the OODB instance being simulated without reconfiguring the application source code interpreter.

- As we were prototyping and developing the UFO OODB simulator, the OODB interpreter remained most independent of the modifications and extensions we made to the simulator.

Based on these observations, we designed the interpreter so that it functions stand-alone or in conjunction with the OODB simulator. During simulation, the interpreter drives the simulator through an interpreter/simulator interface. The following two subsections discuss these two components.

5.2.2.1. UFO Application Interpreter

The role of the UFO application interpreter is simply to execute the application source code, without concern for the configuration of the OODB architecture instance under simulation. In fact, the interpreter is designed to be fully functional without have any OODB architecture instance defined.

We use conventional compiler and interpreter techniques for the UFO interpreter. While a significant design effort was involved, new innovations were not necessary. The application source code is represented and stored as an *abstract syntax tree* in the UFO tool, similar to a parse tree produced by a parser. An additional interpreter structure produced for each application is a set of repositories. Each repository in the set contains structures for the persistent transaction hierarchy and a persistent object heap.

The interpretation of the application code begins in the abstract syntax tree with the first statement of the application prelude. Each statement and each expression in the statements are executed according to the semantics outlined in Section 5.1.2. Each repository retains a persistent handle on the *root object* in the repository. Execution in the persistent object space is bootstrapped when a boot procedure is applied to the root object. Details on the implementation of the UFO application interpreter are given in the next chapter.

5.2.2.2. Interfacing the Application Interpreter to the OODB Simulator

The configuration nodes for an OODB software architecture instance are represented as an abstract syntax tree in the UFO tool. In order to simulate the execution of an application on an architecture instance, the interpreter described in the previous section extends its scope to interface with the simulation constructs in the OODB instance. These simulation extensions do not change the semantics of the code interpretation. Their role is simply to profile selected operations performed by the architectural components. For example, the interpreter will create, commit, and abort transactions while the simulator will keep track of how many transaction creates, commits, and aborts that each transaction manager, persistent cell manager, object server, and client participate in, plus the overall time spent in each component on the different operations.

In the OODB language interpreter, data objects are the central design focus while in the OODB architecture simulator, data cells are the central design focus. Recall that cells support runtime locality through clusters of data objects. Capitalizing on this locality is essential for enhanced OODB performance, so it typically gets a great deal of attention in OODB architectures.

An important issue for us then is how to interface the interpreter with its object focus to the simulator with its cell focus. Rather than extend the interpreter to understand cells, we provided an interface to the simulator for each interpreter operation on a data object. The simulator associates objects with cells and performs all of the architectural operations that result from the operation on the object.

For example, when the interpreter creates a new object, the associated call to the simulator allocates the new object to a cell. Subsequently, when the interpreter performs operations on the object, the simulator evaluates corresponding operations on the enclosing cell. Examples include:

- activation of cells from persistent cell managers to clients and object servers when an inactive object is accessed in the interpreter
- cell cache replacements when the activation of a cell exceeds cache capacity
- creation of a new cell when the first object for a cell is created
- management of cells during transaction creations, commits, and aborts

The data collected and presented from these operations in a simulation profile is described in the next section.

5.2.2.3. Simulation Platform

An OODB will exhibit different performance profiles on different hardware platforms. CPU speed, network speed, disk speed, and operating system efficiency all have an impact on performance. We felt that the UFO simulator should reflect this reality. We provide a set of simulation constants that can be configured prior to simulation. Developers can specify simulation constants for simulating the platform that execute the application and OODB architecture components. The simulation constants supported by the UFO tool are:

- Clock speed for the cpu's running the clients, object servers, and persistent cell managers. This allows, for example, developers to simulate a network configuration where persistent cell managers and object servers run on high-powered server machines while clients run on lower powered desktop machines.

- Disk access and network costs (in cycles) for persistent cell and persistent object manipulation: creation, commit, abort, activation, and deactivation. This allows, for example, developers to simulate fast disk arrays and cheaper disk configurations to compare cost and application performance.

- Costs (in cycles) for inter-cell, intra-cell, and inter-process method invocations. This allows developers to experiment with cost/performance trade-offs in cache, operating systems, and networks.

The UFO tool provides some predefined simulation constants. Developers can request that these "canned" values be used for simulation constants or they can tailor their own simulation constants.

5.2.3. Dynamic Simulation Profiles

After a simulation, the UFO tool produces a profile summary for feedback to developers. Developers use this information to identify performance problems and resource limitations for the OODB instance, relative to the simulated application scenario. If problems are found, developers can modify the requirement variable values accordingly then rederive and simulate the architecture instance.

A particular challenge that we faced as we designed the simulator was distinguishing between useful information that the simulator could provide developers and irrelevant information that would only distract developers from salient information. Experience with the tool helped us identify system property information that was most useful as we compared system properties to requirements for an architecture instance.

Even with being selective in filtering the simulation data from the UFO tool, the data can be large and complex. To aid in this problem, we explored techniques and trade-offs for manual versus automated profile analysis. For example, the UFO tool might perform some automated analysis of the simulation profiles in order to identify problems and to provide developers with specific suggestions on how to refine requirement variable values to achieve better performance.

Manual analysis of UFO simulation profiles for OODB architecture instances requires expertise, both in terms of architecture performance issues and how these relate to refining requirements to enhance performance. Manually analyzing performance profiles can also be time consuming for developers. Therefore, to keep the overall development costs low, a tool like UFO should provide automated performance analysis and feedback whenever possible.

While this type of automated support can be very powerful, we found that it is difficult to design and implement. It requires identifying and encoding the type of knowledge that is typically possessed by system performance experts. Therefore, a thorough implementation of automated performance analysis and feedback would likely require an expert system approach.

The following two sections describe in more detail types of information that UFO provides for manual profile analysis and from automated profile analysis.

5.2.3.1. Profile Summary for Manual Review

In this section we describe the simulation profile data supported by the UFO tool, plus examples of how developers can use the information to refine the OODB architecture to better conform to their application.

Execution time for a simulation

The overall execution times are reported for the client, the object server, and the persistent cell manager subsystems, plus the overall combined times for the full OODB architecture. Developers use this information to make gross comparisons between different architecture instances and look for major performance problems in the major subsystems.

Intra-cell method calls, inter-cell method calls, and inter-process method calls

Detailed information is provided about the locality exhibited during method invocations. Ideally, the majority of object-to-object method invocations take place within cell boundaries and the minority are inter-process. The UFO presents data about the number of invocations and the time associated with invocations made and received at the following granularities:

- per client
- per object server
- overall architecture
- per cell
- per class
- per method declaration
- per method application

Problems that are observed by developers at larger granularities can be tracked down to their source at smaller granulatities. This is one of the key areas for improving performance by improving locality in the requirements definitions.

Creations, activations, deactivations

Similar to the method invocation data, detailed information is provided about the locality exhibited for object and cell creation, activation, and deactivation. Ideally, the activation and deactivation of cells should be rare compared to accessing the objects in the cells after they are activated. The UFO presents data about the number of cell and object creations, activation, and deactivations, plus the associated times, at the following granularities:

- per client
- per object server
- overall architecture
- per cell

Problems that are observed by developers at larger granularities can be tracked down to their source at smaller granulatities. This is another one of the key areas for improving performance by improving locality in the cells and by partitioning cells onto clients or object servers according to the degree of locality that can be achieved in the cells (object servers can better handle cells with weaker locality than can clients).

Maximum dynamic cell size per cell type

Cells are dynamic in that their size can grow and shrink as objects are created and deleted at runtime. The maximum cell size for each type of cell is monitored and reported to catch cases when cells are getting too large to be efficiently activated and deactivated. Subdividing large cells or allocating them to object servers rather than clients can improve performance in some cases.

Transaction creates, commits, and aborts

Transaction profiles are also detailed in the feedback from the UFO tool, both for persistent transactions and transient transactions. Cells are the unit of granularity for commits and aborts, but we also monitor the total number of objects involved in commits and aborts. The data for raw counts, plus the overall times involved, are reported for the transaction manager, the persistent cell managers, and for the overall architecture. This information helps developers identify cases where transaction processing resources can be increased or decreased.

Repository creates, deletes, bindings

The final area of profile data is for repository operations. This information is primarily valuable in cases where architectural processes such as persistent cell managers are dedicated to one repository. Large numbers of repositories can lead to large numbers of total processes executing in an architecture.

5.2.3.2. Automatic Profile Analysis

After some exploration, we determined that extensive research in this area was beyond the scope of this thesis. However, we did want to illustrate the power of automated performance analysis and feedback. We demonstrated the capability in two performance areas, while we defer to future work the study of what is required for a more complete coverage of automated analysis (see Chapter 11., Future Work).

Low Cell Utilization

For each cell declaration allocated to the client, if the total of all method calls received by objects in all of the cell instances is less than 50% of the total number of objects activated during all cell instance activations, then report the following feedback:

Low utilization cell in client. Change cluster utilization requirement to Low for this cell.

The recommended requirement variable value change will cause the cell instances for the cell declaration to be allocated to an object server in the OODB architecture, where it can be cached between client sessions. This may lead to better amortized costs for object access.

Oversized Cells

For each cell declaration allocated to the client, if the total of any cell instance becomes greater than one megabyte, then report:

Oversized cell in client. Change cluster size requirement to Large for this cell.

The recommended requirement variable value change will cause the cell instances for the cell declaration to be allocated to an object server in the OODB architecture, where it can be cached between client sessions. This may lead to lower activation and deactivation costs for large cells.

5.2.4. Inconsistency Detection

When the UFO tool is used to evaluate off-the-shelf OODBs, inconsistencies between the baseline architectural parameter values for an application and the architecture parameter values supported by an off-the-shelf OODB are detected and reported by the UFO tool. The feedback is presented to developers, both in terms of architectural parameter inconsistencies and the baseline application requirements that cannot be satisfied due to the inconsistencies. Given this feedback, developers have the option of either refining the architectural parameters to eliminate the inconsistencies or abandoning the evaluation if they decide that the off-the-shelf OODB does not sufficiently support the requirements for their application.

In some cases the tool can detect the potential for an inconsistency, but cannot make an absolute determination. For example, some OODBs do not support the nesting of long transactions, but static analysis of the application prototype code cannot determine if or when the dynamic control flow might result in recursive long transaction invocations. These types of potential inconsistencies are reported as warnings. Developers must manually evaluate warnings to determine whether or not they are indeed inconsistencies.

Note that by evaluating off-the-shelf OODBs using the architectural parameters, we are considering only architectural functionality and structure, not performance issues that are exposed later during simulation. We made this choice because the analysis of dynamic performance feedback is considerably more complex. We determined that the effort required to automatically quantify the relative suitability of an OODB based on complex profile data was beyond the scope of this research.

Our approach to detecting inconsistencies for off-the-shelf OODBs requires selecting and evaluating one OODB at a time. We originally explored the possibility of a fully automated search and evaluation of all the off-the-shelf OODBs for a given application and then automatically selecting the one with the best results. We found that this approach had several significant problems. First, it is difficult to quantify the suitability of an OODB that has inconsistency warnings since the warnings may represent a serious incompatibility or may represent no problem whatsoever. Second, since our approach relies heavily on feedback from simulation profiles, the real suitability of an OODB cannot be determined from architectural parameters only.

We identified two types of inconsistencies that we could report from the tool:

1. **Missing functionality.** Functionality that is supported in the baseline architectural parameters for an application but that is not supported by the off-the-shelf OODB under evaluation. In this case, the developers will have to decide whether the functionality is an absolute requirement or whether the application can be successfully deployed without the functionality. If not, then the OODB can be excluded from consideration for the application.

2. **Excess functionality.** Functionality that is not indicated in the baseline architectural parameters for an application but that is always part of the off-the-shelf OODB under evaluation. In this case, the tool indicates to developers the extraneous functionality associated with the OODB. The developers can then observe how the excess functionality impacts system size and performance in subsequent modeling and simulation.

Following are several example of inconsistencies and warnings supported by the UFO tool for the three off-the-shelf OODBs modeled: Objectivity, ITASCA, and ObjectStoreLite. Each example contains the name of the architectural parameter and the off-the-shelf OODB for which the inconsistency check is done, the condition under which the inconsistency is reported, the corresponding baseline requirements that cannot be supported, and the suggested modifications to the baseline architectural parameters in order to eliminate the inconsistency.

PCM_Set_AP (for Objectivity)

If the PCM_Set_AP contains more than 1 PCM

Then Report The Multiple Cluster Set Partitions requirement is inconsistent with Objectivity. Objectivity supports only 1 PCM per repository. Allocate all Clients and Object Servers to 1 PCM.

Object_Server_Set_AP (for ObjectStoreLite)

If the Object_Server_Set_AP is not an empty set

Then Report The Sparse Cluster Sets requirement is inconsistent with ObjectStoreLite. ObjectStoreLite doesn’t support computational object servers. Allocate all cells to the Client.

Object_Server_Set_AP (for ITASCA)

If the Object_Server_Set_AP is an empty set

Then Report The Dense Cluster Set requirement is inconsistent with ITASCA. ITASCA does all computation in object servers. Allocate all cells to the Object Servers.

Long_Transaction_AP (for ITASCA)

If Long_Transaction_AP is TRUE

Then Report Warning. ITASCA supports only a single nested long transaction. Check application code for multiple long transactions.

5.3. Adapting Software Architecture Modeling and Simulation Techniques to OODB Architectures

OODBs themselves are tools for developing larger software systems and as such have software development techniques associated with them. One of the early challenges that we faced was that in order to apply software architecture modeling and simulation techniques to OODBs, the two different software development techniques had to be merged. In this section we discuss the issues associated with adapting OODB application development techniques to software architecture modeling and simulation and conversely how we adapted software architecture modeling and simulation to OODB development techniques.

Readers interested in applying the software architecture modeling and simulation approach to classes of systems in domains other than OODBs will be particularly interested in this set of issues. While aspects of the UFO technology such as architectural parameters and configuration nodes are largely independent of domain, the merger discussed in this section is one area where domain-specific characteristics are particularly important in the software architecture modeling and simulation approach.

5.3.1. Conventional Application Development with OODBs

Figure 32. illustrates a typical approach to building applications using an OODB. Starting with the task oval on the left, the application source code is written using the application programming interface (API) provided by the OODB. This API is typically a conventional object-oriented language, such as C++ or Smalltalk, that has been extended with database constructs. A specialized compiler provided with the OODB is then used to translate the source code into executable application code that runs cooperatively with the OODB runtime. The compiler specializations accommodate the OODB extensions to the language. The OODB runtime typically consists of one or more servers to manage persistent storage, concurrent access, and other OODB functions.

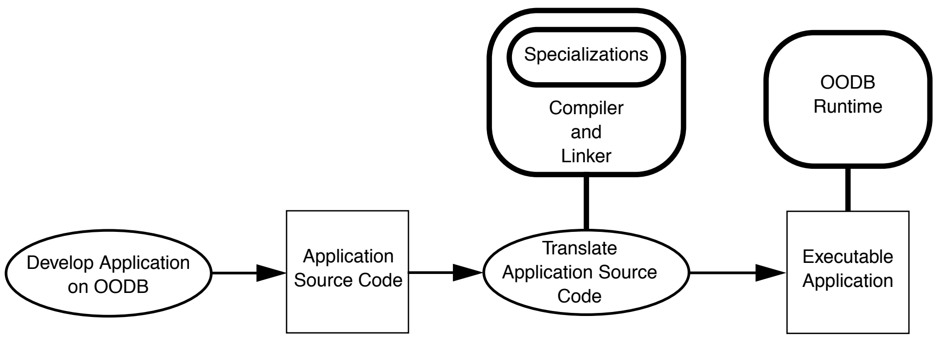


Figure 32. Typical Application Development with OODBs

5.3.2. Application Development with UFO

We next illustrate how we merged UFO's software architecture modeling and simulation technology with that of OODBs. The resulting hybrid represents our solution for modeling and simulating OODB instances in the design space of OODB system architectures.

Recall from Section 5.1.2. that one of the first difficult issues we confronted was that each OODB we studied had a different application programming interface (API). In order to model the design space for OODBs, we didn't want to implement this multitude of different languages that all did essentially the same thing. Furthermore, in our simulations we wanted to be able to write application code once and then simulate the application on different off-the-shelf OODBs without having to modify the application code to match the language for that off-the-shelf OODB.

The approach we took was to generalize the APIs for the OODB instances in our implementation in order to create a single API that captures the important features of many OODB APIs. Any given OODB architecture instance may only support a subset of the generalized API. This is reasonable since most applications will only use a subset of the features available in the API. The key to this approach is to assure that the OODB instance simulated for an application supports the same set or a superset of the API features utilized by the application.

Recall that we refer to the generalized OODB API as the *virtual OODB*. The virtual OODB allows application prototypes to remain independent of the OODB instance that is realized for modeling and simulation. This independence provides the primary advantage of the virtual OODB: different OODB architectures can be realized and explored without changing the application source code.

5.3.2.1. Defining Baseline Requirements with UFO

Figure 33. shows how OODB baseline requirements are defined using UFO. The top half of the diagram corresponds to the basic modeling and simulation approach shown in Figure 4. on page 7, while the bottom half of the diagram corresponds to the OODB approach from Figure 32. above. The modifications required to merge these two techniques are denoted with heavy bold lines. This merger supports building applications using the virtual OODB API, defining OODB requirements for the application, modeling and simulating an OODB instance that satisfies these requirements, feedback on the system properties and simulation profiles, and iterative refinement of the requirements.

The task ovals in the diagram are labeled with numbers to indicate the relative order in which the tasks are carried out. Starting with task number 1 at the lower left of the diagram, the source code for an application is written using the virtual OODB. This task is drawn in bold to indicate a point of merger between UFO and conventional OODBs. In this case, the UFO virtual OODB represents the generalization for different conventional OODB APIs.

After the application source code has been created, the requirements for the OODB are defined in task 2. These requirements come from two sources, application source code and developers. Requirements such as whether or not long-transaction functionality is used by the application can be automatically extracted from the source code. This explains our decision to place the requirements definition phase *after* the application source code has been written. Requirements that can't be automatically extracted from the source code, such as whether the application will be used as a single-user or multi-user application, must be elicited directly from the developer.

Once the requirement variable values are defined, they are mapped to architectural parameters in task 3, and then the architectural parameters are mapped to a software architecture instance in task 5. Static OODB modeling properties that can be derived from the static architecture instance, such as the architectural configuration and component sizes, are fed back to the developer in task 6. This is one source of system properties that assists developers in refining and validating OODB requirements for an application.

The OODB architecture simulator produced from task 7 is a primary point of merger between conventional OODBs and UFO, as illustrated by the bold arrows leaving task 7. Note in the figure that the realizations for the software architecture instance, output from task 7, are specializations to the OODB simulator preprocessor and runtime. We designed these specializations such that the bulk of the simulator implementation remains unchanged as the specialization change for different OODB architectures.

As illustrated by the case study in the previous chapter, developers use the feedback from the UFO modeling and simulation tool to refine the requirements, repeat the mappings and simulations, and reevaluate the profiles until they have converged on a set of baseline requirements that accurately matches the requirements for the application. The level of effort to use a tool based on the UFO design can be relatively low since tasks 3, 5, 6, 7, 8, and 9 can be fully automated and task 2 can be partially automated.

5.3.2.2. Evaluating Off-the-Shelf OODBs with UFO

Using the UFO tool to evaluate off-the-shelf OODBs is similar to Figure 33. with the exception of the addition of Task 4, detecting inconsistencies. This task makes use of information about off-the-shelf OODB architectures in order to identify inconsistencies between the baseline architectural parameters and a particular off-the-shelf OODB. Inconsistencies are fed back to the developer so that the inconsistencies can be resolved.

Tasks 1, 2, and 3 are the same as in defining baseline requirements. Then, once the inconsistencies are resolved for a particular off-the-shelf OODB, tasks 5, 6, 7, 8, and 9 proceed as in requirements validation. The feedback about the static modeling properties and the dynamic simulation properties is used to compare the application requirements and with the feedback from modeling and simulating other off-the-shelf OODB architectures.

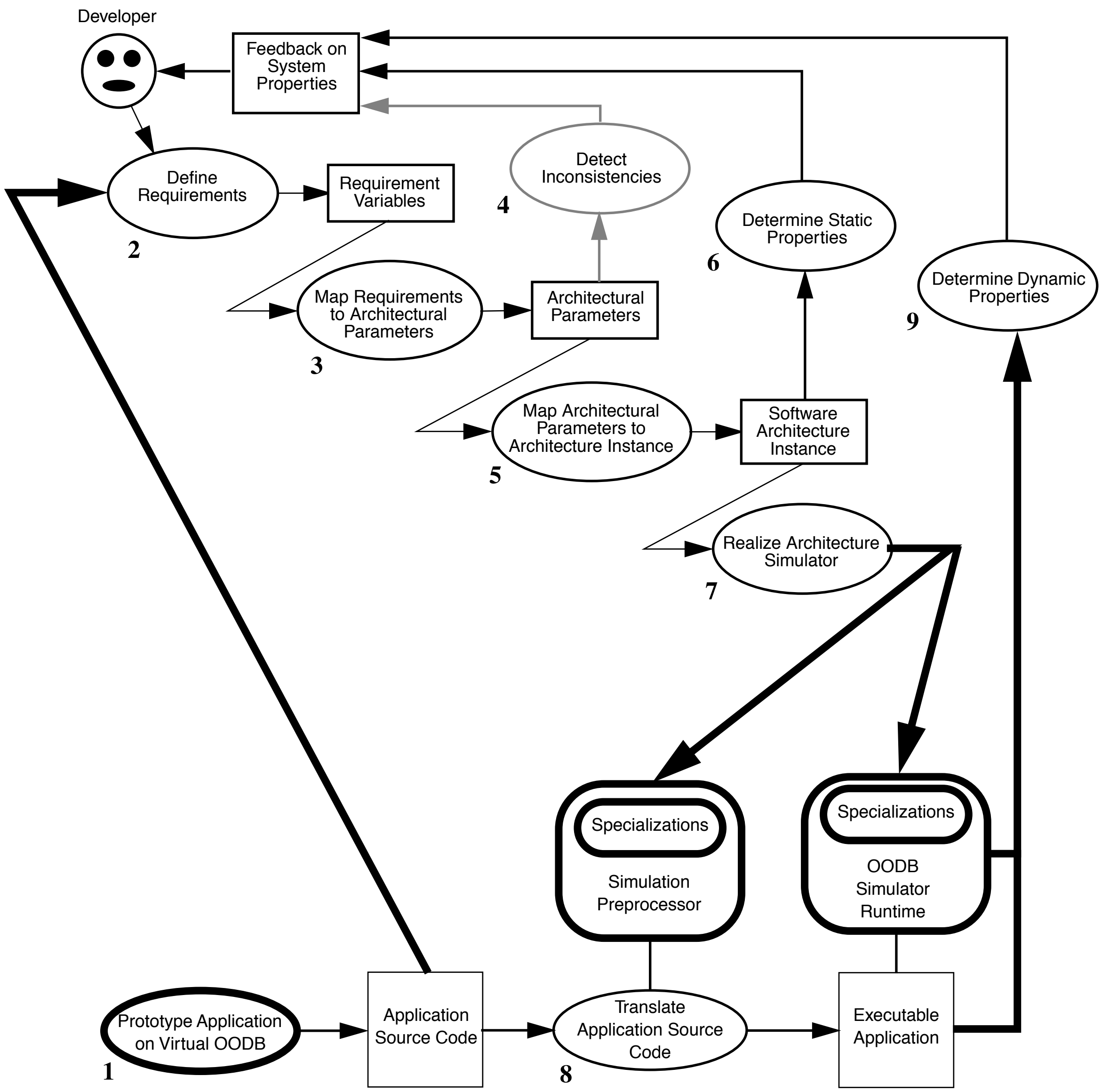


Figure 33. Merging the OODB Domain with Architecture Modeling and Simulation

Developers can explore multiple off-the-shelf OODB architectures for conformance to application requirements. When inconsistencies are detected between the application requirements and an OODB in task 4, developers are notified of requirements not satisfied by the OODB architecture. Developers identify the off-the-shelf OODB that results in the fewest inconsistencies with respect to the application requirements in order to find the OODB with the best conformance. This is consistent with what is observed in software development with conventional OODB. Developers are often forced to compromise an optimal set of application requirements in order to use an off-the-shelf OODB that doesn't precisely satisfy the optimal application requirements. The choice on which OODB to use is based on the perceived match to application requirements. The UFO tool is designed to more accurately and cost effectively identify the off-the-shelf OODB with the best match.

After an off-the-shelf OODB has been selected, the application can be developed on that OODB. However, application source code written using UFO's virtual OODB API cannot be compiled with conventional off-the-shelf OODBs, so any application source code written for the UFO modeling and simulation will have to be ported to the API of the selected OODB. While there is at least one industry effort under way to produce a single standardized OODB API[9], current OODBs each have a unique API. If at some point in the future an industry standard API becomes available, then the UFO virtual OODB API can be built using the standard and application code can be used interchangeable between UFO and other off-the-shelf OODBs that adhere to the standard.

Chapter 6. Implementation of the UFO Tool Set

In this chapter we describe our implementation of the UFO tool set. The implementation is based on the design presented in the previous chapter. Our discussion focuses on the technological issues associated with the implementation strategy for a software architecture modeling and simulation tool set that meets our objectives of configuring and validating high conformance OODB architectures at a low cost.

In Section 6.1. we outline our implementation objectives, base implementation technology, and implementation boundaries. Section 6.2. addresses our implementation's tightly integrated data model and tool set. The base implementation technology provided by the Gandalf system is discussed in Section 6.3. In Sections 6.4. through 6.14., each of the different tools within the UFO environment is described.

6.1. Overview of the Implementation

6.1.1. Objectives

Our objective with the UFO implementation is to create a software tool set to demonstrate the effectiveness and practicality of software architecture modeling and simulation technology. As defined by the thesis statement, the tools should reduce the cost of developing software architecture instances within a class of software systems and also improve our ability to instantiate software architectures with high conformance to application requirements. The UFO tool set is implemented to support our claims that automated mappings from requirements to architecture instances combined with modeling and simulation feedback will result in lower development costs and higher conformance between requirements and architecture instances.

The following features and tools in the implementation help to accomplish our stated objectives:

- a requirements-definition template tool that guides developers in selecting requirement variable values
- automated extraction of certain requirement variable values from the source code to reduce the requirements definition effort
- automated mappings
  - requirement variable values to architectural parameter values
  - architectural parameter values to software architecture instances
  - software architecture instances to simulators
- inconsistency detection and feedback for comparative analysis of off-the-shell OODB architectures
- OODB architecture static modeling and feedback
- OODB architecture simulation, profile feedback, and automated profile analysis

6.1.2. Base Technology

We used the Gandalf System[11] as the base technology to implement the UFO tool set. Our decision to use Gandalf was based on the following reasons:

- Gandalf is a software environment generator technology that allows rapid, iterative prototyping of software development tools such as those required in our UFO implementation.

- Gandalf supports tightly coupled tools and data, allowing us to integrate the UFO editing, mapping, modeling, and simulation tools.

- Our familiarity with the Gandalf System allowed us to accurately access the risks, advantages, and prototyping effort associated with the base technology.

- Tools produced using the Gandalf System provided a structure-oriented user interface that provides users with significant help and guidance. This makes the tools easier to learn and use.

The UFO tool implementation using Gandalf was approximately an eight person*month effort. The final implementation consisted of the following.

- the data schema and data visualization declarations, which required approximately 5250 lines of Gandalf source code

- operational source code for approximately 400 different data types, which required approximately 20,000 lines of Gandalf source code

This was produced using Gandalf structure-oriented editors, so significant portions of the code were automatically produced in the editing templates. From the 5250 lines of schema descriptions and 20,000 lines of source code, Gandalf generated 37,250 lines of C source code and combined this with 116,000 lines of reusable Gandalf code libraries to produce the executable UFO environment. Therefore, the 25,250 lines of source that we developed resulted in 153,250 lines of C code.

6.2. The Integrated Data Model and Tools

In the UFO implementation, we used Gandalf's integrated data management system for all of the data structures in all phases in all of the UFO tools. The advantage of having a single, unified data representation is the tight integration of tools in the environment. All information is available at all times to all of the tools. This contrasts with traditional batch-mode tools, where selected information from one tool is batch transformed into information for the next tool, but not the converse. So, for example, in UFO the semantic analysis tool is concurrently active during the editing phase to assist developers correct static semantic errors as they are editing a UFO program.

UFO's integrated data and tools are illustrated in Figure 34. The partitioning of the tools and data collections directly reflects the design and rationale from the previous section. Related collections of data are illustrated with bold rectangles while the tools that operate on the data are illustrated with rounded rectangles. An arrows from a tool to a data collection indicates that the tool accesses the data collection. Multiple arrows pointing to a data collection indicate multiple tools integrated via the data collection.

Starting in the upper left of the diagram, the *UFO Language Editor* tool stores application programs created by developers in an *Application Driver Code* data collection. The *Static Semantic Analysis* tool associates *Static Semantic Structures* with the *Application Driver Code*. The *Static Semantic Analysis* tool is tightly coupled with the *UFO Language Editor* tool through the data that they share in the *Application Driver Code*.

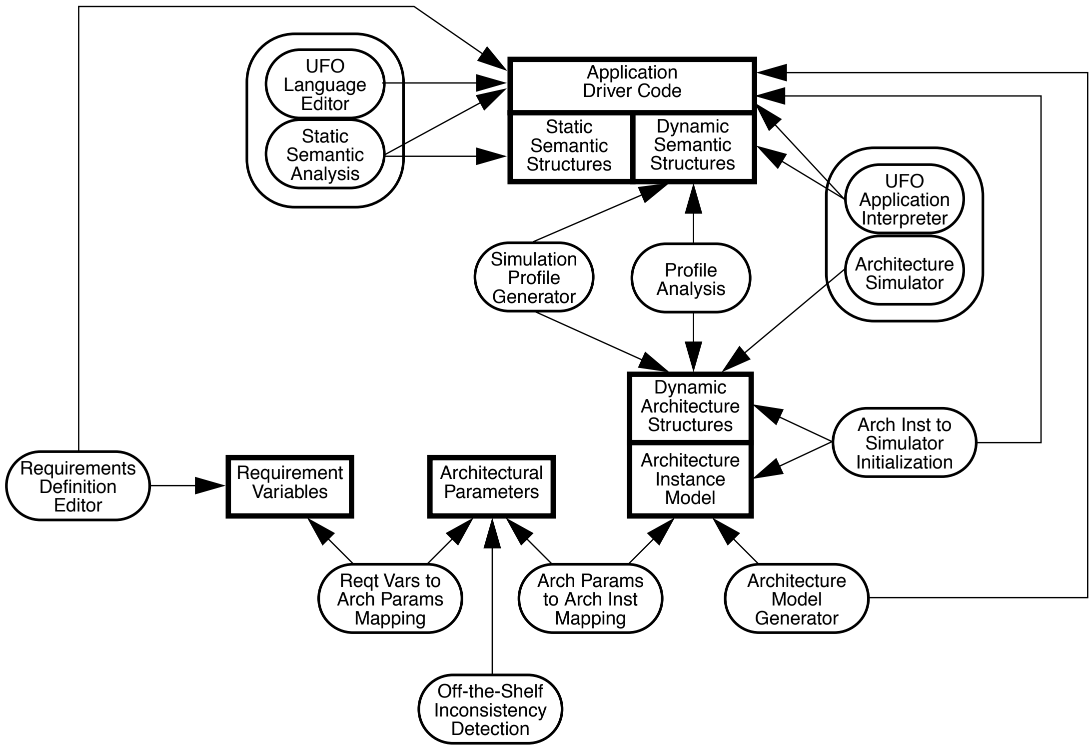


Figure 34. UFO's Integrated Data Model and Tools

At the lower left of the diagram, the *Requirements Definition Editor* tool is used by developers to create data for *Requirement Variables*. The tool also accesses the *Application Driver Code* to automatically extract certain requirement variable values.

The *Reqt Vars to Arch Params Mapping* tool maps *Requirement Variables* to *Architectural Parameters*. The *Off-the-Shelf Inconsistency Detection* tool accesses the *Architectural Parameters* to identify inconsistencies in the parameters relative to a particular off-the-shelf OODB.

The *Arch Params to Arch Inst Mapping* tool maps *Architectural Parameters* to an *Architecture Instance Model*. The *Architecture Model Generator* tool displays the *Architecture Instance Model* in a textual report format.

The *Arch Inst to Arch Realiz Mapping* tool augments an *Architecture Instance Model* with *Dynamic Architecture Structures* to create the data for an architecture realization that can be simulated.

In the upper right corner of the diagram, the *UFO Application Interpreter* tool accesses the *Application Driver Code* and the *Dynamic Semantic Structures* data to execute the application code. The *Architecture Simulator* tool, which is tightly coupled with the *UFO Application Interpreter*, will simulate the operation of the OODB architecture by accessing the *Dynamic Architecture Structures*.

After a simulation, the *Simulation Profile Generator* will generate a textual report from the simulation data stored in the *Dynamic Semantic Structures* on the *Application Driver Code* and the *Dynamic Architecture Structures*. The *Profile Analysis* tool accesses the *Dynamic Semantic Structures* on the *Application Driver Code* and the *Dynamic Architecture Structures* in order to identify and report potential optimizations that may be made by refining the requirement variable values.

In Sections 6.4. through 6.14. we discuss the implementation of each of these tools in more detail along with their associated data. However, before we begin those discussions, a bit more description is needed of the Gandalf implementation technology used to implement the tools and data structures.

6.3. Brief Introduction to the Gandalf System

Since its origin in the early 1980's, Gandalf[11] has traditionaly been referred to as a *software development environment generator*. Characterized in terms of popular commercial technology today, it might be described as an object-oriented database with a textual user interface generator for displaying and modifying the contents of the database. The applications targeted by the Gandalf System have typically been software development tools such as structure-oriented editors and software configuration management systems.

An application is implemented in Gandalf with three orthogonal pieces: *data management*, *data visualization*, and *computation*. The data management piece deals with the data type declarations for the application. The data visualization piece deals with displaying an editable, textual view of the data. The computation piece deals with application operations performed on the data.

In the following subsections we briefly outline these three major components of the Gandalf technology. Following that we describe how each of the UFO tools were implemented using the Gandalf data management, data visualization, and computation components.

6.3.1. Data Management

Data management in Gandalf is handled by an object-oriented database. The schema for a Gandalf database consists of declarations for *composite types*, *primitive types*, and *abstract types*. Composite type declarations describe complex objects that have other objects as sub-components. Primitive types describe primitive objects such as integer, strings, and constants. Abstract types provide polymorphism, where any one of a collection composite or primitive types can be instantiated at runtime for an abstract type.

Figure 35. shows an example Gandalf schema fragment, representing a `CASE` statement stored in the Gandalf database. The schema is divided into three sections, one for composite types, one for primitive types, and one for abstract types.

Composite Types:

```
CASE = <when>
WHEN = boolean_expression statement_list
STATEMENTS = <statement>
...
```

Primitive Types:

```
TRUE = {static}
FALSE = {static}
VARIABLE = {rep}
...
```

Abstract Types:

```
when = WHEN
boolean_expression = VARIABLE FUNCTION TRUE FALSE EQ AND OR NOT
statement_list = STATEMENTS
statement = ASSIGN PROCEDURE CASE LOOP
...
```

Figure 35. Gandalf Schema Example

There are three composite types in the example, `CASE`, `WHEN`, and `STATEMENTS`. The left-hand-side (LHS) of a composite declaration gives the name of the type. The right-hand-side (RHS) can have one of two forms, *variable arity* and *fixed arity.*

A variable arity RHS declares a sequence of zero or more objects of one abstract type. The `CASE` declaration is an example of a variable arity RHS, corresponding to a sequence of zero or more `when` objects. It has a single abstract type name enclosed in angle brackets.

A fixes arity RHS for a composite type declares a fixed-length tuple of objects, where each member of the tuple has a declared abstract type. The `WHEN` declaration is an example of a fixed arity RHS, corresponding to a two member tuple with a `boolean_expression` object and a `statement_list` object.

There are three primitive types in the example, `TRUE`, `FALSE`, and `VARIABLE`. The LHS of a primitive declaration gives the name of the type. The RHS can have one of two forms, *static* and *rep*. A static primitive denotes an object with a statically defined constant value. In the example, `TRUE` and `FALSE` are static primitives. A representation primitive denotes an object with a value that is set at runtime. In the example, VARIABLE is a rep primitive type, corresponding to the name of a variable.

There are four abstract types in the example, `when`, `boolean_expression`, `statement_list`, and `statement`. The LHS of an abstract type declaration gives the name of the type. The RHS is a list of composite types and primitive types that can be instantiated for the abstract type at runtime. Abstract types are only used on the RHS of composite types. In the example, the `boolean_expression` abstract type can be instantiated with either a `VARIABLE` object (a primitive type), a `FUNCTION` object (a composite type), a `TRUE` object (a primitive type), a `FALSE` object (a primitive type), an `EQ` object (a composite type), an `AND` object (a composite type), an `OR` object (a composite type), or a `NOT` object (a composite type).

6.3.2. Data Visualization

The Gandalf data visualization feature allows data objects in a Gandalf database to be displayed to users in different textual representations. This is analogous to report generators provided with many relational databases. Furthermore, Gandalf allows database objects to be modified, created, and deleted through the same textual interface.

Each composite and primitive type declaration in the schema language has a *view declaration* associated with it that specifies how to display the objects of that type. A view declaration contains a mixture of literal text and view commands. Literal text is displayed as is, while the view commands perform other text formatting and display operations. Figure 36. illustrates the view declarations for the `CASE` statement objects in the previous example. The figure also contains some sample text generated under this view declaration.

View Declarations

```
CASE = <when>
      views:
        (0) "case@+@n@0@n@q@-@nendcase"
WHEN = boolean_expression statement_list
      views:
        (0) "when (@1) then@+@n@2@-"
```

Example of Displayed Text

```
case
   when (command == "make") then
      ME.make( );
   when (command == "clear") then
      ME.clear( );
endcase
```

Figure 36. Gandalf View Declaration Example

Under the `WHEN` type declaration, for example, the view declaration is given as "`when (@1) then@+@n@2@-`". This string is interpreted as follows:

- "`when (`" is a literal string that is displayed as is
- "`@1`" is a view command to recursively display the first sub-object (the `boolean_expression` object in this example)
- "`) then`" is a literal string that is displayed as is
- "`@+@n`" is two view commands to (1) increase the level of indentation in the displayed text and to (2) insert a newline
- "`@2`" is a view command to recursively display the second sub-object (the `statement_list` object in this example)
- "`@-`" is a view command to increase the level of indentation starting at the next line of displayed text

The result of this view declaration can be seen in the example of displayed text in Figure 36.

Note that abstract types don't have view declarations since they are never directly instantiated as objects in the database, but rather simply define which composite and primitive object types can be created as sub-objects in a composite object.

6.3.3. Computation

Similar to other OODBs, the Gandalf OODB uses methods associated with database objects for computation. In Gandalf, methods are written in a language call ARL, which stands for Action Routine Language.

The one feature of ARL that is a notable distinction from other OODB languages is the way that it is coupled with the user interface. While most OODB languages have two predefined methods for each type of object, the constructor and destructor, ARL has twelve. In addition to a constructor and a destructor method, the other ten predefined methods are triggered by editing operations on the objects in the database through the user interface. Examples include when the user's cursor moves in or out of the displayed representation of an object, when an existing database object value is modified, and when an object is copied and pasted. Application implementors can define methods to be executed when any of these predefined methods are invoked.

Another way of initiating computation in Gandalf is with *extended commands*. These commands are accessible through a command menu in the user interface. Application implementors can define any number of extended commands for their application.

6.4. UFO Language Editor

Recall from Figure 33. on page 90 that the first step in using the UFO tool set is to develop the application prototype and simulation scripts using the Virtual OODB language. The UFO Language Editor is a syntax-directed editor for this task.

The language implemented in the syntax-directed editor is as described in Section 5.1.2., starting on page 56. In this and the other tool descriptions in this chapter, we will present implementation in terms of the three technical components of the Gandalf implementation technology: data, visualization, and computation.

6.4.1. Data

As illustrated in Figure 34., the UFO Language Editor tool reads and writes in the Application Driver Code data collection. Using Gandalf, the application driver code data is structured as an abstract syntax tree, similar to a parse tree representation of an application program.

Constructs in the UFO language were implemented with 56 composite types, 35 primitive types, and 49 abstract types. Figure 37. shows a sample of some of the typical types in the UFO schema for the application driver code.

Composite Types:

```
CLASS = class_name component_decls exported_routines local_routines constructor destructor
DESTRUCTOR = formal_parameters variable_decls statement_list
VARIABLE_DECL = variable_name type
```

Primitive Types:

```
CLASS_NAME = {rep}
INTEGER = {rep}
TRUE = {static}
```

Abstract Types:

```
lhs = VARIABLE_PUT COMPONENT_PUT RESULT ROOT_PUT
arithmetic_expression = VARIABLE_GET COMPONENT_GET PARAMETER_GET FUNCTION_APPLICATION INTEGER PLUS
                        MINUS TIMES DIVIDE MOD STR_TO_INT
```

Figure 37. UFO Schema Sample

6.4.2. Visualization

As described in Section 6.3.2., with the Gandalf technology the visual representation of the application driver code is generated from the application driver code data using a view declaration. The user never directly edits the program text, but rather fills in code templates to create underlying data, which in turn is displayed back through the view. Since the visual representation is generated from the abstract syntax tree data, the application driver code will always be syntactically correct (assuming that we write the view declarations correctly).

The UFO language editor simultaneously uses two views declarations to generate class definitions. One view is for the body of the class and the other is for the class interface. The class interface is generated from the same data as the class body, but the interface view simply displays the signature of the routines that are specified as “exported”. Therefore, whenever a developer creates a new exported routine, the signature becomes immediately visible in the class interface. With this approach, the tool eliminates the potential for inconsistencies between the interface and the body of a class definition.

6.4.3. Computation

Gandalf has built in auto-construction routines that will automatically create a nested object whenever the type of the nested object can be uniquely determined from the schema. So, for example, when a new UFO class object is created, Gandalf will automatically create nested constructor and destructor objects since the schema specifies that each UFO class has exactly one constructor and destructor component.

The UFO Language Editor does no computation or automation beyond what is automatically provided by Gandalf. In the next section, we describe another tool closely coupled with the language editor that implements extensive static semantic checking and automation on the application driver code data.

6.5. Static Semantic Analysis

The UFO Static Semantic Analysis tool provides the same type of static semantic error checking as a conventional compiler. Furthermore, the UFO language editor and the static semantic analysis tool are tightly coupled through shared data, so the static semantic analysis tool can help to automate some editing tasks and to prevent users from ever introducing static semantic errors in the application driver code.

6.5.1. Data

As shown in Figure 34., we extend the data representation of the application driver code with static semantic structures. The static semantic analysis tool accesses both the application driver code data and static semantic data structures in order to do error checking, and in some cases to prevent the editor from completing operations that would lead to an error in the application driver code.

The static semantic data structures include the following:

- Each object in the application driver code representation is given a *status* attribute that indicates whether the static semantic correctness of the object is currently *OK*, *not OK*, or *unknown*. This value changes as the application driver code is modified.

- Declaration sites and use sites are linked and maintained for the following named constructs in the UFO language:

    - Classes and Root classes
    - Class components
    - Functions, Procedures, and Boot procedures
    - Parameters
    - Local variables

We implemented the static semantic constructs for declaration/use site maintenance using 1 Gandalf composite type, 1 primitive type, and 1 abstract type.

6.5.2. Visualization

None of the data in the static semantic structures is presented to the user through a view. Static semantic errors that are detected by the tool are reported to the user through an error window.

6.5.3. Computation

The different type of static semantic computations that we do can be grouped into error detection, error prevention, and automated editing. These will be addressed in the following subsections. All of these static analysis computations are structured in a common framework that is used to trigger localized semantic analysis under well defined conditions editing.

6.5.3.1. The Static Semantic Analysis Framework

The static semantic analysis framework provides a uniform scheme for doing semantic checks and maintenance for all objects in the application code representation. This framework simplifies our job of writing static semantic checks by encapsulating with each UFO program construct a uniform method for checking and maintaining semantic consistency.

Each object has a *status* attribute and a *Check* method. When *Check* is called at the root of a subtree in the abstract syntax tree, it is called recursively to the child objects, and then to their child objects, until the leaves are reached. Then the semantic checks are made locally at each leaf object, the results propagated up one level to the parents of the leaves, and then the local semantic checks are made at each leaf parent. This continues until the semantic status propagates up to the subtree root where the semantic check was initiated.

For example, if the *Check* method is called on an arithmetic *plus* object, the *Check* method is recursively called on the left expression and the right expression. If the two expressions evaluated correctly, then the *plus* static semantic check is made to see if the left and right expressions have compatible types for an arithmetic *plus*. If the local type check is positive, then the status attribute of the *plus* object is set to OK.

The *status* attribute for each object can have one of three values:

- OK, meaning that local semantic checks have been done and no errors were detected
- NOTOK, meaning that local semantic checks have been done and errors were detected
- UNKNOWN, meaning that local semantic checks have not been done

When the status of any descendent of an object is NOTOK or UNKNOWN, then the status of that object is always UNKNOWN. Only when the status of all descendents of the object is OK are the local semantic checks done on an object. After the local semantic checks are done on an object, its status is set to either OK or NOTOK.

Static semantic analysis is automatically triggered by one of Gandalf's predefined methods. The `Exit` method is called whenever the user's editing cursor moves out of the displayed subtree for an object. Checking is triggered on `Exit` since this usually implies that the user is finished editing within the subtree of that object. We trigger the static semantic analysis from the `Exit` method for the following modular program units: `CLASS`, `ROOT_CLASS`, `CONSTRUCTOR`, `DESTRUCTOR`, `FUNCTION_DECL`, `PROCEDURE_DECL`, `PROGRAM_DECL`, and `PRELUDE`.

6.5.3.2. Static Semantic Error Detection

One form of static semantic analysis in UFO is error detection. Error detection simply reports static semantic errors to the user. However, unlike a compiler, the UFO's static semantic error detection is tightly coupled with the program editor so that errors can be detected and reported when the user makes the error.

The following types of error detection are implemented in UFO:

- Declaration site and Use site inconsistency for classes, functions, procedures, components, variables, and parameters. Errors are reported for items used but not declared and for items declared but not used.

- Name clashes within a scope for components, variables, and parameters. A declared name must be unique across all three of these in each scope.

- Type checking. For each typed object in the language, the types at use sites are checked for consistency with types at declaration sites.

- Signature checking. For each function and procedure in the language, the signatures (number and types of parameters and returns) at use sites are checked for consistency with signatures at declaration sites.

6.5.3.3. Static Semantic Error Prevention

Another form of static semantic analysis in UFO is error prevention. Similar to error detection, error prevention detects static semantic errors as they occur, but goes a step further in that it prevents the associated errors from going into the application driver code. This is only possible in cases where the source of the error can be uniquely determined. The following types of error prevention are implemented in UFO:

- Name clashes within a scope for:
  - Classes. Each class must have a unique name. An attempt to create a duplicate name will fail and produce an error message.
  - Functions and procedures. Within a class, functions and procedures must have unique names. An attempt to create a duplicate name will fail and produce an error message.
- Some UFO language constructs are only legal in certain contexts. Illegal constructions are blocked for:
  - the RESULT keyword used outside the scope of a function declaration
  - the ROOT keyword used outside the scope of a transient transaction
  - any persistent transaction construct used outside the scope of binding block or inside of a transient transaction block
  - any repository command used outside of the prelude or inside of a binding block
  - any function or procedure used in the prelude
  - a NEW operation outside the dynamic scope of a transient transaction
  - an AbortTransientTransaction operation outside the dynamic scope of a transient transaction
  - a BindingBlock construct outside of the prelude

6.5.3.4. Automated Editing Based on Static Semantic Analysis

Another form of static semantic analysis in UFO includes automatic modifications to the abstract syntax tree for the application driver code. This, of course, is only appropriate when the modifications can be unambiguously computed by static semantic analysis. UFO currently applies automatic editing in one simple case. The static semantic analysis tool automatically creates the constructor return type for a class from the class name whenever the class name is created or modified

6.5.3.5. Extended Commands

Static semantic checks in UFO can also be triggered manually from a menu by the user. There are two different forms of manual checks, implemented as Gandalf extended commands. The first is for triggering static semantic checks on a user highlighted portion of the program text. The second is for triggering static semantic checks from the root of the entire abstract syntax tree, forcing all errors to be reported.

6.6. UFO Language Interpreter

As developers create their application driver code, they can execute, test, and debug it using the *UFO Language Interpreter*. The interpreter fully implements the dynamic semantics of the UFO language as described in Section 5.1.2., but it does none of the OODB simulation and profiling. However, as we show later in this chapter, the interpreter also serves as the front end that drives the OODB simulator.

6.6.1. Data

Figure 34. illustrates that the interpreter operates on two data collections, the Application Driver Code and the Dynamic Semantic Structures. The dynamic semantic structures are extensions to the application driver code and static semantic structures that support execution of application programs. The dynamic semantic structures were implemented in Gandalf with 17 composite types, 22 primitive types, and 22 abstract types.

The data in this collection represents and implements UFO repositories, long and short transactions, the object memory heap, and the call stack. Some data structures from the static semantic structures, such as the declaration and use site links, span the boundary between static and dynamic structures. They are used by both the static semantic checker and the interpreter.

6.6.2. Visualization

None of the data in the dynamic semantic structures is presented to the user through a view. The interactions with the application user from the UFO language terminal I/O are handled by the interpreter with a command line window.

6.6.3. Computation

The interpreter tool is invoked by the user from the extended command menu. In keeping with our tight coupling between tools, the interpreter runs directly off of the application driver code representation produced by the language editor. Each object in the application program has a method call *Execute* that performs the local dynamic semantic computation, possibly invoking the Execute method on other objects as prescribed by the language semantics.

The execution begins at the object representing the language prelude. The Execute method on the prelude object creates and initializes call stack objects for the prelude variables and then calls Execute on the prelude statement list object. When that call returns, program execution is finished.

The prelude statement list object simply does the recursive invocation of Execute on each statement object in the list. One such statement might be a CASE statement. The CASE statement object sequentially iterates through the WHEN clause objects, and for each will invoke the Execute method on the boolean guard expression object. If, after evaluation, the expression has a value of TRUE, the statement for that WHEN clause will have its Execute method invoked and the CASE statement will terminate.

The UFO language construct, NEW, is a function that creates a new object and calls the user defined constructor for the class of the new object. When the Execute method is called on a NEW expression, a result pointer is allocated for the call stack. The static semantic data structures contain a pointer from each NEW expression back to the class for the new object. Using the class definition as a “template”, a new object is allocated and initialized from the object heap in the dynamic semantic data area. The constructor for that object and class is then triggered with the Execute method.

With UFO nested persistent transactions, there can be many different versions of an object due to modifications in different transactions. The conventional approach to implementing transactioned objects is to have a *lock table* associated with each transaction that keeps track to locked and modified object for the transaction. In the UFO implementation, however, we maintain all of the transactioned versions for an object within the object data structure. Therefore, given an object and the current transaction we can uniquely access the components for that object in the transaction. Using this approach, the transaction commit operation modifies the internal state of each modified object in a transaction rather than updating transaction lock tables.

6.7. Requirements Definition Editor

After developers create the application driver code, their next step is to use the UFO *Requirements Definition Editor* to define the requirements for the application, which is shown as task 2, *Define Requirements*, in Figure 33. on page 90. The requirements definition editor is implemented as a Gandalf syntax-directed editor.

The "language" implemented in the requirements definition editor is defined by a requirement variables abstract syntax. The language is viewed by the user of the requirements definition editor as a requirements definition *template*.

6.7.1. Data

As shown in Figure 34., the requirements definition editor accesses data from the Application Driver Code and the Requirement Variables data collections. Access to the application driver code, which is read only, allows the requirements definition editor tool to automatically derive some of the requirement variable values from the content of the application driver code.

The requirement variables data collection was implemented with 12 composite types, 22 primitive types, and 29 abstract types. The implementation of the requirement variables data follows directly from the design description in Section 5.1.8.

6.7.2. Visualization

The view declaration used by the requirements definition editor to display the requirement variables data is a *requirement variables template*. This template is shown in Figure 8. on page 35 prior to any requirement variables data being created, and in Figure 18. on page 47 after some example data has been created using the editor.

6.7.3. Computation

There is only one computation associated with UFO's requirements definition editor tool. This computation will scan the application driver code data collection in order to derive values for two requirement variable values. The computation is automatically invoked when a new requirement variable template is initialized with an extended command.

The two requirement variable values extracted are *Long Transactions*, which is a boolean value indicating whether or not long transaction support is required in the OODB architecture, and *Reentrant Partitions*, which is a call graph representation that is used to identify processes in the OODB architecture that are required to be reentrant.

The computation for determining whether or not long transactions are required is straightforward. The portion of the abstract syntax tree representing the application program prelude is scanned for any long transaction constructs, such as CreatePersistentTransaction. If none are found, then the Long Transaction requirement variable data value is created as FALSE, otherwise TRUE.

The computation for the reentrant partition call graph requirement variable creates a graph representing the method invocations among objects in different cluster partitions. Cycles are detected in the graph to indicate reentrant cluster sets (i.e., cluster sets that receive incoming method invocations while waiting for the completion of an outgoing method invocation).

6.8. Requirement Variable to Architectural Parameter Mapping

After developers create the requirement variable values, the next step is to use the UFO *Requirement Variable to Architectural Parameter Mapping* tool to define the architectural parameters for the OODB. This is shown as task 3, *Map Requirements to Architectural Parameters*, in Figure 33. on page 90. The mapping is implemented as a Gandalf extended command.

6.8.1. Data

As shown in Figure 34., the requirement variables to architectural parameters mapping accesses data from the Requirements Variables and Architectural Parameters data collections. Access to the requirement variables is read only. This data is used by the mapping to derive the architectural parameter values.

The implementation of the requirement variables data collection was described in the previous section. The architectural parameters data collection was implemented with 13 composite types, 31 primitive types, and 32 abstract types. The implementation of the architectural parameters data follows directly from the description in Section 5.1.6.

6.8.2. Visualization

The architectural parameters are typically not of interest to developers using UFO. However, we provide a simple view for the architectural parameters, primarily for the purposes of debugging. Figure 11. on page 39 shows an example of the architectural parameter data view.

6.8.3. Computation

The requirement variables to architectural parameters mapping is implemented as a Gandalf extended command and is therefore invoked by users from a menu. When the mapping is invoked, it first checks to see if the requirement variables data is complete. If so, the mapping begins a top down construction of the architectural parameters data.

6.9. Off-the-Shelf OODB Inconsistency Detection and Feedback

The OODB Inconsistency Detection and Feedback tool provides developers using UFO with information about differences between a set of architectural parameter values and an off-the-shelf OODB architecture. This is illustrated as task 4, *Detect Inconsistencies*, in Figure 33. on page 90.

6.9.1. Data

As shown in Figure 34., the inconsistency detection tool accesses data from the Architectural Parameters data collection. This access is read only. The architectural parameters are created by the requirement variables to architectural parameters mapping, described in the previous section.

6.9.2. Visualization

The inconsistency detection tool does not provide a view of the data to users, but rather inconsistencies that are detected by the tool are reported to the user through a message window. Section 4.3.1.1. on page 43 shows examples of inconsistency reports.

6.9.3. Computation

Through a Gandalf extended command menu, users select the off-the-shelf evaluation mode and the off-the-shelf OODB architecture of interest. Currently ObjectStoreLite, Objectivity, and ITASCA are the off-the-shelf OODBs supported. After the requirement variables to architectural parameters mapping is complete, the inconsistency detection and feedback tool is automatically invoked and any inconsistencies detected are reported to the user. The implemented inconsistency analysis is described in Section 5.2.4.

6.10. Architectural Parameter to Software Architecture Instance Mapping

After developers have mapped to architectural parameter values, their next step is to use the UFO *Architectural Parameter to Architecture Instance Mapping* tool to define the architecture instance description for the OODB. This is shown as task 5, *Map Architectural Parameters to Architecture Instance*, in Figure 33. on page 90. The mapping is implemented as a Gandalf extended command.

6.10.1. Data

As shown in Figure 34., the architectural parameters to architecture instance mapping accesses data from the Architectural Parameters and Architecture Instance Model data collections. Access to the architectural parameters is read only. This data is used by the mapping to derive the architecture instance data.

The architecture instance data collection was implemented with 38 composite types, 48 primitive types, and 63 abstract types. The implementation of the architecture instance data follows directly from the description in Section 5.1.5. Recall from that section that architecture instances are defined in terms of configuration nodes. Each configuration node is implemented as an object in the Gandalf abstract syntax tree.

6.10.2. Visualization

Visualization of the architecture instance data, produce by the mapping from architectural parameters to architecture instances, is described in the next tool section, *Static Modeling Feedback*.

6.10.3. Computation

The architectural parameter to architecture instance mapping is implemented as a Gandalf extended command and is therefore invoked by users from a menu. When the mapping is invoked, it first checks to see if the architectural parameter data is complete. If so, the mapping begins a top down construction of the configuration node data. The mapping is implemented according to the pseudo code description in Section 5.1.7.

6.11. OODB Architecture Model Generator

The OODB Architecture Model Generator tool provides developers using UFO with information about the structure and size of OODB architecture instances produced by the mapping from architectural parameters to architecture instances. This is illustrated as task 6, *Collect Static Properties*, in Figure 33. on page 90.

6.11.1. Data

As shown in Figure 34., the Architecture Model Generator accesses data from the Architecture Instance Model and Application Driver Code data collections. The configuration nodes in architecture instance model data representation are create by the architectural parameter to architecture instance mapping, described in the previous section. The configuration nodes have attributes that specify and accumulate size information about the architectural components represented by the configuration node. The architecture model generator tool reads and writes to these attributes in order to construct the static sizing model of an OODB architecture instance. The application driver code is scanned by the tool to produce executable code sizing estimates. The architecture size data was implemented according to the description in Section 5.2.1.2.

6.11.2. Visualization

The displayed representation of OODB architecture instances is done as two different Gandalf views. One view focuses on the architecture structure represented by the configuration nodes, while the other view focuses on the architecture sizing represented by the attribute data. Figure 19. on page 48 shows an example of the architecture structure view while Figure 18. on page 47 shows an example of the sizing view. The different views are selected via extended commands that are available in the UFO tool menu.

6.11.3. Computation

The computation associated with the architecture model generator tool calculates and accumulates the size attributes on the configuration nodes. The tool's algorithm is a bottom up traversal of the configuration nodes that calculates attribute values at each configuration node, based on local data and data from child configuration nodes. Each configuration node object has a method, ArchSummary, that performs the sizing computations for that object.

Size data falls generally into one of three categories:

- architectural component size. Local size contributions are stored as constants in the ArchSummary method code for the configuration nodes. Total architectural component size is calculated as local size plus size contributions from child configuration nodes.
- cache size. Cache size is initialized to predefined values in the configuration nodes and can be adjusted by developers if necessary.
- code space. Code space is calculated from the lines of code application driver code in classes allocated to cells in a particular configuration node.

6.12. Software Architecture Instance to Simulator Initialization

After developers have mapped to the architecture instance model, they can use the UFO *Architecture Instance to Simulator Initialization* tool to initialize the OODB simulator, which is shown as task 7, *Realize Architecture Instance*, in Figure 33. on page 90. This tool is implemented as a combined Gandalf extended command and syntax-directed editor.

6.12.1. Data

As shown in Figure 34., the architecture instance to simulator initialization tool accesses data from the Architecture Instance Model, Application Driver Code, and Dynamic Architecture Structures data collections. Access to the architecture instance and the application driver code is read only. This data is used by the mapping to derive the dynamic architecture structures data.

The dynamic architecture structures data serve three purposes, (1) definition of simulation constants, (2) relationships between object references and cells, and (3) simulation profile attributes. These dynamic architecture structures were implemented with 10 composite types, 10 primitive types, and 28 abstract types, plus attributes on many of the configuration nodes.

The simulation constants define computer system characteristics that will be used by the simulator, such as CPU clock speeds, disk access times, network speed, and inter-process communication costs. The specific data is defined in Section 5.2.2.2. Developers can interactively set and modify these data values prior to simulation.

The portion of the dynamic architecture structures data dealing with relationships between object references and cells is used by the simulator to help track when computation moves between clients and object servers in the architecture. Every time a method invocation is simulated between two objects, this collection of data is used to determine if the objects are in different cells and if the objects are on different processors.

Another portion of the dynamic architecture structures are the attributes for collecting simulation profile data. These attributes will be described in greater detail in the Architecture Simulator tool and Simulation Profile Generator tool sections, where they are initialized, modified, and displayed to the user.

6.12.2. Visualization

The simulation constants data is displayed to the user through a view in a Gandalf syntax-directed editor. Developers can edit and view the simulation constants using this editor. An example of the simulation constants view from the editor is shown in Figure 12. on page 40.

The object to cell relationship data is also displayed to the user through a view in a Gandalf editor. This tool creates "role declarations" for each application object type allocated to a cell. Role declarations associate the destination of an object pointer component in a class with a cell. Developers use the editor to enter a cell and role destination for each object reference component in each role declaration.

6.12.3. Computation

The only computation associated with the architecture instance to simulator initialization tool is the automated initialization of the role declarations in the object to cell relationship data. This part of the tool scans through each cell declared in each client and object server, and for each application class allocated to a cell creates a *role declaration*. The role declarations are modeled after the class declarations in the application driver code, and are created by scanning the application driver code data.

6.13. UFO Architecture Simulator

With the application driver code, the OODB architecture instance, and the initialized simulation structures in place, developers use the *UFO Architecture Simulator* to explore the dynamic properties of the architecture. The simulator fully implements the dynamic semantics of the UFO reference architecture as described in Section 5.1.3. Design of the simulator is described in Section 5.2.2. The simulator is implemented as an extension to the UFO language interpreter, described previously in Section 6.6.

6.13.1. Data

Figure 34. illustrates that the simulator, in conjunction with the interpreter, operates on three data collections, the Application Driver Code, the Dynamic Semantic Structures, and the Dynamic Architecture Structures. These three data collections, all described in previous tool sections, are the culmination of the developers' efforts to configure an optimal OODB architecture for the application. During execution, the simulator tool will accumulate performance profile data in the dynamic semantic structures and dynamic architecture structures that will help developers evaluate the OODB architecture. The architecture performance data is implemented according to the description in Section 5.2.3.

6.13.2. Visualization

None of the data used during simulation is presented to the user through a view. The next tool section describes how the performance profile data is fed back to users after simulation.

6.13.3. Computation

The simulator tool is invoked by the user from the extended command menu. The simulator is an extension to the UFO language interpreter that not only interprets the language semantics, but also interacts with the OODB architecture instance to simulate the architecture operation. In the interpreter are simulation "hooks" that are treated as stubs during interpretation but drive the architecture during simulation.

As a simple example of simulator execution, when the interpreter executes the UFO CreateRepository command, a call is made to SimCreateRepository. During interpretation, this call is treated as a stub that returns immediately, but during simulation, the call will increment a counter attribute on the repository manager configuration node.

For a more complex example, consider the following simulation scenario for assigning to a component value in an OODB object.

1. The UFO interpreter calls Execute on Component_Put language construct. This method implements the language semantics and then calls SimPutComponent to simulate the OODB architecture operations.
2. SimPutComponent increments the component write counts at the component declaration and component use sites. It then calls AccessCell, which checks to see if the object is marked as active.
3. The object is currently not active, so ActivateCell is called. ActivateCell checks to see if cache can accommodate another cell. The cache is currently full, so ActivateCell replaces least recently used cells until there is enough space for the new cell.
4. As each least recently used cell is replaced in the cache, the cell is marked as inactive and the cache replacement count is incremented on the cache configuration node. If a replaced cell is dirty, then PassivateCell is called to simulate the write back to persistent storage.
5. PassivateCell will increment the cell passivation count for the cell type, increment the object passivation count by the number of objects currently in the cell, calculate the cell size, and update the maximum cell size attribute on the cell type declaration is the passivated cell size is larger than the current maximum.
6. ActivateCell then marks the new cell as active and increments the cell activation count and object activation count.
7. Finally, SimPutComponent marks the cell as dirty to indicate the component modification in that cell.

6.14. Simulation Profile Generator

The OODB Simulation Profile Generator tool provides developers with profile information on a simulation run. This is illustrated as task 9, *Collect Dynamic Properties*, in Figure 33. on page 90.

6.14.1. Data

As shown in Figure 34., the Simulation Profile Generator accesses data from the Dynamic Architecture Structures and the Dynamic Semantic Structures data collections, both described in previous sections. The simulation profile generator tool reads these attributes, performs calculations based on simulation constants, and accumulates results in other attributes in order to construct the simulation profile data. The architecture performance data was implemented according to the description in Section 5.2.3.

6.14.2. Visualization

The displayed representation of the simulated OODB architecture profile is done as a Gandalf view. After the profile attributes have been fully derived, the view displays the profile attributes on the configuration nodes in order to show the performance profile of the architecture instance and displays the profile attributes on the dynamic semantic structures on the application driver code in order to trace the performance loads back to their source in the application. Figure 14. through Figure 17. on page 42 through page 46 show examples of the architecture profile report view.

6.14.3. Computation

During simulation, only simple counts are collected at the low level configuration nodes in the architecture instance. This helps to simplify the simulator implementation and enhance performance. It is only after simulation and when a user requests a profile report are the counts accumulated, the higher level profile data calculated, and the report generated.

The tool's algorithm is a bottom up traversal of the data structures. Attribute values at each level of the architecture are calculated, based on local data, data from sub-components, and the simulation constants. Each data object that is part of the profile computation has a method, ProfileDigest, that performs the accumulation and computations for that object.

For example, during simulation, every time that a cell in a client is activated, a counter on the appropriate cell declaration is incremented by one and another counter on the cell declaration is incremented by the number of object in the cell at that time. After simulation, when the ProfileDigest method is call on the cell declaration object, the following calculation is performed:

- the number of cell activations is multiplied by the cell activation cost from the simulation constants
- the number of object activations is multiplied by the object activation cost from the simulation constants
- the two activation cost values are added together and divided by the cpu speed from the simulation constants
- the result is stored in an activation time attribute

When the ProfileDigest method is called on the client object, the method will access each cell declaration configuration node allocated to the client to accumulate the total cell activations, the total object activations, and the total activation time. These results are all store in attributes on the client object. Results are similarly accumulated at higher and higher levels in the architecture until the overall costs for the architecture are accumulated. The results at all levels are displayed in the profile report.

The simulation profile generator is invoked by the user from the extended command menu. The tool first performs the computation and accumulation of the profile data, and then changes to the profile report view.

Accumulated and calculated profile data falls generally into one of the following categories. See Section 5.2.3.1. for details on the simulation profile data design.

- computations of execution times accounted for in the various subsystems and overall
- counts and times accumulated for intra-cell, inter-cell, and inter-process method calls
- counts and times accumulated for object and cell creations, activactions, and deactivations
- maximum dynamic cell size for all cells
- counts and times accumulated for transaction creates, commits, and aborts
- counts accumulated for repository creates, deletes, and bindings

6.15. Automated Profile Analysis and Feedback

The Automated Profile Analysis and Feedback tool provides developers using UFO with information about potential architectural optimizations based on information in a simulation profile. This is an aid to manual analysis of the simulation profile reports, generated as described in the previous section. This is illustrated as task 9, *Collect Dynamic Properties* and the subsequent *Feedback on System Properties* in Figure 33. on page 90.

6.15.1. Data

As shown in Figure 34., the profile analysis tool accesses data from the Dynamic Architecture Structures and the Dynamic Semantic Structures data collections. This access is read only and occurs after simulation and after the simulation profile generator tool has accumulated and calculated all of the profile report data.

6.15.2. Visualization

The profile analysis tool does not provide a view of the data to users, but rather suggestions that are determined appropriate by the tool are reported to the user through a message window. Section 7.2.3.1. on page 146 shows an example of a suggested refinement to OODB requirements, generated as feedback to the user by the profile analysis tool.

6.15.3. Computation

The profile analysis tool is automatically invoked from the simulation profile tool after a simulation profile report has been generated. The profile analysis supported by the tool is described in Section 5.2.3.1.

The tool's algorithm is a bottom up traversal of the configuration nodes in the architecture instance being simulated. Each configuration node has a method, Feedback, that performs the profile analysis for the profile data in that configuration node to make a local determination if any suggested refinements are appropriate.

6.16. Boundaries

The UFO tool implementation closely follows the design outlined in Chapter 5. The one exception is that the implementation simulates multiple users interactions sequentially rather than concurrently. Since the UFO model of concurrency uses serializable transactions, simulating with serialized user interactions rather than concurrent interactions does not change the operational semantics of the UFO language. However, two things that will not be reflected with serialized simulations are (1) concurrency conflicts and (2) performance degradation due to excessive numbers of concurrent users.

Our reason for not implementing concurrency in the UFO simulator was simply that Gandalf does not support concurrent execution threads. We felt that the complexity and level of effort required to manually implement the concurrent simulation threads was not justified by the limited benefits, identified in the previous paragraph, of having simulated concurrency.

Chapter 7. Experiments and Results

In this chapter we describe the experiments that we performed to determine how well our modeling and simulation approach and the UFO tool support our hypothesis. In the experiments we estimate the effort associated with using UFO to define and refining OODB requirements and to evaluate and select off-the-shelf OODBs. We also estimate how well the resulting OODB architectures conform to the application requirements.

We used the UFO tool for experiments on two different OODB-based applications. The first application is the *workflow diagram editor* described in the case study in Chapter 4. This application requires a single-user OODB that is optimized for long editing activities on one or more clusters of data objects. The second application is a *workflow management system* that uses that workflow diagrams produced with the editor to help monitor progress on projects. The second application requires a multi-user OODB that is optimized for a high throughput of short updates on small clusters of objects. These experimental applications are based on experiences developing similar commercial products using OODB technology.

As noted above, the applications store similar data structures in an OODB but have very different OODB requirements and correspondingly different architectures. The different access patterns and user expectations associated with the two applications lead to different OODB requirements and different OODB architectures. The first application requires a client-based OODB, where object clusters are moved to the client machine for access to the objects. The second application requires a server-based OODB where object clusters reside on a collection of servers that indirectly provide object access to a large number of concurrent clients.

As we will see from these experiments, the UFO tool allows us to accurately define requirements and select off-the-shelf OODBs at a fraction of conventional costs.

7.1. Experiment 1: An OODB for a Workflow Diagram Editor

The description of the first experiment is organized into subsections with the following order and content:

- Conceptual view of application
- Design of the classes
- Design and implementation of simulation scripts to drive the experiments with typical scenarios
- Use of the UFO tool to prototype requirements. Use of resulting modeling and simulation data to refine requirements.
- Use of the UFO tool to evaluate off-the-shelf OODBs. Feedback, modeling, and simulation data to compare and select an OODB for the application.
- Modeling and simulation data to illustrate conformance. Estimates on the development costs. Resulting cost/conformance profiles.

The concepts in the workflow diagram editor application are illustrated in Figure 38. The *Primitive Task* is a defined unit of work within a project. For example, shingling the roof might be defined as a primitive task in building a house. A *Resource* is a person or group of people assigned to carry our a primitive task. For example, a roofing company might be assigned to carry out the task of shingling a roof. Multiple primitive tasks can be grouped together to form a larger unit of work called a *Composite Task*. For example, a composite task for roofing a house might consist of primitive tasks for (1) framing the roof, (2) putting on the deck, (3) putting up the flashing, (4) applying the felt, and (5) shingling. Primitive tasks can both produce and use *Products*. In particular, products produced by one primitive task can be used by subsequent tasks, such as the roof frame being used to apply the roof deck.

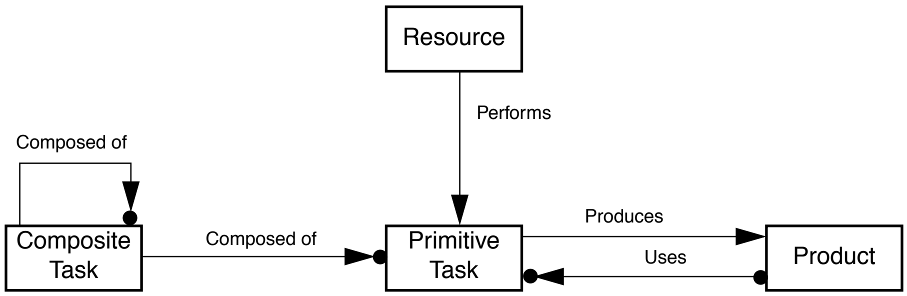


Figure 38. Entity-Relationship Diagram of the Workflow Editor Application

7.1.1. The Classes in the Application Driver Code

The workflow diagram editor concepts were implemented using eleven classes in the UFO language. This provided a realistic yet manageable size of application for simulation and experimentation – approximately 1800 lines of UFO source code.

The classes and their inter-relationships are illustrated in Figure 39. and are described in the following eleven paragraphs. Only the class name and Component parts of each class are listed. Recall that the Components part is the private data for object instances of the class. All of these classes are for persistent objects that are stored in the OODB.

RootClass VPML_Root. This class is for the root object in a workflow diagram. The graph component points to the outermost composite task in the data structure. The other components are all for maintaining the global editing context for the editor.

```
RootClass VPML_Root
...
   Components
      graph: CompositeTask;
      composite_context: CompositeTask;
      current_comp: CompositeTask;
      current_prim: PrimitiveTask;
      marked_prod: Product;
```

Class CompositeTask. This class is for the composite task entity illustrated in Figure 57. The *name* component is the name for the composite task. The *description* is used to store the textual description of what the task is to accomplish. The *composites* and *primitives* both point to the head of a linked list of the nested composite tasks and primitive tasks within the composite task. The *parent* points to the composite task that contains the composite task, while the *sibling* points to the next composite task in the linked list of composite tasks in the parent.

```
Class CompositeTask
...
   Components
      name: STRING;
      description: STRING;
      composites: CompositeTask;
      primitives: PrimitiveTask;
      sibling: CompositeTask;
      parent: CompositeTask;
```

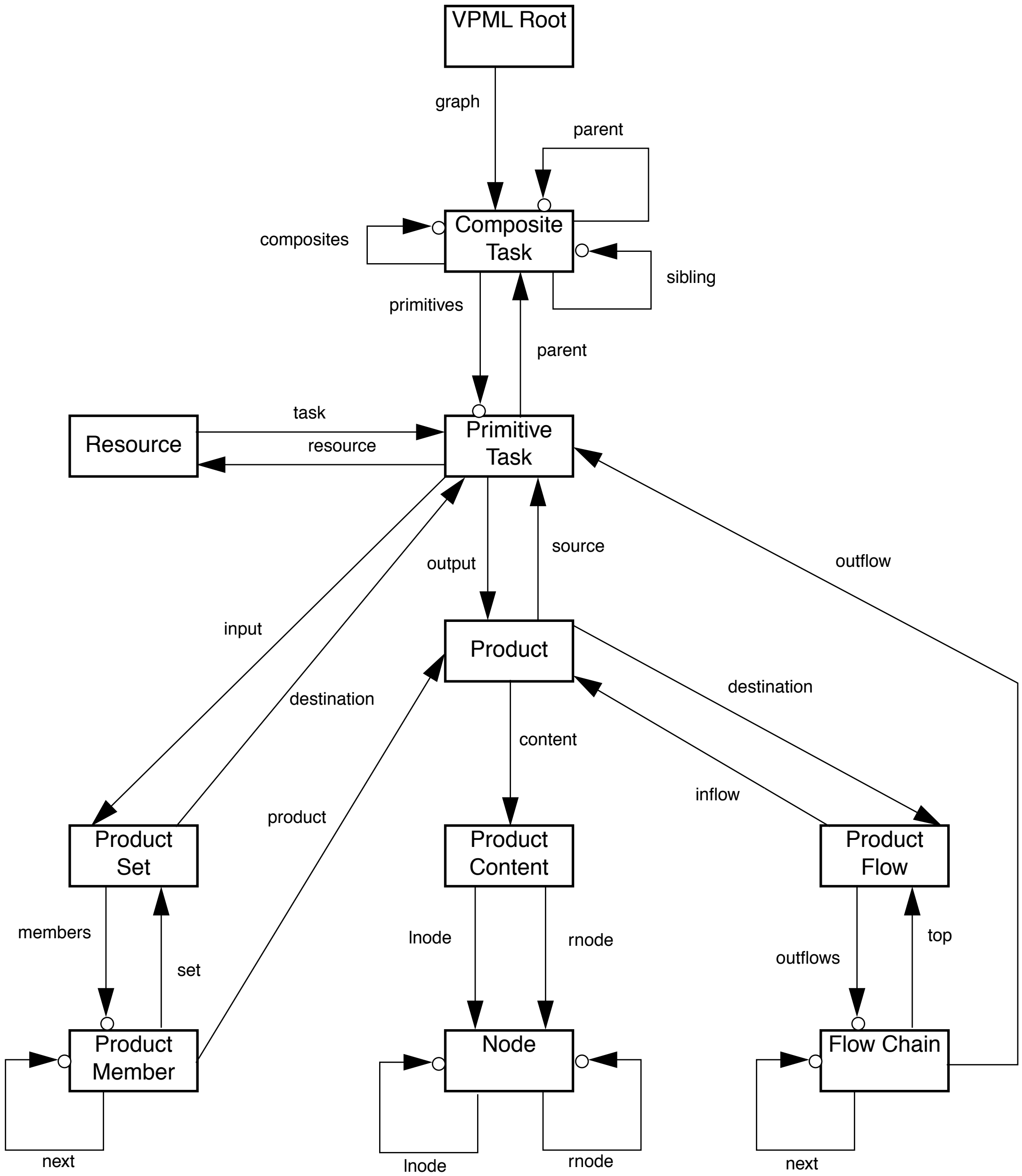


Figure 39. Class Diagram of the Workflow Editor Application

Class PrimitiveTask. This class is for the primitive task entity illustrated in Figure 57. The *name* component is the name for the primitive task. The *description* is used to store the textual description of what the task is to accomplish. The *parent* points to the composite task that contains the primitive task, while the *sibling* points to the next primitive task in the linked list of primitive tasks in the parent. The *resource* component points to the resource entity in Figure 57. The *product* component points to the product entity produced by the task, while the *input* component points to the data structure that maintains the set of products that serve as inputs to the task.

```
Class PrimitiveTask
   ...
   Components
      name: STRING;
      description: STRING;
      parent: CompositeTask;
      resource: Resource;
      output: Product;
      input: ProductSet;
      sibling: PrimitiveTask;
```

Class Resource. This class is for the resource entity illustrated in Figure 57. The *name* component is the name for the resource. The *description* is used to store the textual description of the type of resource needed to accomplish the associated task. The *task* points to the primitive task that the resource is to accomplish.

```
Class Resource
   ...
   Components
      name: STRING;
      description: STRING;
      task: PrimitiveTask;
```

Class Product. This class is for the product entity illustrated in Figure 57. The *name* component is the name for the product. The *description* is used to store the textual description of the product. The *source component* points to the primitive task that produces the product. The *destination* component points to a data structure that maintains the list of primitive tasks for which the product serves as an input.

```
Class Product
   ...
   Components
      name: STRING;
      content: ProductContent;
      description: STRING;
      source: PrimitiveTask;
      destination: ProductFlow;
```

Class ProductSet. This class serves as the anchor for a linked list that maintains a set of products that serve as input to a primitive task. The *destination* component points to the primitive task. The *members* component points to the head of the linked list, which will be an object of type *ProductMember.*

```
Class ProductSet
   ...
   Components
      destination: PrimitiveTask;
      members: ProductMember;
```

Class ProductMember. This class serves as the elements in the linked list for the *ProductSet* (see previous section). The *set* component points back to the *ProductSet* that anchors the linked list. The *next* component points next element in the linked list. The *product* points to a product in the set.

```
Class ProductMember
   ...
   Components
      set: ProductSet;
      next: ProductMember;
      product: Product;
```

Class ProductFlow. This class and the *FlowChain* class are the converse of the *ProductSet* and the *Product-Member* – the *ProductFlow* represents the set of *Uses* dataflows (see Figure 38.) from a product to primitive tasks. The *inflow* component points to the product. The *destination* points to the head of the linked list of *FlowChain* elements that connect to the primitive tasks.

```
Class ProductFlow
   ...
   Components
      inflow: Product;
      outflows: FlowChain;
```

Class FlowChain. This class serves as the elements in the linked list for the *ProductFlow* (see previous section). The *top* component points back to the *ProductFlow* that anchors the linked list. The *next* component points next element in the linked list. The *outflow* points to a primitive task in the set.

```
Class FlowChain
   ...
   Components
      top: ProductFlow;
      next: FlowChain;
      outflow: PrimitiveTask;
```

Class ProductContent. This class serves as the root for dummy content in a product during simulation. In a real editor, this content might be HTML, ASCII, or other types of data. In our simulation, the product content is simply a binary tree of *Nodes*.

```
Class ProductContent
   ...
   Components
      left: Node;
      right: Node;
```

Class Node. This class is for the nodes in binary tree the dummy product content. The *height* component simply represents the point where real content could be stored.

```
Class Node
   ...
   Components
      height: INTEGER;
      left: Node;
      right: Node;
```

7.1.2. The Simulation Scripts

After the application classes are implemented, the next step is for developers to create the simulation scripts. Of course, in this case we play the role of the developers and create the simulation scripts. These scripts are used later in the experiment to simulate typical scenarios for using the application. The scripts provide a repeatable means of comparing and contrasting the same simulation of the workflow diagram editor running on different OODB architectures.

The simulation scripts we created for the experiment simulate a user creating and browsing a workflow diagram. The workflow diagram produced by these scripts has 5 top-level composite tasks, 24 primitive tasks, 24 products with content, 24 resources, and 34 product flow connections. This provided a practical yet manageable sized simulation for the experiments (about an hour for the simulation on an architecture instantiation).

This particular workflow diagram describes a business approach for identifying and deploying new software products. The graphical representation is shown in Figure 40. The large rounded rectangles are composite tasks, the ovals are primitive tasks, the rectangles are products, and the underlined strings are resources. An arrow going from a primitive task to a product indicates that the product is produced by the task. An arrow going from a product to a primitive task indicates that the product is used to carry out the task. Details on the semantics of the workflow diagram, including the labels and other notations on the arrows, will be given in the next experimental description, where the workflow diagram is used to help manage the ongoing work on a project.

The simulation scripts used for this experiment are simply method invocations into the user interface command dispatcher. That is, they invoke the same methods in the application as a user interacting through the user interface of the application. There are 350 method invocations in the scripts. Following is an excerpt that creates the first few workflow constructs illustrated in Figure 40.

```
ME.NewComposite("Define Markets"; "Identify and analyze target markets");
ME.EnterComposite( );
ME.NewPrimitive("Identify Sub-market"; "Find a market niche");
ME.NewResource("Market Analyst"; "Has knowledge of market domain and potential");
ME.NewProduct("Sub-market Description"; "Description of sub-market area"; 4; FALSE);
ME.DisplayProduct( );
ME.SetMark( );
ME.NewPrimitive("Analyze Sub-market"; "Study and characterize sub-market");
ME.DataFlow(""; 0; FALSE);
```

7.1.3. Prototyping OODB Requirements for the Workflow Diagram Editor

With the workflow application code and the simulation scripts in place, we then prototyped the OODB requirements in order to converge on the set of OODB requirements that best conform to the workflow management editor application. These requirements will serve as a baseline in the next part of the experiment where we attempt to identify an off-the-shelf OODB that closely matches the requirements.

Using the modeling and simulation features in the UFO tool, the requirements prototyping activity consists of the following steps:

1. manual and automated requirements definition
2. mapping from requirements to an executable architecture simulation
3. running the simulation to get feedback on the system properties of the architecture
4. if necessary, cycling back to step 1 to refine the requirements based on the simulation feedback

During this portion of the experiment, we demonstrate the following features of the UFO tool that are intended to reduce cost and increase conformance:

- automated definition of some requirement variable values, accomplished by scanning the application source code
- guidance for manually defining the remaining requirement variable values

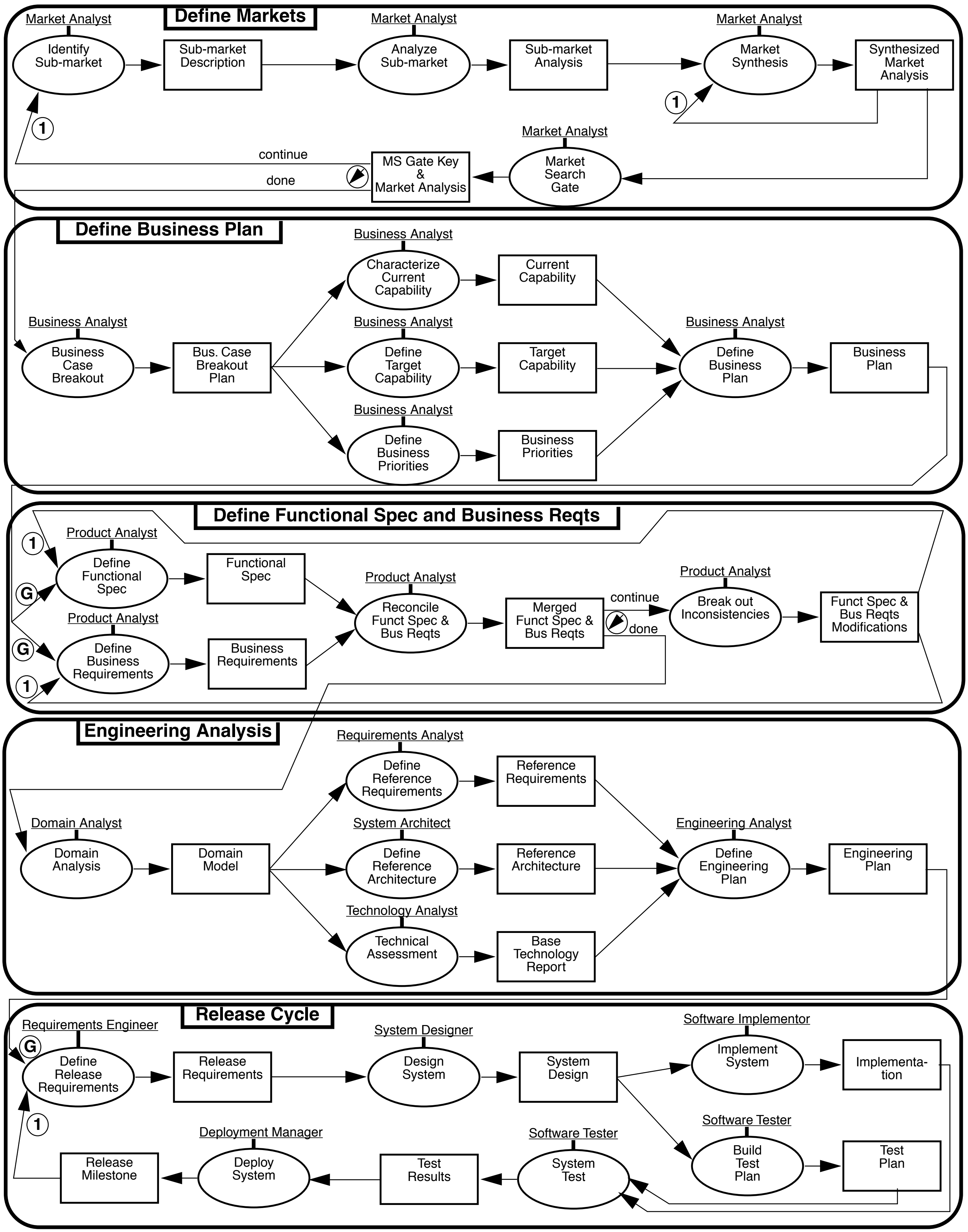


Figure 40. Workflow Diagram used for Experiments

Section 4.2. from the Case Study describes a scenario similar to the one used on this portion of the experiment. Therefore, we assume that the reader is sufficiently familiar with the UFO tool scenarios and we do not repeat the tutorial-level description, but rather present the specifics of the experimental procedures and results.

The first step is to define OODB requirements for the workflow application using the requirement variable template. The initial form of the template is shown in Figure 41.

```
Long Transactions: FALSE

Concurrent Users: $multi_user_rv

Locality Clusters:
   $locality_cluster_rv

Sparse Cluster Sets
   $cluster_set_rv

Dense $dense_cluster_set_rv

Cluster Set Partitions
   $cluster_set_partition_rv

Global Transaction Throughput: $global_transaction_throughput_rv

Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

Figure 41. Requirement Variables Template

The UFO tool first makes a pass over the workflow editor application source code to automatically derive values for two of the requirement variables, *Long Transactions* and *Reentrant Partitions*. As described in the UFO tool implementation chapter, Chapter 6., the long transactions value is derived by searching for any *CreatePersistentTransaction* constructs in the source code (the only way that nested long transactions can be created in the application). Since there are none in the workflow editor, the value is set to FALSE.

The value for the reentrant cluster set variable is an internal representation that is derived from the static method call graph among the workflow application classes. Cycle detection is used to determine whether or not reentrant processes are required in cases where the architecture is partitioned into separate processes.

This automated derivation of requirement variable values impacts both the development cost and conformance to requirements. First, the automation removes the developers' derivation cost for these requirement variable values. Second, the automation removes the opportunity for developers to introduce errors (and therefore lower conformance) in these requirement variable values.

The remaining values in the template cannot be automatically derived and must be manually entered. The UFO tool guides the definition of the requirement variable values in the following ways:

- the template-based tool clearly indicates the values that are needed, which is particularly useful for the complex hierarchical requirements

- context-sensitive help in the UFO tool describes the legal values for each context in the template

In addition, a users' manual could be written to provide guidelines for selecting the appropriate values for requirement variables.

Figure 42. shows the requirement variables template with all remaining values defined. These values were defined by us, playing the role of developers on the workflow diagram editor application. Selecting appropriate values required knowledge of the workflow diagram editing domain, the expected use profiles, and the user expectations on the system.

```
REQUIREMENT VARIABLES

        Long Transactions: FALSE

        Concurrent Users: FALSE

        Locality Clusters:
           Cluster: Root
              Roles:
                 Role root_root is Class VPML_Root
              Utilization: HIGH
              Size: SMALL
              Primary Reliability:
                 HIGH
              Secondary Reliability:
                 NONE

           Cluster: Composite
              Roles:
                 Role comp_root is Class CompositeTask
                 Role prim is Class PrimitiveTask
                 Role res is Class Resource
                 Role prod is Class Product
                 Role pset is Class ProductSet
                 Role pmem is Class ProductMember
                 Role pflo is Class ProductFlow
                 Role fchn is Class FlowChain
              Utilization: HIGH
              Size: SMALL
              Primary Reliability:
                 HIGH
              Secondary Reliability:
                 NONE

           Cluster: PContent
              Roles:
                 Role pcont_root is Class ProductContent
                 Role lnode is Class Node
                 Role rnode is Class Node
              Utilization: HIGH
              Size: SMALL
              Primary Reliability:
                 HIGH
              Secondary Reliability:
                 NONE


        <No Sparse Cluster Sets>

        Dense Cluster Set DCS
           Clusters:
              Root
              Composite
              PContent
           Global Cluster Set Contention: LOW


        Cluster Set Partitions
           Partition CSP
              Cluster Sets:
                 DCS
              Global Partition Contention: LOW


        Global Transaction Throughput: LOW

        Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

Figure 42. Requirement Variable Values for the Workflow Diagram Editor

The value for *Concurrent Users* is set to *FALSE* since the workflow diagram editor is not intended to support concurrent users editing a single workflow diagram.

Three *Locality Clusters* are identified, *Root*, *Composite*, and *PContent*. These particular clusters were selected based on the potential for access locality during workflow diagram editing on the objects in each cluster.

The *Root* cluster contains a single object of class *VPML_Root* (Section 7.1.1.). This object plays the role of the root of the entire persistent data structure for a workflow diagram and therefore is always the first object accessed when a new workflow data repository is opened. Editing commands are dispatched as methods on this object, so it makes sense to have it as a stand-alone object in the root cluster.

The *Composite* cluster contains a single *CompositeTask* object plus all of the objects that are logically contained within that *CompositeTask*. These contained objects will be from classes *PrimitiveTask*, *Resource*, *Product*, *ProductSet*, *ProductMember*, *ProductFlow*, and *FlowChain*. For example, referring to Figure 40., the workflow repository created and used in this experiment would have one cluster for each of the five composite tasks shown (“Define Markets”, “Define Business Plan”, and so forth), where each of the *Composite* clusters would contain all of the objects illustrated inside of a composite task. The workflow diagram editor displays the content of a single composite at a time for editing. This inherent editing locality motivates the definition of the *Composite* cluster.

The third cluster type, PContent, is used for a ProductContent object and all Node object that represent the descriptive content of a product in a workflow diagram. Similar to the Composite cluster, the PContent cluster represents a logical locality for the workflow diagram editor – the editor displays the content of a single ProductContent at one time for editing.

Each of the three clusters are declared to be HIGH utilization (greater than 50%), SMALL size (less than 1 megabyte), HIGH primary reliability (reliable primary persistence), and NONE for secondary reliability (no automated backups).

Since all three of the clusters are declared to be HIGH utilization and SMALL size, none of the clusters go into Sparse Cluster Sets. This is indicated by the <*No Sparse Cluster Sets*> value for that portion of the requirement variables.

Conversely, all of the cluster go into the *Dense Cluster Set* requirement variable. The *Global Cluster Set Contention* is set to LOW on the *Dense Cluster Set* since there is no contention in a single user application.

Since there is only a single dense cluster set, the *Cluster Set Partitions* requirement variable contains only a single partition labeled CSP to contain the dense cluster set DCS.

Finally, the value for *Global Transaction Throughput* requirement variable is LOW since this is a single user application.

The development cost associated defining requirements in this experiment included the application requirements analysis cost and the cost of reflecting the results of the analysis in the requirements template. The majority of the effort was in analysis of how the application would be used, so that we could define the Locality Clusters. Even this was straightforward since the editing locality for this application implied natural locality clusters for the requirements. The other requirement variable values could be easily determined knowing that this is a single user application with an interactive editing interface. We allocate an estimated one half day of development cost to the requirements analysis activity. The cost of entering the information in the requirement variables template tool was negligible.

The experiment next moved to mapping the requirement variables to architectural parameters. This phase is fully automated by the UFO tool. Therefore, there are no development costs other then issuing the command to the tool.

Mapping architectural parameters to a software architecture instance is likewise automated by the UFO tool. The architectural parameters are used to instantiate the data structures for modeling the OODB architecture in the UFO simulator.

After the mapping, the current implementation of the UFO tool requires that the components in each class be annotated to indicate which type of object in which type of locality cluster it refers to. These annotations provide hints to the OODB runtime as to when to create new clusters and when pointer dereferences may cross cluster or processor boundaries. Sample annotations are shown in Figure 43. The right arrow notation, "=>", indicates that the component on the left points at runtime to the object role on the right. The manual annotation activity consists of filling in the cluster and object role on the right of the arrow. This activity (which could be partially automated by extensions to the current tool) took approximately 10 minutes for this experiment.

```
Cell: Composite
   ...

   Role prim is Class PrimitiveTask
   name: STRING
   description: STRING
   parent: CompositeTask => Composite.comp_root
   resource: Resource => Composite.res
   output: Product => Composite.prod
   input: ProductSet => Composite.pset
   sibling: PrimitiveTask => Composite.prim
   state: CHARACTER
```

Figure 43. Component-to-Role Declaration

The final mapping from architectural instance to executable simulation is also automated by the tool. The tool provides the developer with a template to define the simulation constants for the simulated CPUs that the OODB architecture will run on. The constants used for this experiment are shown in Figure 44.

```
Non-reentrant Client:
...
   Processor Constants:
      Processor Speed:   33 MHz
      Intra-Cell Call Cost:    In: 50 cycles,   Out: 50 cycles
      Inter-Cell Call Cost:    In: 100 cycles,   Out: 50 cycles
      RPC Call Cost:           In: 550 cycles,   Out: 550 cycles
      Object Activation Cost:  100 cycles
      Object Passivation Cost: 100 cycles
      Object Creation Cost:    500 cycles
      Cell Activation Cost:    1000 cycles
      Cell Passivation Cost:   1000 cycles
      Cell Creation Cost:      600 cycles
      Cell Lock Check Cost:    50 cycles
...
Persistent Cell Manager. CSP
...
  Persistent Cell Manager Constants:
      Processor Speed:          66 MHz
      Object Activation Cost:   200 cycles
      Object Passivation Cost:  200 cycles
      Object Creation Cost:     1000 cycles
      Object Commit Cost:       200 cycles
      Object Abort Cost:        10 cycles
      Cell Activation Cost:     10000 cycles
      Cell Passivation Cost:    10000 cycles
      Cell Creation Cost:       15000 cycles
      Cell Commit Cost:         10000 cycles
      Cell Abort Cost:          5000 cycles
      Cell Lock/Unlock Cost:    5000 cycles
```

Figure 44. Processor Constants

A summary of the resulting architecture is shown in Figure 45. This architecture is one of the simplest possible with the UFO architecture simulator – a single client, a single persistent cell manager, an embedded repository manager, an embedded transaction manager, no locking, no persistent transactions, no distributed transactions, no automated backups, and eager write-through.

A summary of the architecture properties is shown in Figure 47. This includes the multiplicity of the runtime processes in the architecture and their runtime sizes, plus sizes of the various architectural components the comprise the processes.

Simulation scripts were used to simulate a user creating and browsing the workflow diagram illustrated in Figure 40. The simulation ran 70 minutes on a Sun4c workstation. The output from the simulation was a 76 page execution profile for the application and the OODB architecture. Most of this profile consists of low-level details at the statement level of the application source code, while the more important high-level architectural profile data is presented in a few pages of the simulation profile.

The high-level profiles for the overall architecture, the client, and the persistent cell manager are summarized in Figure 47. Two important points of interest in these profiles are:

- The intra-cell calls dominate the inter-cell calls by a factor of 30. This indicates good internal locality for the clusters defined in the requirements.
- The ratio of activated objects to combined intra-cell, inter-cell, and RPC calls received is 3.6. This indicates high utilization of the objects activated in cells, which implies that the clusters defined in the requirements have good temporal locality.

7.1.3.1. Summary for Experiment 1 Requirements Definition

The parsimony of the architecture combined with the fact that the clusters exhibit good runtime locality indicate that the OODB requirements were accurately defined and that the resulting architecture conforms well to the application requirements on the OODB. We use this set of requirements as the baseline in the remainder of the experiment.

The estimated development cost for defining and refining the OODB requirements for the application were:

- one-half staff*day to define UFO requirement variable values for the application
- one-quarter staff*day to map from requirements to architecture simulator, plus run the simulation
- one-quarter staff*day to review the simulation profile
- one staff*day overhead to explore refinements on the requirements

Thus, we estimate the development cost to be about two days for a developer moderately familiar with the UFO tool. With this minimal level of effort we were able to demonstrate that the defined set of OODB requirements for the workflow editor application are accurate. In contrast, the conventional approach to defining and refining OODB requirements would involve the following labor-intensive tasks:

- *Domain exploration and analysis*. Starting with a “clean slate”, study the OODB domain and characterize the relevant OODB requirement issues.
- *Requirements definition*. For the workflow editor application, develop a clear, complete, and accurate set of OODB requirements.
- *Application prototyping*. Develop a prototype and do the profiling to test the requirements.
- *Requirements refinement*. Iterate through these steps as needed until the requirements are stable.

In the next chapter we further explore and make estimates on the relative cost savings of the UFO versus conventional approach.

```
Non-reentrant Client:

   Embedded Repository Manager:

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:
      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root

      Cell: Composite
         Role Declarations:
            Proxy Role comp_root is Class CompositeTask
            Role prim is Class PrimitiveTask
            Role res is Class Resource
            Role prod is Class Product
            Role pset is Class ProductSet
            Role pmem is Class ProductMember
            Role pflo is Class ProductFlow
            Role fchn is Class FlowChain

      Cell: PContent
         Role Declarations:
            Proxy Role pcont_root is Class ProductContent
            Role lnode is Class Node
            Role rnode is Class Node

   Root Role: Root.root_root

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
      Write-through: EAGER
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
   <Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED
   <Embedded in client.  See Client description.>
```

Figure 45. Editor Architecture Summary with Baseline Requirements

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 1
Total size of shared processes: 2000000 Bytes

Non-reentrant Client:

Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:

   Total size: 50000 Bytes

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:

   Total size: 200000 Bytes

      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.

   Total size: 1100000 Bytes

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2000000 Bytes

   Persistent Cell Manager: CSP

   Write-through: EAGER
   Total size: 2000000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object Locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS


Repository Manager.   Integration: EMBEDDED

<Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED

<Embedded in client.  See Client description.>
```

Figure 46. Editor Architecture Properties with Baseline Requirements

```
Note: All time measurements are expressed in micro-seconds.

Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 4747
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 146882
Activations:     Objects: 13441    Cells: 98
                 Time: 99277
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 19524
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1464740
```

Overall Architecture Summary Profile

```
Non-reentrant Client:

   Intra-Cell Calls Made:     46604    Time: 70607
   Intra-Cell Calls Received: 46604    Time: 70605
   Inter-Cell Calls Made:     1559    Time: 2361
   Inter-Cell Calls Received: 1567    Time: 4747
   RPCs Made:                 0    Time: 0
   RPCs Received:             0    Time: 0

   Object and Cell Services:

      Creations:        Objects: 4588    Cells: 32
                        Time: 70095
      Activations:      Objects: 13441    Cells: 98
                        Time: 43699
      Passivations:     Objects: 4593    Cells: 37
                        Time: 15039

   Total Execution Time: 779088
```

Client Summary Profile

```
Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Write-through: EAGER
      Creations:    Objects:  4588    Cells:  32
                    Time:  76787
      Activations:  Objects:  13441    Cells:  98
                    Time:  55578
      Passivations: Objects:  4593    Cells:  37
                    Time:  19524
      Commits:      Objects:  4593    Cells:  37
                    Time:  19524
      Aborts:       Objects:  0    Cells:  0
                    Time:  0
      Total Execution Time:   685652
```

Persistent Cell Manager Summary Profile

Figure 47. Editor Simulation Profiles with the Baseline Requirements

7.1.4. Selecting an Off-the-Shelf OODB for the Workflow Diagram Editor

In the next part of the experiment, we used the UFO tool to compare the architectural properties of three off-the-shelf OODBs, using as a baseline the architectural properties that we defined in the first part of the experiment. We used the same workflow diagram editor application code and simulation scripts as in the first part of the experiment. With the baseline requirements as the starting point for evaluating each OODB, we then used the static feedback and the dynamic simulation feedback to identify sub-optimal architectural properties in the off-the-shelf OODBs and to identify the "best" OODB choice for the workflow diagram editor application.

We selected the three off-the-shelf OODB architectures for the experiment to represent a range of functionality, size, complexity, and performance. ITASCA is at one extreme, with a large-scale, multi-user, distributed architecture. ObjectStoreLite is at the other extreme with a light-weight, single-user, single-machine architecture. Objectivity lies between these two extremes and is architecturally oriented towards medium sized multi-user applications with large granularity clustering and concurrency control.

7.1.4.1. Objectivity

We first set the UFO tool to target the Objectivity OODB architecture and started with the baseline requirement variable values from the previous section (see Figure 42.). When the mapping from requirement variables to architectural parameters was invoked, the following inconsistencies were reported between the stated requirements and Objectivity:

```
Long Transactions: FALSE
** INCONSISTENCY: Objectivity always supports long transactions.
Set architectural parameter to TRUE.
FALSE Long Transactions requirement value not supported.

Object Locking: FALSE
** INCONSISTENCY: Objectivity always supports object locking.
Set architectural parameter to TRUE.
FALSE Concurrent Users requirement value not supported.

Transaction Manager Integration: EMBEDDED
** INCONSISTENCY: Objectivity does not support embedded transaction managers.
Remove embedded transaction manager and create a shared transaction manager.
FALSE Concurrent Users requirement value cannot be supported.
```

The architectural parameters were modified as indicated, which resulted in the following warning:

```
Repository Manager Integration: SHARED
** WARNING ** Objectivity always uses embedded repository managers.
Remove this process from the Objectivity profile.
```

This warning is accounted for in the architecture profile.

Next, the mapping from architectural parameters to architecture instance and the mapping from architecture instance to architecture simulator were invoked to create the architecture instance for simulation. The same processor constants were used as in the first part of the experiment (see Figure 44.). The architecture summary is shown in Figure 48., the architecture properties are shown in Figure 49., and the simulation profile summary is shown in Figure 50.

We will review these results in Section 7.1.4.4. to compare and contrast Objectivity, ITASCA, and ObjectStoreLite after we present the results from these other two OODBs.

The development activity associated with resolving the reported inconsistencies and warnings, for refining the requirements to be consistent with the Objectivity architecture, and for collecting the simulation data is estimated to be about one staff*day for a developer moderately familiar with the UFO tool.

```
Non-reentrant Client:

   Embedded Repository Manager:

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: SHARED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

    Object and Cell Services.
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root

      Cell: Composite
         Role Declarations:
            Proxy Role comp_root is Class CompositeTask
            Role prim is Class PrimitiveTask
            Role res is Class Resource
            Role prod is Class Product
            Role pset is Class ProductSet
            Role pmem is Class ProductMember
            Role pflo is Class ProductFlow
            Role fchn is Class FlowChain

      Cell: PContent
         Role Declarations:
            Proxy Role pcont_root is Class ProductContent
            Role lnode is Class Node
            Role rnode is Class Node

   Root Role: Root.root_root

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED
   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
      Write-through: EAGER
      Persistent nested transactions: SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object locking: SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
   <Embedded in client.  See Client description.>

Transaction Manager.   Integration: SHARED

   Nested persistent transactions: SUPPORTED
   Distributed transactions: NOT SUPPORTED
```

Figure 48. Summary of Objectivity-based Editor Architecture

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 2
Total size of shared processes: 3750000 Bytes

Non-reentrant Client:

Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:
   Total size: 50000 Bytes

      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Object and Cell Services.

   Total size: 1300000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER


<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED

Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2750000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2750000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: SUPPORTED.    Size: 500000
      Distributed transactions: NOT SUPPORTED
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS



Repository Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>

Transaction Manager.   Integration: SHARED

Total size: 1000000 Bytes

   Nested persistent transactions: SUPPORTED.    Size: 500000
   Distributed transactions: NOT SUPPORTED
```

Figure 49. Properties of Objectivity-based Editor Architecture

```
Note: All time measurements are expressed in micro-seconds.

Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 7121
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 149307
Activations:     Objects: 13441    Cells: 98
                 Time: 106702
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 22327
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1972351
```

Overall Architecture Summary Profile

```
Non-reentrant Client:

   Intra-Cell Calls Made:     46604    Time: 70607
   Intra-Cell Calls Received: 46604    Time: 70605
   Inter-Cell Calls Made:     1559    Time: 2361
   Inter-Cell Calls Received: 1567    Time: 7121
   RPCs Made:                 0    Time: 0
   RPCs Received:             0    Time: 0

   Object and Cell Services:

      Creations:        Objects: 4588    Cells: 32
                        Time: 70095
      Activations:      Objects: 13441    Cells: 98
                        Time: 43699
      Passivations:     Objects: 4593    Cells: 37
                        Time: 15039

   Total Execution Time: 909370
```

Client Summary Profile

```
Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Write-through: EAGER
      Creations:    Objects:  4588    Cells:  32
                    Time:  79212
      Activations:  Objects:  13441    Cells:  98
                    Time:  63003
      Passivations: Objects:  4593    Cells:  37
                    Time:  19524
      Commits:      Objects:  4593    Cells:  37
                    Time:  22327
      Aborts:       Objects:  0    Cells:  0
                    Time:  0
      Total Execution Time:   1062981
```

Persistent Cell Manager Summary Profile

Figure 50. Simulation Profiles for Objectivity-based Editor

7.1.4.2. ITASCA

Next, we set the UFO tool to target the ITASCA OODB architecture and restarted with the baseline requirement variable values as before (see Figure 42.). When the requirement variable to architectural parameter mapping was invoked, the following inconsistencies were reported for ITASCA:

```
** INCONSISTENCY: ITASCA always supports long transactions.
Manually set architectural parameter to TRUE.
FALSE Long Transactions requirement value not supported.

** INCONSISTENCY: ITASCA always supports object locking.
Set architectural parameter to TRUE.
FALSE Concurrent Users requirement value not supported.

**INCONSISTENCY: ITASCA always supports distributed transactions.
Set architectural parameter to TRUE.
See message regarding sparse cluster sets for unsupported requirements.

** INCONSISTENCY: ITASCA does all computation in object servers.
Allocate all cells to Object Servers.
Cannot support Dense Cluster Set requirement.

** INCONSISTENCY: ITASCA does not support embedded transaction managers.
Remove embedded transaction manager and create a shared transaction manager.
FALSE Concurrent Users requirement value cannot be supported.

** INCONSISTENCY: ITASCA does not support embedded repository managers.
Remove embedded repository manager and create a shared repository manager.
FALSE Concurrent Users requirement value cannot be supported.
```

The architectural parameters were modified as indicated by these messages.

Next, the mapping from architectural parameters to architecture instance and the mapping from architecture instance to architecture simulator were invoked. The same processor constants were used as in the first part of the experiment (see Figure 44.). The resulting architecture summary is shown in Figure 51., the architecture properties are shown in Figure 52., and the simulation profile summary is shown in Figure 53.

We will review these results in Section 7.1.4.4. to compare and contrast Objectivity, ITASCA, and ObjectStoreLite for use with the workflow diagram editor application.

As with Objectivity, the development activity associated with resolving the reported inconsistencies and warnings, for refining the requirements to be consistent with the ITASCA architecture, and for collecting the simulation data is estimated to be about one staff*day for a developer moderately familiar with the UFO tool.

7.1.4.3. ObjectStoreLite

Next, we set the UFO tool to target the ObjectStoreLite OODB architecture and restarted with the baseline requirement variable values as before (see Figure 42.). When the mapping from requirement variables to architectural parameters was invoked, no inconsistencies were reported between the stated requirements and ObjectStoreLite.

Next, the mappings were invoked to create the architecture instance for simulation. The same processor constants were used as in the first part of the experiment (see Figure 44.). The architecture summary is shown in Figure 54., the architecture properties are shown in Figure 55., and the simulation profile summary is shown in Figure 56.

We will review these results in the next to compare and contrast Objectivity, ITASCA, and ObjectStoreLite for use with the workflow diagram editor application.

Since there were no inconsistencies or warnings reported for ObjectStoreLite, the development activity associated with collecting the simulation data for the architecture is estimated to be about one half staff*day for a developer moderately familiar with the UFO tool.

```
Reentrant Client:

   Object and Cell Services.

      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED
      Write-through policy: EAGER

   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root

   Root Role: Root.root_root

Object Server Set.   Integration: SHARED

   Threaded Object Server: SCS
      Object and Cell Services.

         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED
         Write-through policy: EAGER

      Cell Allocations:
         Cell: Composite
            Role Declarations:
               Proxy Role comp_root is Class CompositeTask
               Role prim is Class PrimitiveTask
               Role res is Class Resource
               Role prod is Class Product
               Role pset is Class ProductSet
               Role pmem is Class ProductMember
               Role pflo is Class ProductFlow
               Role fchn is Class FlowChain
         Cell: PContent
            Role Declarations:
               Proxy Role pcont_root is Class ProductContent
               Role lnode is Class Node
               Role rnode is Class Node

Persistent Cell Manager Set.   Integration: SHARED
   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
         SCS
      Write-through: EAGER
      Persistent nested transactions: SUPPORTED
      Distributed transactions: SUPPORTED
      Object locking: SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: SHARED
   Object server set administration: SHARED
   Persistent cell manager set administration: SHARED

Transaction Manager.   Integration: SHARED
   Nested persistent transactions: SUPPORTED
   Distributed transactions: SUPPORTED
```

Figure 51. Summary for ITASCA-based Editor Architecture

```
Size per client: 2634720 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 4
Total size of shared processes: 10320800 Bytes

Reentrant Client:
Total size: 2634720 Bytes
Size accounted for by methods: 34720 Bytes
Size accounted for by cache: 1000000 Bytes

   Object and Cell Services.
   Total size: 1350000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED.    Size: 50000
      Write-through policy: EAGER

Object Server Set.   Integration: SHARED
Total processes per object server set: 1
Total size per object server set: 5970800 Bytes

   Threaded Object Server: SCS
   Total size: 5970800 Bytes
   Size accounted for by methods: 20800 Bytes
   Size accounted for by cache: 4000000 Bytes

      Object and Cell Services.
      Total size: 4450000 Bytes
         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED.    Size: 50000
         Write-through policy: EAGER

Persistent Cell Manager Set.   Integration: SHARED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2950000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2950000 Bytes

      Processes allocated to PCM:
         CLIENT
         SCS
      Persistent nested transactions: SUPPORTED.    Size: 500000
      Distributed transactions: SUPPORTED.    Size: 200000
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
Total size: 200000 Bytes
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: SHARED
   Object server set administration: SHARED
   Persistent cell manager set administration: SHARED

Transaction Manager.   Integration: SHARED
Total size: 1200000 Bytes
   Nested persistent transactions: SUPPORTED.    Size: 500000
   Distributed transactions: SUPPORTED.    Size: 200000
```

Figure 52. Properties for ITASCA-based Editor Architecture

```
Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     228    Time: 345
Inter-Cell Calls Received: 236    Time: 1072
RPCs Made:                 1331    Time: 22183
RPCs Received:             1331    Time: 24199
Creations:       Objects: 4588    Cells: 32
                 Time: 149307
Activations:     Objects: 6    Cells: 6
                 Time: 1581
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 22327
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  2035604
```

Overall Architecture Summary Profile

```
Intra-Cell Calls Made:     939    Time: 1422
Intra-Cell Calls Received: 939    Time: 1422
Inter-Cell Calls Made:     0    Time: 0
Inter-Cell Calls Received: 8    Time: 36
RPCs Made:                 1331    Time: 22183
RPCs Received:             0    Time: 0
Creations:        Objects: 1    Cells: 1
                  Time: 33
Activations:      Objects: 6    Cells: 6
                  Time: 200
Passivations:     Objects: 6    Cells: 6
                  Time: 200
Total Execution Time: 100029
```

Client Summary Profile

```
Intra-Cell Calls Made:     45665    Time: 69185
Intra-Cell Calls Received: 45665    Time: 69183
Inter-Cell Calls Made:     228    Time: 345
Inter-Cell Calls Received: 228    Time: 1036
RPCs Made:                 0    Time: 0
RPCs Received:             1331    Time: 24199
Creations:        Objects: 4587    Cells:  31
                  Time: 70062
Activations:      Objects: 0    Cells:  0
                  Time: 0
Passivations:     Objects: 4587    Cells:  31
                  Time: 14839
Total Execution Time: 1177035
```

Object Server Summary Profile

```
Creations:    Objects:  4588    Cells:  32
              Time:  79212
Activations:  Objects:  6    Cells:  6
              Time:  1381
Passivations: Objects:  4593    Cells:  37
              Time:  19524
Commits:      Objects:  4593    Cells:  37
              Time:  22327
Aborts:       Objects:  0    Cells:  0
              Time:  0
Total Execution Time:   758540
```

Persistent Cell Manager Summary Profile

Figure 53. Simulation Profiles for ITASCA-based Editor

```
Non-reentrant Client:

       Embedded Repository Manager:
          Maximum number of concurrent clients on a repository: 1
          Maximum number of repositories per installation: 1000000000
          Transaction manager administration: EMBEDDED
          Object server set administration: NO OBJECT SERVERS
          Persistent cell manager set administration: SHARED

       Embedded Transaction Manager:
          Nested persistent transactions: NOT SUPPORTED
          Distributed transactions: NOT SUPPORTED

       Object and Cell Services.
          Threads: SINGLE
          Object locking: NOT SUPPORTED
          Persistent nested transactions: NOT SUPPORTED
          Cache size: 1000000 Bytes
          Cache replacement policy: LRU
          Distributed transactions: NOT SUPPORTED
          Write-through policy: EAGER

       Cell Allocations:
          Cell: Root
             Role Declarations:
                Proxy Role root_root is Class VPML_Root

          Cell: Composite
             Role Declarations:
                Proxy Role comp_root is Class CompositeTask
                Role prim is Class PrimitiveTask
                Role res is Class Resource
                Role prod is Class Product
                Role pset is Class ProductSet
                Role pmem is Class ProductMember
                Role pflo is Class ProductFlow
                Role fchn is Class FlowChain

          Cell: PContent
             Role Declarations:
                Proxy Role pcont_root is Class ProductContent
                Role lnode is Class Node
                Role rnode is Class Node

       Root Role: Root.root_root

    <No Object Servers>

    Persistent Cell Manager Set.   Integration: SHARED

       Persistent Cell Manager. CSP
          Processes allocated to PCM:
             CLIENT
          Write-through: EAGER
          Persistent nested transactions: NOT SUPPORTED
          Distributed transactions: NOT SUPPORTED
          Object locking: NOT SUPPORTED
          Backup policy: NO AUTOMATED BACKUPS

    Repository Manager.   Integration: EMBEDDED
       <Embedded in client.  See Client description.>

    Transaction Manager.   Integration: EMBEDDED
       <Embedded in client.  See Client description.>
```

Figure 54. ObjectStoreLite-based Editor Architecture Summary

```
Size per client: 2405520 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 1
Total size of shared processes: 2000000 Bytes

Non-reentrant Client:
Total size: 2405520 Bytes
Size accounted for by methods: 55520 Bytes
Size accounted for by cache: 1000000 Bytes

   Embedded Repository Manager:
   Total size: 50000 Bytes
      Maximum number of concurrent clients on a repository: 1
      Maximum number of repositories per installation: 1000000000
      Transaction manager administration: EMBEDDED
      Object server set administration: NO OBJECT SERVERS
      Persistent cell manager set administration: SHARED

   Embedded Transaction Manager:
   Total size: 200000 Bytes
      Nested persistent transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED

   Object and Cell Services.
   Total size: 1100000 Bytes

      Threads: SINGLE
      Object locking: NOT SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: NOT SUPPORTED
      Write-through policy: EAGER

<No Object Servers>

Persistent Cell Manager Set.   Integration: SHARED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2000000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2000000 Bytes

      Processes allocated to PCM:
         CLIENT
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: NOT SUPPORTED
      Object Locking: NOT SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>

Transaction Manager.   Integration: EMBEDDED
<Embedded in client.  See Client description.>
```

Figure 55. ObjectStoreLite-based Editor Architecture Properties

```
Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 4747
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0
Creations:       Objects: 4588    Cells: 32
                 Time: 146882
Activations:     Objects: 13441    Cells: 98
                 Time: 99277
Passivations:    Objects: 4593    Cells: 37
                 Time: 34563
Commits:         Objects: 4593    Cells: 37
                 Time: 19524
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  1464740
```

Overall Architecture Summary Profile

```
Intra-Cell Calls Made:     46604    Time: 70607
Intra-Cell Calls Received: 46604    Time: 70605
Inter-Cell Calls Made:     1559    Time: 2361
Inter-Cell Calls Received: 1567    Time: 4747
RPCs Made:                 0    Time: 0
RPCs Received:             0    Time: 0

Object and Cell Services:

   Creations:        Objects: 4588    Cells: 32
                     Time: 70095
   Activations:      Objects: 13441    Cells: 98
                     Time: 43699
   Passivations:     Objects: 4593    Cells: 37
                     Time: 15039

Total Execution Time: 779088
```

Client Summary Profile

```
Creations:    Objects:  4588    Cells:  32
              Time:  76787
Activations:  Objects:  13441    Cells:  98
              Time:  55578
Passivations: Objects:  4593    Cells:  37
              Time:  19524
Commits:      Objects:  4593    Cells:  37
              Time:  19524
Aborts:       Objects:  0    Cells:  0
              Time:  0
Total Execution Time:   685652
```

Persistent Cell Manager Summary Profile

Figure 56. ObjectStoreLite-based Editor Simulation Profiles

7.1.4.4. Collating the OODB Simulation Results for the Workflow Diagram Editor

To compare and contrast the conformance of Objectivity, ITASCA, and ObjectStoreLite to the workflow diagram editor, we first look at the architectural differences reported in the UFO modeling and we then look at the relative size and performance data reported by the tool. The baseline requirements and architecture are used as a standard for comparison.

Table 3. summarizes structural and functional conformance:

- (column 1) the name of the OODB being modeled
- (column 2) the architectural inconsistencies detected in the baseline architecture when compared to the OODB. These are the inconsistencies reported by the tool for each OODB.
- (column 3) the baseline requirements that can't be satisfied due to the inconsistencies. These are reported by the tool along with the reported inconsistencies in column 2.
- (column 4) the architectural differences between the OODB and the baseline architecture. These differences were identified by comparing the architectural summaries of the baseline and other OODBs (Figure 45., Figure 48., Figure 51., and Figure 54.).

This table demonstrates that ObjectStoreLite is a perfect architectural match to the workflow diagram editor application. Objectivity had extraneous architectural support for long transactions, object locking, plus stand-alone process for transaction management. ITASCA had these three non-conformance areas plus 4 others.

Table 4. provides size and performance profiles on the baseline architecture and the three OODBs in order to characterize the impact of the architectural conformance issues identified in Table 3. This size and performance data comes from the architectural properties and simulation profiles summarized in Figure 47. & Figure 47., Figure 49. & Figure 50., Figure 52. & Figure 53., and Figure 55. & Figure 56.

Since there were no architectural differences between the baseline and ObjectStoreLite, their modeling and simulation data are the same. Objectivity provides the next best conformance for the application, with an additional shared process (the stand-alone transaction manager), a 90% larger shared process size, a 40% larger total runtime size, and only 75% of the execution speed. The details in the architecture summaries and simulation profiles show that the size and performance differences are accounted for by extraneous functionality in the Objectivity architecture. ITASCA conformance was even lower due to the extensive functionality that it provides – three additional shared processes, a 520% larger shared process size, a 290% larger total runtime size, and only 75% of the execution speed.

Table 3. Reported Inconsistencies and Architectural Impacts

OODB Model	Architectural Inconsistencies Reported	Unsupported Baseline Requirements	Architectural Differences

Objectivity	Objectivity always supports long transactions.
	
	Objectivity always supports object locking.
	
	Objectivity does not support embedded transaction managers.
		`The Long Transaction requirement is inconsistent with Objectivity.`
		
		`The Concurrent Users requirement is inconsistent with Objectivity.`
			Extra long transactions support in Client, Transaction Manager, and Persistent Cell Manager.
			
			Extra object locking support in Client and Persistent Cell Manager.
			
			Transaction manager a stand-along process rather than embedded in client.

ITASCA	ITASCA always supports long transactions.
	
	ITASCA always supports object locking.
	
	ITASCA always supports distributed transactions.
	
	ITASCA does all computation in object servers.
	
	ITASCA does not support embedded transaction managers.
	
	ITASCA does not support embedded repository managers.
		`The Long Transaction requirement is inconsistent with Objectivity.`
		
		`The Concurrent Users requirement is inconsistent with Objectivity.`
		
		`The Dense Cluster Set requirement is inconsistent with Objectivity.`
			Extra long transactions support in Client, Transaction Manager, and Persistent Cell Manager.
			
			Extra distributed transactions support in Client, Transaction Manager, and Persistent Cell Manager.
			
			Extra object locking support in Client and Persistent Cell Manager.
			
			Extra object server process serving as the primary computational server for the application.
			
			Extra multi-threading support in object server.
			
			Transaction manager a stand-along process rather than embedded in client.
			
			Repository manager a stand-along process rather than embedded in client.

ObjectStoreLite	None	None	None

Table 4. OODB Properties for the Workflow Diagram Editor Application

OODB Model	Runtime Client Size	Number of Shared Processes	Runtime Shared Process Size	Total Runtime Size	Total Execution Time
Baseline	2.4 Mbytes	1	2.0 Mbytes	4.4 Mbytes	1.5 Seconds
Objectivity	2.4 Mbytes	2	3.8 Mbytes	6.2 Mbytes	2.0 Seconds
ITASCA	2.6 Mbytes	4	10.3 Mbytes	12.9 Mbytes	2.0 Seconds
ObjectStoreLite	2.4 Mbytes	1	2.0 Mbytes	4.4 Mbytes	1.5 Seconds

7.1.4.5. Summary for Experiment 1 OODB Selection

The estimated development cost for simulating and comparing the off-the-shelf OODB system properties for the three OODBs were:

- one-half staff*day per OODB to execute the mappings from requirements to simulator and to address inconsistencies reported by the tool
- one-quarter staff*day per OODB to run the simulation
- one-quarter staff*day per OODB to review the simulation profile
- one staff*day overhead to do the comparative analysis among the different OODBs

Thus, we estimate the cost for evaluating the three OODBs to be about four days for a developer moderately familiar with the UFO tool. With this minimal level of effort we were able to characterize how well each of the OODBs satisfied the requirements of the workflow editor application. In contrast, the conventional approach to evaluating OODB requirements would involve the following labor-intensive tasks:

- Installing and learning how to use each OODB.
- Connecting the application to the each OODB.
- Application and OODB profiling.

In the next chapter we further explore and make estimates on the relative cost savings of the UFO versus conventional approach.

7.2. Experiment 2: An OODB for a Workflow Management System

Similar to the first experiment, the description of the second experiment is organized into subsections with the following order and content:

- Conceptual views of application
- Design of the extra classes needed for the second experiment
- Design and implementation of simulation scripts to drive the experiments with typical scenarios
- Use of the UFO tool to prototype requirements. Resulting modeling and simulation data plus feedback to refine and refine requirements.
- Use of the UFO tool to evaluate off-the-shelf OODBs. Feedback, modeling, and simulation data to compare and select an OODB for the application.
- Modeling and simulation data to illustrate conformance. Estimates on the development costs. Resulting cost/conformance profiles.

The second experiment is based on a software application called the *Workflow Management System*. It is a multi-user application that uses workflow diagrams for project management. The workflow diagrams produced by the editor in the first experiment are given an execution semantics. The workflow management system “executes” a workflow diagram, keeping track of which tasks have been completed, which tasks are in progress, which tasks are ready to start, and which tasks are waiting for input products from upstream tasks. Users interact with the workflow management system, requesting information about tasks that are active or ready, indicating that they are taking on a new task, creating data products that will be used by downstream tasks, and indicating that they have completed a task.

As an example, the workflow diagram in Figure 40. begins executing with no active tasks and one ready task. The ready task, *Identify Sub-market* in the upper-left, starts in the ready state because it is initialized with an input *token*, as indicated with the number 1 in the circle on the input flow to that task. Whenever all of the input flows to a task receive one or more tokens, the task becomes ready.

Tokens are generated in one of two ways, (1) initial tokens are created each time a workflow diagram begins execution, and (2) a token is generated down the flow from a product of an active task whenever a user indicates that the task has been completed. An input token from an input flow is consumed by a task whenever the task goes from the inactive state to the ready state. However, if an input flow is labeled as *gated*, denoted by the ‘G’ in a circle, the token “sticks” at the input and is never consumed. That is, once the first token arrives at a gated input, there will always be a token satisfying that input. The *Define Release Requirements* task in the lower left of the diagram has a gated input. When the first token arrives at this input, the workflow diagram goes into a continuous loop that restarts each time a token becomes available on the other input.

Users can view the list of ready tasks and the list of active tasks. A user can activate a ready task, which means that the user assumes responsibility for completing the task. When the task is completed, the user indicates this to the workflow management system, in which case an output token is generated and the task goes back to the inactive state.

The workflow diagrams can have conditional branches, such as the output from *MS Gate Key & Market Analysis* in the lower center of the top composite task in Figure 40. When the user indicates that the *Market Search Gate* task is complete, they must also indicate which of the two branches to follow, *continue* or *done*.

Typical use of the workflow management system is characterized by multiple concurrent users making short, small updates to an executing workflow diagram. For this experiment, we focus on workflow management system installations with large numbers of users, such as insurance claim processing sites or large corporate software development sites with thousands of users.

The experimental description that follows first describes the workflow management system implementation, mostly in terms differences in and extensions to the workflow editor implementation. Next, is the description of how the UFO tool was used to converge on a baseline set of OODB requirements for the application. In particular, this experiment illustrates techniques for defining requirements for multi-user applications. Then, the architectural models for the three off-the-shelf OODBs are compared to the baseline requirements using the UFO tool. Finally, we estimate the development costs associated with converging on the baseline requirements and in selecting an off-the-shelf OODB.

7.2.1. .Additional Classes and Components in the Workflow Manager Application

The workflow management system was implemented using the eleven classes from the workflow editor, plus two new classes for managing lists of ready and active tasks, plus additional class components and methods to manage the execution of a workflow diagram. The two new classes are illustrated in Figure 57. The gray boxes and arrows are for several of the original eleven classes shown in the top of Figure 39., while the black boxes and arrows are for the two new classes.

The workflow management system implemented consisted of about 2900 lines of UFO source code. The class components for the two new classes are described in the following two paragraphs. Following that, the extensions to the components in the other eleven classes are shown. The basic classes from the editor are maintained in the workflow management system. The operations for creating objects with the editor are adapted for importing the workflow diagrams that will be executed.

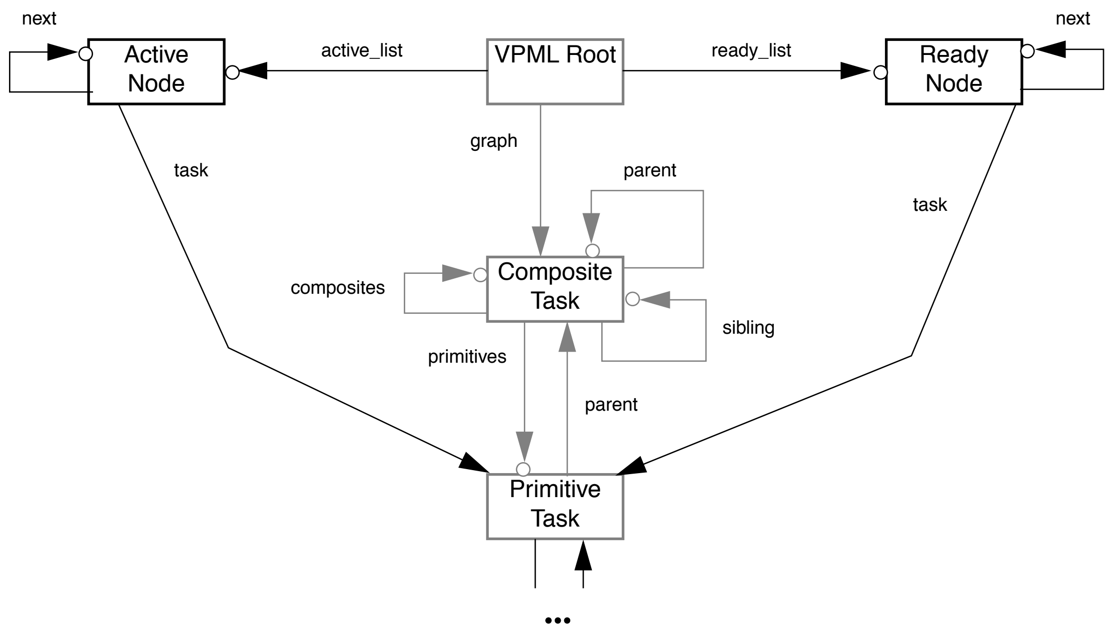


Figure 57. Amendments to the Class Diagram for the Workflow Manager Application

Class ReadyNode. This class is for the Ready Node entity illustrated in Figure 57. It implements the list of primitive tasks that have all of their input tokens available but that have not yet been activated by a user. The *task* component points to the primitive task object that is ready. The *next* and *previous* components implement the linked list. The *name* component is the name of the primitive task.

```
Class ReadyNode
...
   Components
      task: PrimitiveTask;
      next: ReadyNode;
      previous: ReadyNode;
      name: STRING;
```

Class ActiveNode. This class is for the Active Node entity illustrated in Figure 57. It implements the list of primitive tasks that have been activated by a user but that have not yet been designated as complete. The *task* component points to the primitive task object that is active. The *next* and *previous* components implement the linked list. The *name* component is the name of the primitive task.

```
Class ActiveNode
...
   Components
      task: PrimitiveTask;
      next: ActiveNode;
      previous: ActiveNode;
      name: STRING;
```

RootClass VPML_Root. This class from the editor is extended with four components. *Working_task* points to the active task that the user is currently working on. *Ready_list* and *active_list* point to the heads of the ready and active task lists. *Enacting* indicates whether or not the workflow diagram is still being edited or whether it is being executed (once execution starts, the diagram can no longer be modified).

```
RootClass VPML_Root
...
   Components
      ...
      working_task: PrimitiveTask;
      ready_list: ReadyNode;
      active_list: ActiveNode;
      enacting: BOOLEAN;
```

Class PrimitiveTask. This class from the editor is extended with one component. The *state* component indicates whether the task is inactive, ready, or active.

```
Class PrimitiveTask
...
   Components
      ...
      state: CHARACTER;
```

Class Product. This class from the editor is extended with one component. The *state* component indicates whether the product is uninitialized or initialized.

```
Class Product
   ...
   Components
      ...
      state: CHARACTER;
```

Class ProductMember. This class from the editor is extended with two components. The *token_que_length* indicates the number of tokens in an input flow, and *gated* indicates whether the input flow is gated or not.

```
Class ProductMember
   ...
   Components
      ...
      token_que_length: INTEGER;
      gated: BOOLEAN;
```

Class ProductFlow. This class from the editor is extended with three components. The *selector* component indicates whether or not the product flow is a selector with multiple output flows. If it is a selector, the *selected* value indicates whether or not the user has selected which output flow to use on an active task. If it is selected, the *selection* value is the name of the selected flow.

```
Class ProductFlow
   ...
   Components
      ...
      selector: BOOLEAN;
      selected: BOOLEAN;
      selection: STRING;
```

Class FlowChain. This class from the editor is extended with one component. The *selection* component provides the name of the selection for each of the alternatives in an output flow.

```
Class FlowChain
   ...
   Components
      ...
      selection: STRING;
```

7.2.2. The Simulation Scripts

The simulation scripts that we created to drive the workflow management system for this experiment were a sequence typical task activations, product updates, task completions, and task browsing. Since the UFO simulator does not support the simulation of concurrent users, the operations were performed as sequential user interactions. We show later how to evaluate the sequential simulation results in terms of concurrent user loads.

The simulation was on the workflow diagram illustrated in Figure 40. The execution went around the loop in the top composite task (*Define Markets*) five times, through all primitive tasks in the next composite task (*Define Business Plan*), around the loops in *Define Functional Spec and Business Reqts* four times, through all primitive tasks in *Engineering Analysis*, and around the *Release Cycle* loop five times. This resulted in the activation and close out of 76 tasks, or 152 different user interactions.

Figure 58. shows an excerpt from the simulation scripts used for this experiment. The method invocations alternately activate and complete (close out) tasks. The method definitions for *SimActivateTask* and *SimCloseOutTask* are shown in Figure 59. Each consists of several simple steps that a user would typically take when interactive with the application.

```
ME.SimActivateTask("Identify Sub-market");
ME.SimCloseOutTask("Identify Sub-market"; 4; "");
ME.SimActivateTask("Analyze Sub-market");
ME.SimCloseOutTask("Analyze Sub-market"; 5; "");
ME.SimActivateTask("Market Synthesis");
ME.SimCloseOutTask("Market Synthesis"; 6; "");
ME.SimActivateTask("Market Search Gate");
ME.SimCloseOutTask("Market Search Gate"; 2; "continue");
ME.SimActivateTask("Identify Sub-market");
ME.SimCloseOutTask("Identify Sub-market"; 4; "");
ME.SimActivateTask("Analyze Sub-market");
ME.SimCloseOutTask("Analyze Sub-market"; 5; "");
```

Figure 58. Example Code from Simulation Scripts

7.2.3. Prototyping OODB Requirements for the Workflow Management System

As in the first experiment, we next focus on defining and refining OODB requirements for the application. In this experiment we demonstrate more advanced techniques for using feedback from modeling and simulation, including automated suggestions on refining OODB requirements that are generated during the UFO tool's automated analysis of simulation profiles.

Prototyping the OODB requirements for the workflow management system proceeded through three stages. The initial set of requirements were similar to those used for the workflow diagram editor in the first experiment. The next refinement moved some of the object clusters into computational servers to optimize data movement among processes. The final refinement replicated some of the architectural components in order to meet high levels of multi-user concurrency.

```
Procedure SimActivateTask(name: STRING)

   ME.ListReady(FALSE);
   ME.ActivateTask(name);
   ME.ConnectToTask(name);
   ME.QueryTask(FALSE);
end Procedure SimActivateTask

Procedure SimCloseOutTask(name: STRING; psize: INTEGER; sel: STRING)

   ME.ListActive(FALSE);
   ME.ConnectToTask(name);
   ME.QueryTask(FALSE);
   ME.CreateProductContent(psize);
   case
      when (! sel == "") then
         ME.AssignSelector(sel);
   endcase
   ME.CloseOut( );
end Procedure SimCloseOutTask
```

Figure 59. Method Definitions for SimActivateTask and SimCloseOutTask

7.2.3.1. Initial OODB Requirements for the Workflow Management System – Client-based Architecture

The initial requirement variable values are shown in Figure 60. These are the same as the baseline requirements from the first experiment, with the two exceptions:

- the concurrent users requirement is set to TRUE
- the additional classes of objects (*ReadyNode* and *ActiveNode*) are allocated to the *Root* cluster

We mapped the requirements to an architecture instance for simulation. The simulation scripts were then used to drive the simulation. The same processor constants were used as in the first experiment (see Figure 44.). The total to run the simulation was one hour and twenty minutes. We then used the UFO tool to generate and analyze the profile data. The tool issued the following feedback message regarding the *Composite* cell:

```
** FEEDBACK ** LOW utilization cell in client.
Change cluster Utilization requirement to LOW.
```

Reviewing the profile summary for the *Composite* cell, we find the following raw data:

```
Cell: Composite
   Intra-Cell Calls Made:     2022    Time: 3060
   Intra-Cell Calls Received: 2022    Time: 3060
   Inter-Cell Calls Made:     255    Time: 385
   Inter-Cell Calls Received: 541    Time: 2458
   RPCs Made:                 0    Time: 0
   RPCs Received:             0    Time: 0
   CREATIONS:        Objects: 0    Cells: 0
                     Time: 0
   ACTIVATIONS:      Objects: 6032    Cells: 155
                     Time: 22975
   PASSIVATIONS:     Objects: 6031    Cells: 154
                     Time: 22942
   Total Execution Time: 54880
   Maximum Cell Size: 9472
```

```
Long Transactions: FALSE

Concurrent Users: TRUE

Locality Clusters:
   Cluster: Root
      Roles:
         Role root_root is Class VPML_Root
         Role ready is Class ReadyNode
         Role active is Class ActiveNode
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: Composite
      Roles:
         Role comp_root is Class CompositeTask
         Role prim is Class PrimitiveTask
         Role res is Class Resource
         Role prod is Class Product
         Role pset is Class ProductSet
         Role pmem is Class ProductMember
         Role pflo is Class ProductFlow
         Role fchn is Class FlowChain
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: PContent
      Roles:
         Role pcont_root is Class ProductContent
         Role lnode is Class Node
         Role rnode is Class Node
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE


<No Sparse Cluster Sets>

Dense Cluster Set DCS
   Clusters:
      Root
      Composite
      PContent
   Global Cluster Set Contention: LOW


Cluster Set Partitions
   Partition CSP
      Cluster Sets:
         DCS
      Global Partition Contention: LOW


Global Transaction Throughput: LOW

Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

Figure 60. Initial Requirement Variable Values for Workflow Management System

The feedback message refers to the ratio of the intra-cell, inter-cell, and rpc calls received to the object activations. When this ratio drops below 50%, UFO notifies developers that a cluster is being sparsely utilized (i.e., that significantly more time is being spent activating objects than using them). In this case the ratio was 40%. Since the clustering for the cell is good (i.e., the intra-cell calls dominate the intra-cell calls), the tool recommends that the cluster definition be retained but labeled as a low utilization cluster in the requirements.

7.2.3.2. First Refinement on OODB Requirements – Object Server-based Architecture

The recommended refinement is made to the requirements. Since the *Composite* cluster is now a low utilization cluster, it must be put in a sparse cluster set. A sparse cluster set, *SCS*, is created for this purpose. The refined requirement variable values are shown in Figure 61.

The mappings were invoked to produce the architecture instance for simulation from the new requirements. The resulting architecture summary is shown in Figure 62. Items to note include:

- The *Composite* cell has been moved from client to Object Server, SCS. Now the *Composite* cells can be activated and kept in object server cache for use by multiple clients. This is intended to improve the amortized activation cost by activating once and using many times.

- Distributed transactions are now supported in architecture components (in client, object server, persistent cell manager, and transaction manager) since now both the client and the object server participation in transactions.

- Object locking is supported (in client, object server, and persistent cell manager) because of the multi-user requirement.

The architecture properties are shown in Figure 63. and the simulation profile summary is shown in Figure 64. Note that since the object server caches the *Composite* cell objects, they don't have to be activated each time they are used. In this case the *Composite* cell objects were placed in the cache when they were created by the import operation (prior to the start of the profiling). As a result, the object server summary profile in Figure 64. shows zero activation time, implying that no subsequent activations were needed. This, therefore, shows that the UFO tool's suggested refinement to the initial architectural parameters eliminated the problem of excessive activation time on the low-utilization *Composite* cell.

Since the UFO simulator processes client sessions sequentially, the profiles and projections for concurrent client sessions are derived manually from the sequential data. We use the profile data in Figure 64. to extrapolate concurrent loads on the different components in the OODB architecture.

Each user will have a dedicated workstation to be used as the client machine for the workflow management system. One or more object servers and the persistent cell managers can be shared among the clients. We must project how to structure the architecture to provide sufficient capacity in the object servers and persistent cell managers.

The targeted use of this application is for project management for a large corporate software development site with approximately 6000 employees using the workflow management system. In the average case, each employee will complete two tasks per day in a workflow diagram. Each employee must report their progress before close of business each week, and the typical scenario is for all employees to dedicate the last 15 minutes of work each week to updating their progress in the workflow diagrams. Therefore, the application requires capacity for 6000 persons entering 10 task activation transactions and 10 task closeout transactions in 15 minutes, for a total of 8000 transactions per minute (tpm) for all users, or 1.3 tpm for each of the 6000 users.

These requirements are shown in the third column of Table 5. Since each user is on a different client machine, the client requirement is 1.3 tpm. The object server and persistent cell manager components will be shared and must support the combined transaction rate of 8000 tpm.

```
Long Transactions: FALSE

Concurrent Users: TRUE

Locality Clusters:
   Cluster: Root
      Roles:
         Role root_root is Class VPML_Root
         Role ready is Class ReadyNode
         Role active is Class ActiveNode
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: Composite
      Roles:
         Role comp_root is Class CompositeTask
         Role prim is Class PrimitiveTask
         Role res is Class Resource
         Role prod is Class Product
         Role pset is Class ProductSet
         Role pmem is Class ProductMember
         Role pflo is Class ProductFlow
         Role fchn is Class FlowChain
      Utilization: LOW
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: PContent
      Roles:
         Role pcont_root is Class ProductContent
         Role lnode is Class Node
         Role rnode is Class Node
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

Sparse Cluster Sets
   Cluster Set SCS
      Clusters:
         Composite
      Global Cluster Set Contention: LOW

Dense Cluster Set DCS
   Clusters:
      Root
      PContent
   Global Cluster Set Contention: LOW

Cluster Set Partitions
   Partition CSP
      Cluster Sets:
         DCS
         SCS
      Global Partition Contention: LOW

Global Transaction Throughput: LOW

Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

Figure 61. 1st Refinement of Requirement Variable Values

```
Reentrant Client:
       Object and Cell Services.
          Threads: SINGLE
          Object locking: SUPPORTED
          Persistent nested transactions: NOT SUPPORTED
          Cache size: 1000000 Bytes
          Cache replacement policy: LRU
          Distributed transactions: SUPPORTED
          Write-through policy: EAGER
       Cell Allocations:
          Cell: Root
             Role Declarations:
                Proxy Role root_root is Class VPML_Root
                Role ready is Class ReadyNode
                Role active is Class ActiveNode
          Cell: PContent
             Role Declarations:
                Proxy Role pcont_root is Class ProductContent
                Role lnode is Class Node
                Role rnode is Class Node
       Root Role: Root.root_root

    Object Server Set.   Integration: SHARED
       Reentrant Object Server: SCS
          Object and Cell Services.
             Threads: SINGLE
             Object locking: SUPPORTED
             Persistent nested transactions: NOT SUPPORTED
             Cache size: 4000000 Bytes
             Cache replacement policy: LRU
             Distributed transactions: SUPPORTED
             Write-through policy: EAGER
          Cell Allocations:
             Cell: Composite
                Role Declarations:
                   Proxy Role comp_root is Class CompositeTask
                   Role prim is Class PrimitiveTask
                   Role res is Class Resource
                   Role prod is Class Product
                   Role pset is Class ProductSet
                   Role pmem is Class ProductMember
                   Role pflo is Class ProductFlow
                   Role fchn is Class FlowChain

    Persistent Cell Manager Set.   Integration: SHARED
       Persistent Cell Manager. CSP
          Processes allocated to PCM:
             CLIENT
             SCS
          Write-through: EAGER
          Persistent nested transactions: NOT SUPPORTED
          Distributed transactions: SUPPORTED
          Object locking: SUPPORTED
          Backup policy: NO AUTOMATED BACKUPS

    Repository Manager.   Integration: SHARED
       Maximum number of concurrent clients on a repository: 1000000
       Maximum number of repositories per installation: 1000000000
       Transaction manager administration: SHARED
       Object server set administration: SHARED
       Persistent cell manager set administration: SHARED

    Transaction Manager.   Integration: SHARED
       Nested persistent transactions: NOT SUPPORTED
       Distributed transactions: SUPPORTED
```

Figure 62. 1st Refinement Architecture Summary

```
Size per client: 2560360 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 4
Total size of shared processes: 8881360 Bytes

Reentrant Client:
Total size: 2560360 Bytes
Size accounted for by methods: 60360 Bytes
Size accounted for by cache: 1000000 Bytes

   Object and Cell Services.
   Total size: 1250000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED.    Size: 50000
      Write-through policy: EAGER

Object Server Set.   Integration: SHARED
Total processes per object server set: 1
Total size per object server set: 5531360 Bytes

   Reentrant Object Server: SCS
   Total size: 5531360 Bytes
   Size accounted for by methods: 31360 Bytes
   Size accounted for by cache: 4000000 Bytes

      Object and Cell Services.
      Total size: 4250000 Bytes
         Threads: SINGLE
         Object locking: SUPPORTED
         Persistent nested transactions: NOT SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED.    Size: 50000
         Write-through policy: EAGER

Persistent Cell Manager Set.   Integration: SHARED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2450000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2450000 Bytes

      Processes allocated to PCM:
         CLIENT
         SCS
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: SUPPORTED.    Size: 200000
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
Total size: 200000 Bytes
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: SHARED
   Object server set administration: SHARED
   Persistent cell manager set administration: SHARED

Transaction Manager.   Integration: SHARED
Total size: 700000 Bytes
   Nested persistent transactions: NOT SUPPORTED
   Distributed transactions: SUPPORTED.    Size: 200000
```

Figure 63. 1st Refinement Architecture Properties

```
Intra-Cell Calls Made:     36541    Time: 55359
Intra-Cell Calls Received: 36541    Time: 55360
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 155    Time: 703
RPCs Made:                 786    Time: 13098
RPCs Received:             786    Time: 14289
Creations:       Objects: 14348    Cells: 75
                 Time: 458877
Activations:     Objects: 4849    Cells: 174
                 Time: 74205
Passivations:    Objects: 6361    Cells: 403
                 Time: 111823
Commits:         Objects: 6361    Cells: 304
                 Time: 88366
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  3911203
```

Overall Architecture Summary Profile

```
Intra-Cell Calls Made:     34519    Time: 52299
Intra-Cell Calls Received: 34519    Time: 52300
Inter-Cell Calls Made:     0    Time: 0
Inter-Cell Calls Received: 150    Time: 681
RPCs Made:                 536    Time: 8932
RPCs Received:             250    Time: 4544
Creations:         Objects: 14348    Cells: 75
                   Time: 218756
Activations:       Objects: 4849    Cells: 174
                   Time: 19966
Passivations:      Objects: 330    Cells: 249
                   Time: 8545
Total Execution Time: 1370706
```

Client Summary Profile

```
Intra-Cell Calls Made:     2022    Time: 3060
Intra-Cell Calls Received: 2022    Time: 3060
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 5    Time: 22
RPCs Made:                 250    Time: 4166
RPCs Received:             536    Time: 9745
Creations:         Objects: 0    Cells:  0
                   Time: 0
Activations:       Objects: 0    Cells:  0
                   Time: 0
Passivations:      Objects: 6031    Cells:  154
                   Time: 22942
Total Execution Time: 158043
```

Object Server Summary Profile

```
Creations:    Objects:  14348    Cells:  75
              Time:  240121
Activations:  Objects:  4849    Cells:  174
              Time:  54239
Passivations: Objects:  6361    Cells:  403
              Time:  80336
Commits:      Objects:  6361    Cells:  304
              Time:  88366
Aborts:       Objects:  0    Cells:  0
              Time:  0
Total Execution Time:   2382454
```

Persistent Cell Manager Summary Profile

Figure 64. 1st Refinement Simulation Profiles

Table 5. Multi-Use Performance Profile and Requirements

Architectural Component	Simulated Performance (transactions per minute)	Required Performance (transactions per minute)
Client	6,600 tpm	1.3 tpm
Object Server	57,000 tpm	8,000 tpm
Persistent Cell Manager	3,800 tpm	8,000 tpm

From the simulation data in Figure 64., we can calculate the transaction throughput rate for the clients, object servers, and persistent cell managers. The simulation performed 150 transactions. The total execution time for each architectural component is shown in the figure: 1,370,000 microseconds for a client, 158,000 microseconds for the object server, and 2,380,000 microseconds for the persistent cell manager. The translation of these values into transactions per minute is shown in the second column of Table 5.

Comparing the simulated performance in column two to the required values in column three, we see that the client and object server in the architecture both exceed the requirements, but the persistent cell manager provides less than 50% of the required capacity. To address this problem, the *Global Partition Contention* requirement variable value must be set to *HIGH* for the Cluster Set Partition *CSP* (corresponding to the cells managed by the persistent cell manager). This will result in a separate persistent cell manager processor being dedicated to each workflow management diagram repository rather than a single persistent cell manager processor shared among all workflow management diagram repositories.

7.2.3.3. Final Refinement on OODB Requirements – Replication in the Architecture

Section 5.1.8. showed that setting the global partition contention to high on a cluster set partition increases capacity and availability for the OODB. Therefore, since we are setting the value to high in order to increase capacity, we also get higher availability. With this in mind, we decided to increase availability in the two other global contention requirements: *Global Cluster Set Contention* on the *Sparse Cluster Sets* and *Global Transaction Throughput*. The resulting requirement variable are shown in Figure 65.

The mappings were invoked to produce the architecture instance for simulation from the requirements. The resulting architecture summary is shown in Figure 66. Note that the *Integration* on the *Object Server Set*, the *Persistent Cell Manager*, and the *Transaction Manager* now have the value of *DEDICATED*. This means that each repository that is created for a workflow diagram will have a dedicated processor for the object server, persistent cell manager, and transaction manager. The corresponding architecture properties are shown in Figure 67. The second and third lines from the top show that each repository will now have three dedicated processes and 8.8 megabytes of runtime process space associated with it. This increase in resources provides the desired increase in capacity that we needed and the increase in availability that we requested.

The resulting simulation profile summary is shown in Figure 68. The results are nearly identical to the previous simulation since the architectural properties that we changed – multi-user capacity and availability – are not metered in the simulator. With no further requirement refinements suggested by the architectural properties and simulation profiles, we use this set of requirements as the baseline requirements for the OODB in the workflow management system.

```
Long Transactions: FALSE

Concurrent Users: TRUE

Locality Clusters:
   Cluster: Root
      Roles:
         Role root_root is Class VPML_Root
         Role ready is Class ReadyNode
         Role active is Class ActiveNode
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: Composite
      Roles:
         Role comp_root is Class CompositeTask
         Role prim is Class PrimitiveTask
         Role res is Class Resource
         Role prod is Class Product
         Role pset is Class ProductSet
         Role pmem is Class ProductMember
         Role pflo is Class ProductFlow
         Role fchn is Class FlowChain
      Utilization: LOW
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

   Cluster: PContent
      Roles:
         Role pcont_root is Class ProductContent
         Role lnode is Class Node
         Role rnode is Class Node
      Utilization: HIGH
      Size: SMALL
      Primary Reliability:
         HIGH
      Secondary Reliability:
         NONE

Sparse Cluster Sets
   Cluster Set SCS
      Clusters:
         Composite
      Global Cluster Set Contention: HIGH

Dense Cluster Set DCS
   Clusters:
      Root
      PContent
   Global Cluster Set Contention: LOW

Cluster Set Partitions
   Partition CSP
      Cluster Sets:
         DCS
         SCS
      Global Partition Contention: HIGH

Global Transaction Throughput: HIGH

Reentrant Cluster Sets: <Automatically Derived. Internal Representation Not Shown>
```

Figure 65. Final Refinement of Requirement Variable Values

```
Reentrant Client:
   Object and Cell Services.
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED
      Write-through policy: EAGER
   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root
            Role ready is Class ReadyNode
            Role active is Class ActiveNode
      Cell: PContent
         Role Declarations:
            Proxy Role pcont_root is Class ProductContent
            Role lnode is Class Node
            Role rnode is Class Node
   Root Role: Root.root_root

Object Server Set.   Integration: DEDICATED

   Reentrant Object Server: SCS
      Object and Cell Services.
         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: NOT SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED
         Write-through policy: EAGER
      Cell Allocations:
         Cell: Composite
            Role Declarations:
               Proxy Role comp_root is Class CompositeTask
               Role prim is Class PrimitiveTask
               Role res is Class Resource
               Role prod is Class Product
               Role pset is Class ProductSet
               Role pmem is Class ProductMember
               Role pflo is Class ProductFlow
               Role fchn is Class FlowChain

Persistent Cell Manager Set.   Integration: DEDICATED

   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
         SCS
      Write-through: EAGER
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: SUPPORTED
      Object locking: SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: DEDICATED
   Object server set administration: DEDICATED
   Persistent cell manager set administration: DEDICATED

Transaction Manager.   Integration: DEDICATED
   Nested persistent transactions: NOT SUPPORTED
   Distributed transactions: SUPPORTED
```

Figure 66. Final Refinement Architecture Summary

```
Size per client: 2560360 Bytes
Number of processes per active repository: 3
Total size of processes per repository: 8781360 Bytes
Number of fixed (shared) processes: 1
Total size of shared processes: 200000 Bytes

Reentrant Client:
Total size: 2560360 Bytes
Size accounted for by methods: 60360 Bytes
Size accounted for by cache: 1000000 Bytes
   Object and Cell Services.
   Total size: 1250000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: NOT SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED.    Size: 50000
      Write-through policy: EAGER

Object Server Set.   Integration: DEDICATED
Total processes per object server set: 1
Total size per object server set: 5631360 Bytes

   Reentrant Object Server: SCS
   Total size: 5631360 Bytes
   Size accounted for by methods: 31360 Bytes
   Size accounted for by cache: 4000000 Bytes
      Object and Cell Services.
      Total size: 4350000 Bytes
         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: NOT SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED.    Size: 50000
         Write-through policy: EAGER

Persistent Cell Manager Set.   Integration: DEDICATED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2450000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2450000 Bytes
      Processes allocated to PCM:
         CLIENT
         SCS
      Persistent nested transactions: NOT SUPPORTED
      Distributed transactions: SUPPORTED.    Size: 200000
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
Total size: 200000 Bytes
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: DEDICATED
   Object server set administration: DEDICATED
   Persistent cell manager set administration: DEDICATED

Transaction Manager.   Integration: DEDICATED
Total size: 700000 Bytes
   Nested persistent transactions: NOT SUPPORTED
   Distributed transactions: SUPPORTED.    Size: 200000
```

Figure 67. Final Refinement Architecture Properties

```
Intra-Cell Calls Made:     36541    Time: 55359
Intra-Cell Calls Received: 36541    Time: 55360
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 155    Time: 703
RPCs Made:                 786    Time: 13098
RPCs Received:             786    Time: 14289
Creations:        Objects: 14348    Cells: 75
                  Time: 458877
Activations:      Objects: 4849    Cells: 174
                  Time: 74205
Passivations:     Objects: 6361    Cells: 403
                  Time: 111823
Commits:          Objects: 6361    Cells: 304
                  Time: 88366
Aborts:           Objects: 0    Cells: 0
                  Time: 0
Total Execution Time:  3921524
```

Overall Architecture Summary Profile

```
Intra-Cell Calls Made:     34519    Time: 52299
Intra-Cell Calls Received: 34519    Time: 52300
Inter-Cell Calls Made:     0    Time: 0
Inter-Cell Calls Received: 150    Time: 681
RPCs Made:                 536    Time: 8932
RPCs Received:             250    Time: 4544
Creations:         Objects: 14348    Cells: 75
                   Time: 218756
Activations:       Objects: 4849    Cells: 174
                   Time: 19966
Passivations:      Objects: 330    Cells: 249
                   Time: 8545
Total Execution Time: 1370706
```

Client Summary Profile

```
Intra-Cell Calls Made:     2022    Time: 3060
Intra-Cell Calls Received: 2022    Time: 3060
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 5    Time: 22
RPCs Made:                 250    Time: 4166
RPCs Received:             536    Time: 9745
Creations:         Objects: 0    Cells:  0
                   Time: 0
Activations:       Objects: 0    Cells:  0
                   Time: 0
Passivations:      Objects: 6031    Cells:  154
                   Time: 22942
Total Execution Time: 168364
```

Object Server Summary Profile

```
Creations:    Objects:  14348    Cells:  75
              Time:  240121
Activations:  Objects:  4849    Cells:  174
              Time:  54239
Passivations: Objects:  6361    Cells:  403
              Time:  80336
Commits:      Objects:  6361    Cells:  304
              Time:  88366
Aborts:       Objects:  0    Cells:  0
              Time:  0
Total Execution Time:   2382454
```

Persistent Cell Manager Summary Profile

Figure 68. Final Refinement Simulation Profiles

7.2.3.4. Summary for Experiment 2 Requirements Definition

The estimated development cost for defining and refining the OODB requirements for the application were:

- one-half staff*day to define UFO requirement variable values for the application
- one-quarter staff*day to map from requirements to architecture simulator, plus run the simulation
- one-quarter staff*day to review the simulation profile
- one staff*day to identify the first refinements to the requirements and to run the simulation
- one staff*day to identify the final refinements to the requirements and to run the simulation

Thus, we estimate the development cost to be about three days for a developer moderately familiar with the UFO tool. With this minimal level of effort we were able to demonstrate that the defined set of OODB requirements for the workflow management system application are valid. In the next chapter we further explore and make estimates on the relative cost savings of the UFO versus conventional approach.

7.2.4. Selecting an Off-the-Shelf OODB for the Workflow Management System

As in the first experiment, we next used the UFO tool to select among the three off-the-shelf OODBs (ObjectStoreLite, Objectivity, and ITASCA). We use the baseline requirements from the previous section as the standard for comparison.

7.2.4.1. ObjectStoreLite

We first set the UFO tool to target the ObjectStoreLite OODB architecture. When the mapping from requirement variable values to architectural parameter values was invoked, the following inconsistencies were reported:

```
** INCONSISTENCY: ObjectStoreLite does not support dedicated transaction managers.
...
** INCONSISTENCY ** ObjectStoreLite does not support object locking.
...
** INCONSISTENCY ** ObjectStoreLite always uses embedded repository manager.
...
** INCONSISTENCY: ObjectStoreLite supports only NON-REENTRANT clients.
...
** INCONSISTENCY: ObjectStoreLite doesn't support computational object servers.
...
** INCONSISTENCY: ObjectStoreLite does not support dedicated PCMs.
...
```

The reported inconsistencies imply that the ObjectStoreLite architecture is completely inappropriate for the workflow management application. For example, the first reported inconsistency arise from the fact that the ObjectStoreLite architecture doesn’t support multiple users. Therefore, we immediately eliminate ObjectStoreLite from consideration for this application.

7.2.4.2. Objectivity

Next, we set the target to Objectivity and invoked that mapping from requirement variables to architectural parameters. The following summarized inconsistencies were reported:

```
** INCONSISTENCY: Objectivity supports only NON-REENTRANT clients.
...
** INCONSISTENCY: Objectivity does not use distributed transactions.
...
** INCONSISTENCY: Objectivity doesn't support computational object servers.
...
** INCONSISTENCY: Objectivity does not support dedicated PCMs.
...
** INCONSISTENCY: Objectivity does not support dedicated transaction managers.
...
** INCONSISTENCY: Objectivity always supports long transactions.
...
** WARNING ** Objectivity always uses embedded repository managers.
...
```

These inconsistencies imply that Objectivity does not support some of the important performance-oriented baseline requirements. Therefore, we eliminated Objectivity from consideration for this application.

7.2.4.3. ITASCA

Finally, we set the target to ITASCA and invoked the mapping requirement variables to architectural parameters. The following inconsistencies were reported:

```
** INCONSISTENCY: ITASCA always supports long transactions.
...
** INCONSISTENCY: ITASCA does not support dedicated transaction managers.
...
** INCONSISTENCY: ITASCA does not support dedicated object server sets.
...
** INCONSISTENCY: ITASCA does not support dedicated PCMs.
...
```

The first inconsistency indicates that ITASCA provides extra functionality that is not need for this application, but this does not preclude its use. The next three inconsistencies can easily be addressed by having a separate ITASCA installation for each repository, thereby creating transaction managers, object servers, and persistent cell managers that are dedicated to individual repositories.

After resolving the inconsistencies, the mappings were invoked to produce the ITASCA architecture instance for simulation from the requirements. The resulting architecture summary is shown in Figure 69. The architecture properties are shown in Figure 70. The simulation profile summary is shown in Figure 71.

7.2.4.4. The OODB Simulation Results for the Workflow Management System

Table 6. provides size and performance profiles on the baseline architecture and ITASCA. This size and performance data comes from the architectural properties and simulation profiles summarized in Figure 67. & Figure 68., and Figure 70. & Figure 71.

```
Reentrant Client:
   Object and Cell Services.
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED
      Write-through policy: EAGER
   Cell Allocations:
      Cell: Root
         Role Declarations:
            Proxy Role root_root is Class VPML_Root
            Role ready is Class ReadyNode
            Role active is Class ActiveNode
      Cell: PContent
         Role Declarations:
            Proxy Role pcont_root is Class ProductContent
            Role lnode is Class Node
            Role rnode is Class Node
   Root Role: Root.root_root

Object Server Set.   Integration: SHARED

   Threaded Object Server: SCS
      Object and Cell Services.
         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED
         Write-through policy: EAGER
      Cell Allocations:
         Cell: Composite
            Role Declarations:
               Proxy Role comp_root is Class CompositeTask
               Role prim is Class PrimitiveTask
               Role res is Class Resource
               Role prod is Class Product
               Role pset is Class ProductSet
               Role pmem is Class ProductMember
               Role pflo is Class ProductFlow
               Role fchn is Class FlowChain

Persistent Cell Manager Set.   Integration: SHARED

   Persistent Cell Manager. CSP
      Processes allocated to PCM:
         CLIENT
         SCS
      Write-through: EAGER
      Persistent nested transactions: SUPPORTED
      Distributed transactions: SUPPORTED
      Object locking: SUPPORTED
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: SHARED
   Object server set administration: SHARED
   Persistent cell manager set administration: SHARED


Transaction Manager.   Integration: SHARED
   Nested persistent transactions: SUPPORTED
   Distributed transactions: SUPPORTED
```

Figure 69. ITASCA Architecture Summary

```
Size per client: 2660360 Bytes
Number of processes per active repository: 0
Total size of processes per repository: 0 Bytes
Number of fixed (shared) processes: 4
Total size of shared processes: 10331360 Bytes

Reentrant Client:
Total size: 2660360 Bytes
Size accounted for by methods: 60360 Bytes
Size accounted for by cache: 1000000 Bytes
   Object and Cell Services.
   Total size: 1350000 Bytes
      Threads: SINGLE
      Object locking: SUPPORTED
      Persistent nested transactions: SUPPORTED
      Cache size: 1000000 Bytes
      Cache replacement policy: LRU
      Distributed transactions: SUPPORTED.    Size: 50000
      Write-through policy: EAGER

Object Server Set.   Integration: SHARED
Total processes per object server set: 1
Total size per object server set: 5981360 Bytes

   Threaded Object Server: SCS
   Total size: 5981360 Bytes
   Size accounted for by methods: 31360 Bytes
   Size accounted for by cache: 4000000 Bytes
      Object and Cell Services.
      Total size: 4450000 Bytes
         Threads: MULTI
         Object locking: SUPPORTED
         Persistent nested transactions: SUPPORTED
         Cache size: 4000000 Bytes
         Cache replacement policy: LRU
         Distributed transactions: SUPPORTED.    Size: 50000
         Write-through policy: EAGER

Persistent Cell Manager Set.   Integration: SHARED
Total processes per persistent cell manager set: 1
Total size per persistent cell manager server set: 2950000 Bytes

   Persistent Cell Manager: CSP
   Write-through: EAGER
   Total size: 2950000 Bytes
      Processes allocated to PCM:
         CLIENT
         SCS
      Persistent nested transactions: SUPPORTED.    Size: 500000
      Distributed transactions: SUPPORTED.    Size: 200000
      Object Locking: SUPPORTED.    Size: 250000
      Backup policy: NO AUTOMATED BACKUPS

Repository Manager.   Integration: SHARED
Total size: 200000 Bytes

   Maximum number of concurrent clients on a repository: 1000000
   Maximum number of repositories per installation: 1000000000
   Transaction manager administration: SHARED
   Object server set administration: SHARED
   Persistent cell manager set administration: SHARED

Transaction Manager.   Integration: SHARED
Total size: 1200000 Bytes
   Nested persistent transactions: SUPPORTED.    Size: 500000
   Distributed transactions: SUPPORTED.    Size: 200000
```

Figure 70. ITASCA Architecture Properties

```
Intra-Cell Calls Made:     36541    Time: 55359
Intra-Cell Calls Received: 36541    Time: 55360
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 155    Time: 703
RPCs Made:                 786    Time: 13098
RPCs Received:             786    Time: 14289
Creations:       Objects: 14348    Cells: 75
                 Time: 458877
Activations:     Objects: 4849    Cells: 174
                 Time: 74205
Passivations:    Objects: 6361    Cells: 403
                 Time: 111823
Commits:         Objects: 6361    Cells: 304
                 Time: 88366
Aborts:          Objects: 0    Cells: 0
                 Time: 0
Total Execution Time:  4531705
```

Overall Architecture Summary Profile

```
Intra-Cell Calls Made:     34519    Time: 52299
Intra-Cell Calls Received: 34519    Time: 52300
Inter-Cell Calls Made:     0    Time: 0
Inter-Cell Calls Received: 150    Time: 681
RPCs Made:                 536    Time: 8932
RPCs Received:             250    Time: 4544
Creations:        Objects: 14348    Cells: 75
                  Time: 218756
Activations:      Objects: 4849    Cells: 174
                  Time: 19966
Passivations:     Objects: 330    Cells: 249
                  Time: 8545
Total Execution Time: 1458551
```

Client Summary Profile

```
Intra-Cell Calls Made:     2022    Time: 3060
Intra-Cell Calls Received: 2022    Time: 3060
Inter-Cell Calls Made:     5    Time: 7
Inter-Cell Calls Received: 5    Time: 22
RPCs Made:                 250    Time: 4166
RPCs Received:             536    Time: 9745
Creations:        Objects: 0    Cells:  0
                  Time: 0
Activations:      Objects: 0    Cells:  0
                  Time: 0
Passivations:     Objects: 6031    Cells:  154
                  Time: 22942
Total Execution Time: 204485
```

Object Server Summary Profile

```
Creations:    Objects:  14348    Cells:  75
              Time:  240121
Activations:  Objects:  4849    Cells:  174
              Time:  54239
Passivations: Objects:  6361    Cells:  403
              Time:  80336
Commits:      Objects:  6361    Cells:  304
              Time:  88366
Aborts:       Objects:  0    Cells:  0
              Time:  0
Total Execution Time:   2868669
```

Persistent Cell Manager Summary Profile

Figure 71. ITASCA Simulation Profiles

Table 6. OODB Properties for the Workflow Diagram Editor Application

OODB Model	Runtime Client Size	Number of Dedicated & Shared Processes	Runtime Dedicated & Shared Process Size	Total Runtime Size	Total Execution Time
Baseline	2.6 Mbytes	3 1	8.8 Mbytes 0.2 Mbytes	11.6 Mbytes	3.9 Seconds
ITASCA	2.7 Mbytes	3* 1* * Dedicated by convention of 1 repository per installation	10.1 Mbytes 0.2 Mbytes	13.0 Mbytes	4.5 Seconds

The ITASCA runtime process size is 12% larger than the baseline, and the execution time is 15% greater. This is attributed to excess functionality, such as the support for long transactions.

7.2.4.5. Summary for Experiment 2 OODB Selection

The estimated development cost for simulating and comparing the off-the-shelf OODB system properties for the three OODBs were:

- one staff*day to execute the mappings from requirements to architecture simulator and address inconsistencies reported by the tool
- one-half staff*day total to run the two ITASCA simulations
- one staff*day to review the initial ITASCA simulation profile and do the multi-user analysis

Thus, we estimate the cost for evaluating the three OODBs to be about two and one half days for a developer moderately familiar with the UFO tool. With this minimal level of effort we were able to characterize how well each of the OODBs satisfied the requirements of the workflow editor application.

In the next chapter we further explore and make estimates on the relative cost savings of the UFO versus conventional approach.

Chapter 8. Analysis

In this chapter we look back at our hypothesis from Section 1.4. and the experimental results from the previous chapter in order to analyze the degree to which the hypothesis is satisfied. Our analysis in this chapter is divided into two parts, supporting the hypothesis for requirements definition and supporting the hypothesis for off-the-shelf OODB selection. These two sections correspond to the two major functions of the UFO tool, each of which was tested in each of the two experiments that we carried out.

As discussed in Section 3.6., the level of effort required to quantify and accurately compare conventional software development costs and conformance values to those of UFO is beyond the scope of resources available for this study. Therefore, we report here on single data points and estimates from the UFO implementation and experiments in order to provide rough approximations for cost and conformance values. We rely on the reader's intuition and possible experience in defining requirements and implementing software systems as we discuss the cost and conformance issues associated with conventional and UFO software development technology.

8.1. Supporting the Hypothesis for Requirements Definition

The hypothesis for the requirements definition portion of the UFO technology states that we provide an engineering approach for converging on a set of requirements that results in both a relatively low development cost and a relatively high conformance between requirements and OODB simulation properties. In this section, we argue that our work supports this hypothesis. The projected cost/conformance profiles for requirements definition are illustrated in Figure 73. The vertical axis in this diagram is the typical cost for defining requirements for a system instance, while the horizontal axis represents the typical degree of conformance between the system properties of a system instance and the requirements. The conformance using requirements definition is projected to be as high as custom system development while the cost is projected to be considerably lower than both the conventional general purpose and custom system approach. We justify this projection in the following sections.

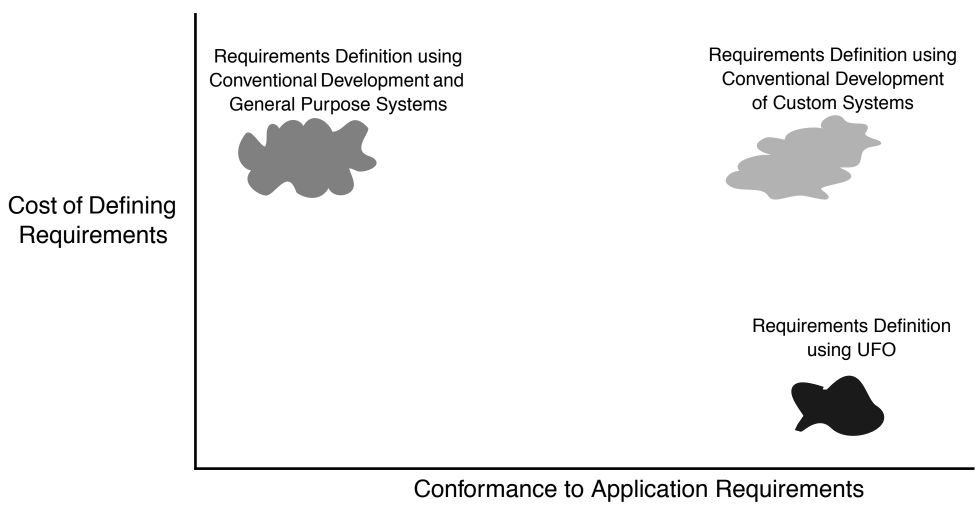


Figure 72. Cost/Conformance Profile for Requirements Definition

8.1.1. Requirements Definition Cost Estimates

With the conventional approach to defining requirements, the associated cost come primarily from three sources: *discovery cost*, *common specification cost*, and *variable specification cost*. The *discovery cost* is the cost incurred to explore the domain for the class of systems, to identify all of the relevant requirement issues, and to identify how each requirement issue applies to a particular instance. Since conventional approaches to custom system development or general purpose systems typically start with a "clean slate" where developers have limited understanding of the domain, the discovery costs can be significant.

Based on our experiences developing UFO, the conventional *discovery cost* for the portion of the OODB domain considered for UFO is estimated to be approximately 2 months for an experienced software architect with no previous experience in the OODB domain. The discovery activities include reading research papers, books, and commercial literature on OODB objectives, concepts, functionality, architectural idioms, and implementation tricks, plus the possibility of cursory explorations with off-the-shelf OODB systems. The deliverable from the discovery phase would be a domain model, a high-level reference architecture, or similar items.

In addition to the discovery costs are the cost associated with specifying the requirements in a clear, complete, and accurate form so that developers can use the requirements in subsequent design and implementation. With conventional approaches there is no distinction between the specifications that are common for a class of systems and the specifications that are variable among the different instances in the class. However, we split the costs into *common specification cost* and *variable specification cost* to aid in later comparisons of conventional approaches to defining requirements and the UFO approach to defining requirements.

Based on our experiences developing UFO, the conventional *common specification cost* and *variable specification cost* for an OODB instance is estimated to be approximately 2 months each for an experienced software architect with no previous experience in the OODB domain. The requirements definition activities include defining required functionality, boundaries and limitations, associations, and so forth. The deliverable would be natural language requirements document on the order of 100 pages, clearly defining what is required of the system instance for each of the requirement issues in the domain.

The UFO tool significantly reduces the conventional cost of defining requirements in the following ways:

- <u>*Reduced discovery cost*</u>. All requirement issues for the class of systems have been identified, explored, and captured in the UFO tool. Therefore, the *discovery cost* in requirements definition is zero.

- <u>*Reduced common specification cost*</u>. The common requirements are the same and predefined for all instances in the UFO tool. Therefore, the *common specification cost* in requirements definition is zero.

- <u>*Reduced variable specification cost*</u>. The UFO tool provides the following support for efficiently specifying variable requirements:
  - guided requirements definition
  - automatic extraction of some requirement variable values
  - feedback on system properties from modeling and simulation for refining requirements
  - automated suggestions for refining requirements

  In the UFO experiments we estimated the *variable specification cost* using the UFO tool to be approximately 2 days with the workflow editor (see Section 7.1.3.1.) and 3 days for the workflow manager (see Section 7.2.3.4.).

8.1.2. Requirements Definition Conformance Estimates

Conformance is primarily determined by three factors: *completeness*, *accuracy*, and *satisfaction*. Each of these have values ranging from 0% to 100%. *Completeness* indicates the degree to which all requirements issues have been addressed in the set of requirements for an instance. For example, if developers overlook or omit a requirement, then *completeness* is less than 100%. *Accuracy* indicates the degree to which the set of requirements specified for an instance accurately reflects the intent of the developers. For example, if developers misunderstand a requirement and specify it incorrectly, then accuracy is less than 100%. *Satisfaction* indicates the degree to which the system properties for an instance satisfy the set of requirements for the instance. For example, if a system property is inconsistent or out of bounds for the requirement on that system property, then *satisfaction* is less than 100%.

The UFO tool increases the potential for high conformance in the following ways:

<u>Completeness</u>. All of the requirement issues for the class of systems have been identified, explored, and captured in the UFO technology. This assures that the *completeness* of the requirements is near 100%. Note that this assumes a mature and stable UFO tool, where omissions have been detected and addressed.

<u>Accuracy</u>. Since the common requirements are the same and predefined for all OODB instances in UFO, the *accuracy* for common requirements will be near 100% for a mature and stable UFO technology. That is, since the common requirements are predefined, there is no opportunity for developers to incorrectly specify a common requirement.

For variable requirements, the UFO tool increases *accuracy* by automatically deriving some requirement values from source code and by guiding definition of the remaining requirements. For example, the requirement variable template editor provides developers with legal values for each requirement variable and can provide on-line help to help developers select the appropriate values.

Accuracy is also increased through feedback from modeling and simulation. Unexpected size, performance, and functionality that is found in system property profiles can often be accounted for in inaccurately specified requirements. The UFO tool can increase *accuracy* by:

- providing modeling and simulation feedback on system properties to determine the accuracy of requirements

- providing feedback in the form of suggestions on how to refine requirement values in order to increase accuracy

For example, in the first experiment on the workflow editor described in Section 7.1., we illustrated how to optimize cluster optimization using feedback from simulation profiling. In the second experiment on the workflow manager, described in Section 7.2., we illustrated how to identify overloaded processors and how to modify requirements to more accurately reflect the system requirements.

<u>Satisfaction</u>. Since the common requirements are addressed with predefined, common structures in the software architectures for all instances in UFO, the *satisfaction* for common requirements will be near 100% for a mature and stable UFO technology. That is, since the common requirements are satisfied by predefined structures in the software architecture, there is no opportunity for developers to incorrectly implement a common requirement.

Similarly, variable requirements in the UFO technology will have a near 100% value for *satisfaction* since the mappings from requirements to software architectures will assure that the requirement is properly addressed in the architecture of the instance.

The primary source of variability in the *satisfaction* value comes from the intangible requirements on size and performance. There are implicit desiderata in software for a system to be negligible in size and instantaneous in response. However, a developer's acceptable threshold for system size or system performance may not always be achievable. In these cases, the UFO tool helps to increase *satisfaction* by providing feedback on system properties such as size and performance. This increase may simply mean that developers must adopt more realistic and practical expectations on size and performance.

8.2. Supporting the Hypothesis for Off-the-Shelf OODB Selection

The hypothesis for the off-the-shelf OODB selection portion of the UFO technology says that we provide an engineering approach for selecting an off-the-shelf system that results in both a relatively low development cost and a relatively high conformance between baseline requirements and off-the-shelf OODB simulation properties. In this section, we argue that our work supports this hypothesis. The projected cost/conformance profiles for OODB selection are illustrated in Figure 73. The vertical axis in this diagram is the typical cost for selecting an OODB for a particular application, while the horizontal axis represents the typical degree of conformance between the system properties of a system instance and the application requirements. We justify these results in the following sections.

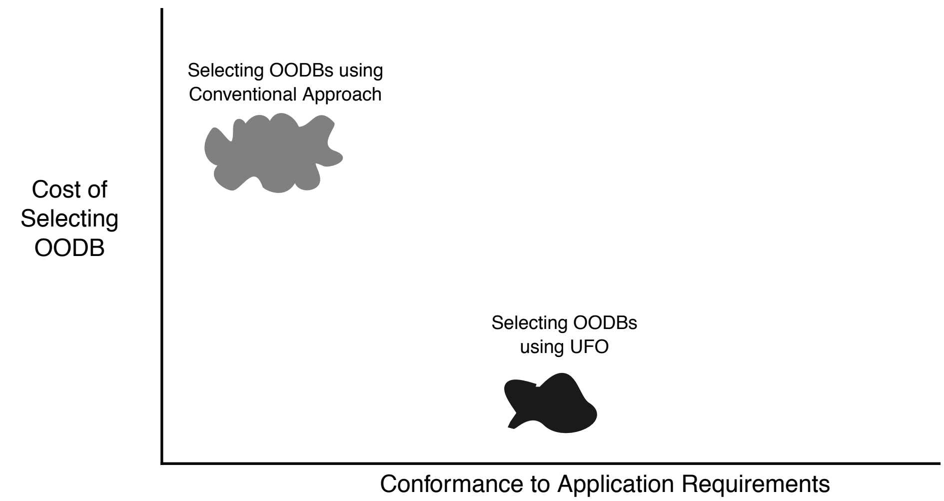


Figure 73. Cost/Conformance Profile for Selecting Off-the-shelf OODBs

8.2.1. Off-the-Shelf OODB Selection Cost Estimates

The costs associated with the conventional approach to selecting off-the-shelf OODBs come primarily from three sources, *search cost*, *profiling cost*, *collation cost*.

The *search cost* is the cost associated with locating the available off-the-shelf systems, reviewing available literature and information, and pre-selecting the collection of appropriate candidates. The *profiling cost* is the cost of acquiring, installing, and benchmarking the candidates in order to collect relevant profile data. The *collation cost* is the cost of doing the comparative analysis, ranking, and selection of an off-the-shelf candidate based on the profile data.

When using the UFO tool for selecting off-the-shelf OODBs, the selection costs are reduced in the following ways:

- *Search cost*. All of the off-the-shelf OODBs that will be evaluated have already been identified and captured in the tool. This, of course, assumes a mature UFO that contains a relatively complete set of off-the-shelf options. Therefore, the *search cost* in the selection cost equation is zero.

- *Profiling cost*. The UFO tool provides the following support for profiling:
  - feedback on inconsistencies between a baseline OODB architecture and the off-the-shelf OODBs being profiled
  - feedback on system properties from modeling and simulation
  - automated profile analysis

  This tool support significantly reduces the *profiling cost* in the selection cost equation. Later in this section we will estimate these values for our experiments.

- *Collation cost*. The cost of comparing the different off-the-shelf OODB modeling and simulation results from the UFO tool is reduced by presenting modeling and simulation data for each OODB candidate in a consistent and uniform way.

Based on the UFO experiments and practical experience evaluating OODBs we can roughly estimate these values. We estimate the *search cost* to be approximately 1 month to identify 3 to 5 good candidates. The *profiling cost* using conventional evaluation of selected off-the-shelf OODBs would be on the order of 6 months for experienced software engineers without previous experience with OODBs. The *collation cost* associated with taking the profile results and ordering the OODB candidates by their relative merit for a given application is estimated to be approximately 1 month.

In the workflow editor experiment using UFO, we estimated the *profiling cost* to be 3 days for 3 candidate OODBs and the *collation cost* to be 1 day (see Section 7.1.4.5.). In the workflow manager experiment, the UFO estimates were 1.5 days for *profiling cost* and 1 day for *collation cost* (see Section 7.2.4.5.).

8.2.2. Off-the-Shelf OODB Selection Conformance Estimates

The conformance equation for OODB selection will be essentially the same as with requirements definition, with the additional consideration that the *satisfaction* may be further reduced by any mismatches between the baseline requirements and the off-the-shelf OODB that most closely matches the baseline.

In Section 7.1.4.4. for the workflow editor we selected an off-the-shelf OODB that perfectly matched the baseline OODB. Table 4. in that section shows that ObjectStoreLite has the same *Total Runtime Size* and *Total Execution Time* as the baseline OODB. Therefore, *satisfaction* in this case is not reduced by inconsistencies.

In Section 7.2.4.4. for the workflow manager we found an off-the-shelf OODB with a close but not perfect match to the baseline OODB. Table 6. in that section shows that ITASCA has a *Total Runtime Size* that is 12% larger than the baseline and a *Total Execution Time* 15% greater than the baseline. Therefore, the *satisfaction* of the baseline requirements by ITASCA is not perfect. However, the UFO tool helps find the off-the-shelf OODB instance with the highest degree of satisfaction even if a perfect match is not available.

Chapter 9. Evaluation and Discussion

In this chapter we review and evaluate our experiences in defining software architecture modeling and simulation technology, aimed at improving the development cost/conformance profiles within a class of similar systems. We present some of the more significant lessons that we learned from these experiences and highlight some of the key issues associated with creating and using the technology.

There are six major areas that are examined in the following subsections. First, we examine the issues and challenges associated with capturing the relationships between requirements, architectures, and system properties. Recall that these relationship are used to define the mappings in the tool and therefore play a central role in the approach. The second section deals with our choice of discriminators for the OODB domain, the tradeoffs involved in making these choices, and why the choices were appropriate for our work. The next section examines the effectiveness of the abstractions that we used in our technology. We argue that abstraction is the unifying principal behind the software development efficiencies offered by UFO. The next two subsections deal with the practicality and extensibility issues associated with our technology, projecting what would be involved in taking UFO from a research prototype to a production quality software development tool. In the final section we evaluate some of the challenges and successes in our low-level experiences implementing the UFO tool.

9.1. Capturing Relationships Between Requirements, Architectures and Properties

The key to our solution for improved cost/conformance profiles is to capture the conventional relationships between system requirements, software architectures, and system properties in an engineering support tool. Capitalizing on these relationships, engineering support tools help developers to make principled choices within the design space for a class of systems. In this section we review our experiences in capturing these relationships and exploiting these relationships in a software architecture modeling and simulation tool.

9.1.1. The Triad of System Requirements, Architectures, and Properties

Early stages of this research focused only on system requirements, system architectures, and the relationships between them. The hope was that developers could accurately specify their requirements *a priori*, and then the relationships between requirements and architectures could be used to create a deterministic mapping from requirements to system architectures with system properties that fully satisfied the requirements. However, after some experience we found that it was not always feasible for developers to fully understand and accurately specify system requirements, even with the help of an engineering support tool that guided the definition of requirement variable values.

For example, with the UFO tool, developers might predict cluster utilization to be low on a particular cluster of objects. However, the application simulation might reveal heavy utilization of the cluster that caused performance bottlenecks in the architecture that were not predicted in the requirements.

To help developers more rapidly and accurately test and refine their requirements, we replaced the binary relationship between requirements and architectures with the ternary relationship between requirements, architectures, and properties. This augmented our early view with an explicit representation of system properties, a path to extract properties from architectures, and a feedback path from system properties back to requirements.

We found that the combination of requirements, architectures, and system properties allows for significantly better engineering support for rapidly deploying high-conformance architectures than does simply considering requirements and architectures. We believe that this is because the cycle of relationships between requirements, architectures, and system properties more accurately reflects and supports the way that conventional software is developed in practice – experiences and observations regarding software system properties are commonly used to iteratively refine requirements and optimize software system architectures.

9.1.2. Relationship between Requirements and Architectures

In our initial attempt at capturing the relationships between OODB requirements and OODB architectures, we used a single mapping from requirement concepts to architecture realizations. This mapping was a complex data flow graph that we referred to as a dependency graph. Closer examination of the dependency graph indicated that the mapping could be decomposed into four simpler mappings, each addressing a separate concern. These are the four mapping phases illustrated in Figure 4. on page 7:

- Elicit requirements. Automatic extraction of requirement variable values from the application source code and definition of requirement variable values by developers.
- Derive architectural parameters. From requirement variable values to architectural parameter values.
- Instantiate software architectures. From architectural parameter values to software architecture instance (instantiated configuration nodes).
- Realize software architectures. From configuration nodes to software architecture realization (OODB modeling and simulation).

The decomposed mappings simplified the encoding of relationships. The complex and convoluted dependency graph now became a composition of simple and straightforward mappings, each with separate and clear concerns. These mappings are easier to create, understand, modify, and maintain. We attribute this to the fact that the mapping phases more accurately reflect the different activities and relationships that exist in conventional development of software architectures.

9.1.3. Relationship between Architectures and System Properties

The most accurate means of extracting properties from a software system is with direct measurements and profiles on an executable system instance. However, this can be expensive if prototypes must be manually built or if multiple evaluation copies of off-the-shelf systems are purchased. For this thesis, we focus on modeling and simulation as a cost effective means of determining system properties. Although this was a pragmatic measure for completing the thesis work (i.e., we didn't have the resources to build a generator for executable software system architectures), modeling and simulation is also a realistic approach that can be applied in a cost effective way in practice.

The key to making the modeling and simulation effective is to accurately reflect the salient properties of the OODB domain. However, note that it is primarily those properties that are a function of the requirements variables that need to be addressed. That is, system properties that are common and constant in all possible system instances are much less interesting than the system properties that vary and discriminate the different system instances. For example, in OODB subsystems whose structure, size, or performance can vary depending requirement variable values, we provide a breakdown of how the components in the subsystem contribute to the overall structure, size, or performance. For subsystems that don't vary from instance to instance, no breakdown is needed or provided.

It was a challenge to "calibrate" the UFO modeling and simulation tool. Much of the information that we needed regarding size and performance of OODB architectures was not published. Furthermore, we did not have the time and resources to purchase, study, and prototype off-the-shelf OODBs in order to get representative data. Our approach to solving this problem was to rely on low-level performance simulations such as remote procedure call costs, disk access costs, and so forth. In practice more accurate property models might be required.

We simply note here that the cost and effort associated with calibrating a simulation and modeling tool such as UFO are significant. However, this up-front cost and effort is amortized over each use of the tool.

9.1.4. Relationship between Properties and Requirements

The relationship between instance properties and requirements is a test for conformance. That is, it is a test for how well a system instance satisfies the requirements of the application. Low conformance between requirements and system properties can be accounted for by

- the system instance being used in a way that was not anticipated by the developer specifying the requirements. For example, the application simulation might access data in a pattern that does not match the locality predictions of the developer specifying locality clusters.

- a developer error in understanding a requirement or in specifying a value. For example, developers might accidently specify a cluster they believe will be heavily utilized as a low utilization cluster in the requirements.

The most effective way to capitalize on the relationship between properties and requirements in a modeling and simulation tool is to automatically test for and detect inconsistencies, and then have the tool automatically suggest refinements in the requirements that might improve conformance. An example of this in the UFO tool is the test for high versus low utilization on clusters. The heuristic for testing high utilization is that the number of intra-cell, inter-cell and RPC calls received on a cell must be 50% greater than the number of object activations on the cell. This can easily be extracted from the simulation profile for an instance and compared to the high or low utilization requirement given for the cell. If the utilization for a cluster in the performance profile does not match the utilization requirement definition for the cluster, the tool will recommend that the requirement variable value be changed. This will result in the data cluster being moved to an OODB component that can more efficiently manage it.

Analysis of some system properties can be extremely complex, requiring considerable knowledge about the application and about computer systems in general. It may be intractable to implement automated analysis in the tool in these cases, so developers must manually evaluate the profiles that are produced by the tool to determine appropriate refinements to the requirements. This approach can still be effective, but is more time consuming and leaves room for oversight and errors.

For example, one could imagine an algorithm that would find the optimal clustering and partitioning requirements based on the low-level method invocation profile among data objects. The analysis, design, and implementation of that algorithm would be very complex, possibly a worthwhile thesis top of its own. Further complicating this problem, however, is that a developer doing the clustering and partitioning analysis uses knowledge about locality in the application domain and preferences of application users to find clusters that may not be globally optimal but optimize localized critical operations. To automate this type of analysis would require an expert system plus extensions to the requirements definitions portion of the tool to specify the locally critical operations.

In general, the decision as to whether to implement automated profile analysis or to use manual analysis depends on the trade-off between the cost of implementing the automated analysis up-front versus the cost of repeatedly doing the manual analysis after each simulation. As discussed in Section 5.2.3.2., we limited the automated profile analysis effort for the thesis to two performance areas. This allowed us to demonstrate the potential of automation while still containing the scope of the thesis effort.

9.1.5. Discovering Relationships prior to Encoding

We have developed techniques for encoding the concepts and relationships in requirements, architectures, and properties for a class of systems. However, one of the challenges that we did not explicitly address in this work lies in discovering the concepts and relationships in a class of systems, prior to encoding them in a simulation and modeling tool. This type of discovery process is commonly referred to as *domain analysis* in the software reuse field.

Although we did not directly focus on how to do the domain analysis to discover the relationships for a class of systems, the technology that we have developed indicates the type of information that domain analysis must produce and it suggests the activities required for producing that information. For example, the UFO modeling and simulation tool provides a framework within which to capture:

- variability in OODB requirements
- parameters that express architectural variability in OODB architectures
- a mapping between requirement variability and architectural variability
- configurations of architectural components that represent different OODB instances
- a mapping between architectural parameters and configurations of architectural components
- simulation of the architectural components in an OODB instance

These items provided us with a separation of concerns as we did the analysis on the OODB domain. Each item has a clear focus that drives the analysis. Each item has a framework for capturing and expressing information about the OODB domain.

In Chapter 11., Future Work, we also describe the potential for future work in defining a general methodology and technology for driving and capturing domain analysis in modeling and simulation tools similar to UFO.

9.2. Choosing the Discriminators for OODBs

The UFO requirement variables, architectural parameters, and configuration nodes are all discriminators among OODB instances within the class of OODB systems. That is, they represent features from the OODB domain that we deemed significant for distinguishing some OODB instances from others in the class.

Our choice of discriminators was driven by two primary objectives. Most importantly, we wanted the most significant and practical discriminators for modeling and simulating OODB architectures in the UFO tool. Second, we wanted discriminators that provided the biggest payoff from the investment we made to incorporate them in the tool. In other words, within the limited scope of this thesis we didn't want to spend time characterizing and implementing discriminators that were relatively insignificant in the OODB domain or that were overly complex relative to the value that they provided to the tool.

Fortunately these two objectives did not conflict. The discriminators that we determined to be most significant in the OODB domain, such as locality, distribution, and concurrency control, were also manageable within the scope of this thesis. The discriminators that we determined to be most complex and difficult to implement, such as multiple long transaction models and multiple OODB languages, were also less significant in discriminating among architectures that satisfy OODB requirements.

The existing UFO tool, the reference model, requirement variable definitions, architectural parameter definitions, and configuration node definitions provide a framework for analyzing and incorporating new OODB discriminators and OODB instance models. The UFO tool was designed and implemented to support evolution. Therefore, with continued effort new discriminators can be added to the UFO tool to model fine-grained details of OODBs and to keep current with the evolution of the OODB domain.

9.3. Effectiveness of Abstractions in Software Architecture Modeling and Simulation

While we have reported on several specific sources for savings in software development cost with our software architecture modeling and simulation technology, there is a unifying principle that explains why development costs are less: raising the level of abstraction for developing software.

In [13] we reported on the importance of abstraction in determining the effectiveness of software reuse techniques. Indeed, the findings there led to this thesis. In that work we claimed that the intellectual effort required to develop a software system can be minimized when a software reuse technique accomplishes the following two objectives:

1. Reduce the amount of intellectual effort required to go from the conceptualization of a system to an explicit representation of the system. This is done by moving the abstractions used in the explicit representation closer to the abstractions that developers use to reason about system implementation.

2. Reduce the amount of intellectual effort required to go from the explicit representation of the system to an executable implementation of the system. This is done by automating all or part of the mapping from explicit representation to executable realization.

The software architecture modeling and simulation technology that we describe in this thesis is a software reuse technique that focuses on these two objectives.

The first abstraction objective is accomplished through the requirements definition part of the UFO tool. Since it is typical for developers to initially reason about software systems in terms of system requirements, a good way to minimize the amount of intellectual effort required to go from the conceptualization of a system to an explicit representation of the system is to make a requirements specification the explicit representation of the system. With a tool such as UFO, this is particularly beneficial since:

- developers only have to specify the discriminating requirements for a class of systems. The common requirements for the system instances are implicit, or abstracted away from.
- the tool defines for users the discriminating requirement variables, their structure, and their legal values
- some of the requirement variables can be automatically derived by the tool
- the modeling, simulation, and feedback of system properties provides tool support to developers as they converge on a validated set of requirements

Each of these helps to reduce the conventional cost of taking a software system from a concept to an explicit requirements specification for the system.

The second abstraction objective is accomplished with automation going from requirements variable values to realizations of software architecture instances. With UFO, the tool provided automated guidance in evaluating and selecting off-the-shelf OODB instances that best satisfied application requirements. This helps to reduce the conventional cost of going from a requirements specification of a system to an executable realization of the system.

In Chapter 11., Future Work, we discuss other ways in which a tool like UFO can automate the transition from a requirements specification to an executable realization. One is to automatically generate custom system designs from the software architecture instance models in the tool. Another is to automatically generate an executable custom system from the software architecture instance models in the tool by configuring a collection of reusable software components (using the configuration node technique from UFO). Note that in this case, the abstraction level for implementing an executable system has been raised to the level of merely specifying requirement variable values. The rest of the system generation fully automated.

9.4. Practicality

In section 1.5. on page 9 we described a self-imposed challenge to make UFO a practical software engineering tool. In this section we examine the practicality of transitioning and using the technology in mainstream software engineering practice. We explore practicality through the following two issues:

- What level of expertise is required to <u>use</u> a software architecture modeling and simulation tool?
- What level of expertise is required to <u>implement</u> a software architecture modeling and simulation tool?

9.4.1. Level of Expertise Needed for Users

A developer with a solid understanding of conventional software engineering technology and methods should have sufficient skills to effectively use a tool like UFO. In particular, the user must understand the roles and relationships between requirements, architectures, classes of systems, and instances from a class of systems. Note the users of the tool don't have to understand how these concepts and relationships are implemented in the tool, but rather how to use a tool that provides these concepts and relationships as abstractions to the user.

A users' manual for the tool would likely contain the following information, so users must be competent in using this information.

- General information about the class of system. Presents an overview of the intended function, general requirements, reference architecture, and basic system properties. Sets the boundaries of interest, in terms of what is included and what is not included.

- Discriminating and common features for the target instances. Refines the general view of the domain to indicate that various target instances that are supported by the tool. This information includes the common requirements, architectural features, and system properties, plus the requirements that help to discriminate among the instances, the variance in the architectural features for the different instances, and the system properties associated with the different instances.

- Details on using the tool for requirements definition. Describes in detail each of the slots in the requirements template of the tool, including the intent and impact of the requirement variable, the legal values, and heuristics for choosing appropriate values.

- Details on interpreting the output from the tool. The tool may automate some of the property extraction and suggest the appropriate refinements to requirements. However, in cases where the developer must manually extract relevant system properties and interpret this in terms of refinements on requirements, more detailed knowledge, expertise, and effort may be required of the developer. The manual provides rationale and heuristics on how to do the analysis and refinement.

From this description of a users' manual for a software architecture modeling and simulation tool, we see that the skills needed to understand and use a tool like UFO are not beyond those needed by a conventional system architect or system designer. In fact, since the users' manual and tool provide considerable technical support for a class of systems, the overall skill set needed by a project using software architecture modeling and simulation technology is likely to be less than the skill set needed to implement a comparable system instance using conventional analysis, design, and implementation techniques.

An additional new skill that is required of the uses is the ability to analyze and interpret modeling profiles, simulation profiles, and feedback suggestions and then to impart any desired refinements to the requirement variables. This skill is similar to the ad hoc techniques for system refinement in conventional software development, but having a tool like UFO involved may require different ways of thinking about how to impart desired changes to a system instance by way of changing requirement variable values.

9.4.2. Level of Expertise Needed for Implementors

While the expertise required to use a software architecture modeling and simulation tool is less than that required using conventional techniques, the expertise required to implement such a tool is likely to be greater. The implementors must be *generalists* that can not only understand the requirements, architectural, and implementation issues associated with a single system instance, but also understand the issues associated with the entire domain for a class of similar systems. Note, however, that there will be many more developers using a software architecture modeling and simulation tool than there will be implementors building it. Therefore, only a relatively small number of specialists are needed to implement tools for a much larger population of developers using them.

A similar characterization of skills has been noted for the field of domain engineering. That field also looks at general techniques for characterizing domains, or classes of systems. However, as just noted in Section 9.1.5., our technology has a very specific focus in terms of the type of information that is need from the domain and how it will be used. This framework or structure associated with our technology provides more guidance in discovering the entities and relationships during domain analysis, so we might expect the implementors' job might be more well defined than that of a domain analyst.

9.5. Extensibility

We explore the extensibility of the UFO work through the following questions:

- How well does the OODB software architecture tool (UFO) generalize to other classes of systems?
- How well does the UFO approach scale to very large architectures? When does complexity become an issue?

9.5.1. Generalizing the OODB Work to Other Classes of Systems

The idea of improving cost and conformance for a class of systems is not unique to the OODB class of systems. Our experiences suggest that the technology that we developed for the UFO tool can be generalized and applied to other classes of systems.

Software development costs can be kept low for a class of systems via the automation provided by a software architecture modeling and simulation tool. Conformance can be kept high through the mapping from requirements to architecture, the common parts of the architecture that satisfy the common requirements, and the feedback from the system properties. However, the specific details on what conformance means will vary from domain to domain. For example, in the OODB class of systems, conformance depends on runtime locality in the declared object clusters. This may not be a conformance issue in other domains such as interactive video systems.

Based on our experiences with UFO and the OODB domain, we have identified several characteristics for a class of systems that are necessary in order for the software architecture modeling and simulation technology to be successfully applied.

- The class of systems must contain a significant number of instances. That is, the effort required to implement a modeling and simulation tool must be amortized across all of the instances that will be modeled by the technology. If there are only a small number of instances that will every be created in the tool, then the up-front cost may not be justified by the overall savings. The overall cost of building a tool compared to the average savings per instantiation give the projected payoff schedule.

- Diverse and broad classes of architectures may not be amenable to software architecture modeling and simulation technology. This is because commonality plays a less significant part in the instances, because a high degree of variability is more difficult to capture and implement in the tool (resulting in more up-front cost), and because the high variability requires more effort when using the tool.

- The target instances for the class of systems must be predictable and identifiable up-front when the tool is developed. If not, developers will frequently find that the instances they need have requirements, architectures, and system properties outside the scope of the tool.

9.5.2. Scaling Software Architecture Modeling and Simulation Technology

Our work with UFO is experimental in nature and accordingly limited in scope. It is a fair question then as to how well our work scales from experimental to practical scope. Did we make unrealistic simplifications in order to control complexity? In what areas and at what level does complexity become a limiting factor?

We noted earlier that designing and implementing a tool such as UFO is considerably more complex than using the tool to define and validate requirements and to select corresponding architectural instances. Therefore, in this section we focus on the complexity and scaling issues associated with building software architecture modeling and simulation technology, not using it.

For the thesis, we found OODBs to be a realistic class of systems for applying software architecture modeling and simulation ideas to, not just a “toy” domain for experimentation.[1] There is significant commonality among the instances, there are a significant number of different instances that are valuable in practice, and OODBs are relatively large sized software systems. As we look towards larger scale efforts, we explore three dimensions, (1) modeling greater degrees of variance, (2) modeling domains with larger system sizes, and (3) greater levels of detail in the architecture modeling.

9.5.2.1. Scaling to Greater Degrees of Variance

Scaling the technology with greater degrees of variance results in:

- a larger number of requirement variables
- a larger number of architectural parameters
- a larger number of configuration nodes
- more complex mappings from requirement variables to architectural parameters
- more complex mappings from architectural parameters to architecture instance models
- a more complex architecture instance simulator
- more complex simulation profile analysis and feedback reporting

[1] This has been borne out in professional practice on projects where OODBs were evaluated and selected for application development.

As we reported earlier in this chapter, our initial attempts at creating a dependency graph between requirements and architectures led to significant complexity. When we subdivided the mapping into four separate concerns, the task became much easier. For example, the two mappings from requirement variables to architectural parameters and from architectural parameters to architecture instance models consist of a collection small algorithms, encapsulated around each architectural parameter or configuration node. As shown in Sections section 5.1.7. on page 74 and section 5.1.9. on page 78, these algorithms are on the order of one fourth to one half page of pseudo-code, and the implemented algorithms are of a similar, manageable size.

As the variability scales linearly, then we would expect the number of requirement variables, architectural parameters, and configuration nodes to also increase linearly. Likewise, the number of mapping algorithms would increase with the number of variables, parameters, and nodes that they map. The size of the mapping would increase linearly or sub-linearly since the algorithms map from a finite number of variables or parameters to a single parameter or configuration node. We estimate that variability could easily scale by an order of magnitude before the number of requirement variables, architectural parameters, and configuration node, or the size of the mapping algorithms started to introduce serious complexity issues.

One part of the UFO design and implementation that might not scale well is the approach to evaluating and comparing off-the-shelf systems. Our approach is to model and simulate each off-the-shelf system and then manually and automatically compare the models and profiles to a validated standard for an application. This approach of instantiating and comparing one instance at a time could be very time consuming and error prone if the number of off-the-shelf instances were to grow to twenty or more. In Chapter 11., Future Work, we describe more appropriate techniques for selecting from among large numbers of off-the-shelf system variants.

9.5.2.2. Scaling to Larger Architectures

System size of the system instances being modeled and simulated is much less of an issue for our technology than is the amount of variability and detail being modeled. For example, consider a class of systems whose instances are four times larger than in the OODB domain, but whose variability was one fourth that of the OODB domain. This would result in fewer requirement variables, fewer architectural parameters, fewer configuration nodes, and smaller mappings. This would lead to less complexity when compared to the UFO tool in terms of the design and implementation of a modeling and simulation tool to support the new class of systems. Therefore, the fact that the system size that we are modeling and simulating is larger does not imply that the implementation of a modeling and simulation tool is more difficult. In fact, for this example the only impact of a larger system size would be that configuration nodes would model larger software components than configuration nodes in the OODB domain (i.e., the size attribute of the configuration nodes used in the modeling profile would be a larger integer).

In Chapter 11., Future Work, we discuss enhancements to our technology that would generate system design or configure executable software systems. In this type of technology, larger system size would become an issue since the actual design structures and/or system software components would have to be implemented.

9.5.2.3. Scaling to Greater Levels of Detail in Modeling

The level of detail that we provided in the UFO OODB architecture modeling best corresponds to the level of detail found in a high-level system design. In practice, this level of detail might be scaled to correspond to a low-level system design. This increased detail would provide more accurate predictions from the models, simulations, and requirement validations.

The impact associated with greater modeling detail is similar to that described for scaling to greater variance in Section 9.5.2.1.: a larger number of requirement variables, a larger number of architectural parameters, a larger number of configuration nodes, more complex mappings from requirement variables to architectural parameters, more complex mappings from architectural parameters to architecture instance models, a more complex architecture instance simulator, and more complex simulation profile analysis and feedback reporting.

9.6. Lessons from the UFO Design and Implementation

In this section we focus on some specific issues that arose during the design and implementation of the UFO tool. These low-level details will be of interest to implementors of software architecture modeling and simulation tools similar to UFO.

9.6.1. UFO's Virtual OODB Language

In section 5.1.2. on page 56 we described the application development language for UFO. We referred to this language as the *virtual OODB* language since it was a generalized API for all possible OODB instances in the UFO parameterized architecture. We examine this generalized API here since it was a significant challenge to define and because it plays a key role in the UFO technology (or any similar software architecture modeling and simulation implementation).

The rationale for having the virtual OODB is so that the application driver code can be written once and then the application code tested against many different OODB architecture refinements without modification. If each OODB instance were to have a different language for writing applications, then with each refinement in the architecture it might be necessary to modify the application source code. This would make the technology much more tedious and error prone, plus it would be more difficult to compare the modeling and simulation data since the application code might be different for each profile.

An implication of the virtual OODB approach is that each OODB instance supports all or part of the API. That is, some of the less functional instances will not support the full virtual OODB language. This is acceptable since many applications do not need the full functionality of the virtual OODB. The only constraint is that in order to be modeled, simulated, and evaluated an OODB instance must support all of the functionality used by an application. This is easily handled by listing the required functionality for an application in the requirement variables. An example in UFO is the *long transactions* requirement variable.

A challenge to the virtual OODB language is finding a single generalization for all of the instances that we wanted to consider. This is complicated further when commercial off-the-shelf OODBs are considered since each of these systems may provide a slightly different flavor of functions such as long transactions. Furthermore, different OODBs may use different programming languages as the basis for their API.

For UFO, we simplified the situation by using a single programming language with a single semantics for functionality such as long transactions and concurrency control. For commercial off-the-shelf systems, we modeled their architectures as close as we could, but the application driver code written in UFO cannot be used directly with any commercial off-the-shelf system. However, we feel that there is sufficient architectural similarity between the UFO models and the actual OODBs so that the UFO models are likely to be reasonable predictors for comparing the profiles of off-the-shelf systems.

In practice, there are several potential ways to improve on the UFO virtual OODB language. One is to simply add more detail in modeling the subtle differences between functionality such as long transactions, such that all of the differences are represented in the API, the requirements, and the parameterized architecture. The issues associated with add more detail to modeling are discussed in Section 9.5.2.3.

9.6.2. Static Declaration of Locality Clusters

Because of the overhead associated with moving objects in and out of persistent storage, object access locality plays important role in OODBs. When objects are clustered with good locality, the object movement overhead can be amortized over multiple objects in the cluster.

OODBs typically use runtime “hints” or heuristics to assign objects to clusters enhance locality clustering. In some cases, these clustering hints cannot be honored at runtime. For example, the OODB may not be able to fit an object into a cluster that is already full.

For the purposes of comparing different OODB profiles in UFO, this unpredictable clustering behavior was undesirable. For example, we didn’t want an OODB’s performance to be degraded simply because object alignment in clusters was poor for a particular simulation run. Therefore, for the UFO simulation, we used a stronger mechanism to get more predictable locality profiles across all OODB architectures. We used a common static declaration of clusters in the requirement variables in order to get repeatable runtime object clustering across different object architectures.

This approach provided some interesting challenges in the simulator implementation. In particular, when a new object is created, we had to be able to determine from the static locality declarations exactly what runtime cell to put the object in. We did this by using the position of the new object relative to another object that pointed to it. For example, if developers declared that an object of type B was to be clustered with an object of type A whenever the A object had a pointer to the B object, then we used this information at runtime to cluster a B object with an A object whenever it was created relative to the A object. In order to accomplish this we had to restrict the language so that all object creations were the sole right-hand-side expression of an assignment that had an object component on the left-hand-side. This restriction is of course too strong for a production OODB language, but posed no significant problems or limitations during our prototyping and profiling.

A possible extension to our approach would be to include the different clustering mechanisms found in different OODBs as part of the architectural variability in UFO. This would impact the implementation all the way from the virtual OODB language, to the requirements, to the parameterized architecture, and the simulator. This extension would allow developers to explore the impact of different cluster schemes on their applications.

9.6.3. Defining Configuration Nodes

Once we had identified the concepts in the OODB domain and had defined the UFO tool structures (such as the requirements template, architectural parameters, architecture instances, configuration nodes, mappings, and so forth), we found that it was often straightforward and intuitive to express the OODB domain concepts in terms of the UFO implementation constructs. However, the approach for partitioning the parameterized architecture into configuration nodes was not initially obvious (see section 5.1.5. on page 67 for a description of configuration nodes). We include below several simple guidelines for making these decisions.

- Common, invariant subsystems can be represented as a single terminal configuration node without further decomposition or specialization. Only those subsystems that may vary among instances need to be decomposed into smaller configuration nodes or have specializations.

- For variant subsystems, there are three basic techniques for representing the variations using configuration nodes: specialization, decomposition, and alternation. These techniques can be applied recursively and heterogeneously. Following are guidelines for choosing which technique to apply when modeling a variant subsystem.

    - <u>Specialization</u>. Analogs to specialization include translation, transformation, subtyping, or generic instantiation on a single base component. Use a specialization configuration node for a variant subsystem when there are relatively minor and isolated differences among the subsystem variations.
    - <u>Decomposition</u>. The analog to decomposition is partitioning a subsystem into modules or classes. Use a decomposition configuration node (i.e., a configuration node with nested child configuration nodes) for a variant subsystem when the each variant feature in the subsystem is modular and the overall variance represents a significant portion of the subsystem.

- Alternation. Alternation is simply providing multiple subsystems to choose from. Alternative subsystems are represented with configuration nodes that are alternative terminal nodes at a child of a parent configuration node. Use child alternatives when there is a relative large amount of difference in the subsystem variants and when these differences are not modular.

Chapter 10. Conclusions

In this thesis we developed a tool for modeling and simulating OODB architectures. We showed that this tool, called UFO (for Uniquely Fabricated OODBs), and the approach we defined for using it satisfies our hypothesis:

> It is possible to apply an engineering approach, based on the modeling and simulation of OODB architectures, to efficiently guide developers in making principled choices within the architectural space for the class of OODB software systems, such that we improve our ability to identify OOBD architecture instances with high conformance to application requirements.

The tool focuses on two specific software development tasks: defining the application requirements on an OODB and selecting an off-the-shelf OODB that best satisfies these requirements. We showed that the tool significantly improves our ability to carry out these tasks compared to conventional ad hoc approaches.

10.1. Justifying the Hypothesis

We showed how the cycle of relationships between (1) application requirements on OODBs, (2) OODB architectures, and (3) the system properties of OODBs serve as the technical basis for the UFO tool. Developers iteratively traverse this cycle to converge on a set of *baseline requirements* that accurately define an application's requirements on an OODB. The cycle is then repeatedly traversed again to evaluate how well different off-the-shelf OODBs satisfy the baseline requirements for an application.

To manage the complexity of the relationships between requirements, architectures, and system properties, we decomposed the problem into three smaller sub-problems.

- We captured the relationships between requirements and software architectures in our *parameterized software architecture*. The parameterized software architecture takes values for discriminating OODB requirements its input parameters and maps these to configurations for an OODB modeler and simulator. The parameterized software architecture is further decomposed into three sub-mappings, each with separate concerns: (1) requirements variables to architectural parameters, (2) architectural parameters to architecture instances, and (3) architecture instances to OODB modelers and simulators.

- We captured the relationships between OODB software architectures and OODB system properties in our OODB modeling and simulation tool. The OODB modeler presents feedback to developers about the static properties of an OODB architecture, such as the number and size of the system processes and the functionality supported by the OODB. The OODB simulator presents feedback to developers about the dynamics properties of an OODB architecture operating in the context of a particular application. Examples of dynamic properties captured by the UFO tool include execution times, locality, and resource utilization.

- We captured some of the relationships between OODB system properties back to requirements in an automated profile analysis. The profile analysis presents developers with feedback about how well an OODB architecture satisfies certain requirements and how the requirements might be refined to more accurately express the requirements that the application has on the OODB.

To clearly illustrate how our tool could be used in a practical setting we presented in Chapter 4. a hypothetical case study of a development team using UFO to define requirements on an OODB for their application and to evaluate and select an off-the-shelf OODB to satisfy those requirements. The scenario shows the issues that developers have to address and the steps that are taken when using UFO.

We ran four experiments to help justify our claim that we satisfied the hypothesis of our work. These experiments, described in Chapter 7., were used to collect data about the level of effort required to use UFO to define requirements and select off-the-shelf OODBs for two different applications. In Chapter 8. we present an informal analysis using the experimental data to estimate the reduction in development effort that UFO offers and the accuracy of the results from UFO. This informal analysis is divided along the two functions of the tool, defining requirements and selecting OODBs. Our informal analysis projected that UFO could potentially reduce the development effort for defining OODB requirements and selecting off-the-shelf OODBs to a few percent of the conventional costs of doing analysis, requirements definition, prototyping, and OODB evaluation. We also informally projected that UFO could lead to requirements and OODB selections that are accurate and complete to within a few percent for a given application.

10.2. Contributions

The primary contribution of this work is to demonstrate that a modeling and simulation tool can play an important role in helping software developers make principled engineering choices within the architectural design space for a class of similar systems. We have shown how such a tool can reduce the overall application development effort and increase the overall accuracy in defining requirements for an OODB and in selecting off-the-shelf OODBs that satisfy the requirements. The following paragraphs discuss more specific contributions within this context.

- **UFO tool**. We created a tool that application developers can use to define and test application requirements for OODBs and to select off-the-shelf OODBs that satisfy these requirements. We demonstrated the use of this tool with case study scenarios and experiments.

- **Role of modeling and simulation feedback**. We have shown the value of providing modeling and simulation feedback to developers as they define baseline requirements for instances within a class of similar systems and as they evaluate how well different instances satisfy the baseline requirements. This feedback allows developers to *iteratively* refine their baseline requirements to more accurately reflect their intent. This feedback also allows developers to *iteratively* cycle through the evaluation and selection of appropriate instances for a given application.

- **Tool framework for classes of similar systems**. We have identified and implemented in the UFO tool a *framework* for the capturing the constructs and relationships from the software architectures in a class of similar systems. As a whole, this collection of relationships appears complex, difficult to capture, express, manage, and implement. One of the significant contributions of our work was to characterize the structure of these concepts and relationships and to *factor them into modular and simpler abstractions*. By doing so we reduce the complexity of designing and implementing a tool like UFO. For example, the abstractions provided a guiding framework for characterizing the different concepts and relationships the OODB domain.

    - At the highest level of abstraction, we partitioned the problem into a triangular cycle of relationships between requirements and software architectures (the parameterized software architecture), between software architectures and system properties (the modeler and simulator), and between system properties and requirements (the profile analysis).

    - Within each of these three we further factored the problem into smaller abstractions, including mappings from requirements to architectural parameters, from architectural parameters to configuration nodes, from configuration nodes to a modeler and simulator, the feedback on inconsistencies between off-the-shelf instances and baseline requirements, system properties feedback from the modeler, system properties feedback from the simulator, and automated analysis of the system properties from the profiles relative to the baseline requirements.

- An additional contribution of factoring the problem into smaller abstractions is that other disciplines can make use of the individual constructs and relationships without inheriting the whole UFO technology. For example, in the next chapter we discuss future work that in some cases capitalizes on using individual subcomponents of the UFO tools and in other cases capitalizes on extending individual subcomponents.

- **Constructs and relationships for OODBs**. Another important contribution of this work is our characterization of OODBs as a class of systems:

  - *Commonality*. We defined a reference architecture that captures salient common architectural features in the OODB domain.

  - *Discriminators*. We defined requirement variables, architectural parameters, and configuration nodes that discriminate among the salient differences within the class of OODB systems.

  - *Feedback*. We defined modeling and simulation profiles for OODBs and also inconsistency reports for mismatches between baseline requirements and off-the-shelf OODBs. This feedback expresses the salient information that developers need to define and refine OODB requirements and to evaluate off-the-shelf OODBs.

Chapter 11. Future Work

In this chapter we look towards building on the accomplishments of this thesis with new or expanded research efforts. We describe six different topics related to this thesis that have research potential. The first of these, outlined in Section 11.1. explores the opportunity for doing quantitative validation of the UFO modeling and simulation tool and the effectiveness of its use. Section 11.2. looks at two additional ways of realizing our software architecture instance model, generating a system design, and composing an executable collection of software components. Section 11.3. describes the opportunity for advanced mechanisms to evaluate and compare large numbers of off-the-shelf systems. Section 11.4. suggests that the developer can be removed from the feedback loop in our approach so that a software architecture dynamically adapts to changing requirements in a deployed application. In Section 11.5., we look at the potential for a support environment that would help developers build tools like UFO for many different classes of software architectures. Finally, in Section 11.6. we describe the possibilities for integrating multiple software architectures into composite architectures that could be modeled and simulated as a whole.

11.1. Validation of UFO

We have indicated earlier that a thorough quantitative validation of the UFO approach and technology is beyond the scope of this thesis, so we relied on qualitative arguments to support the hypothesis. In this section we explore the possibility of doing a quantitative validation of UFO.

We identified two primary areas where further validation would be beneficial. The first is validation of the UFO modeler and simulator. This type of validation, described in Section 11.1.1., would determine how well the modeler and simulator predicts system properties of off-the-shelf systems. Second, in Section 11.1.2., we discuss validation of the overall software development approach advocated in this thesis. This type of validation would (1) quantitatively verify that the requirements definition and off-the-shelf OODB selection costs associated with UFO are less than with conventional software development approaches, (2) quantitatively verify that OODB requirements and selected off-the-shelf OODBs conform more closely to application requirements using UFO as compared with conventional approaches, and (3) quantitatively verify that choices made by developers using the UFO tool are as good as or better than choices made by OODB domain experts.

11.1.1. Validating the UFO Modeler and UFO Simulator

As shown in Figure 4., UFO provides three feedback paths from the OODB architecture modeler and simulator, (1) inconsistency detection between baseline architectural parameters for an application and an off-the-shelf OODB, (2) static architectural modeling properties such as size and structure for an OODB architecture, and (3) dynamic simulation properties. Validation of the UFO modeler and simulator corresponds to validating the accuracy of the information from these feedback paths. That is, validation would determine how well these feedback paths predict inconsistencies, size, functionality, and performance of real OODB systems.

The validity of the feedback information is a function of its *completeness* and its *correctness*. Completeness indicates whether or not the feedback paths contain all of the relevant information needed by developers. Correctness indicates whether or not the information from the feedback paths accurately represent the real properties of the systems being modeled or simulated.

11.1.1.1. Completeness of the Feedback Paths

The definition of completeness depends on the objectives for the modeler and simulator. For example, the following increasingly rigorous objectives would require increasingly rigorous definitions of "complete" feedback information:

1. identify the OODB architectures that closely conform to the baseline requirements and eliminate those that are blatant mismatches

2. accurately determine the relative ordering of all of the off-the-shelf OODBs relative to the baseline requirements

3. accurately predict the performance and runtime size data associated with the OODB architecture instances

If the objectives for the modeler and simulator aren't being met (we describe in Section 11.1.2. how to validate whether or not the objectives are being met) and the information from the feedback paths is determined to be "correct" (we describe in Section 11.1.1.2. how to validate correctness of the feedback information), then it follows that the feedback paths do not contain complete information. That is, essential information is missing that would otherwise support the modeling and simulation objectives.

Information needed for completeness might be identified in several ways. First is the approach that we took with UFO. By studying the OODB domain we determined the information that we subjectively determined to be most relevant for inconsistencies, modeling data, and simulation data. Second is to survey consultants and other OODB evaluation experts as to what information they use. And finally, OODB architects could provide useful information about the implications that different architectural features would have on externally observable system properties.

11.1.1.2. Correctness of the Feedback Paths

Correctness of the feedback paths is the degree to which the modeling and simulation information accurately reflects the properties of an OODB architecture instance. As with completeness, the definition of correctness depends on the objectives for the modeler and simulator. It may be sufficient for the feedback data for a collection of OODBs to reflect the relative ordering of system properties or a stronger set of objectives may require the feedback paths to accurately predict the size and performance data.

Validating the correctness of the OODB modeler and simulator would require experiments with off-the-shelf OODBs operating under a representative spectrum of different application requirements. Prototype applications would be run on both an off-the-shelf OODB and the modeler and simulator for the same OODB. The results of the two would then be compared.

With sufficient experimentation and refinement of a modeler and simulator, correctness objectives could likely be met. However, in a practical tool it would be important to easily add new off-the-shelf OODBs to the modeler and simulator without compromising the correctness and without having to incur the cost of extensive re-validation experiments and refinements. Therefore, it would be important to define a set of benchmarks that could be run on a new OODB and a means of configuring the modeler and simulator to produce correct values for the new off-the-shelf OODB.

11.1.2. Validating the Effectiveness of the UFO Tool

In Chapter 8. we presented a qualitative validation of our hypothesis for the effectiveness of the UFO tool and approach. In that chapter we identified factors that impacted costs and conformance, and then intuitively argued why UFO improved these factors relative to conventional approaches.

In order to do a quantitative validation, we would first need to more rigorously define and test the factors that impact cost and conformance. For example, in Section 8.2.1. we identified *search cost*, *profiling cost*, and *collation cost* as factors that determine the cost of selecting an off-the-shelf OODB. While these appear to be a reasonable characterization of the factors, there may be different factorizations or additional factors that would help to better represent selection costs. With a well defined characterization of cost and conformance factors in place, we could more accurately perform experiments to compare the cost and conformance using UFO and conventional approaches.

The validation experiments would involve experimental groups developing applications that use OODBs. Different experimental groups would use different software development approaches related to the evaluation and selection of OODBs. Then the associated costs and conformance associated with each approach would be measured and compared. The following three development approaches would provide a useful spectrum: (1) conventional application development using developers without OODB expertise, (2) conventional application development with the help of OODB experts or consultants, (3) application development using the UFO tool.

- The conventional application development group would not use the UFO tool or the modeling and simulation approach, but rather would use conventional software development techniques to analyze, prototype, design, select OODBs, and implement their applications.

- The application development group with OODB domain experts would use conventional software development techniques, but would rely on the knowledge and experience of the OODB experts to analyze, prototype, design, select OODBs, and implement their applications.

- The application development group using UFO would use the modeling and simulation approach and tool to help define requirements, simulate system instances, select off-the-shelf OODBs, and implement an executable application.

The data from the three groups would be collected according to the cost and conformance factors and model discussed above, including the amortized cost for developing the UFO architecture modeling and simulation technology. The hypothesis could then be validated according to the cost and conformance results.

A collection of applications requiring a representative spectrum of OODBs would be necessary to accurately carry out the validation experiments. That is, the application requirements from the different applications used in the experiments should result in the selection of a significant number of the available off-the-shelf OODBs.

11.2. Software Architecture Realizations

During the early stages of our research, we identified three different software representations that could be produced by mappings from software architecture instances: (1) OODB simulators, (2) OODB designs, and (3) executable OODBs. The first was addressed in this thesis while the second two were determined to be out of scope. We discuss here the research potential of the remaining two items.

11.2.1. Generating a Design from a Software Architecture Instance

With the current UFO tool, developers first converge on a set of baseline requirements and then attempt to identify an off-the-shelf OODB that closely conforms to the baseline requirements. In the case where none of the off-the-shelf options are satisfactory, a custom OODB must be developed. The current UFO tool provides little support for custom development other than the baseline requirements and the high-level textual description of the architecture instance that is derived from the baseline requirements. A promising extension to the UFO tool is an OODB design generator for custom OODBs.

Figure 74. illustrates how the current UFO approach would be modified so that OODB designs could be generated. Tasks 1 through 6 are identical to Figure 33. on page 90. Task 7 generates a design description from a software architecture instance. For task 8, developers take the design description and manually implement the custom OODB, producing OODB compiler specializations and an OODB runtime.

The configuration nodes that are used to internally represent the OODB architecture instances in UFO are also a good candidate from which to generate OODB designs. Each configuration node in UFO could have a design description associated with it, plus design descriptions for the specializations that can be applied to the configuration node. The design descriptions would include internal details that are not modeled in the current UFO modeling tool, such as internal software components and interfaces. The reference architecture description in Section 5.1.3. could serve as the unifying framework for the overall design description, with the configuration nodes supplying the specific details on the different architectural variants.

Two interesting research issues related to generating OODB designs include:

- What type of design notation is best suited for the generated designs? Graphical, textual, formal?
- What level of detail can be achieved in the generated design descriptions? High level designs, low level designs, or both?

11.2.2. Generating an Executable System from a Software Architecture Instance

Another potential extension to UFO is to directly generate a custom executable OODB from the software architecture instances. This approach has several significant implications. First, there would no longer be a need for OODB modeling and simulation. All of the system properties could be collected directly from the generated OODB. Second, there would be no need to consider off-the-shelf OODBs since the generated OODBs could be used for applications rather than evaluating and selecting off-the-shelf OODBs.

Figure 75. shows how executable OODBs could be generated from a collection of reusable software components using UFO. Tasks 1 through 6 are identical to Figure 74. in this chapter and Figure 33. on page 90. Task 7 configures the executable OODB from a collection of reusable components, based on a software architecture instance. The resulting OODB serves the same purpose as an off-the-shelf OODB in tasks 8 and 9.

OODB generation could be accomplished by configuring reusable software components according to the configuration nodes in a software architecture instance model. Each of configuration nodes in the UFO parameterized architecture would have an associated reusable software component. The reusable software components would have the same specialization and child subcomponents as modeled in the configuration nodes.

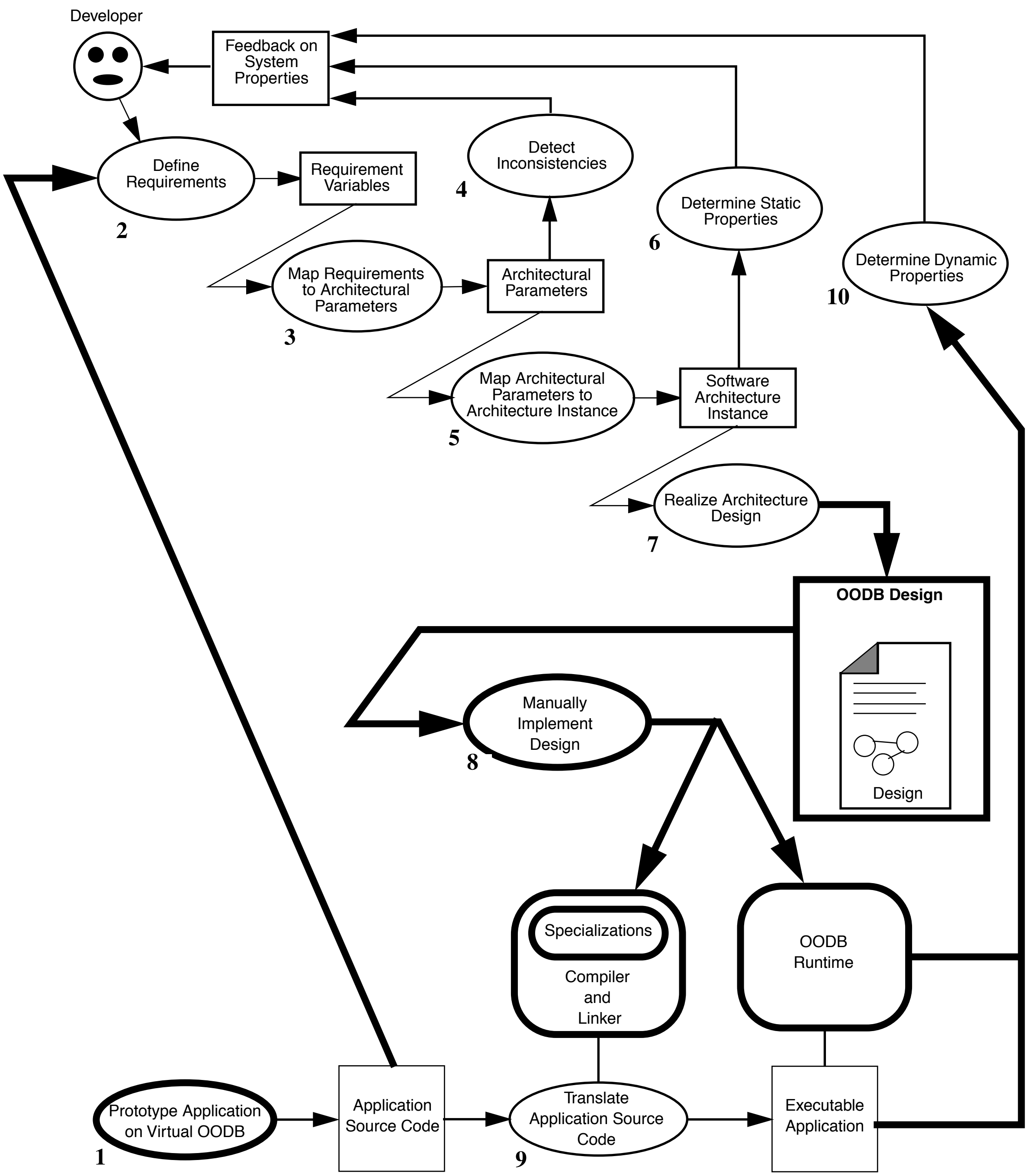


Figure 74. Using UFO to Generate Designs from Requirements

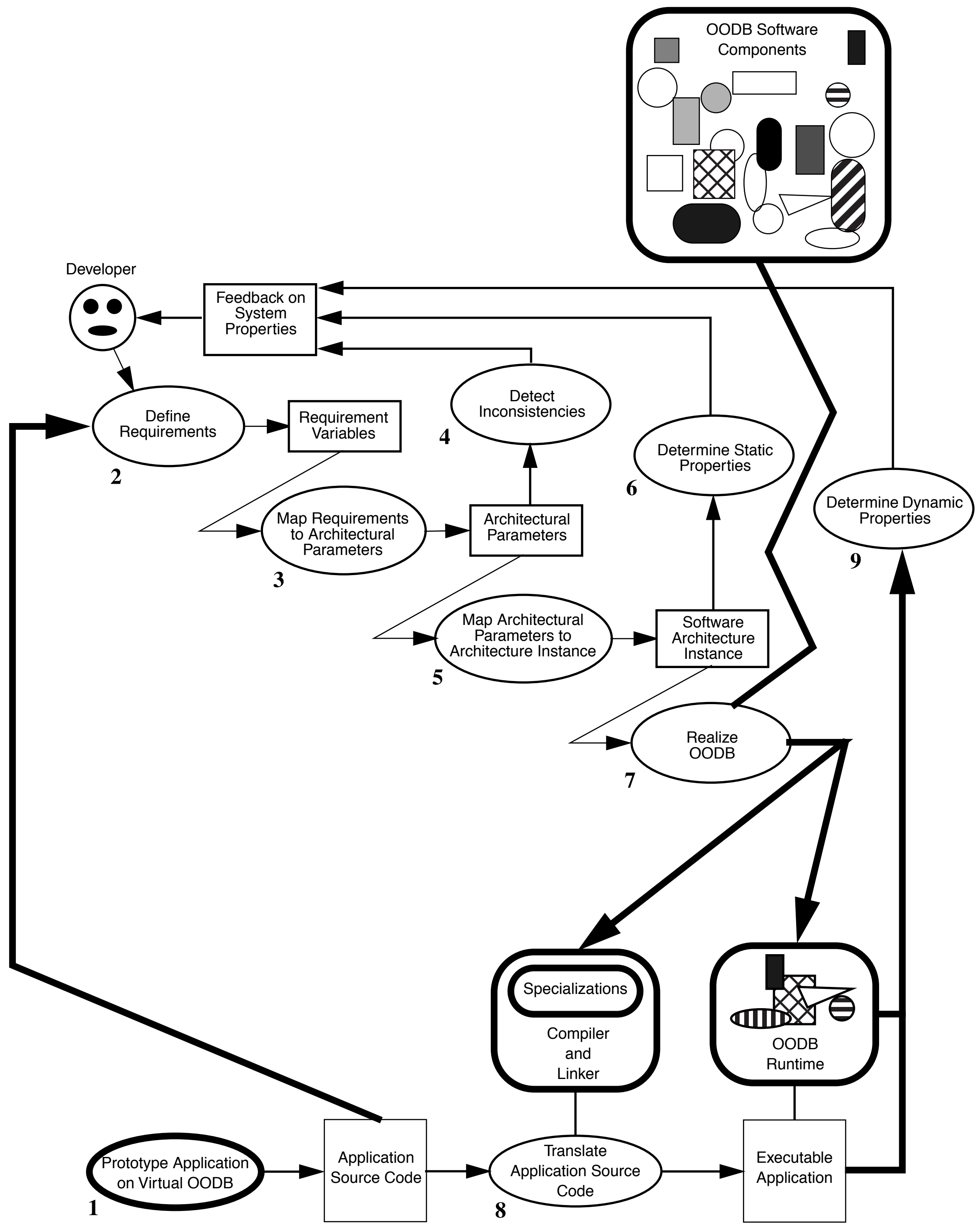


Figure 75. Using UFO to Generate an Executable OODB from Requirements

With this approach, the UFO parameterized architecture and reusable software components are similar to the existing software reuse concepts *software frameworks* and *software component libraries*. The proposed extension to UFO could address some of the shortcomings of these reuse techniques.

Software component libraries are collections of software modules with functionality that can commonly be reused in different software development projects. Hash tables, sets and sequences, and multi-dimensional data points are examples of reusable software components. A noted problem with software components is that they don't provide architectural structure or guidance for selecting and composing a consistent set of components to satisfy the requirements for an application. These structuring and composition tasks are often the most difficult and time consuming parts of software development, so reusable software components often don't offer significant savings in development costs.

Software frameworks can offer advantages over software component libraries. Frameworks provide reusable and flexible software "skeletons" that can be specialized within the design space for a class of systems. In order to use a framework in an application, the undefined portions in the code skeleton are filled out by adding components with predefined interfaces in predefined places. While frameworks provide structure for composing reusable components, they don't provide guidance in selecting a consistent set of components that satisfy a specific set of requirements for an application.

UFO technology, on the other hand, provides both structure and guidance for composing a collection of software components (i.e., configuration nodes) into a software architecture that satisfies the requirements for a specific application, similar to the work of Batory[14]. In addition, UFO also provides support for exploring the validity of those decisions through the feedback paths to developers.

Another potential research issues related to generating executable OODBs is configuration management. As the UFO reference model, configuration nodes, and mappings evolve during development and maintenance, so must the reusable software components that go into making the executable OODBs. It is important that the reusable component library contains a collection of software components that are consistent with each other and furthermore are consistent with the UFO configuration nodes and mappings. This presents an interesting software configuration management problem in that the component library evolves over time as well as the mappings that define all possible combinations components in the generated OODBs. How can implementors of the UFO tool be assured that they have not introduced a change to a component or mapping that results in an inconsistent configuration of components in an OODB?[1]

11.3. Evaluating and Comparing Off-the-Shelf Instances

The current approach to evaluating and comparing off-the-shelf OODB architectures with UFO is to model and simulate each candidate and then *manually* compare the results. There is an opportunity for research on *automated* tools for evaluating, comparing, and pruning off-the-shelf candidates relative to the requirements for a particular application. This might be particularly useful if the number of available off-the-shelf instances increases to a level where manual evaluation becomes tedious or error prone.

We have identified several items that could be explored in such research efforts. The first would be automated evaluation and comparison of off-the-shelf architectures based on static modeling properties and dynamic simulation properties. The current UFO tool would be used to define baseline requirements and to map from baseline requirements to a baseline software architecture instance. A new tool would be developed to automatically indicate how well each of the off-the-shelf candidates conforms to the baseline architecture based on static properties such as excess or missing functionality or excessive system size or numbers of processes. Another new tool could also automate the simulation and profiling of an application on different off-the-shelf architectures, and then order candidates based on dynamic criteria such as overall execution time or execution time on

1. The author is studying and prototyping a configuration management system of this type in a commercial development project.

certain operations indicated by developers to be critical. Poor matches could be immediately pruned and developers could focus on evaluating and selecting from the best candidates.

Another potential approach is to create a UFO tool with requirement variables, architectural parameters, a mapping from requirement variables to architectural parameters, configuration nodes, and a mapping from architectural parameters to configuration nodes that model only off-the-shelf architecture instances. Using this tool, developers could define their application requirements and then invoke the mappings in order to instantiate an off-the-shelf architecture that satisfies the requirements. The following situations could arise:

- One off-the-shelf architecture would be instantiated that corresponds to exactly one product satisfying the requirements.
- No off-the-shelf architectures would be instantiated, indicating that no products satisfy the requirements. The tool or mappings in this case could indicate where each of the off-the-shelf instances were pruned and why so that requirements can be relaxed in minimal ways in order to instantiate one or more instances.
- One or more off-the-shelf architectures could be instantiated that corresponds to multiple products satisfying the requirements. These candidates could be further evaluated with simulation to identify the best candidates, using the automated profile analysis to order the candidates according how well they satisfy the requirements.

To automated the evaluation and selection of off-the-shelf OODBs, the requirement variables could be extended to provide the tool with the type of information that developers currently use to do manual evaluation and selection. For example, developers might be interested in optimizing the performance of certain critical operations in their application and much less interested in the overall execution time. These preferences could be expressed by indicating performance-critical operations in the requirements.

11.4. Automated Architectural Evolution

With UFO, developers can model and simulate OODB architectures that satisfy a set of application requirements *prior* to application development. This is done prior to development so that baseline requirements are confirmed and so that an appropriate OODB architecture can be selected to best satisfy those requirements. However, what happens when the application requirements for an OODB change *after* the application is developed and deployed? With the current UFO, this would mean repeating the UFO analysis cycles (defining requirements, simulating, selecting OODBs) and converting the existing application and existing data in the OODB repositories to a new OODB.

An adaptation of UFO technology would automate the detection of requirement changes, analysis of the impact of changing requirements, and conversion of a deployed OODB. With this technology, the UFO tool and its associated iterative cycles from requirements to architectures to system properties and back to requirements would begin prior to application development (as it does in the current tool), but would remain active after the application was deployed.

There are numerous practical reasons why the application requirements on an OODB might change after the application is deployed:

- The number of application users might increase or decrease, leading to insufficient or excessive OODB resources for the application.
- Enhancements to the application such as new features or modified features may indirectly impact the OODB requirements.
- As a project using the application evolves, so may the types of interactions with the application.
- The user community for the application may become more sophisticated after some experience using the application, resulting in changes to the typical user interactions with the tool.

Several major pieces of technology would have to be developed for automated architectural evolution in response to changing requirements. First, the OODBs produced or selected by UFO would have to continuously gather and feedback runtime profile data. Second, analysis of this feedback would have to be fully automated to detect sub-optimal performance profiles and associated refinements to the requirements. Third, the tool would have to automatically make the refinements to the requirements and rederive the architecture instance model. Next the tool would have to select or generate an OODB from the architecture instance. And finally, the tool would have to generate and perform the necessary conversion functions to efficiently upgrade the existing application OODB with the new one.

Currently the two options when an OODB no longer satisfies evolving application requirements are for the application users to tolerate sub-optimal performance of the application due to a sub-optimal OODB or to incur the high costs of analysis, redesign, modification, and conversion of the application implementation and the existing application data. Automated architectural evolution offers an attractive alternative to these options.

Learning systems might offer useful concepts and technology for automated architectural evolution. For example, Staudt-Lerner demonstrated how a software application could be constructed with monitors to gather usage profiles and agents that would analyze the usage profiles and attempt to tailor the application to a user's individual style of interaction[15].

11.5. Creating Software Architecture Modeling and Simulation Tools

Although the work in this thesis focused on modeling and simulating OODB architectures, we paid careful attention throughout not to limit the applicability of the approach to only the OODB domain. It has always been our belief that the concepts and technology developed for UFO could be applied to a broad range of different software architectures. An opportunity with significant potential for future branches of UFO research is to generalize the approach to support many domains other than just OODBs.

Rather than simply repeating the UFO OODB effort for each domain, a much more useful approach would be to generalize the practices and technology that we used for UFO to provide *meta-tools* for efficiently creating software architecture modeling and simulation tools analogous to UFO. For example, one can imagine a special-purpose language and editor for defining configuration nodes and another special-purpose language and editor for defining mappings from architectural parameters to configuration nodes. These special-purpose languages would be independent of the application domain.

This approach is analogous to the Gandalf environment generator system[11]. Similar to what we are proposing here, Gandalf is a meta-tool for building structure-oriented editors. It provides a collection of special purpose languages for expressing salient features for a structure-oriented editor application.

One of the more difficult challenges in capturing the software architecture for a class of systems is knowing what to look for and capture in the domain. The meta-tools for capturing software architectures could provide structure and guidance to the domain analysis task. For example, users would need to define a reference architecture to capture commonality in the domain and discriminators such as requirement variables, architectural parameters, configuration nodes, and mappings to capture the variance in the domain. The languages and tools for capturing these could provide significant guidance in how to think about the domain analysis problem and the type of information that users need to look for in a class of systems.

11.6. Composite Software Architectures

With a capability in place to capture multiple software architectures (as described in the previous section) a new set of opportunities and challenges arise for modeling and simulating *composite* software architectures. A composite software architecture is a collection of software architectures that are integrated to create a single larger architecture. The objective would be to create technology to model and simulate a composite software architecture as a whole.

Modeling and simulating composite software architectures is useful for addressing the architecture mismatch problem described by Garlan et.al.[16]. At the analysis and design stage of software application development it is often useful to identify off-the-shelf subsystems that can be used to implement the application. However, there is a risk that mismatches across the architectural interfaces may make it difficult or impossible to effectively integrate the off-the-shelf subsystems into an effective composite architecture. A tool that could model and simulate the integration of software architectures into a single composite architecture would be useful in that it could detect these architectural mismatches early in the software analysis and design stages rather than in later prototyping, implementation, or testing stages.

At least two approaches could be explored for architecture composition. One would recursively apply the UFO approach to software architectures. In the same way that we compose configuration nodes within a well defined framework for OODB architectures, there could be higher level frameworks that would compose software architectures. This approach, however, implies that a higher level framework for an application must exist before developers can use it. If they are trying to develop an application with an unanticipated composition of software architectures, this approach will not work.

Another approach would allows developers to define arbitrary compositions of software architectures. This would correspond to the definition of a framework on-the-fly. Tools similar to those described in the previous section, *Creating Software Architecture Modeling and Simulation Tools*, might be useful for quickly defining a unique framework for integrating, modeling, and simulating a composite software architecture for a particular application.

References

[1] Burkhard, Neil, McCabe, Rich, Campbell, Grady, Wartik, Steve, O'Connor, Jim, Valent, Joe and Facemire, Jeff. "Reuse-Driven Software Processes Guidebook." Technical Report SPC-92019-CMC, Software Productivity Consortium, Herndon, Virginia, November, 1993.

[2] Dijkstra, E.W. "Notes on Structured Programming." In Structured Programming, Dijkstra, E.W., Dahl, O.J. and Hoare, C.A.R., Ed(s). Academic Press, London, England, 1972, pages 1-82.

[3] Parnas, David L. "On the Design and Development of Program Families." IEEE Transactions on Software Engineering. SE-2:1-9, 1976.

[4] Lane, Thomas G. "User Interface Software Structures." PhD Thesis CMU-CS-90-101, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May, 1990.

[5] Shaw, Mary and Garlan, David. "Software Architecture." Prentice Hall, Upper Saddle River, New Jersey, 07458, 1996.

[6] Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John. "Design Patterns." Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[7] Boehm, Barry W. and Scherlis, William L. "Megaprogramming." In Proceedings of the Software Technology Conference 1992, DARPA, Los Angeles, CA, April, 1992, pages 63-82.

[8] Barry, Douglas. "Charting the Feature Coverage of ODBMSs." Object Magazine. 28-42, February, 1997.

[9] Atwood, Tom, Duhl, Joshua, Ferran, Guy, Loomis, Mary and Wade, Drew. "The Object Database Standard: ODMG-93." Cattel, R.G.G., Ed(s). Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[10] Arango, Guillermo and Prieto-Diaz, Ruben. "Domain Analysis Concepts and Research Directions" In Domain Analysis and Software Systems Modeling, Arango, Guillermo and Prieto-Diaz, Ruben, Ed(s). IEEE Computer Society Press, Los Alamitos, CA 90720, 1991, pages 9-32.

[11] The Gandalf Project. "The Gandalf System Reference Manual." Technical Report CMU-CS-93-X & CMU-CS-86-130, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, April 1993.

[12] "CORBA Object Request Broker." Technical Report http://www.omg.org, Object Management Group, December 1997.

[13] Krueger, Charles W. "Software Reuse." ACM Computing Surveys. 24, 2:131-183, June 1992.

[14] Batory, Don, Singhal, Vivek, Sirkin, Marty and Thomas, Jeff. "Scalable Software Libraries." ACM Press, New York, NY, December 1993, pages 191-199.

[15] Staudt Lerner, Barbara. "Automated Customization of User Interfaces." PhD Thesis CMU-CS-89-178, Carnegie Mellon University, Pittsburgh, PA, September 1989.

[16] Garlan, David, Allen, Robert and Ockerbloom, John. "Architectural Mismatch: Why Reuse is So Hard." IEEE Software. 12, 6:17-26, November 1995.

[17] Tepfenhart, William M., IEEE Software, 31-35, January, 1997

[18] Monroe, Robert, Kompanek, Andrew, Melton, Ralph and Garlan, David. "Architectural Styles, Design Patterns, and Objects." IEEE Software. 14, 1:43-52, January 1997.

[19] Shaw, Mary. "Comparing Architectural Design Styles." IEEE Software. 12, 6:27-41, November 1995.

[20] Abowd, Gregory, Allen, Robert and Garlan, David. "Using Style to Understand Descriptions of Software Architecture." In 18, 5, David Notkin, Ed(s). Software Engineering Notes, ACM Press, Los Angeles, CA, December 1993, pages 9-20.

[21] Neighbors, J.M. "DRACO: A Method for Engineering Reusable Software Systems." In Software Reusability: Volume I - Concepts and Models, Biggerstaff, T.J. and Perlis, A.J., Ed(s). ACM Press, New York, NY, 1989, pages 295-320.

[22] Prieto-Diaz, R. "Domain Analysis for Reusability." In Proceedings of COMPSAC '87, IEEE Computer Society, 1987, pages 23-29.

[23] Arango, Guillermo. "DOMAIN ANALYSIS - From Art Form to Engineering Discipline." In Proceedings of the 5th International Workshop on Software Specifications and Design, IEEE Computer Society Press, Los Alamitos, CA, May, 1989, pages 152-159.

[24] Hess, James A., Novak, William E., Carrol, Patrick C., Cohen, Sholom G., Hollibaugh, Robert R., Kang, Kyo C. and Peterson, A. Spencer. "A Domain Analysis Bibliography." Technical Report CMU-SEC-90-SR-3, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, July 1997.

[25] Perry, Dewayne E. and Wolf, Alexander L. "Foundations for the Study of Software Architecture." Software Engineering Notes. 17, 4:40-52, October 1992.

[26] "Proceedings of the First Intl Workshop on Architectures for Software Systems." Technical Report CMU-CS-95-151, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1995.

[27] Mettala, Erik and Graham, Marc. "The Domain Specific Software Architecture Program." Los Angeles, CA, April 1992, pages 204-236.

[28] Tracz, Will. "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)." ACM SIGSOFT Software Engineering Notes. 19, 2:52-56, April 1994.24.

[29] Tracz, Will. "DSSA (Domain-Specific Software Architecture) Pedagogical Example." ACM SIGSOFT Software Engineering Notes. 20, 3:49-62, July 1995.

[30] Batory, Don, Coglianese, Lou, Goodwin, Lou and Shafer, Steve. "Creating Reference Architectures: An Example from Avionics." ACM Press, New York, NY, August 1995, pages 27-37.

[31] "Readings in Object-Oriented Database Systems." Zdonik, Stanley B. and Maier, David, Ed(s). Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[32] Carey, Michael J., DeWitt, David J., Graefe, Goetz, Haight, David M., Richardson, Joel E., Schuh, Daniel T., Shekita, Eugene J. and Vandenberg, Scott L. "The EXODUS Extensible DBMS Project: An Overview." Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990, pages 474-499.

[33] Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C. and Wise, T.E. "GENESIS: An Extensible Database Management System." IEEE Transactions of Software Engineering. 14, 11: November 1988.

[34] Batory, D.S. "Modeling the Storage Architectures of Commercial Database Systems." ACM Transacations on Database Systems. 10, 4:463-528, December 1985.

[35] Carey, Michael J., DeWitt, David J., Richardson, Joel E. and Shekita, Eugene J. "Object and File Management in the EXODUS Extensible Database System." August 1986, pages 91-100.

[36] Batory, D.S. and Mannino, M. "Panel on Extensible Database Systems." ACM Press, New York, NY, May 1986, pages 187-190.

[37] Keller, Arthur M. and Wiederhold, Gio. "Modularization of the DADAISM Ada Database System Architecture." Technical Report unpublished, Standord University, Palo Alto, CA, February 1989.

[38] Simmel, Sergiu S. and Godard, Ivan. "The Kala Basket." ACM Press, New York, NY, 1991, pages 230-246.

[39] Joseph, John V., Thatte, Satish M., Thompson, Craig W. and Wells, David L. "Object-Oriented Databases: Design and Implementation." Proceeding of the IEEE. 79, 1:42-64, January 1991.

[40] Wells, David. "DARPA Open Object-Oriented Database." Technical Report unpublished, Texas Instruments, Dallas, TX, September 1991.

[41] Hornick, Mark F. and Zdonik, Stanley B. "A Shared, Segmented Memory System for an Object-Oriented Database." ACM Transactions on Office Information Systems. 5, 1:70-85, 1987.

[42] Boehm, Barry and Scacchi, Walt. "Simulation and Modeling for Software Acquisition (SAMSA): Plans and White Paper." Technical Report http://sunset.usc.edu/SAMSA/shitepaper.html, University of Southern California, June 1995.

[43] Boehm, Barry and Scacchi, Walt. "Simulation and Modeling for Software Acquisition (SAMSA): Air Force Opportunities (Extended Report)." Technical Report http://sunset..usc.edu/SAMSA/SAMSA_FAM_final_report.html, University of Southern California, March 1996.