Software Performance Engineering for Object-Oriented Systems: A Use Case Approach

Connie U. Smith[†] and Lloyd G. Williams[§]

[†]Performance Engineering Services
PO Box 2640, Santa Fe, New Mexico, 87504-2640
(505) 988-3811, http://www.perfeng.com/~cusmith

[§]Software Engineering Research
264 Ridgeview Lane
Boulder, CO 80302
(303) 938-9847

Abstract

Many object-oriented systems fail to meet performance objectives when they are initially constructed. These performance failures result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows. Most performance failures are due to a lack of consideration of performance issues early in the development process. However, early consideration of performance in object-oriented systems is straightforward. The Use Case scenarios produced by developers during analysis and design serve as a starting point for performance analysis. This paper describes a systematic approach to the performance engineering of object-oriented systems based on Use Case scenarios. This approach is cost-effective and has a low impact on the software development process. A simple case study illustrates the process.

1 Introduction

Object-oriented techniques have become widely accepted for designing and implementing software systems in application areas ranging from client-server to real-time, embedded systems. Object-oriented software systems are typically easier to understand, easier to adapt to new requirements, and have a higher potential for reuse than those developed with procedural approaches.

Unfortunately, many object-oriented systems fail to meet performance objectives when they are initially constructed. These performance failures result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows. Moreover, "tuning" code to improve performance is likely to disrupt the original design, negating the benefits obtained from using the object-oriented approach. Finally, it is unlikely that "tuned" code will ever equal the performance of code that has been engineered for performance. In the worst case, it will be impossible to meet performance goals by tuning, necessitating a complete redesign or even cancellation of the project.

Our experience is that most performance failures are due to a lack of consideration of performance issues early in the development process, in the architectural design phase. Poor performance is more often the result of problems in the design rather than the implementation. However, the trend in the object-oriented community is to defer consideration of performance until after the system has been implemented. The following quote from Auer and Beck (Auer and Beck, 1996) is typical:

> Performance myth: "Ignore efficiency through most of the development cycle. Tune performance once the program is running correctly and the design reflects your best understanding of how the code should be structured. The needed changes will be limited in scope or will illuminate opportunities for better design."

This "fix-it-later" attitude is not unique to the object-oriented community. It is rooted in the view that performance is difficult to predict and that the models needed to predict the performance of an emerging system are complex and expensive to construct. Predicting the performance of object-oriented systems can, in fact, be very difficult. The functionality of object-oriented systems is decentralized. Performing a given function is likely to require collaboration among many different objects from several classes. These interactions can be numerous and complex and are often obscured by polymorphism, making them difficult to trace. The current trend toward distributing objects over a network compounds the problem.

Despite these difficulties, our experience is that it is possible to *cost-effectively* engineer object-oriented systems that meet performance goals. By carefully applying the techniques of software performance engineering (SPE) throughout the development process, it is possible to produce object-oriented systems that have adequate performance *and* exhibit the other qualities, such as reusability, maintainability, and modifiability that have made object-oriented development (OOD) so effective (Smith and Williams, 1993).

In this paper we focus on Use Cases since they provide the basis for a bridge between object-oriented methods and SPE (Smith and Williams, 1997). An instance of a Use Case represents a particular execution of the system. Use Case instances are described using Scenarios. Scenarios from Use Cases are translated into SPE performance scenarios. Performance scenarios are, in turn, used to construct and evaluate a variety of performance models.

SPE is a method for constructing software systems to meet performance objectives (Smith, 1990). Performance refers to the response time or throughput as seen by the users. The SPE process begins early in the software life cycle and uses quantitative methods to identify a satisfactory architecture and to eliminate those that are likely to have unacceptable performance. SPE continues throughout the development process to: predict and manage the performance of the evolving software, monitor actual performance against specifications, and report problems as they are identified. SPE begins with deliberately simple models that are matched to the current level of knowledge about the emerging software. These models become progressively more detailed and sophisticated as more details about the software are known. SPE methods also cover performance data collection, quantitative analysis techniques, prediction strategies, management of uncertainties, data presentation and tracking, model verification and validation, critical success factors, and performance design principles.

This paper describes the application of SPE to object-oriented systems. We begin with a review of related work. Overviews of OOD and SPE follow. We then present an overview of the SPE process for object-oriented systems. A simple example illustrates the process.

2 Related Work

As noted in the introduction, object-oriented methods typically defer consideration of performance issues until detailed design or implementation (see e.g., (Rumbaugh, et al., 1991), (Jacobson, et al., 1992), (Booch, 1994)). Even then, the approach tends to be very general and ad hoc. There is no attempt to integrate performance engineering into the development process.

Some work specifically targeted at object-oriented systems has emerged from the performance community. Smith and Williams (Smith and Williams, 1993) describe performance engineering of an object-oriented design for a real-time system. However, this approach applies general SPE techniques and only addresses the specific problems of object-oriented systems in an ad hoc way.

Hrischuk et. al. (Hrischuk, et al., 1995) describe an approach based on constructing an early prototype which is then executed to produce *angio traces*. These angio traces are then used to construct *workthreads* (also known as *timethreads* or *use case maps* (Buhr and Casselman, 1992), (Buhr and Casselman, 1994), (Buhr and Casselman, 1996)), which show object method invocations. Service times for methods are estimated. This differs from the approach described here in that their approach derives scenarios from prototype execution and generates the system execution model from the angio traces. Our approach is intended for use long before executable prototypes are available; and it reflects a view of the software that explicitly models more general scenarios with execution path frequencies and repetitions.

Baldassari et. al. describe an integrated object-oriented CASE tool for software design that includes a simulation capability for performance assessment (Baldassari, et al., 1989), (Baldassari and Bruno, 1988). The CASE tool uses petri nets for the design description language rather than the general methods described above, thus the design specification and the performance model are equivalent and no translation is necessary. Using these capabilities requires developers to use both the PROTOB method and CASE tool. The approach described here is general in that it may be used with a variety of object-oriented analysis and design methods.

3 Object-Oriented Development

Object-oriented development is an approach to software specification, design and construction that is based on identification of the objects that occur naturally in the application and implementation domains. The specification, design and code are then organized to reflect the structure inherent in those objects and their interactions.

A number of approaches to object-oriented analysis and/or design have appeared over the past several years (see e.g., (Shlaer and Mellor, 1988), (Shlaer and Mellor, 1992), (Booch, 1991), (Rumbaugh, et al., 1991), (Jacobson, et al., 1992), (Selic, et al., 1994), (Rational Software Corporation, 1997)). Despite their apparent differences, these approaches share several significant commonalties. They all involve construction of a set of conceptual models of the system under development. These conceptual models are based on object-oriented concepts such as classes, objects, methods (operations), and inheritance. Each of them also employs, at one time or another, different views of the classes and objects that are being modeled.

The principal views are embodied in static and dynamic models. Static models (Class and Object Diagrams) describe the classes and objects that are relevant to the problem and the relationships among them. Dynamic models (State Diagrams) describe the patterns of behavior that apply to objects belonging to a given class. A number of methods have also adopted Use Case Diagrams to describe interactions between the system and its environment or between objects within the system.

In this paper, we focus on Use Cases since these provide the basis for a bridge between object-oriented methods and SPE (Williams and Smith, 1995). A Use Case is a specific way of using the system (Jacobson, et al., 1992), (Rational Software Corporation, 1997). Each Use Case consists of a set of sequences of actions that the system performs to achieve some desired result. An instance of a Use Case represents a particular execution of the system.

Use Case instances are described using Scenarios. A Scenario is a sequence of actions describing the interactions between the system and its environment (including the user) or between the internal objects involved in a particular execution of the system. The scenario shows the objects that participate and the messages that flow between them. A message may represent either an event or an invocation of one of the object's methods (operations). In object-oriented methods, scenarios are used to:

- describe the externally visible behavior of the system,
- involve users in the requirements analysis process,
- support prototyping,
- help validate the requirements specification,
- understand interactions between objects, and
- support requirements-based testing.

As described in (Williams and Smith, 1995), scenarios provide a common point of departure between object-oriented requirements or design models and SPE models. Scenarios may be represented in a variety of ways (Williams, 1994). Here we use Message Sequence Charts (MSCs) to describe scenarios in object-oriented models. The MSC notation is specified in ITU standard Z.120 (ITU, 1996). Several other notations used to represent scenarios are based on MSCs (examples include: (Rumbaugh, et al., 1991), (Jacobson, et al., 1992), (Booch, 1994), and (Rational Software Corporation, 1997)). However, none of these incorporates all of the features of MSCs needed to establish the correspondence between scenarios in object-oriented modeling and scenarios in SPE.

Figure 1 illustrates a high-level MSC for a simple automated teller machine (ATM). Each object that participates in the scenario is represented by a vertical line or axis. The axis is labeled with the object name (e.g., anATM). The vertical axis represents relative time which increases from top to bottom; the MSC notation does not include a representation of absolute time. Interactions between objects (events or method invocations) are represented by horizontal arrows.

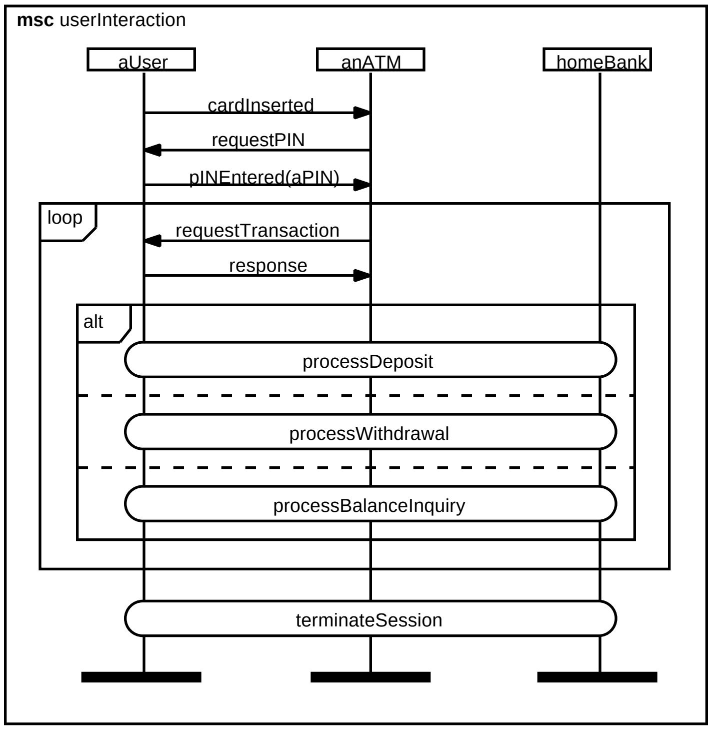

Figure 1. Message Sequence Chart for a user interaction with the ATM

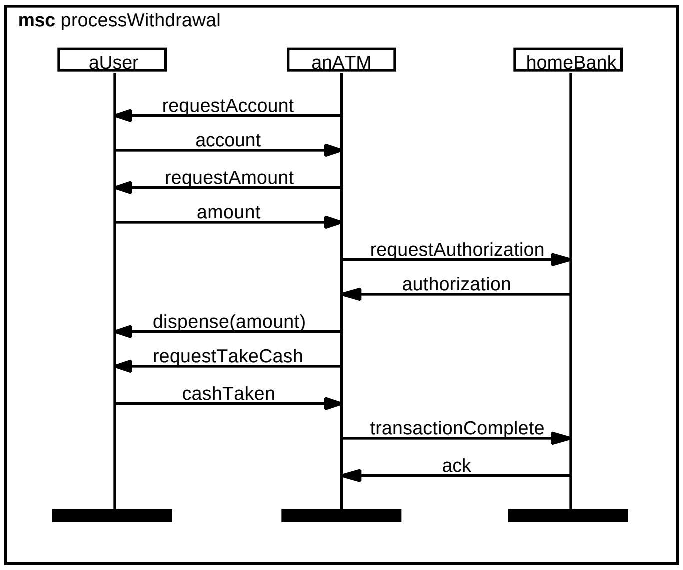

Figure 2. Message Sequence Chart processWithdrawal

Figure 1 describes a general scenario for user interaction with the ATM. The rectangular areas labeled "loop" and "alt" are known as "inline expressions" and denote repetition and alternation. This Message Sequence Chart indicates that the user may repeatedly select a transaction which may be a deposit, a withdrawal, or a balance inquiry. The rounded rectangles are “MSC references” which refer to other MSCs. The use of MSC references allows horizontal expansion of Message Sequence Charts. The MSC that corresponds to ProcessWithdrawal is shown in Figure 2.

A Message Sequence Chart may also be decomposed vertically, i.e., a refining MSC may be attached to an instance axis. Figure 3 shows a part of the decomposition of the anATM instance axis. The dashed arrows represent object instance creation or destruction. Arrows originating or terminating at the edge of the chart are those from the higher-level MSC.

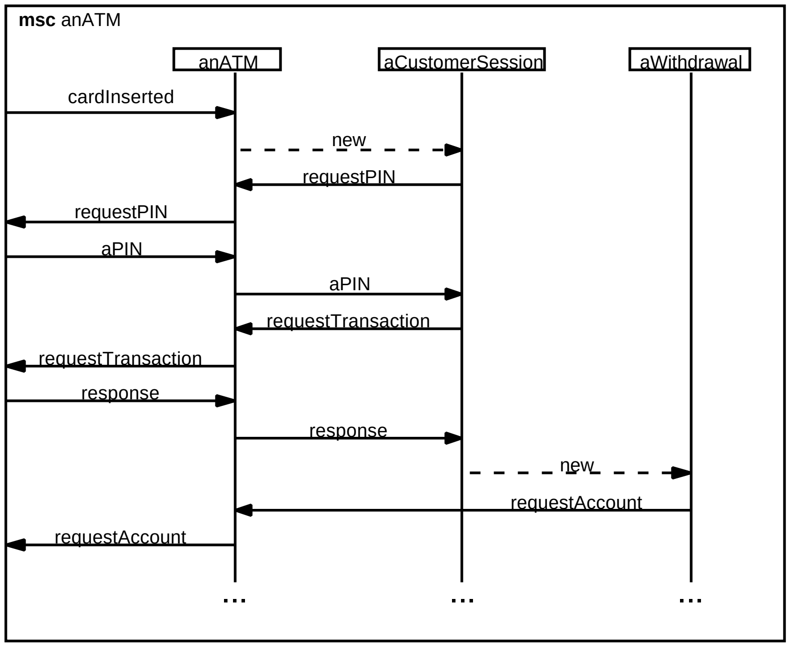

Figure 3. MSC anATM

MSC references and decomposition help to control complexity by hiding details until they are needed. They also make it easier to assemble scenario fragments into full scenarios. Finally, decomposition allows a developer to elaborate a scenario by including additional objects (as in Figure 3) as the design becomes more detailed.

Scenarios from Use Cases provide the basis for constructing performance scenarios. Performance scenarios are, in turn, used to construct performance models. Section 4 describes these performance models and their use in managing performance throughout the development process.

4 Software Performance Engineering

Software performance engineering is a quantitative approach to constructing software systems that meet performance objectives. SPE prescribes principles for creating responsive software, the data required for evaluation, procedures for obtaining performance specifications, and guidelines for the types of evaluation to be conducted at each development stage. It incorporates models for representing and predicting performance as well as a set of analysis methods.

SPE uses deliberately simple models of software processing with the goal of using the simplest possible model that identifies problems with the system architecture, design, or implementation plans. These models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. As the software process proceeds, the models are refined to more closely represent the performance of the emerging software.

The precision of the model results depends on the quality of the estimates of resource requirements. Because these are difficult to estimate early in the software process, SPE uses adaptive strategies, such as upper- and lower-bounds estimates and best- and worst-case analysis to manage uncertainty. For example, when there is high uncertainty about resource requirements, analysts use estimates of the upper and lower bounds of these quantities. Using these estimates, analysts produce predictions of the best-case and worst-case performance. If the predicted best-case performance is unsatisfactory, they seek feasible alternatives. If the worst case prediction is satisfactory, they proceed to the next step of the development process. If the results are somewhere in-between, analyses identify critical components whose resource estimates have the greatest effect and focus on obtaining more precise data for them. A variety of techniques can provide more precision, including: further refining the design and constructing more detailed models or constructing performance benchmarks and measuring resource requirements for key components.

Two types of models provide information for design assessment: the *software execution model* and the *system execution model*. The software execution model represents key aspects of the software execution behavior. It is constructed using execution graphs (Smith, 1990) to represent workload scenarios. Nodes represent functional components of the software; arcs represent control flow. The graphs are hierarchical with the lowest level containing complete information on estimated resource requirements. Figure 4 shows the execution graph corresponding to the user interaction scenario from Figures 1 - 3. The graph shows that, following GetCustomerInfo and GetPIN, the ATM will repeat the ProcessTransaction node n times. ProcessTransaction and TerminateSession are expanded nodes; they are expanded in a separate graph. Figure 5 shows the expansion of ProcessTransaction.

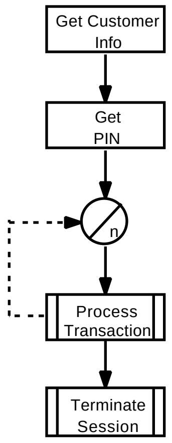

Figure 4. Execution Graph for user interaction with the ATM

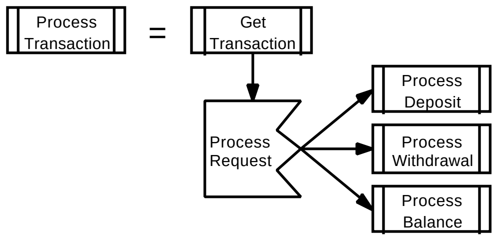

Figure 5. Expansion of processTransaction

A comparison of Figures 1 through 3 with Figures 4 and 5 illustrates the strong correspondence between Use Case scenarios, as represented in MSCs, and performance scenarios, as represented in execution graphs.

Solving the software model provides a static analysis of the mean, best- and worst-case response times. It characterizes the resource requirements of the proposed software alone, in the absence of other workloads, multiple users or delays due to contention for resources. If the predicted performance in the absence of these additional performance-determining factors is unsatisfactory, then there is no need in constructing more sophisticated models.

If the software execution model indicates that there are no problems, analysts proceed to construct and solve the system execution model. This model is a dynamic model that characterizes the software performance in the presence of factors, such as other workloads or multiple users, that could cause contention for resources. The results obtained by solving the software execution model provide input parameters for the system execution model. Solving the system execution model provides the following additional information:

- more precise metrics that account for resource contention
- sensitivity of performance metrics to variations in workload composition
- effect of new software on service level objectives of other systems
- identification of bottleneck resources
- comparative data on options for improving performance via: workload changes, software changes, hardware upgrades, and various combinations of each

The system execution model represents the key computer resources as a network of queues. Queues represent components of the environment that provide some processing service, such as processors or network elements. Environment specifications provide device parameters (such as CPU size and processing speed). Workload parameters and service requests for the proposed software come from the resource requirements computed by solving the software execution model. The results of solving the system execution model identify potential bottleneck devices and correlate system execution model results with software components.

If the model results indicate that the performance is likely to be satisfactory, developers proceed. If not, the model results provide a quantitative basis for reviewing the proposed design and evaluating alternatives. Feasible alternatives can be evaluated based on their cost-effectiveness. If no feasible, cost-effective alternative exists, performance goals may need to be revised to reflect this reality.

This discussion has outlined the SPE process for one design-evaluation cycle. These steps repeat throughout the development process. At each phase, the models are refined based on the more detailed design and analysis objectives are revised to reflect the concerns that exist for that phase (Smith, 1990).

5 SPE for OOD

Software performance engineering for object-oriented systems includes the following steps:

1. *Establish performance objectives*: Performance objectives specify the quantitative criteria for evaluating the performance characteristics of the system under development. These objectives may be expressed in several different ways, including: response time, throughput, or constraints on resource usage. For information systems, response time is typically described from a user perspective, i.e., the number of seconds required to respond to a user request. For real-time systems, response time is given as the amount of time required to respond to a given external event. Throughput requirements are specified as the number of transactions or events to be processed per unit time.

2. *Identify important Use Cases*: The important Use Cases are those that are critical to the operation of the system or which are important to responsiveness as seen by the user. Typically, this is only a subset of the Use Cases that are identified during object-oriented analysis.

3. *Select key performance scenarios*: It is unlikely that all of the scenarios for each critical Use Case will be important from a performance perspective. For each important Use Case, the key scenarios are those which are executed frequently or those which are critical to the perceived performance of the system.

4. *Translate scenarios to execution graphs*: Once the key performance scenarios have been identified, the MSC representation is translated to an execution graph. Currently, this is a manual process. However, the close

correspondence between scenarios as expressed in MSCs and execution graphs suggests that an automated translation may be possible.

Estimates of the amount of processing required for each step in the execution graph are obtained form the class definition for each object involved. This information is contained in the class diagram, or the logical view of the system architecture (Kruchten, 1995). As described above, early in the development process, these may be simply best / worst case estimates. Later, as each class is elaborated, the estimates become more precise.

5. *Add resource requirements*: The processing steps in an execution graph are typically described in terms of the software resources (e.g., operating systems calls or database accesses) used. Resource requirements map these software resource requirements onto the amount of service they require from key devices in the hardware configuration.

   Resource requirements depend on the environment in which the software executes. Information about the environment is obtained from the physical view of the architecture (Kruchten, 1995). In the UML, this corresponds to the Deployment Diagram.

6. *Solve the models*: As noted above, solving the execution graph characterizes the resource requirements of the proposed software alone. If this solution indicates problems, analysts consider design alternatives to address the problems. If not, then analysts proceed to solve the system execution model.

These steps are illustrated with the following case study.

6 Case Study

This case study examines an interactive system, known as ICAD, to support computer-aided design (CAD) activities. Engineers will use the application to construct and view drawings that model structures, such as aircraft wings. The system also allows users to store a model in a database and interactively assess the design's correctness, feasibility, and suitability. The model is stored in a central, relational database and several versions of the model may exist within the database.

A drawing consists of nodes and elements. Elements may be: beams, which connect two nodes; triangles, which connect three nodes; or plates, which connect four or more nodes. Additional data is associated with each type of element to allow solution of the engineers' model. A node is defined by its position in three-dimensional space (x, y, z), as well as additional information necessary for solution of the model.

Several different Use Cases have been identified for ICAD, including Draw (draw a model) and Solve (solve a model). For this example we focus on the Draw Use Case and one particular scenario, DrawMod (Figure 6). In the DrawMod scenario, a typical model is drawn on the user's screen. A typical model contains only nodes and beams (no triangles or plates) and consists of 2050 beams. The performance goal is to draw a typical model in 10 seconds or less.

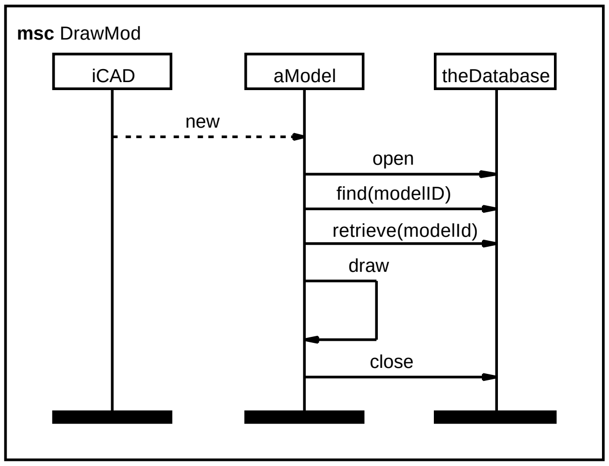

Figure 6. The DrawMod Scenario

The following sections consider three alternative designs for this application and their performance.

6.1 Design 1

The first design uses objects to represent each beam and node. This design offers a great deal of flexibility, makes it possible to treat all types of elements in a uniform way, and allows the addition of new types of elements without the need to change any other aspect of the application. The Class Diagram for Design 1 is illustrated in Figure 7.

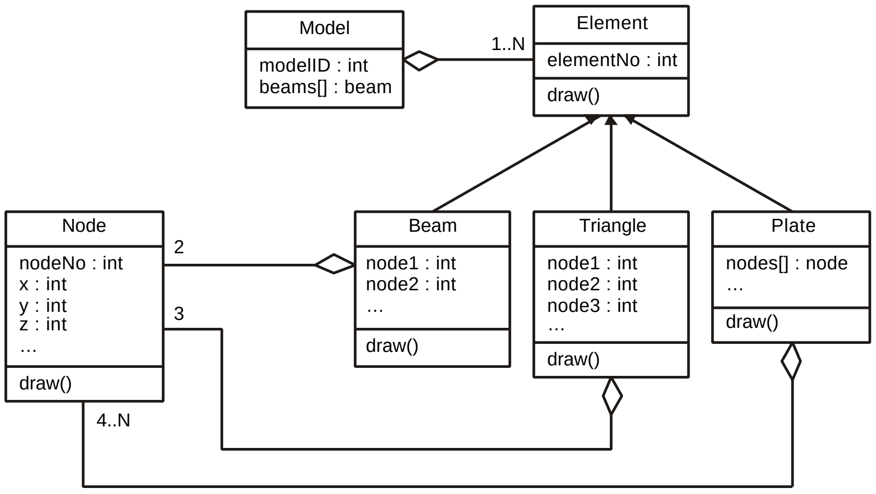

Figure 7. Class Diagram for Design 1

Given the design in Figure 7, the DrawMod scenario expands to that in Figure 8. The unlabeled dashed arrows indicate a return from a nested set of messages. This notation is not part of the MSC standard, it is taken from the UML (Rational Software Corporation, 1997).

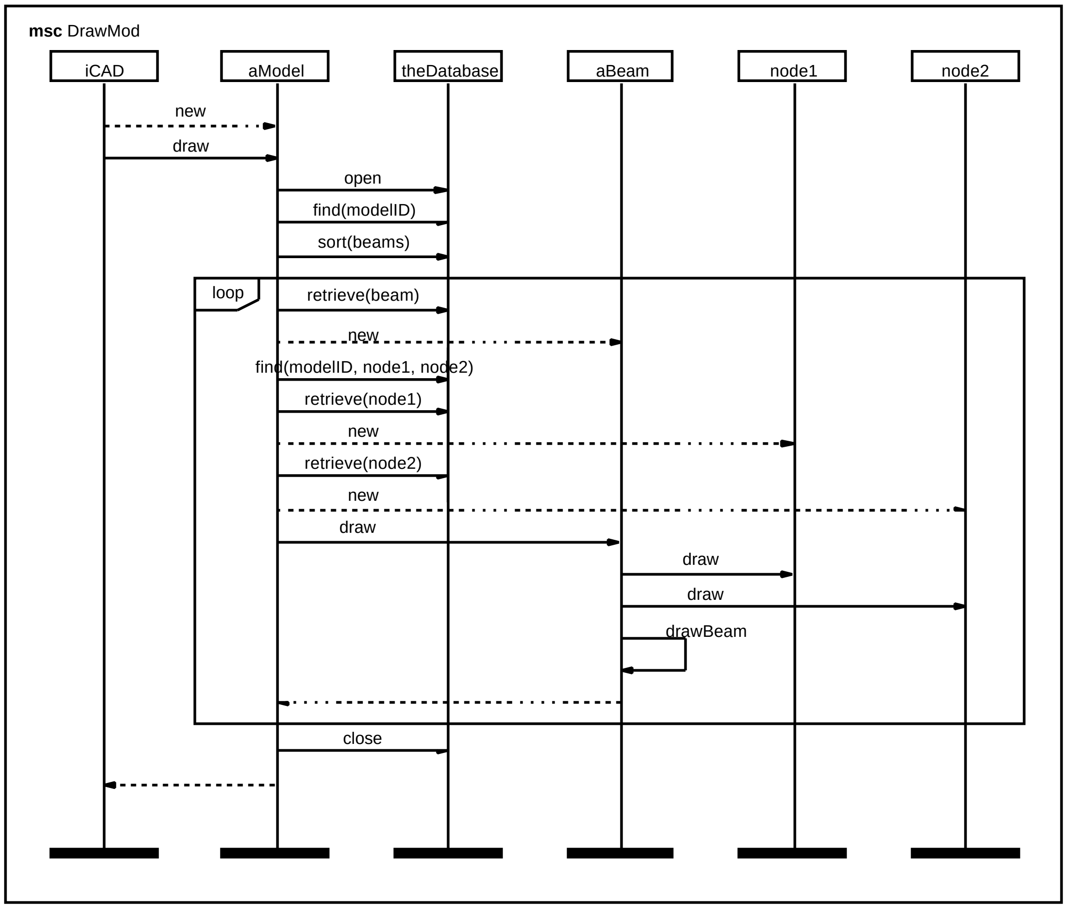

Figure 8. DrawMod scenario for Design 1

This paper illustrates model solutions using the *SPE•ED*™ performance engineering tool (Smith and Williams, 1997). A variety of other performance modeling tools are available, such as (Beilner, et al., 1988), (Beilner, et al., 1995), (Goettge, 1990), (Grummitt, 1991), (Rolia, 1992), (Turner, et al., 1992). However, the approach described here will need to be adapted for tools that do not use execution graphs as their modeling paradigm.

Figure 9 shows *SPE•ED*s screen with the execution graph corresponding to the scenario in Figure 8. The expanded nodes in the tool's graphs are shown with color. The "world view" of the software model appears in the small navigation boxes on the right side of the screen. The top level of the model is in the top-left navigation box; its nodes are black. The top-right navigation (turquoise) contains the Initialize processing step (the steps preceding find(modelID) in the MSC). Its corresponding expanded node in the top-level model is also turquoise. The expansion of the yellow DrawBeams processing step contains all the steps within the loop in the MSC. Again, there is a close correspondence between the object interactions in the MSC scenario in Figure 8 and the execution graph in Figure 9.

After creating the processing steps in the execution graph, analysts then specify resource requirements for each step. Then, *SPE•ED* produces solutions for both the software execution model and the system execution model. The specification of resource requirements as well as the model solutions are described in (Smith and Williams, 1997). The parameters in this case study are based on the example in (Smith, 1990); the specific values used are omitted here.

Figure 10 shows a combination of four sets of results for the “No Contention” Solution - the elapsed time for one user to complete the Drawmod scenario with no contention delays in the computer system. This best-case solution indicates whether it is feasible to achieve performance objectives with this approach. The solution in the top-left portion of the Figure shows that the best-case elapsed time is 992.33 seconds. The time for each processing step is next to the step. The color bar legend in the upper-right corner of the quadrant shows the values associated with each color. Values higher than the 10 second performance objective will be red, lower values are respectively cooler colors. The “Resource usage” values below the color bar legend show the time spent at each computer device. Of the approximately 992 seconds, 990 is due to time required for I/O at the “DEVs” disk device. The DrawBeam processing step requires 991 seconds for all 2050 iterations. The time per iteration, 0.483 seconds, is in the top-right quadrant along with the time for each processing step in the loop. The bottom two quadrants show the break-down of the computer device resource usage for the top level model and the DrawBeam submodel. Most of the I/O time (988 seconds) is in the DrawBeam step, the bottom-right quadrant shows that the I/O is fairly evenly spread in the submodel: 0.12 secs. for both RetrieveBeam and FindNodes, 0.24 secs. for SetUpNode.

The results show that Design 1 clearly will not meet the performance goal of 10 seconds, so we explore other possible designs.

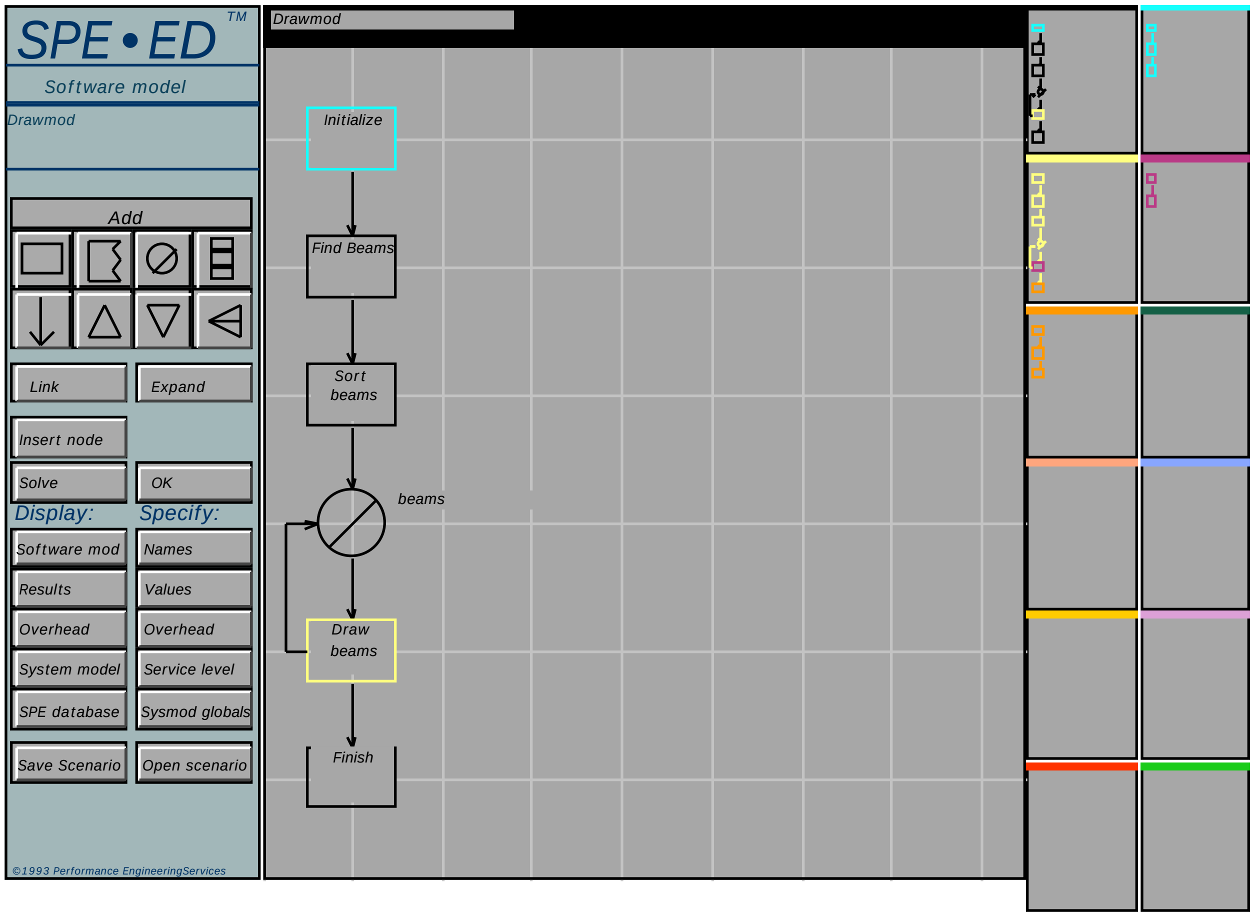

Figure 9. Execution Graph for DrawMod scenario for Design 1

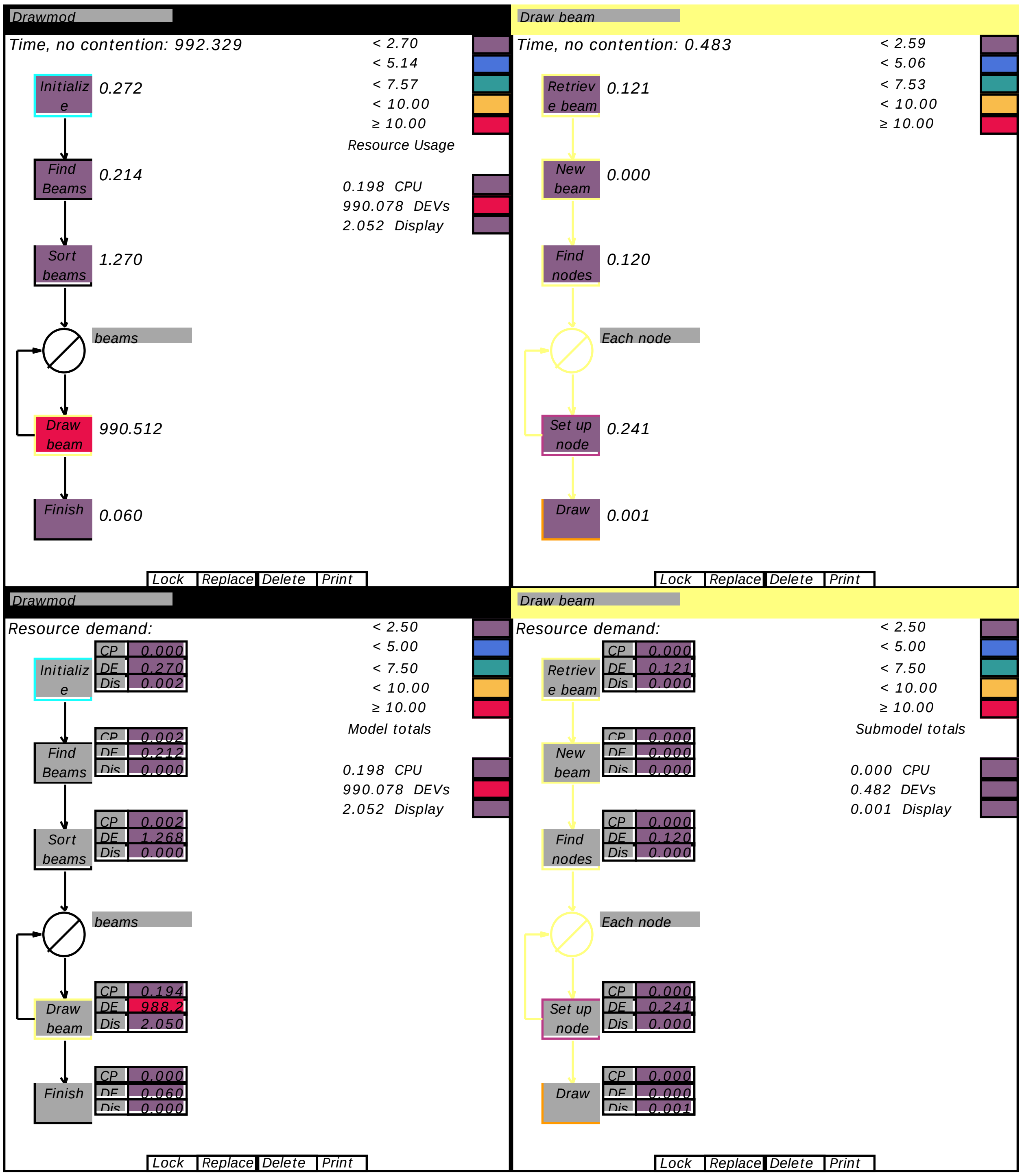
Drawmod
Time, no contention: 992.329
< 2.70
< 5.14
< 7.57
< 10.00
≥ 10.00
Resource Usage
0.198 CPU
990.078 DEVs
2.052 Display
Initialize 0.272
Find Beams 0.214
Sort beams 1.270
beams
Draw beam 990.512
Finish 0.060
Lock Replace Delete Print
Draw beam
Time, no contention: 0.483
< 2.59
< 5.06
< 7.53
< 10.00
≥ 10.00
Retrieve beam 0.121
New beam 0.000
Find nodes 0.120
Each node
Set up node 0.241
Draw 0.001
Lock Replace Delete Print
Drawmod
Resource demand:
< 2.50
< 5.00
< 7.50
< 10.00
≥ 10.00
Model totals
0.198 CPU
990.078 DEVs
2.052 Display
Initialize CP 0.000 DE 0.270 Dis 0.002
Find Beams CP 0.002 DE 0.212 Dis 0.000
Sort beams CP 0.002 DE 1.268 Dis 0.000
beams
Draw beam CP 0.194 DE 988.2 Dis 2.050
Finish CP 0.000 DE 0.060 Dis 0.000
Lock Replace Delete Print
Draw beam
Resource demand:
< 2.50
< 5.00
< 7.50
< 10.00
≥ 10.00
Submodel totals
0.000 CPU
0.482 DEVs
0.001 Display
Retrieve beam CP 0.000 DE 0.121 Dis 0.000
New beam CP 0.000 DE 0.000 Dis 0.000
Find nodes CP 0.000 DE 0.120 Dis 0.000
Each node
Set up node CP 0.000 DE 0.241 Dis 0.000
Draw CP 0.000 DE 0.000 Dis 0.001
Lock Replace Delete Print

Figure 10. Performance Results for Design 1.

6.2 Design 2

Design 1 uses an object for each beam and node in the model. While this gives the design a great deal of flexibility, using an object for each node and beam is potentially expensive in terms of both run-time overhead and memory utilization.

We can reduce this overhead by using the Flyweight pattern (Gamma, et al., 1995). Using the Flyweight pattern in ICAD allows sharing of beam and node objects and reduces the number of each that must be created in order to display the model. Each model now has exactly one beam and node object. The node and beam objects contains *intrinsic* state, information that is independent of a particular beam or node (such as coordinates). They also know how to draw themselves. *Extrinsic* state, coordinates and other information needed to store the model are stored separately. This information is passed to the beam and node flyweights when it is needed.

The Flyweight pattern is applicable when (Gamma, et al., 1995):

· the number of objects used by the application is large,
· the cost of using objects is high,
· most object state can be made extrinsic,
· many objects can be replaced by fewer, shared objects once the extrinsic state is removed, and
· the application does not depend on object identity.

The SPE evaluation will determine if the ICAD application meets all of these criteria.

Instead of using an object for each Beam and Node, we use a shared object based on the Flyweight pattern. The state information is removed from the Beam and Node classes and is stored directly in Model. The Class Diagram for this design is shown in Figure 11.

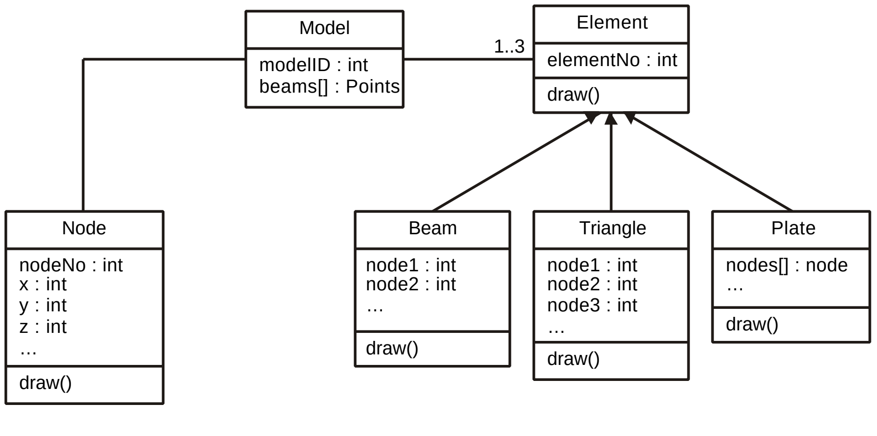

Figure 11. Class diagram for Design 2

The scenario resulting from this set of classes is shown in Figure 11. As shown in Figure 11, constructors for Node and Beam are executed only once, resulting in a savings of X constructor invocations.

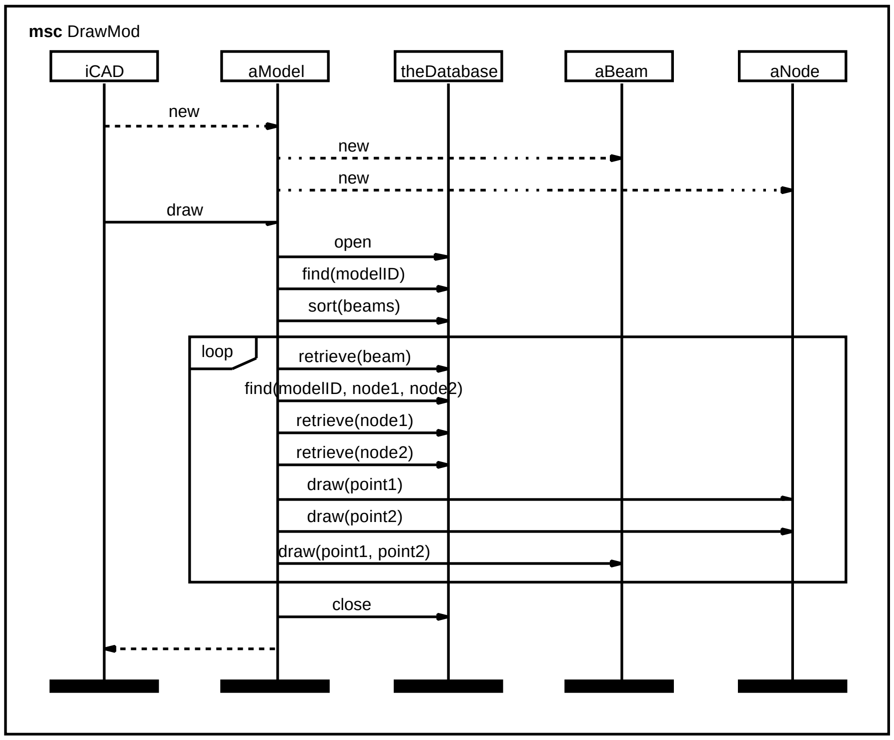

Figure 11. DrawMod scenario for Design 2

The changes to the execution graph for this design are trivial. The graph nodes corresponding to the “New” processing steps move from the yellow subgraph that represents the DrawBeam processing step to the turquoise subgraph corresponding to the Initialize processing step. This takes the corresponding resource requirements out of the loop that is executed 2050 times.

The overall response time is reduced from 992.33 to 992.27 seconds. The results of the software execution model for this design indicate that using the Flyweight pattern did not solve the performance problems with ICAD. Constructor overhead is not a significant factor in the DrawMod scenario. The amount of constructor overhead used in this case study was derived from a specific performance benchmark and will not generalize to other situations. It is compiler, operating system, and machine dependent; in our case constructors required no I/O. It is also design-dependent; in our example there is no deep inheritance hierarchy. It is also workload-dependent; in this case the number of beams and nodes in the typical problem is relatively small. Nevertheless, we choose to retain the Flyweight-based design; it will help with much larger ICAD models where the overhead of using an object for each beam and node may become significant, making the design more scalable.

The evaluation of Design 2 illustrates two important points: modifying performance models to evaluate design alternatives is relatively easy; and it is important to quantify the effect of design alternatives rather than blindly follow design “guidelines” that may not apply. Note that the relative value of improvements depends on the order that they are evaluated. If the database I/O and other problems are corrected first, the relative benefit of flyweight will be larger.

The problem in the original design, excessive time for I/O to the database, is not corrected with the Flyweight pattern, so the next design focuses on reducing the I/O time due to database access.

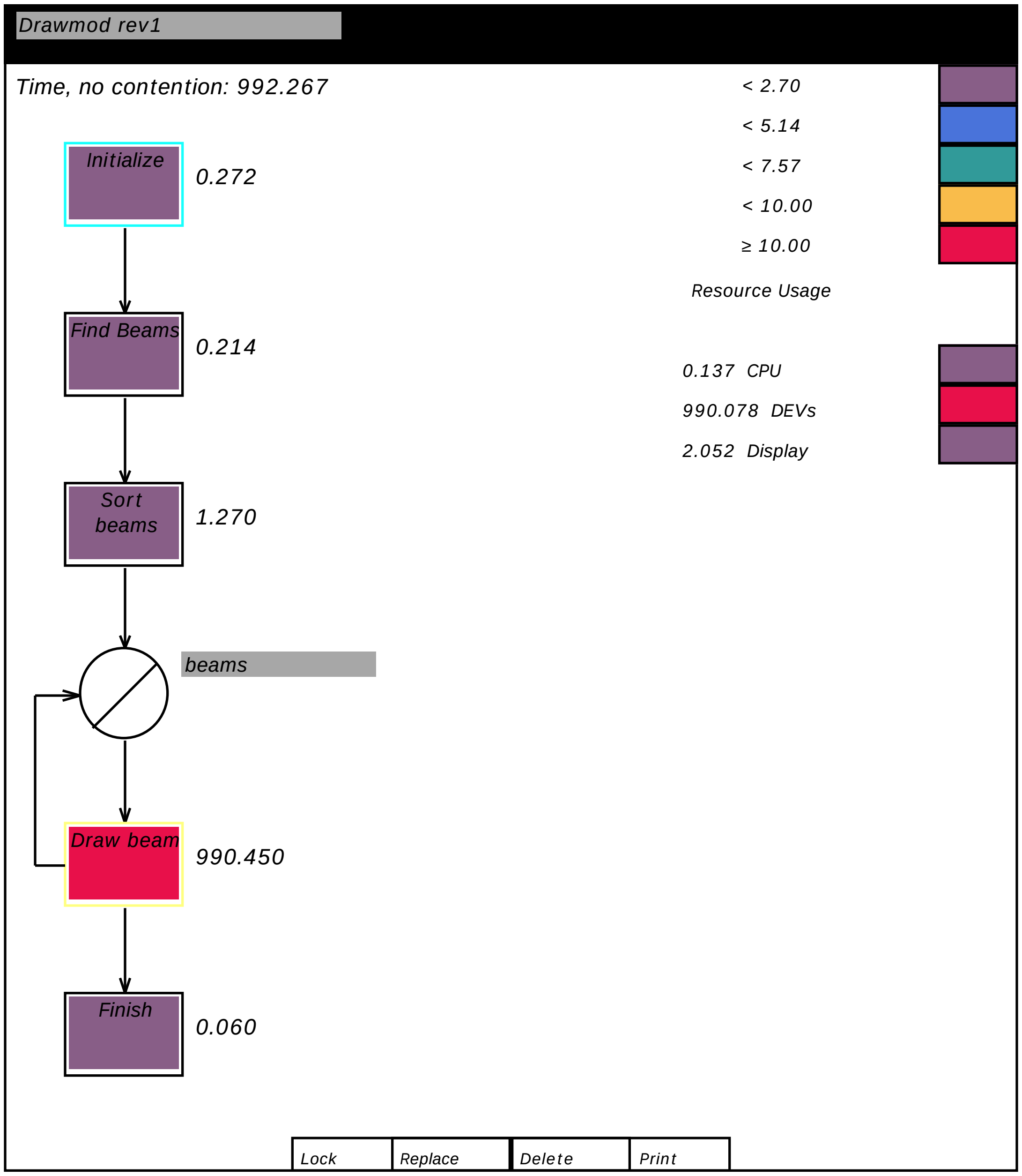

Figure 12. Results for DrawMod scenario for Design 2

6.3 Design 3

This design uses the same architecture as Design 2 (Figure 10) but modifies the database management system with a new operation to retrieve a block of data with one call: retrieveBlock(). Design 3 uses this new operation to retrieve the beams and nodes once at the beginning of the scenario and stores the data values for all beams and nodes with the model object rather than retrieve the value from the database each time it is needed. This new operation makes it possible to retrieve blocks containing 20K of data at a time instead of retrieving individual nodes and beams[1]. A single block retrieve can fetch 64 beams or 170 nodes at a time. Thus, only 33 database accesses are required to obtain all of the beams and 9 accesses are needed to retrieve the nodes.

The class diagram for Design 3 does not change from Design 2. Figure 13 shows the MSC that corresponds to the new database access protocol. The bold arrows indicate messages that carry large amounts of data in at least one direction. Although this notation is not part of the MSC standard, we have found it useful to have a way of indicating resource usage on scenarios that are intended for performance evaluation.

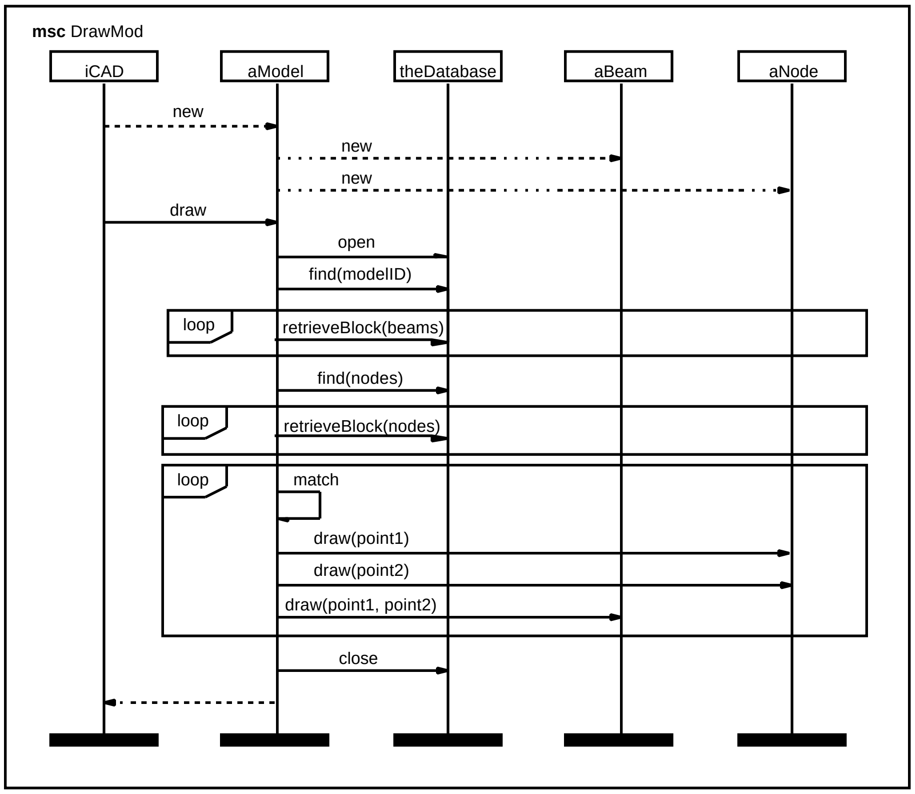

Figure 13. DrawMod scenario for Design 3

[1] Note: A block size of 20K is used here for illustration. The effect of using different block sizes could be evaluated via modeling to determine the optimum size.

Figure 14 shows the execution graph corresponding to Figure 13 along with the results for the “No Contention” solution. The time for Design 3 is approximately 8 seconds – a substantial reduction.

Other improvements to this design are feasible, however, this serves to illustrate the process of creating software execution models from object-oriented designs and evaluating trade-offs. It shows that it is relatively easy to create the initial models, and the revisions to evaluate design alternatives are straightforward. Analysts will typically evaluate other aspects of both the software and system execution models to study configuration sizing issues and contention delays due to multiple users of a scenario and other workloads that may compete for computer system resources. In these additional performance studies, the most difficult aspect has been getting reasonable estimates of processing requirements for new software before it is created. The process described here alleviates this problem. Once this data is in the software performance model, the additional studies are straightforward and are not described here. Information about these additional performance models are in (Smith and Williams, 1997).

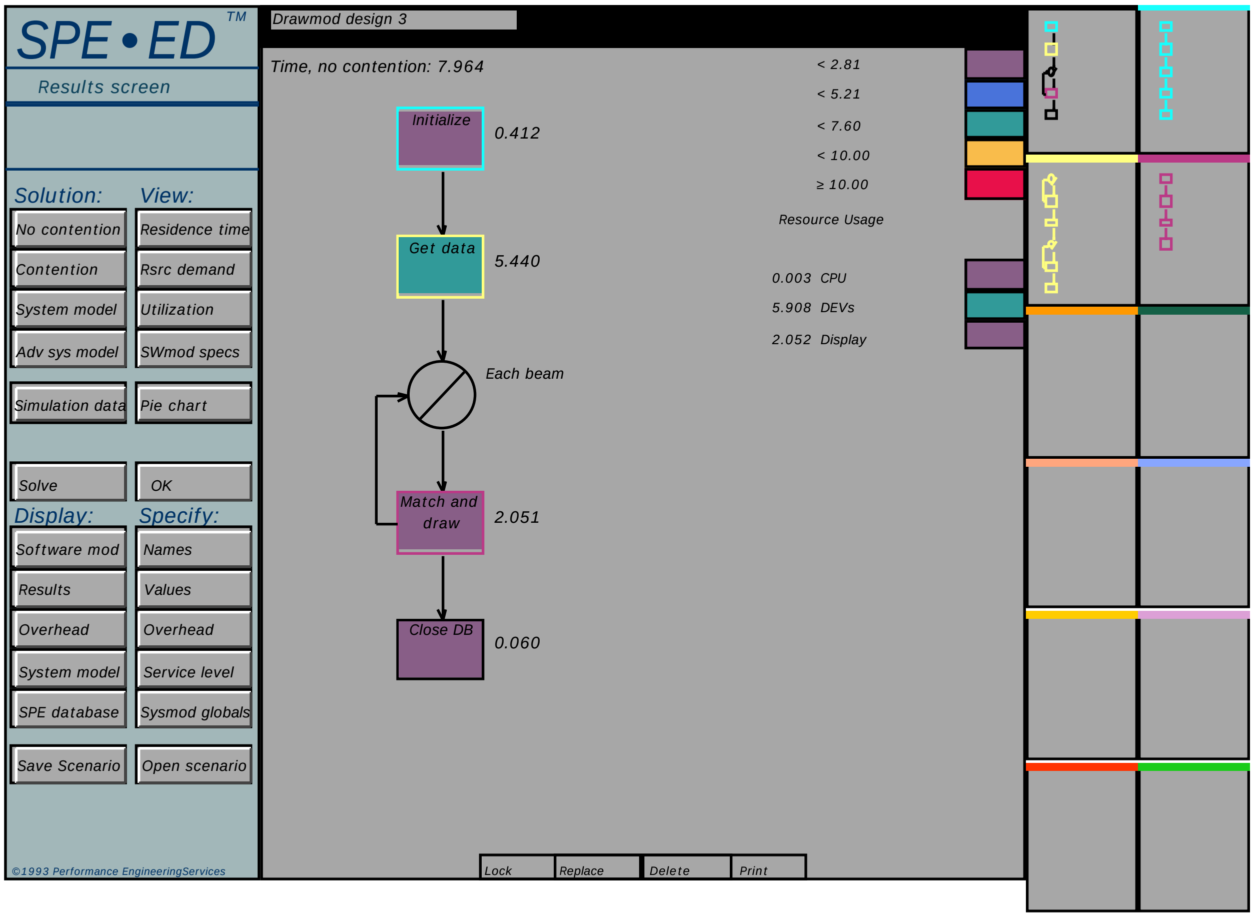

Figure 14. Execution Graph for DrawMod scenario for Design 3.

7 Summary and Conclusions

This paper has described a systematic approach to the performance engineering of object-oriented software that is critical to preventing performance failures. Performance failures may result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, missed market windows, and, in the worst case, the need to completely re-design the product or even cancel the project.

Object-oriented systems offer unique challenges for performance engineering due to the complexity of interactions between objects and the trend toward distribution of objects over a network. However, our experience has shown that it is possible to *cost-effectively* engineer object-oriented systems that meet performance goals. This paper has described the process of software performance engineering for object-oriented systems and illustrated that process with a simple case study.

The key to this process is the employment of Use Case scenarios as a link between object-oriented analysis and design models and performance models. Use case scenarios can be translated to execution graphs which serve as a starting point for performance modeling. This connection is a key step in enabling the performance evaluation of new object-oriented software systems. We illustrated the connection with a simple best-case *software* execution model. The software execution model is sufficient for many architecture and design evaluations. Even when more complex performance analysis is required, this first-step evaluation is essential to narrow the problem space and focus on the problems requiring analysis. Thus, the modeling approach uses deliberately simple models that are matched to the current level of knowledge about the emerging software. These models become progressively more detailed and sophisticated as more details about the software are known. Adaptive strategies are used to manage uncertainty. Thus, the modeling effort matches the level of knowledge about the emerging system and is less intrusive upon the development process.

The Use Case scenarios are vital to this SPE approach; they are also an important step in the design process. Scenarios have been shown to be useful to designers for reasoning about the problem and its solution. Other advantages of Use Cases are described in Section 3. Using scenarios as a starting point for the SPE analysis means that designers do not have to produce additional artifacts and performance analysts have a familiar description of workloads. This approach also facilitates communication between designers and performance analysts, lowering one of the barriers to using SPE.

The case study in this paper omits an important step in the SPE process: specification of software resource requirements. While it is a vital step in the process, the process of identifying specifications necessary, gathering data in performance walkthroughs, and specifying the performance data in the SPE tool is documented elsewhere (Smith, 1990), (Smith and Williams, 1997). We are currently exploring an approach for integrating the specification of performance requirements into Message Sequence Charts.

The case study illustrates the value of systematically connecting the software architecture and design models to SPE performance models. It provides quantitative data for alternatives to ensure that performance goals can be achieved with the selected design thus precluding tuning that may disrupt the design. It permits project managers to invest in alternatives that have a beneficial effect. Our technique for connecting design and SPE performance models preserves the benefits of both systematic design methods and systematic application of SPE methods. This preservation results from our particular combination of design and performance experience. Preserving the benefits of design and performance engineering is essential to effectively meet performance objectives of new systems with a design that is scalable, robust, maintainable, reusable and has other quality attributes.

This work is part of a larger project to make it easier for developers to perform initial performance assessments. One of the principal barriers to the widespread acceptance of SPE is the gap between the software developers who need performance assessments and the performance specialists who have the skill to conduct comprehensive performance engineering studies with today's modeling tools. Thus, extra time and effort are required to coordinate the design formulation and the design analysis. This limits the ability of developers to explore design alternatives. The matching of Use Case scenarios and performance scenarios, together with the use of a tool, such as *SPE•ED*, that automates key aspects of the SPE process represent a significant step toward achieving this goal.

As noted in Section 4, the translation of MSCs to execution graphs is currently a manual process. However, the close correspondence between scenarios as expressed in MSCs and execution graphs suggests that an automated translation may be possible. A future project will explore this possibility. A previous project developed an SPE meta-model that defines the information requirements for SPE (Williams and Smith, 1995). The SPE meta-model can be used by CASE tool vendors to add the capability to collect performance data as part of the design information. By collecting performance data within the design tool and automatically translating MSCs to execution graphs, it will be possible to export data from the CASE tool to any SPE tool that supports the meta-model. Thus CASE tools need not replicate the performance expertise already available. This offers a more cost-effective approach to supporting SPE. Our other future work will address models of distributed object-oriented systems, the specification of performance requirements, and additional tool features to automate SPE evaluations.

8 References

Auer, K. and K. Beck, Lazy Optimization: Patterns for Efficient Smalltalk Programming, in *Pattern Languages of Program Design, Volume 2*, (J. Vlissides, et al., ed.), Addison-Wesley, 1996.

Baldassari, M., et al., PROTOB: A Hierarchical Object-Oriented CASE Tool for Distributed Systems, *Proceedings of the European Software Engineering Conference, 1989*, Coventry, England, (1989).

Baldassari, M. and G. Bruno, An Environment for Object-Oriented Conceptual Programming Based on PROT Nets, in *Advances in Petri Nets, Lectures in Computer Science No. 340*, ed.), Springer-Verlag, Berlin, 1988.

Beilner, H., et al., Towards a Performance Modeling Environment: News on HIT, *Proceedings of the 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Plenum Publishing, (1988).

Beilner, H., et al., The Hierarchical Evaluation Tool HIT, in *Performance Tools and Model Interchange Formats*, (F. Bause and H. Beilner, ed.), Universität Dortmund, Fachbereich Informatik, Dortmund, Germany, 1995.

Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin / Cummings, Redwood City, CA, 1994.

Booch, G., *Object-Oriented Design with Applications*, Benjamin / Cummings, Redwood City, CA, 1991.

Buhr, R. J. A. and R. S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, Upper Saddle River, NJ, 1996.

Buhr, R. J. A. and R. S. Casselman, Timethread-Role Maps for Object-Oriented Design of Real-Time and Distributed Systems, *Proceedings of OOPSLA '94: Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, 301-316 (1994).

Buhr, R. J. A. and R. S. Casselman, Architectures with Pictures, *Proceedings of OOPSLA '92: Object-Oriented Programming Systems, Languages and Applications*, Vancouver, BC, 466-483 (1992).

Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Goettge, R. T., An Expert System for Performance Engineering of Time-Critical Software, *Porceedings of the Computer Measurement Group Conference*, Orlando, FL, 313-320 (1990).

Grummitt, A., A Performance Engineer's View of Systems Development and Trials, *Proceedings of the Computer Measurement Group Conference*, Nashville, TN, 455-463 (1991).

Hrischuk, C., et al., Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype, *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Durham, NC, 399-409 (1995).

ITU, Criteria for the Use and Applicability of Formal Description Techniques, Message Sequence Chart (MSC), International Telecommunication Union, 1996.

Jacobson, I., et al., *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA, 1992.

Kruchten, P. B., The 4+1 View Model of Architecture, *IEEE Software*, 12(6), 42-50 (1995).

Rational Software Corporation, Unified Modeling Language: Notation Guide, Version 1.1, Rational Software Corporation, Santa Clara, CA, 1997.

Rolia, J. A., Predicting the Performance of Software Systems, University of Toronto, 1992.

Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Selic, B., et al., *Real-Time Object-Oriented Modeling*, John Wiley and Sons, New York, 1994.

Shlaer, S. and S. J. Mellor, *Object Lifecycles: Modeling the World in States*, Yourdon Press, Englewood Cliffs, NJ, 1992.

Shlaer, S. and S. J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, Englewood Cliffs, NJ, 1988.

Smith, C. U., *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA, 1990.

Smith, C. U. and L. G. Williams, Performance Engineering Evaluation of Object-Oriented Systems with SPEED, in *Computer Performance Evaluation: Modelling Techniques and Tools*, No. 1245, (R. Marie, et al., ed.), Springer-Verlag, Berlin, 1997.

Smith, C. U. and L. G. Williams, Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives, *IEEE Transactions on Software Engineering*, 19(7), 720-741 (1993).

Turner, M., et al., Simulating Optimizes Move to Client/Server Applications, *Proceedings of the Computer Measurement Group Conference*, Reno, NV, 805-814 (1992).

Williams, L. G., Information Requirements for Software Performance Engineering, Software Engineering Research, Boulder, CO, 1994.

Williams, L. G. and C. U. Smith, Information Requirements for Software Performance Engineering, in *Quantitative Evaluation of Computing and Communication Systems*, No. 977, (H. Beilner and F. Bause, ed.), Springer-Verlag, Heidelberg, Germany, 1995.