Design of a World-Wide Simulation Web

by

Elaine Yang

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2003

© 2003 Massachusetts Institute of Technology
All rights reserved

Author
Department of Electrical Engineering and Computer Science
May 23, 2003

Certified by
Woodie C. Flowers
Pappalardo Professor Of Mechanical Engineering
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Design of a World-Wide Simulation Web

by

Elaine Yang

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2003

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The World-Wide Simulation Web (WWSW) is an emergent distributed simulation environment whose use model parallels that of the World-Wide Web (WWW). The idea was developed from working with two generations of the Distributed Object-Modeling Environment (DOME). A design for the architecture of the WWSW is proposed. The specification for the WWSW was determined by examining a list of desired features, deciding who the system users were and understanding use scenarios. The architecture design covers object model representation, server structure, client application structure, and graphical user interface guidelines. An implementation is complete of the core object model, client and server applications, and user interface concepts. Initial results have been promising since many desired features such as a flat object model representation, multiple model instances support, model solving, and advanced notification have been demonstrated. However, there is still a lot of work to be done to fully realize the complete WWSW vision.

Thesis Supervisor: Woodie C. Flowers
Title: Pappalardo Professor Of Mechanical Engineering
Acknowledgements

I thank David Wallace for sharing his DOME vision with me and allowing me to work with him on it. It’s been a privilege. Thanks also go to the members of the CADlab over the years for making the work environment a pleasant one. Special kudos to the crew of students that really pitched in and helped out this past spring: Qing Cao, Charles Dumont, Wei Mao, Sittha Sukkasi, and Jacob Wronski. I am grateful to Keith Thoresz and Renu Fondeker for taking over my DOME responsibilities so that I could go to school.

Chris Magee and the Center for Innovation in Product Development sponsored my work throughout my entire time and The Ford Motor Company has been a major sponsor of the DOME project for many years. I hope you reap the benefits.

I also want to thank Woodie Flowers for being the most supportive thesis supervisor one could ask for. Thanks for sticking by me!

Thanks also go to Franklyn Turbak who taught me a lot about programming and work, and inspired and encouraged me to go to school again. I’m finally done!

Also, thanks go to my advisor, John Gutttag, and Anne Hunter who believed in me even when things weren’t looking good. Special thanks go to Anne for her incredible support of Course 6 students through all the years.

Finally, thanks go to my family for their love and support. Mom, Dad, James, Richard, and Nanette — I hope all your dreams of happiness come true.
# Table of Contents

Acknowledgements ........................................................................................................ 5
Table of Contents ............................................................................................................. 7
List of Figures .................................................................................................................. 9

1 Motivation for a World-Wide Simulation Web .......................................................... 11
   1.1 DOME History ..................................................................................................... 11
   1.2 Second-Generation DOME Weaknesses ............................................................ 12
       1.2.1 No Multiple Model Instance Support ......................................................... 12
       1.2.2 One Application, One Mode ...................................................................... 13
       1.2.3 "File System" Representation of Objects in Models .................................. 14
       1.2.4 No Model Solving Support .................................................................... 17
       1.2.5 Miscellaneous Limitations ................................................................... 19
       1.2.6 Outdated Technologies ........................................................................ 20
       1.2.7 Intellectual Property Issues ................................................................... 21
   1.3 Third-Generation DOME .................................................................................. 22

2 World-Wide Simulation Web Specification .................................................................. 23
   2.1 Desired Features ............................................................................................... 23
       2.1.1 Accessible Infrastructure ........................................................................ 23
       2.1.2 Simple to Implement Visualization Support ............................................ 23
       2.1.3 Model Interfaces ..................................................................................... 24
       2.1.4 Real-time Status Information .................................................................. 24
       2.1.5 Multiple Model Instances Support ......................................................... 25
       2.1.6 Collaboration Support ........................................................................... 25
       2.1.7 Version Support ....................................................................................... 26
       2.1.8 Accurate, Consistent and Efficient Calculations .................................... 26
       2.1.9 Standard Technologies ........................................................................... 27
       2.1.10 Distributed Network Security ............................................................... 27
   2.2 Users and Use Scenarios ...................................................................................... 28
       2.2.1 Model Builder .......................................................................................... 28
       2.2.2 Server Administrator ............................................................................... 29
       2.2.3 Model Deployer ...................................................................................... 30
       2.2.4 Model User ............................................................................................. 30
       2.2.5 System Integrator .................................................................................. 31
       2.2.6 Team Manager ....................................................................................... 31
       2.2.7 Playspace Member .................................................................................. 32
       2.2.8 Model Editor .......................................................................................... 32

3 World-Wide Simulation Web Architecture .................................................................. 33
   3.1 An Application Suite ........................................................................................ 34
       3.1.1 The User Applications ............................................................................ 34
       3.1.2 Server Utilities ....................................................................................... 34
   3.2 Server Architecture ............................................................................................. 35
       3.2.1 Model Repository ..................................................................................... 35
       3.2.2 Authentication ......................................................................................... 35
Table of Contents

3.2.3 Model Server ........................................................................35
3.2.4 Plugin API ........................................................................37
3.2.5 Client API ........................................................................37
3.2.6 Issues for Implementers ......................................................37
4 Implementation and Results .......................................................38
  4.1 Selecting Technologies ........................................................38
    4.1.1 Programming Language ..............................................38
    4.1.2 Database ..................................................................39
    4.1.3 Network Protocol ......................................................40
    4.1.4 Relation Implementation ..........................................40
  4.2 Current Implementation .......................................................41
    4.2.1 Client Application .....................................................41
    4.2.2 Server ....................................................................43
5 Making the World-Wide Simulation Web a Reality .........................45
  5.1 Desired Features ................................................................45
    5.1.1 Pluggable Architecture ..............................................45
    5.1.2 Constraints ................................................................46
    5.1.3 Custom Networking Protocol .....................................46
    5.1.4 Distributed Network Security .....................................46
  5.2 Towards an Open Source Software Development Model .................46
  5.3 Conclusions ....................................................................47
References ..................................................................................49
List of Figures

Figure 1.1 Using remote aliases in wrong and right ways .................................................. 15
Figure 1.2 Solving within a single model ........................................................................... 17
Figure 1.3 Solving of two integrated models ..................................................................... 18
Figure 2.1 A native DOME model ..................................................................................... 28
Figure 2.2 A plugin DOME model .................................................................................... 29
Figure 3.1 The World-Wide Simulation Web (WWSW) ...................................................... 33
Figure 3.2 Native DOME model vs Plugin DOME model ................................................ 36
1 Motivation for a World-Wide Simulation Web

In the second half of 2000, the second-generation Distributed Object-based Modeling Environment (DOME2) had served its research purpose and was obsolete. Efforts to patch its inadequacies were hampered by lack of access to the kernel source code and the architecture of the implementation itself. While it was a useful prototype in 1998 and allowed a number of pilot studies to be performed to demonstrate and develop the concept, by 2000 it was showing its age. Enabling technologies had advanced and the choices made in 1998 were no longer the best choices for 2000. Therefore, it was decided that a new version would be designed and implemented.

1.1 DOME History
The idea for DOME belongs to David Wallace, who first developed it in his PhD thesis (Wallace, 1994). The first generation DOME implementation (DOME1), developed 1996-1998, showed that emergent distributed computational simulations were possible (Pahng, et al, 1997; Pahng, et al, 1998). In DOME1, the programmers coded the distributed system. DOME2 (1998-2001) was implemented to explore how users could build distributed simulations in an environment similar to the World-Wide Web (WWW) (Abrahamson, et al, 2000; Borland, et al, 2000). DOME2 tried to answer the question of how would the user build such a system? The DOME2 goal was to develop a system that non-programmers could use.

The DOME2 implementation required that users have access to a running DOME server. This server could either be started on the user’s computer or any other computer on a network that the user would have access to via standard network technologies. The
user would then start the DOME client which was a Java applet run from a standard web browser. Within this applet, an environment was created which allowed users to view and run models on DOME servers, and to build and link models. The system proved to be very easy for new users to learn and successfully met its goal of demonstrating a web-based method for building distributed simulations.

1.2 Second-Generation DOME Weaknesses
DOME2 was used in a number of pilot projects successfully. Many companies were interested in trying it out. However, there were some fundamental capabilities which had been requested which were not implemented and would be very difficult due to the design of DOME2. The idea of how things should work also evolved with more experience with the software. By 1999, the DOME concept had evolved to that of a World-Wide Simulation Web (WWSW) in which participants all over the world should be able to publish, use, and link models together in a manner similar to how users currently publish, use, and link material on the World-Wide Web (WWW). Eventually, a number of limitations in DOME2 made it impossible to continue work with the WWSW concept. These limitations are discussed below.

1.2.1 No Multiple Model Instance Support
The architecture of DOME2 supported a very limited use scenario — there could only be one running instance of each model and everything was in a global scope. While the presentations talked about multiple users being able to each run their own instance of a model, only a single model instance was ever demonstrated. Audience members, myself included, were quite confused as to how the multiple model instance scenario would work given what was presented. It was not until I had worked with the actual
implementation itself did I understand that it was not possible to have separate running instances of a model. It was an inherent limitation of the kernel\(^1\) to which we did not have access.

1.2.2 One Application, One Mode
The DOME2 implementation included a single client application. In this client application, there was just one mode\(^2\). Everything could be done at any time. When the client application was running, one could create a model, edit it, and run it, all in the same window. While this was convenient, it was also not scalable. The single mode implementation required that one needed to access a DOME server in order to build a model because the model was running during the build process. The implementation supported multiple clients to the same model, so while one person was building on one machine, many people could be watching the process unfold. This was neat, but costly, since the build process generally involved quite a bit of editing, and synchronizing all the network clients to the model involved network resources that could be better used elsewhere. Generally, when one is editing a document, one does so locally and then shares the completed document with others. A bigger Achilles heel was that the single-mode model meant that versioning and locking models were practically impossible. There was no clear point in which one could say that the model was “finished and ready for use” and then “here is the next version”. A model was always editable (since distinct edit and view modes did not exist). This meant if one built a model and made it available

\(^1\) The DOME kernel provided the core DOME functionality including an object representation and networking services. The kernel code was written and owned by Matthew Wall.

\(^2\) A program mode restricts users to a particular set of functions. For example, in vi, the classic unix text editor, there was a text entry mode in which characters typed were interpreted as part of the document and a command mode in which characters typed were interpreted as commands. There were also multiple edit modes (insert, append, and open modes).
for others, anyone who used the model could edit it even if the original creator did not intend for the model to be edited.

1.2.3 “File System” Representation of Objects in Models
The object representation upon which DOME2 was based bound objects in the model in a strict hierarchy. The objects in a model behaved similarly to files on popular computer operating systems’ file systems. DOME containers were used to organize model objects similar to how folders are used to organize files. The original model object was located in a particular container. However, aliases to that model object could be created in any container similar to aliases/shortcuts/symbolic links to files found in modern operating systems. Two issues with the implemented “file system” model of organizing model objects made model building especially difficult.

The first problem was that remote aliases did not work the same way as local aliases. In DOME2, aliases could either be created to objects in the current model (local alias) or to objects in other models (remote aliases) which could be on a different machine than the current machine. In this way, models could be linked across the DOME network since an alias to a remote object allowed one to work with that object. One expects that the state of an alias reflects the state of the original object, and vice versa. Changes to the value of either the alias or the original object should result in both objects reflecting the same value. This was true of local aliases. However, remote aliases were not bi-directional. A change in the alias would propagate to the remote, original object. However, changes to the original object would not propagate to remote aliases. As a result, one needed to be very careful about where to create remote aliases when connecting two models (Figure 1.1).
a) Two models to be linked together. Goal is to have C drive X.

b) Create remote alias in Model2 to C in Model1. Define relationship that C drives X. Does not work because new values of C do not propagate to the remote alias for C.

c) Create remote alias in Model1 to X in Model2. Define relationship that C drives X. Works since new values of alias to X are propagated to the original X.

Figure 1.1 Using remote aliases in wrong and right ways.
This was a constant source of errors when building distributed networks of models. A second problem with remote aliases was that one needed to delete all remote aliases before deleting the original object. Otherwise, the models involved would crash. This was problematic since it was not always easy to remember where all the remote aliases to an object exist. This would be a similar requirement to not being able to remove a web page from the WWW unless all hyperlinks to that web page were removed first. Local aliases, though, could be deleted after the deletion of the original object. This is similar to removing a web page, a process which breaks links to the page, and then removing the broken links.

The distinction between an original object and an alias made making changes to the model cumbersome. For example, assume the parameter width is added to a model box1 and an alias to width is added to a model box2 and another alias to width is added to a model box3. This means that the width for all three boxes should always be the same. Later we decide that only box2 and box3 should have the same width, and box1 should have a different width. It is not possible to just replace width in box1 leaving the width parameters in box2 and box3 intact since box2 and box3 have aliases to an object and the original object would no longer exist so the aliases are invalid. Instead, the widths for both box2 and box3 need to be changed. They could either be aliases to a parameter not in either box, or one of the boxes could contain the original parameter and the other box would contain an alias to the original parameter.
Model1 drives both M and N of Model2. Without distributed solving, i.e. solving across models, a change in A would execute r1 resulting in changes to M and N in Model1. Model2 would receive the changes to M and N and execute r2 twice — once for each change received. With solving, Model2 would know that the changes to M and N are coupled and therefore would wait for both changes to arrive before executing r2.

Figure 1.3 Solving of two integrated models.

For example, perhaps the length and width of a box is constrained in the program to not differ by more than 5 units. Originally the length and width are 5 and 8. The next values will be 20 and 15. If one of the values is changed but not the other, then the model no longer meets the constraints defined and it an error results when the model is executed. However, if both values are set and then the model is executed, everything is fine. To solve this problem in DOME2, users had to explicitly implement solving logic into their models, which was difficult to do for large complex systems. One of the core DOME concepts was that it should support an emergent system definition as opposed to an explicit system definition. In other words, users should be able to add to a system without knowing about the details of every other part of the system. Implementing solving logic requires users to understand how other parts of the system work thereby completely defeating the goal of being able to build emergent systems.
1.2.4 No Model Solving Support

The DOME2 implementation did not support the solving of models. Solving within a single model was added as a layer on top of the kernel (Figure 1.2) within the CADlab in an effort to have more reasonable computational characteristics.

![Diagram](image)

**Model1**: Parameters A, B, and Z are related via a relation r1. For example, maybe r1 is defined as $Z = A + B$. When A is changed, r1 is executed and Z is updated. When B is changed, again r1 is executed.

**Model2**: When A is changed, r1 is executed resulting in new values for both M and N. Without solving, relation r2 would execute twice — once for each input change received. With solving, though, r2 would only execute once because it would know that given the graph structure for the model, M and N always change together (that is, if M changes, N will change as well and vice versa). Therefore, if either M or N changes, r2 will wait until the other change is received before executing.

![Figure 1.2 Solving within a single model.](image)

Solving for distributed models (Figure 1.3) was not possible. Without distributed solving, the models might execute more often than necessary. This is not a problem if the models execute quickly. However, if a model executes on the order of minutes or hours, unnecessary model executions can quickly become costly! Another problem was that for many models (such as CAD models) ignoring correlations between inputs and executing each change independently would cause the models to crash.
1.2.5 Miscellaneous Limitations
There were numerous other limitations in the architecture. While it supported units compatibility checking, the process of adding new units required full access to the source code and a new recompilation of the entire system. The units implementation itself was limited in that it only supported units which were related via multiplicative factors. So, it was not possible to do a Celsius to Fahrenheit conversion. The relations\(^3\) were all compiled which meant that they ran faster after the first time, but the first hit was very slow! For example, a DOME2 pilot application completed with the Ford Motor Company involved 3000 design variables took several hours to compile and start (Abrahamson, et al., 2000). Distributed models needed to be started in a very particular order. If the order was not followed, the system would crash. There were a number of idiosyncrasies that, if one knew about them, one could work around them. However, there were so many that it really required a lot of concentration to do everything in the right order to avoid breaking the system. These problems did not result from sloppy implementation—they were consequences of an architectural design that did not foresee and could not be adapted to the needs of the WWSW concept as it developed. As a result, errors were common and server or client restarts happened numerous times in the course of daily use. The client implementation was unreliable and slow. The client would often run out of memory and have to be restarted. The client was also unable to keep up with data that was sent back to it. Models would finish execution and the client would still reflect changes from that execution for several seconds afterwards.

\(^3\) A relation was the data structure that allowed users to specify how parameters in the model were related. For example, one might define area = length*width.
1.2.6 Outdated Technologies
The technologies used in the implementation were cumbersome and outdated. In order to run DOME on a machine, a number of software packages had to be installed. This process, by itself, took about 2 hours. Part of the problem was that to build models that could do anything, one needed to have relations in the model. Relations were compiled using C++, so generally a C++ compiler needed to be installed since typical users in a product development environment did not have one.

The kernel was implemented in C++. The client layer was implemented in a combination of C++ and Java. Using C++ meant that the implementation was only available for platforms for which it was compiled. In reality, only the Windows operating system platform was supported since it was never easy enough to port the code to other platforms.

CORBA was used as the networking standard which would allow objects to be used from remote machines. While CORBA was received with a lot of fanfare when it was introduced in the early 1990s, by 2000 it had clearly been surpassed by Java networking technologies. CORBA implementations were not interoperable, CORBA was not easy to use, and the limitations in remote aliases were blamed on CORBA. Properties of CORBA were responsible for the strict requirements on the order in which models had to be started in distributed simulations.

The C++/Java client meant that something needed to be used to bridge the two programming languages. The Java Native Interface (JNI) was chosen because it is the standard way to accomplish the task. However, JNI is also somewhat difficult to use. Requiring its use limited the number of people who could work on adding new
Motivation for a WWSW

functionality between the client and the server since people had difficulty debugging JNI code.

Overall, the technologies used in the DOME implementation meant that there was a lot that had to be learned to do programming for DOME. There were very few people who knew enough to develop new code and the CORBA/JNI code was mostly copied from place to place. Finally, although the code was written in C++, it used a number of C standards. The difference meant that string manipulations were harder than they needed to be because the C++ string templates were not supported. Memory management was also more prone to error since C code requires more vigilance than C++ code in that area.

1.2.7 Intellectual Property Issues
The final problem with the DOME implementation was one of intellectual property issues. The DOME implementation was divided into two parts, the kernel and the client. The kernel was written and owned by Matthew Wall based upon his interpretation of DOME1. The CADlab had no access to the kernel source code. In 1999, Matt started a company using the kernel and his efforts quickly turned towards developing the next version of the software. The CADlab was left with the kernel and no support. New features were requested, but they required a major overhaul of the kernel and the changes requested were not the same changes Matt was making for his commercial version. So, kernel development came to a standstill. The client level code was developed in the lab primarily by Nick Borland and Ben Linder. The CADlab had ownership of this code. Much was done by the creators of this code base to get around limitations in the kernel. However, there is only so much patching that can be done. Eventually, there was nothing
left that could be done to the client side to work around architectural and conceptual limitations of the kernel.

1.3 Third-Generation DOME
The third-generation DOME implementation (DOME3) was started at the end of 2000 (Wallace, et al, 2002). For the first year, most of the focus was on brainstorming what was desired and how to accomplish the most wanted features. David Wallace and I worked to together during this phase of development. The second year started off with prototyping of different parts. The object model was designed and implemented. A simple server was set up to test the concepts. Eventually, the current implementation was started. In this version, the object model was also implemented and a framework for the client application and graphical user interface (GUI) was designed. In the third year, Keith Thoresz and Renu Fondeker assumed primary responsibility for the development of DOME3. They, along with David, the students of the CADlab (Qing Cao, Sittha Sukkasi, Jacob Wronski, Charles Dumont, and Wei Mao) and I implemented the rest of the client and server applications. In this work, though, I primarily describe the requirements and architecture of DOME3.

The goal of DOME3 is to build a framework that will enable the WWSW vision to become a reality. It would be a system that allows users to create emergent distributed systems in a manner similar to that used in the WWW, supports proper mathematical solving of models and distributed models, and scales well for both number of users and complexity of models.
2 World-Wide Simulation Web Specification

The first step to starting the design of anything is to figure out what you’re designing. In software, the questions to be answered are what should it do, and who will use it and how. Over the course of many conversations with David Wallace, a picture emerged of what DOME3 should support.

2.1 Desired Features

2.1.1 Accessible Infrastructure
The overall goal is to design an infrastructure to support the WWSW vision. Like the WWW, the infrastructure would ideally be accessible to users of all popular consumer operating systems (at a minimum Windows, Macintosh, Linux, and Unix variants). Technologies chosen for use within the implementation must be easy and quick for users to install and be widely available for the different operating systems. Ideally, different code branches would not have to be maintained to support different operating systems. To make the simulation web work, people need to participate in it. The more participants there are, the more likely participants will find something of use to them in the WWSW. Therefore, having a widely accessible infrastructure so that almost no one is excluded is of utmost importance.

2.1.2 Simple to Implement Visualization Support
One thing that was often desired in the old version of DOME was alternative visualizations. The implementation should be structured so that it is easy to provide users with different ways of looking at the systems they are building. The default “tree view” is good for viewing hierarchical items, but it’s not possible to easily show which items are
the same items in a tree view. The infrastructure should also support custom graphical user interfaces (GUIs). This is important because the DOME implementation can only provide a small set of standard visualizations. They will clearly not meet the needs of everyone. So, it is important that it is easy for relatively beginning programmers (and ideally non-programmers as well!) to be able to add their own custom designed GUIs for visualizing systems.

2.1.3 Model Interfaces
In a similar vein, it is also often desired to customize the set of objects that can be viewed by different audiences. A model may have hundreds of variables, but maybe only a handful are useful to the normal user of the model and a different subset is of interest to a special group of users. To support this concept, it must be possible to define multiple interfaces to the model. An interface is a subset of the values in the model. To enforce consistency, all access to a running model would be made through one of the model’s interfaces. This provides an easy way to enforce access control since privileges can be set on particular interfaces.

2.1.4 Real-time Status Information
Real-time status information from the server to the client was often desired. If the server was not on your machine, you would often wonder if things were ok if they were taking a some time. A number of ideas go in this category. Models can present real time status such as if they are running, stopped, paused, etc. Models can estimate how long they will take to run a computation. If it is more than a few seconds, the user can be presented with information about how long the wait would be. A countdown timer or stopwatch could be used to show users how much longer they would need to wait before seeing the results of
some computation. Finally, it is helpful for users to know what parts of the model will be executing so that they have a sense of what parts of the model will be executed and how much computation is left to be done. Support for advanced notification of parameter status helps users visualize model execution and provides feedback that something is happening before the final answers are available.

2.1.5 Multiple Model Instances Support
Multiple model instances support is needed if the simulation web idea is to become reality. Multiple instances of models should be able to be run independently of each other at the same time. The parallel with the WWW is the following: multiple users can query a search engine at the same time and they each get their own copy of the answers corresponding to their own query. Without multiple instance support, there is only one search engine web page. When one-user types in a query and hits “go,” the answer that user gets back is also returned to all other current viewers of that web page. Only one query can be run at a time. This obviously does not scale well. Users must be able to run models in a space that is not shared with other users.

2.1.6 Collaboration Support
Collaboration support is also needed. Users may want to see the same instance of a running model. However, we need to be able to define multiple collaboration scopes (instead of just one global one as in the old DOME implementation). So, if we have two project teams, the members of each project team need to be able to work together and share the same models, but the two teams should be completely independent. The infrastructure should also be designed so that it will be easy to add groupware such as chat, common whiteboards/pointers, and video-conferencing.
2.1.7 Version Support
Product Data Management (PDM) and Product Life-cycle Management (PLM) have been the ongoing buzzwords in corporate environments for the last few years. PDM/PLM allows companies to manage their data, in particular, version it. In order for the WWSW to be acceptable to the corporate environment, it needs to be able to support versioning either inherently or via adaptors to existing versioning systems. Corporations will want to keep track of which version of an engine model and performance model were used together in a simulation to justify a particular engineering decision. Version support also allows users to go back and use an old version if newer versions turn out to be flawed.

2.1.8 Accurate, Consistent and Efficient Calculations
The architecture should have, built-in, advanced calculation support. A spreadsheet is the lowest common denominator. One designs a spreadsheet and then can run simulations by changing values. The moment a value changes, the spreadsheet is triggered. That behavior is reflected in the old DOME2 implementation. The WWSW, however, needs more than just spreadsheet behavior in order to reasonably manage and solve large simulation systems. As discussed earlier, solving, and in particular, distributed solving, needs to be supported in such a way that it would be easy to swap algorithms in and out of the system to optimize for solving behavior. Unit compatibility checking is a must as all too often mismatches in units arise as problems when working in systems that involve many different models resulting in costly financial fixes or product failures. There also needs to be an easy way for users to add custom units and for those units to be distributed to other users. Constraints on data are also needed so that invalid values are not propagated through the system breaking models since not all models are robust to invalid
data. All of these features help to ensure that the calculation will be more efficient and correct.

2.1.9 Standard Technologies
Although the infrastructure will be built by a small team of programmers initially, the goal would be for the software to evolve into something that many could participate in and add to. In order to fulfill this goal, two design attributes must be kept in mind. First, the technologies used should not be obscure. It needs to be designed well so that new programmers will be able to quickly become productive contributors to the system. Second, everything needs to be designed with “plug-and-play” in mind. Every component of the system should be thought of as something that users in the future will want to be able to contribute to. So, it should be easy for users to add units, data types, relations, models, solvers, plugins (links to external applications), GUIs, etc.

2.1.10 Distributed Network Security
Finally, the system needs to support network security. Network security involves two components: authentication of users and granting privileges on resources. Distributed network security is an active research area today with no standard solutions. Security is key, though, in order for the system to gain corporate acceptance. Companies who participate by putting their models on the WWSW want to be assured that they will not lose control of or leak information about their proprietary models. The privilege system will be complex since fine-grained control is required and there are many types of privileges one could have. However, it needs to be simple enough that users who want it will use it instead of avoiding the system.
2.2 Users and Use Scenarios
When designing a system, it is important to think ahead of time about who the users are and how they will use the system. These are typically depicted in user diagrams and use case diagrams. In this section we will list the envisioned users of the system along with how we think they would want to use the system.

2.2.1 Model Builder
The model builder is a user who builds models. A model builder may want to build native DOME models (models that are entirely built in DOME) or plugin DOME models (models that wrap an existing model in some third-party application). A similar process is used when building native and plugin models. To build a native DOME model (Figure 2.1), a user creates parameters and then links them via relations which allow the user to specify how parameters are related to each other. For example, a user could say that the parameter area is equal to the parameter length squared. For each parameter, the user specifies a data type (such as a real number, an integer, a vector, a matrix, a boolean value, a string, etc) and sets initial values and appropriate constraints (such as units). The parameters and their relations together define the model. In a native DOME model with solving support there is a simulation engine which determines in which order the relations should execute.

![Figure 2.1 A native DOME model.](image)
A plugin DOME model (Figure 2.2) follows a similar procedure except, in this case, the computation is done via a call to an external application. So, only parameters are available in plugin models. The relationship between the parameters is defined by the external model. A mapping from the parameter in DOME to the external model is specified by the user (for example, the length is cell A3 in the spreadsheet). While the relationship between parameters is defined externally, DOME needs to know what it is in order to be able to solve the model correctly. Therefore, the user will also need to define the causal relationships between the parameters in the model as represented by the graph drawn in Figure 2.2.

After building a model, the user defines interfaces for different sets of users. If desired, the user can create an interface that contains the complete model. Once the user is done, the model and its interfaces are saved. All of this is done locally on the users machine. The user can open up the model and edit the model and interfaces.

2.2.2 Server Administrator
DOME servers are needed to host running models (much like web servers are needed to serve web sites). Someone is responsible for installing and maintaining the DOME server. This is the server administrator. The server administrator initially installs and
configures the DOME server on a particular machine. Then, the server administrator can allow users to place models to be hosted on the machine. To do this, each user must be added to the list of users with access privileges to place models on the machine. Users may want to restrict access to their models by specifying which users have access. The DOME server would either have to be configured to authenticate against existing repositories of user information or the list of users would need to be added to the DOME server itself. The server administrator would have the authority to disable or remove users from the system and full control over removing/deleting models from the server. Log data such as how many accesses per model or how many models are running would be available to the server administrator.

2.2.3 Model Deployer
A model deployer is a user who has permission to place a model on a DOME server. This may or may not be the same person as the model builder. For example, one can design a poster and distribute it around town, or one can design a poster and delegate the distribution of the poster to someone else. That is the difference between the model builder and the model deployer. The model deployer would log into the DOME server in which the model is to be placed. Then, he would need to specify which model on his local system to deploy and where on the remote system to place it. Finally, he would have to specify which interfaces to make available and to whom they would be available.

2.2.4 Model User
Assuming there are models on DOME servers, users can access the server and run models. This is very similar to interacting with web pages on the WWW. Users start the browser application and type in the name of a server. This will give them a view of which
models are available. They select the model they want and see which interfaces of the model are available. Finally, they select an interface with which to interact with the model. Once an interface is started, users can change values of input parameters, submit the values to the server, and view the results. In the future, one can imagine that there might be search applications which would help users find the models they want.

2.2.5 System Integrator
Now that we have models available on the simulation web, users will want to link models together. This process is called integration, and one who integrates models together is sometimes referred to as a system integrator. The system integrator will create a project. A project is a model that contains other models. Other than that, it behaves very much like a standalone, simple model. In the project, the system integrator collects resources that it will use from the WWSW. Then, components of those resources are added to the integration model. Finally, the system integrator defines how those resources will be linked together.

Once a project is defined, interfaces can be defined for a project (just like for a model). Once the project is saved, someone with appropriate privileges will need to deploy it onto a DOME server so that users will be able to run the project.

2.2.6 Team Manager
Groups of users will want to use DOME in a collaborative fashion meaning that they will want to be able to all see the same running instance of a group of models. Groups of users who want to collaborate together are defined in DOME playspaces. The person who sets up the playspace (most likely the team manager) would need to have the privilege to do so on a DOME server since playspaces are managed by DOME servers. Once a playspace
is set up, all users are notified of its existence so that they can work in the playspace when they want to collaborate.

2.2.7 Playspace Member
The playspace member starts up a DOME browser and navigates to the desired playspace. The playspace is started. As a member of the playspace, he can start models in the playspace as well as view models that other members have started. Members can use supported groupware tools such as an instant messaging program to communicate with other members who are online at the time.

2.2.8 Model Editor
Models and projects which are deployed will likely need to be modified. They may be modified by the original model builder or by someone else to whom the model deployer gave edit permissions. If the user who desires to edit the model does not already have a copy of the model on his local machine, he can download the model from the server after being authenticated. This process is known as checkout. Once the model is on the local machine, it can be edited. Once the editing is completed, it must be deployed again onto the DOME server. If it is to be deployed to replace the copy on the server, the process is known as redeployment. Otherwise, it will create a separate copy on the server so that both the new and the old model versions would be available at the same time. Either way, all versions of the model are archived. It is just a matter of whether or not the versions are categorized as being two versions of one object (redeployment) or as two objects (new deployment).
3 World-Wide Simulation Web Architecture

A software specification can be satisfied by many different system architectures. In this chapter an architecture is proposed for the specifications outlined in Chapter 2. The World-Wide Simulation Web will consist of two parts: a server application and a client application. The server application enables the concept of distributed integrated simulations available to clients on the Internet. Servers exchange information with other servers to coordinate the models in each distributed simulation. The client allows users to participate in the WWSW. At a minimum, users should be able to create models, use models, and link models. The client would connect to the server in order for a user to be able to run models in the WWSW. Figure 3.1 illustrates the high-level architecture for the WWSW.

The WWSW (the cloud) is a network of servers. Users of the WWSW use client software. Client1 is building a model for the WWSW and does not need to be connected to the server. Client2 is deploying a model so he is moving information to Server2. Client3 is running a standalone model on Server1. Client4 is running a distributed model by connecting to Server3 which, in turn, communicates with Server2.

Figure 3.1 The World-Wide Simulation Web (WWSW).
3.1 An Application Suite
The client application is a catchall term used to represent the software that will enable users to participate in the WWSW. As noted previously, there are many different users for the WWSW, each of whom has different needs. Therefore, the "client application" is actually a suite of applications. Whether one chooses to distribute the suite as a single software application or as many applications is purely an implementation and/or business decision.

3.1.1 The User Applications
The user applications needed to support the WWSW are very similar in concept to the ones currently available for the WWW. Users will need an application to build models and integrated simulations. Users will need an application to move the systems they have built to a server so that it can be used on the WWSW. Users will need an application that allows them to browse for what is available on WWSW servers. Finally, users will need an application that allows them to run models in the WWSW. Running a model means being able to view the model and interact with it. There are many other applications that users would find useful, such as a search engine for the WWSW. However, the goal here is to just specify the minimum set of applications needed to provide the required functionality.

3.1.2 Server Utilities
The server administrator is a different type of participant in the WWSW. Although servers are generally started once and left alone for long periods of time, it is useful to have a suite of server utilities available to help the server administrator if there are problems or if the server needs to be reconfigured. At a minimum, the server administrator should be able to start and stop the server remotely (from a machine other
than the one the server is on). Ideally, he would also be able to customize access to his server and view access, usage, and error logs of the server.

3.2 Server Architecture
The DOME server, like modern web servers, is an application with many roles. A well-designed server will have a modular architecture in which it will be easy to substitute different implementations for each module or to use multiple modules for a particular function.

3.2.1 Model Repository
The DOME server must have a repository of models to serve to users. How that repository is organized is left to the implementer. The DOME server must have a module that supports model management. This includes allowing users to upload models to the server, reload models to the server, and delete models from the server. In order to prevent abuse, it is expected that authentication would be used to determine who would have privileges to change the model repository. Again, it is left to the implementer to decide how fine-grained the access control would be.

3.2.2 Authentication
The DOME server must have an authentication module which would permit server components to use authentication, if desired. The authentication standard is left to the implementer. However, it is expected that the authentication support would be designed to allow different implementations to either be used concurrently or easily swapped.

3.2.3 Model Server
At the core of the DOME server is the model server which executes models on the server. Two basic types of DOME models are supported. A native DOME model uses DOME’s
simulation engine. DOME is similar to packages such as Matlab and Mathematica in that one can build models entirely in DOME and execute them. This is possible because DOME has its own set of objects and a simulation engine which supports calculations using the object model. A plugin DOME model uses the simulation capabilities of an external software program (Figure 3.2).

![Diagram](image)

The native DOME model includes a simulation engine that decides when relations in the model should execute. The plugin DOME model has DOME parameters that are connected to parameters in the external application model. The external model decides when to execute itself. The relationship between parameters in the plugin model (represented by the graph shown) can only be known if the model builder specifies it in DOME.

**Figure 3.2 Native DOME model vs Plugin DOME model.**

The model server must also support the execution of compound models (models which include a collection of DOME models). This involves the need to start all the components in the compound model when it starts and shutting them all down when it is closed. The model server connects the component models together so that model changes propagate to the models that need the changes.

The model server must be able to support clients to the models. Clients would want to be able to view the current status of models, and start, run, and stop models. It must be possible to have multiple instances of the model exist at the same time. This would enable different users to each execute their own copy of the model. The model
server must also support multiple clients to the same model so that users could share a copy of the model.

3.2.4 Plugin API
An Application Programming Interface (API) must be defined to specify the interface between the model server and commercial off-the-shelf (COTS) software. This Plugin API is used to develop “plugins” which are implementations of the API for particular programs (e.g. Excel or Matlab). The Plugin API should be designed to be easy to understand and implement. The mechanism for installing plugins into a DOME server should be straightforward and simple.

3.2.5 Client API
A client API must be defined to specify how clients interact with the DOME server. The client API should be supported in a number of computer languages so that it would be easy to write custom clients to DOME servers which would allow programs in many languages to use the WWSW. The client API should be well designed with care taken to consider ease of use and flexibility.

3.2.6 Issues for Implementers
The server is intentionally under-specified. There are many decisions left to the server implementer. For example, no particular security mechanism is required, though the specifications ask for network security support. Here we have only specified the ones needed to enable the WWSW to work. The details of the additional functionality desired are undefined which is why they are not discussed here.
4 Implementation and Results

The implementation of DOME3 has been the result of the work of many individuals. I was responsible for making the most of the key technology decisions that are explained below. After that, the state of the current implementation will be analyzed for how it does or does not match the specifications.

4.1 Selecting Technologies
Choice of technology is important. A good choice is one that will not be outdated too soon, is easy to use, and has the right amount of flexibility so that what one wants to do can be accomplished reasonably.

4.1.1 Programming Language
The first question was which programming language to use? Since we wanted cross-platform capability (one code base works on multiple operating systems), a cross-platform language was desired. While one can write cross-platform code in C++, it must be compiled separately for each platform and presents maintenance problems. There were a number of cross-platform languages available including Java, Python\(^1\), Perl\(^2\), Ruby\(^3\). Much of the initial prototyping was done in Python. The Python language was simple, easy to learn, and supported networking out of the box (as does Java, Perl, and Ruby, but not C++). It also had a number of nice features that made working with a complex hierarchy of objects easy. The problem, though, was that the GUIs were to be implemented in Java and they needed their own representation of the object model. In effect, the object model was being developed twice – once in Python and once in Java.

\(^1\) http://www.python.org
\(^2\) http://www.perl.com
Implementation and Results

This led to a synchronization nightmare, as features would be added to one side but not the other side. This was quickly found to be untenable so the entire implementation was ported to Java. Java had its own advantages. More people know Java and it has extensive networking libraries. Using Java allows the same object model to be used on both the server and the client application. The features that Java lacked (multiple inheritance and generic datatypes), which python had, were worked around. The tools for working with Java are much more sophisticated than those available for python. The IntelliJ IDEA\textsuperscript{4} IDE has proven to be a huge time saver with its support for real-time syntax checking, refactoring, and debugging.

4.1.2 Database
In this implementation, it was decided to use a database to store server information. A relational database was chosen since it was easy to organize the server data into the structure of a relational database. A Java database was desired since installation would be easy. A SQL database was necessary so that a database upgrade would be possible in the future and so the implementation would not be locked to any proprietary database languages. Upon investigating the available Java SQL databases in early 2003, it was decided that HSQL Database Engine\textsuperscript{5} was the most mature of the open source offerings available. It definitely has limitations in that it doesn’t support a number of more advanced SQL statements which would be useful if they were available. However, it was adequate for the current effort. An open source database was desired so that DOME could eventually be distributed freely.

\textsuperscript{3} http://www.ruby-lang.org/en
\textsuperscript{4} http://www.jetbrains.com
\textsuperscript{5} http://hsqldb.sourceforge.net
4.1.3 Network Protocol

The networking technology chosen was XML-RPC\(^6\). The Apache XML-RPC\(^7\) implementation was used. There were a number of contenders in this field. Ease of use and cross-language support were the most important attributes. This ruled out all-Java technologies like Java Remote Method Invocation (RMI). Even though XML-RPC is an old technology that has not been developed since 1999, its simplicity of use and wide availability in many computer languages make it ideal for simple networking projects. The implementations are old (since the specification has not changed for many years) and have not been updated to take advantage of new language features. There was some attempt to use the more modern SOAP\(^8\) implementations, but the tests led to the conclusion that the substantial overhead associated with SOAP implementations was not appropriate for a simulation environment. Whether this was because the technology was not mature yet or because of user error is not known since I was not involved in that testing. Meanwhile, XML-RPC has been very easy to adopt. As a result, new distributed services have been rapidly implemented on top of it.

4.1.4 Relation Implementation

Relations allow users to define how parameters are related to each other. Ideally, this would be expressed in a syntax which would be easy for non-programmers to learn. In the old implementation, relations were compiled in C++. There was a problem with implementing the object model in Java because Java does not support operator overloading. What that means was that we could not define how to add (or any other mathematical operation) two instances of a DOME datatype. Instead, we would need to

\(^6\) http://www.xmlrpc.com
\(^7\) http://ws.apache.org/xmlrpc
\(^8\)
use methods. For example, instead of being able to say $c = a + b$, we would need to do something like $c.set\text{Value}(a.add(b))$. The second representation requires a lot more programming knowledge and is more prone to user error. This was not thought about when the decision to use Java everywhere was made. Luckily, there is an implementation of python in Java called jython. It allows Java objects to be used in python within a Java program. Python supports operator overloading. Using jython, Java objects can overload operators in jython expressions. So, the relation expressions are written in python. The python syntax is easy to understand for simple expressions, so it is not hard for the non-programmer to learn what they need to know to write relations. Python is also an interpreted language so there isn’t a huge hit at the beginning waiting for the relations to compile. While an interpreted language is slower than compiled code, the difference won’t be noticeable for most applications.

4.2 Current Implementation

The current implementation of DOME includes a server and a single client application.

4.2.1 Client Application

The client application is a single application with 4 modes: build, deploy, run, and server. In build mode, users can build models, projects (aggregate models) and playspaces (collaborative spaces). Users can also checkout existing items on the server in order to be able to edit them locally. In deploy mode, users move the items they have built to the server. At this time, users can set privileges on the item. There are generally two categories of privileges to set: edit and access. A user with complete edit privileges on an item will be able to check it out, modify it, and redeploy the item again. A user with

---

8 http://www.w3.org/2000/xp/Group
complete use privileges will be able to use the item in many contexts. Partial sets of
privileges can also be granted. In run mode, there are two distinct user activities. First,
the browser is used to navigate the WWSW for models and playspaces of interest. Then,
the user may start a model or join a playspace. Once that GUI is displayed, the user will
be able to run models of interest. The client allows users to submit and view changes.
There is support for displaying status and advanced notification messages from the
server. Starting a model many times creates many model instances. Using a model in a
playspace allows multiple users to use the same model. That is how the distinction
between multiple users and multiple model instances is defined in this implementation.
The server mode includes a collection of server utilities. In server mode a server
administrator may define and edit users with passwords and groups of users. Users and
groups may be granted permission to deploy models and/or playspaces on the server. The
server administrator also has permission to remove any item on the server. A user may
use server mode to change his password and to manage his model or playspace filespace.

The user interface must be easy to implement, extend, and customize. It is desired
that the same user interface code be usable on multiple operating systems with very few,
if any, changes per platform. The natural choice was to use Java Swing which was the
most sophisticated of the freely available cross-platform GUI toolkits. In order to meet
our goal of having a GUI that would be easy to customize, the JavaBeans standard was
adopted. This is the standard that enables drag-and-drop GUI builders to be built and we
eventually would like to have one for DOME. Swing itself can be hard to program if
complex GUIs are desired. A number of convenience methods were implemented to help

\[9\text{ http://www.jython.org}\]
so that the user that could be assigned permission to use a model on the server must be
have an account on the system created by the system administrator. A distributed
authentication mechanism, which would permit assigning access privileges to users of
other servers, has not been implemented since there does not exist a standard solution to
the problem today. Finally, model solving is inherent in the implementation of the model
and distributed solving support is implemented in the project. The server has basic
support for each of the categories described in Chapter 2. With the current
implementation, it has been possible to start demonstrating the core WWSW idea to
interested parties again.
DOME GUI programmers maintain consistency across the application and to take away a lot of the grunt work in arranging Swing components. The structure supports new visualizations easily (Design Structure Matrix, graph, and XML visualizations are currently available for models). Because of the JavaBeans standard, it is also easy to add custom GUIs to model interfaces. This is also supported and was very simple to design. Finally, the GUI has largely been a success since it is, for the most part, easy to learn and consistent across platforms.

4.2.2 Server
The Server has many modules. The model server component was implemented by Keith Thoresz. The rest of the modules have been implemented by a number of students in the CADlab. The server is designed to support two virtual file spaces. They are "virtual" in the sense that the actual space structure is kept in the database instead of on the local operating system. One file space supports models while the other supports playspaces. The additional modules in the server include support for browsing the virtual files paces, deploying and redeploying to the virtual file spaces, user/group management, authentication, and permissions checking.

The server versions models and playspaces during the deployment process. It also stores archives of the old versions. As mentioned before, the server supports both multiple users of a model via playspaces and multiple running instances of a model. Multiple interfaces to models may be defined. Users always access a model via a model interface. Therefore, access control to the model may be defined on a per-interface basis. Model deployers may configure certain users access to certain interfaces and they may also grant guest access to interfaces. The user and group database is local to each server
5 Making the World-Wide Simulation Web a Reality

The implementation that exists is reasonably functional, though lacking in elegance underneath. It already provides a lot of functionality beyond what was available in the second-generation DOME implementation. Key limitations in the old version such as solving and multiple model instances support were easily accomplished in the new representation. However, there is still quite a bit of work to be done.

There were a number of features implemented which do not match current conceptual ideas. Some of the ideas were not understood by the implementers. Some of the ideas were only developed after seeing how the current implementation behaved. The application needs to be optimized since loading the application can take on the order of minutes on some machines. The lack of code documentation makes it hard for new programmers to get up to speed quickly.

5.1 Desired Features
In this sections key features of the specification which have not yet been implemented in the current architecture are discussed.

5.1.1 Pluggable Architecture
One of the original goals was to have an architecture in which it would be easy for users to add new components to the system. While a lot of this is in place, it is mostly undocumented so it is not yet easy for users to participate in the code development. One current failure is that the plugin API which was implemented to allow users to easily interface with external programs has been received negatively by all test users. This
aspect will need to be redone in the near future since plugins are a critical component of the system.

5.1.2 Constraints
The current implementation does not yet support constraints for parameters. This is necessary if the models are to be used for real in a distributed web environment. Most users who find a model on the web will not know what the valid values for the input will be. Constraints will allow a user to understand what the valid values are and to ensure that models are not run with invalid values which might result in an application crash.

5.1.3 Custom Networking Protocol
While XML-RPC is proving to be adequate for now, it is clearly outdated. Work needs to be done in defining a networking protocol which would be optimized for DOME. HTTP tunneling of the protocol can be used to address the issue of how to use DOME through firewalls once it has a protocol which is not naturally run over HTTP.

5.1.4 Distributed Network Security
While the current DOME implementation has notions of authentication and privileges, they are quite primitive. Real security with encrypted authentication needs to be incorporated into the system. A working distributed authentication and privilege system needs to be used. This will require new thinking since no such system currently exists as a freely available implementation that once can just drop into one's system.

5.2 Towards an Open Source Software Development Model
One final goal is to be able to turn the project into an open source project. To do this, a system that would support open source development should be implemented. In well-managed open source projects, many developers have access to the code and can submit
changes. However, changes are not merged into the main source tree until it has been
tested by a member of a small group of people who check that the code performs as
advertised and does not break existing functionality. Via this process, many people can
participate in code development and the quality of the code can be maintained.

The current DOME implementation has had a number of contributors of varying
abilities. The code now shows traces of the different personalities. Formatting and name
conventions are not consistent across files or even within a file. It will be necessary to
impose coding standards on the project. The code has also become quite hard to debug.
This is due to a number of new additions to the code which have been made all over the
place without fully understanding what the proper way to add a feature is. There has been
a bit of a “just get it done” mentality in implementing new features. The resources are not
there to support a review process. Many times changes to the code base break other
things, but no one notices until weeks later. Therefore, moving the current development
model to one that supports open source development would benefit the current
development team as well.

5.3 Conclusions
The implementation today accomplishes many of the goals outlined in Chapters 2. It
supports model and project building, deployment, and use. The project execution needs a
bit more work, but it is almost complete. While it is usable, there is a bit to be desired
about the implementation. Many of the key features desired from the old implementation
are complete. However, there is a lot more that needs to be done before the system is
ready to be used widely by many users and in the corporate environment and before the
system can be released as open source software.
References


