Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE TANGRAM MODELING ENVIRONMENT

Leana Golubchik
Gary D. Rozenblat
William C. Cheng
Richard R. Muntz

November 1990
CSD-900034

The Tangram Modeling Environment [1]

Leana Golubchik Gary D. Rozenblat William C. Cheng
Richard R. Muntz

UCLA Computer Science Department

June 28, 1990

[1] This work was partially supported by a MICRO grant from the University of California and the Hughes Aircraft Company, by NSF-CNPq Cooperative Research Grant INT 8902183 and by an equipment grant from DEC.

Abstract

This paper describes a prototype modeling environment, Tangram, which incorporates mathematical solution of queueing networks and Markov chain models as well as a comprehensive simulation capability. The system also supports user defined hierarchical modeling, user specified approximation methods, a graphical interface, and other state-of-the-art features. Rather than a completed, closed system, we view Tangram as an environment in which application specific modeling and performance evaluation sub-systems can be added with minimum effort. The ease with which such extensions are supported is heavily dependent on the object-oriented paradigm which is used throughout the design: the object-oriented paradigm is shown to effectively support code reusability across different applications via the mechanism of inheritance and encapsulation. The envisioned system can be termed a "modeling shell" capable of being easily tailored to specific applications. It supports extension by allowing new tools, solution techniques, and application areas to be added modularly and without modification to existing parts of the system. A detailed example is used throughout the paper to illustrate the ideas.

1 Introduction

Driven by the personal computer revolution, advances in graphics, and the development of the object-oriented programming paradigm, the opportunities for an advanced modeling environment have never been greater. This paper presents an architecture for such an advanced modeling environment. We concentrate on computer systems performance modeling, as that is where our expertise is, but the principles developed here apply to many other types of modeling.

A modeling system, to be successful, has to provide many different types of features. First, it has to be useful over a wide variety of application areas which we will call *domains*. Example domains are queueing network models and Markov chains. Other domains can be associated with particular applications (e.g., communications networks, database systems, or even tailored to a particular product). Some domains will be specializations of other domains, e.g. product form queueing networks models are a specialization of extended queueing network models. Models are constructed from queues or other objects that are customary to that domain. In addition to the objects, a domain encapsulates specialized knowledge relevant to the domain, e.g. choosing a solution approach.

Second, the modeling environment should support various types of users, from novices to experts. In our view, a specialized application oriented domain will often be created by an expert. There are two main reasons for tailoring the system to a particular application domain: (1) to create an application oriented interface for less expert users and, (2) to create a more convenient tool for the particular application by specializing notations, solution techniques, etc.

To create the application domain, the expert may utilize many different types of tools available in the system. When complete, an application domain will generally contain different types of information, including the types of objects from which models in this domain are constructed, consistency rules telling when a model is well formed, what solution approachs are to be used, and how the results are to be displayed.

Third, the modeling environment must be able to incorporate new solution methods.

determined by some combination of time, accuracy, etc.) is one that combines a number of complementary solvers, a protocol is required in which to express the rules by which a solution technique is chosen in any given situation.

In summary, a modeling system consists of a large amount of software including command interpreters, analysis packages, query and display facilities. This software represents a significant investment and one would like it to be applicable to a wide range of application areas, e.g., queueing network models, reliability models, etc. The objective is to develop an approach that allows a modeling environment to be tailored to different application domains while allowing most of the software to be re-usable across application domains.

In this paper, we will further explore the above mentioned issues and goals, discuss the most difficult obstacles associated with these goals, and present our approach to solving these problems, which we are implementing within the Tangram modeling environment. The organization of this paper is as follows. Section 2 will present background information and a set up of the example used throughout the paper to illustrate our ideas. Sections 3 and 4 will present our approach to implementing the goals stated in this section, and section 5 will present various solution techniques available in Tangram. Each of the former three sections has the following structure: general ideas on the topic, followed by an example illustrating these ideas, followed by implementation details. Finally, section 6 presents the concluding remarks.

2 Background

2.1 Related Work

Currently, most modeling packages are designed either around one application (e.g. communication networks) or around one solver (e.g. simulation). Tools designed for analyzing the performance and reliability of computer and communication networks are most similar to Tangram's. While many packages exist for modeling reliability (e.g. SAVE [GOYA86] from IBM, HARP [BAVU87] and SHARPE [SAHN87] from Duke University, ARIES [MAKA82] from UCLA and SURF [COST81] from CNRS), they all have the same problem. They

and to coordinate the operation of the data processors. The processors are connected by a communication network. The particular architecture depicted in Figure 1 is known as a *shared-nothing* architecture, i.e. in which disks and memory are only accessible (directly) by the local processor. This system is intended to run database software, where the workload (the database transactions) will be represented by task precedence graphs (e.g., see Figure 2). The transactions submitted to the database are parsed by the query processors and broken

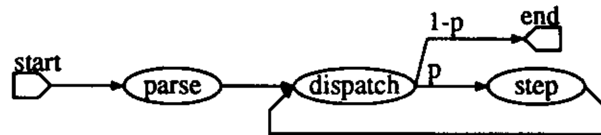

Figure 2: A Job Precedence Graph.

down into database steps; these steps are dispatched to the data processors for execution. Notice that this is a fairly crude specification of a typical database transaction. One of the missing details is the transaction *commit* processing, which we will include in a more detailed example in Section 4.2.

We will use this example to illustrate how Tangram may be tailored to a particular application, in this case a database machine. It is recognized that a certain amount of effort is required to do this customization. Our goal is to minimize this effort. Nevertheless, if one had only one instance of a model for a database machine to run then going directly to one of the existing facilities, e.g. queueueing networks, would be the preferred approach. Providing a customized interface will be justified for example when (1) the modeling effort is part of a system design and implementation effort and many versions and variations of the database machine will be studied or, (2) the system is to be used by people less expert in modeling (e.g. system designers, or later for system configuration or sizing) and the interface needs to be at a conceptual level more compatible with the application area.

2.3 Implementation Basics

While a number of modeling and simulation tools have been purely text based, we feel that a better solution lies in providing the modeler with an object-based graphical environment. The advantages of such systems are realized in a number of ways. Graphics-based systems

systems. A component encapsulates the appearance and semantics of such an element. Thus the user creates, destroys, and otherwise manipulates components that represent entities within the problem space.

- *Tools* support direct manipulation of components. Through the use of animation and other visual techniques, the user's perception of the effects of his actions are reinforced by immediate feedback. Examples of tools include tools for selecting, moving, and connecting components.

- *Commands* encapsulate operations on components or other objects. Commands are stateful and can be executed. They are also similar to messages in traditional object-oriented systems, in that they can be interpreted by components. Unlike tools, commands represent actions that do not require visual feedback or interaction with the user, and are also used to carry out the effects of a manipulation which is managed by a tool. Examples include commands for changing the state of a component, duplicating a component, and grouping several components into a composite component.

Additionally, a number of general abstractions to allow the composition and coordination of the above building blocks are supported. These include the *Editor* class whose instances associate tools, user-accessible commands (usually in the form of menus) with a viewing/drawing area (a *viewer*). One-of-a-kind object classes are provided to coordinate the interaction of simultaneously open editors and to manage persistent storage of components, tools, and commands. A *unidraw* object and a *catalog* object assume those roles, respectively.

As the authors of Unidraw point out, components are, arguably, the most important class of objects in the system. Components are the objects of interest in a particular application and closely "resemble" their real-world counterparts. Unidraw adopts a well-established user interface concept of the distinction between the state and operations that characterize an object and the way the object is presented in a particular context. This is accomplished by the separation of components into *subject* and *view* objects. While the subject encapsulates the context-independent state and operations of the component, the view supports a context-dependent presentation of the subject. Component subjects can maintain multiple graphical and non-graphical views (specified in parallel class hierarchies) and their definition includes how they respond to commands and tool manipulation. Component subjects and views can

- a library of solution methods,
- a (specialized) set of queries
- the knowledge of how to choose solvers to answer queries on the model belonging to the domain itself, and
- the knowledge of how to transform a model in the current domain to corresponding (sub)models in other domains, e.g., a high level system model into a queueing network model.

By encapsulation, we mean that the detailed implementation of these five attributes of a domain is hidden. There are several major advantages to encapsulation. It allows flexibility within a domain and at the same time provides a well defined rigid interface to the outside world. It facilitates the construction of new models, using previously defined components, and the addition of new solution techniques with minimal effort. Constraints applicable to the class of problems under investigation can be enforced automatically in addition to opportunistic selection of solution tools.

Using *object-oriented* programming concepts, domains can be organized in a hierarchy, as illustrated in Figure 3. More *generalized* domains are closer to the root of the hierarchy

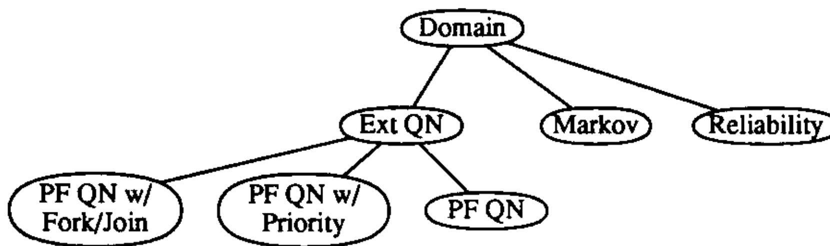

Figure 3: A Domain Hierarchy.

and more *specialized* domains are further away from the root. This hierarchical organization of domains provides two benefits: *inheritance* and *dynamic classification*. Inheritance is used in accordance with traditional object-oriented gains, where a request is passed up the hierarchy until an ancestor, willing and able to comply, is found. Dynamic classification is a "reverse" action. Instead of going up the hierarchy and searching for a generalized solution,

complete set of the domain attributes (mentioned at the beginning of this section). Hence, it is not necessary to wait for changes to occur in all the attributes before creating a new domain.

There are many other considerations that go into domain creation. These include the level of expertise and the modeling intent of the user. It is clear that the end user would not go to the trouble of creating a new domain, however what is not obvious is that it is not always good practice for an expert user to create a new domain whenever one of the attributes changes. The judgement call that needs to be made will depend on the intent of the modeler. If, for instance, the change is in the set of modeling components, but the new component is experimental and will only be used once, then it does not pay to go through the labor of creating a new domain. Rather, the new component should be created on the fly and used as a building-block, until there is more provocation for creating a whole new domain. This decision process is illustrated in the following section through the use of our database example.

3.2 Example

To illustrate the use and advantages of the main ideas discussed in the previous section, we will go through the exercise of creating a new domain, the Database Machine (DBM) domain. Figure 4 depicts the domain hierarchy that we will be using for the remainder of this section.

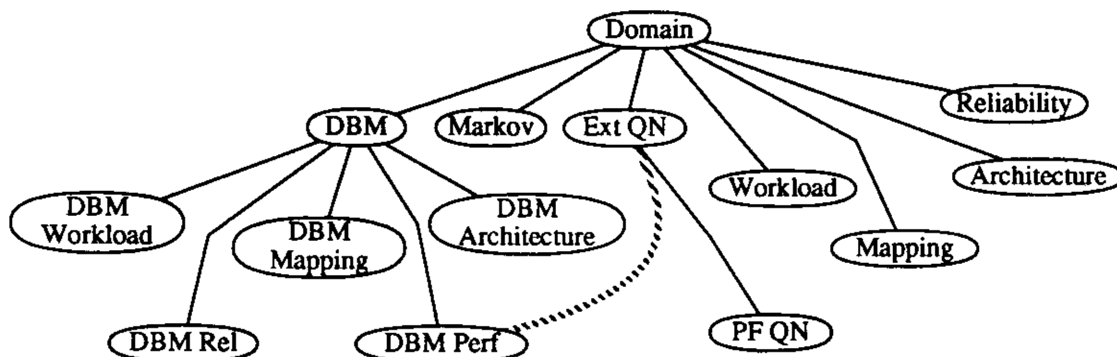

Figure 4: An Example Domain Hierarchy.

or solution domains is required.

The next step is to define a library of model components which will be available in the DBM domain for model construction. When dealing with database systems, most models can be constructed from a workload or an architecture specification, or a combination of both. Therefore, the primitive objects will either be in the DBM Workload domain or the DBM Architecture domain. The workload domain will have primitives corresponding to basic software modules in the system, e.g., parse, dispatch, access, schedule, etc. (Each of these can be primitives in the domain, or they can be instances of the same task primitive with different attributes.) The architecture domain will contain objects such as: query processors, data processors, disk units, communication networks, etc. The first set of primitives is useful in describing behavior of transactions that can be executed on a database machine; the second set of primitives is needed to describe the physical construction of a database machine. Further, we want to be able to mix and match various workloads and architectures, i.e., model several workloads running on the same architecture and vice versa (see section 4.2).

Next, we must consider the possible queries that will be asked of the system. This step involves examining the available set of queries in the parent domain and adding an appropriate set of keywords, necessary for query construction/expression in this new domain. For instance, one can assume that transaction throughput rate, utilization of data and/or query processors will be required. In order to formulate such queries, the user must use the proper keywords and specify, graphically or textually, to which modeling components these keywords apply. A complete set of keywords would depend on the type of solvers available, i.e., on the set of queries that the DBM domain (and possibly its children domains) is capable of recognizing.

As was pointed out earlier, the query should be expressed at a natural or most intuitive level of abstraction, i.e., in the domain of the problem/system. The solution, however, should be computed in the "best" way. This brings us to the next two steps in building a new domain: interfaces with solvers and transformation into other domains. The library of solution methods in DBM domain contains all those solvers that can deal directly with database objects, such as data processors and transactions. For instance, we could take the equations presented in [TAY85] and implement a solver that analyzes locking performance in

dynamicly, at run-time. The *Register* operation is used by a newly created domain instance to notify all other domain instances of its existence, so that its ancestor domains (in the inheritance hierarchy) can delegate, or bind to a solution method in its definition.

Other operations include a *Transform* method which transforms a model to a specified domain, as well as operations for accessing and iterating through the available solvers, described in more detail in Section 5.2.

4 Model Specification Techniques

What is a *system*? A system is a multi-faceted entity, and it means different things to different people. To a system architect, a system is a set of hardware components and their relationships. To a system analyst, a system is a set of tasks executing on the hardware platform. To a performance analyst, a system is a collection of resources shared by a collection of users. To a customer, a system may be summarized as a set of cost/performance trade-offs. In Tangram terminology, these different views of a system are called *facets*, and a *system* is a collection of related facets. Figure 5 depicts a *system* and some of its *facets*.

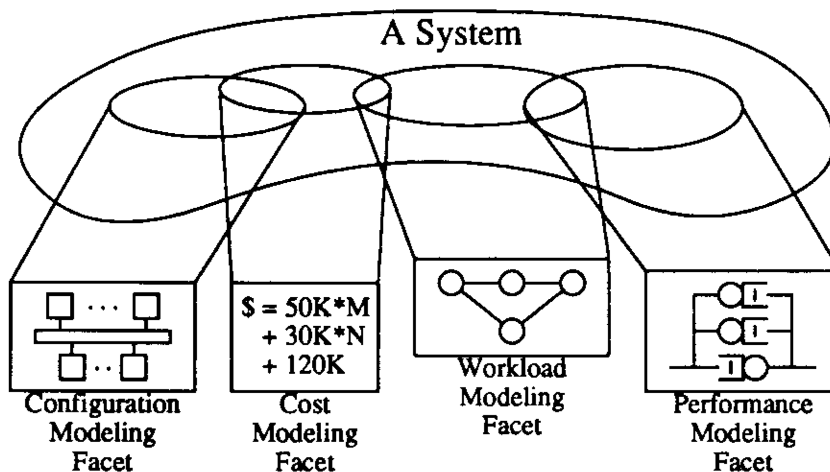

Figure 5: A System and Its Facets.

The interface to the black-box is restricted to a set of *ports* attached to the icon[2]. Thus the ports allow data to flow in and out of a modeling component facilitating data propagation within a model.

We adopt an entity/relationship/attribute view of describing a model [CHEN76], where a model is composed of entities, relationships among these entities, and attributes of both entities and relationships used for further specification. Visually, entities are represented by iconic facets of building-block systems, relationships are represented by lines connecting ports of icons, and attributes are represented by name/value pairs. Colors, line thicknesses, line styles, etc., may be used to graphically distinguish sub-categories of relationships. Different type of ports may also be distinguished graphically by different shapes and/or colors. The iconic/modeling facets combination is used for both top-down and bottom-up hierarchical construction of models. Every entity that is part of a modeling facet make up in some domain is an iconic facet of a more primitive system in that same domain. When a user wishes to view a more detailed description of a particular model entity, he expands its iconic facet (representation) which reveals a corresponding modeling facet in the proper domain. When a user wishes to abstract the details of a portion of the model, he can iconify that building-block which replaces the structural representation with a graphical one.

As was mentioned in section 3 and is described in detail in section 5, problem transformation is a very useful solution technique. In order to be helpful in this task, a modeling environment should provide support for construction of new modeling facets by combining one or more already existing facets via a transformation. For example, a Configuration Modeling Facet may be transformable to a Reliability Modeling Facet. In more complex cases, it is necessary to construct a new domain, a Mapping Domain, whose modeling components are various types of mappings from one domain to another. This will allow a user to combine several modeling facets, from different domains, into a single one, in the Mapping domain, and create ties between various types of modeling objects using mapping components. The use of this concept is illustrated in the following example section.

Another kind of facet is the *solution* facet, and it is used to describe model solution procedures. Solution facets are described in section 5.

[2]One port is shown in each facet in Figure 6; some ports are not shown on the disks and the controller.

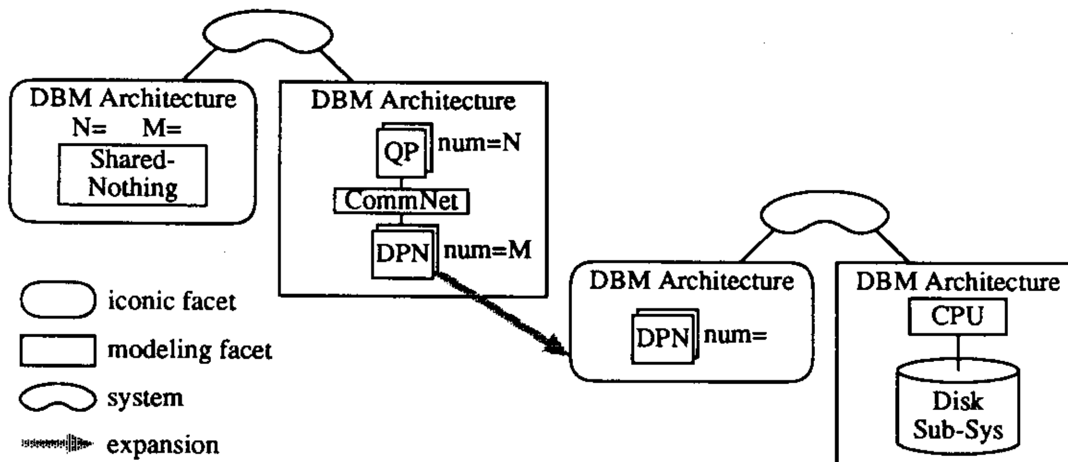

Figure 8: Shared-Nothing DBM Architecture.

For instance, expanding a data processor node reveals a composition of a cpu and a disk sub-system.

The DBM Architecture facet specifies the resources available in our distributed database machine. Another aspect of interest, when describing distributed databases, is the workload imposed on it, independent of its architecture. Therefore, the next facet of interest is the DBM Workload facet. As is shown in Figure 7, the workload on this system is composed of two types of transactions: TPx and TPy. A closer look at one of them[3], for example, TPy, reveals a task precedence graph, indicating the order and types of tasks to be performed to complete a transaction of this type (refer to Figure 9). Much like the architecture of the physical machine, the workload can be decomposed hierarchically. Each of the tasks can be a primitive or a building-block. For instance, it is shown that the *step* task is composed of two primitive tasks – *schedule* and *access*.

The architecture of the database machine and the workload to be executed on it, each presents a description of the database system that is complete for some types of analysis; for example, the architectural representation alone is sufficient for performing reliability analysis. However, many forms of analysis require some kind of mapping of the workload onto the architecture. For example, one way to compute a performance measure, such as a transaction response time, is to build a queueing network model, representing the database

[3]For simplification purposes the remainder of this paper will only consider a single transaction, TPy.

architecture facet.

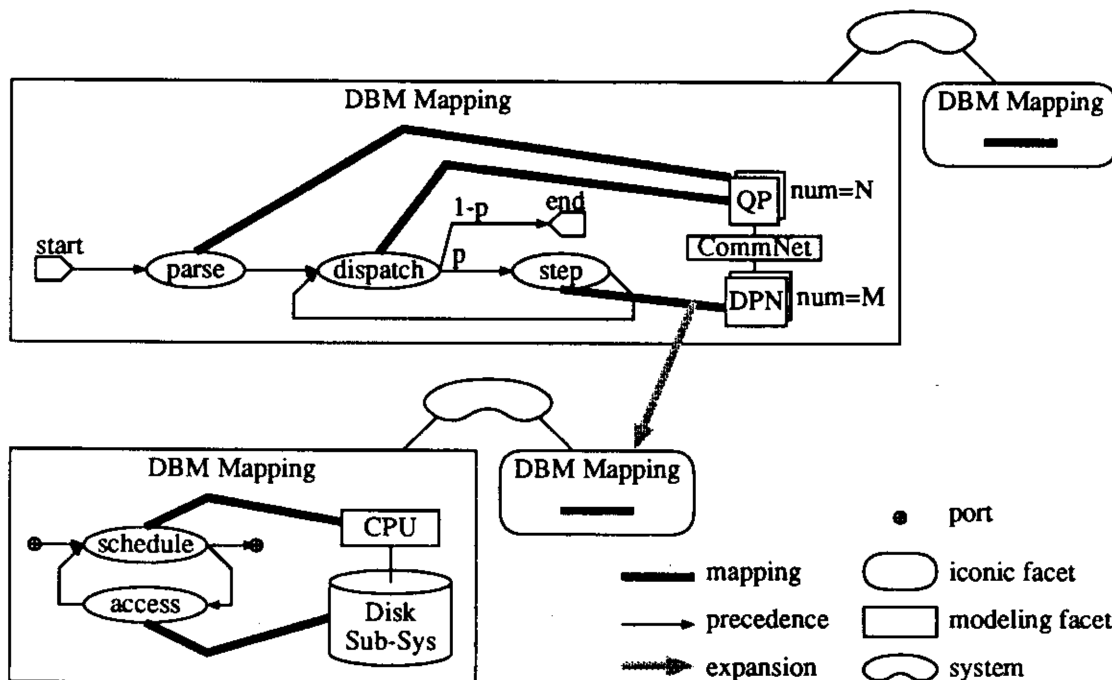

Figure 10: Example DBM Mapping.

Just as hierarchical model construction was useful in building other modeling facets, so it is desirable in building modeling-mapping facets. The building-blocks of the DBM Mapping domain include predefined mappings between tasks and architectural units. A simple example of this idea is illustrated again in Figure 10. Here, at the top level, the *step* task is mapped onto a *data processing node*. A closer look at this mapping, i.e. going down the mapping hierarchy, reveals more detailed and fairly generic mapping that can be used in many systems, namely that the schedule task is mapped onto the CPU of the data processor, and the access task is appropriately mapped onto the disk sub-system. Note, that there needs not exist a one-to-one correlation between the number of abstraction levels or objects in the workload and architecture specifications.

At this point, it would be instructive to work through a detailed construction of a mapping between the two-phase commit task, which consists of the *prepare* and *commit* phases[4], and

[4]Both prepare and commit have the *fork-action-join* construct, where the *fork* represents a query processor

query processors (not shown). The details of the *commit* phase can be defined in exactly the same manner, with the prepare task replaced by the *commit* task, still using exactly the same set of DP id's, i.e., using the same DP id object.

Suppose that our eventual goal is to compute responses to several performance queries, which normally involves creation of queueing network models. We now have enough information to (automatically) construct a queueing network model of the database system represented by the mapping facet. The query and data processors and the disk sub-systems correspond to the resources of the network; the workload (mapped onto the resources) defines the set of classes/chains (jobs) in the network, and their routing information. The model parameters, such as the service time distributions and means, number of customers in a chain, etc., are defined by the attributes of both the architecture and the workload facets. For example, the query processor speed, defined in the architecture facet, together with the number of instructions required by the dispatch task, defined in the workload facet, determine the service time of a class of customers at a center.

The modeling-queueing facet is constructed automatically by the DBM Perf domain. The query and data processors and the disk units, primitives in the DBM domain, are transformed into processor sharing queues, primitives in the Extented Queueing Network domain. The workload is mapped onto these resources as follows. Transactions of type TPy enter the network as class 1 customers. After completion of the *parse* task a customer changes to class 2 and goes into the *dispatch-step* cycle. Once the cycle is complete, with probability p, a customer changes to class 3 and enters the *prepare* stage of the two-phase commit process. With probability q one or more of the data processors decide not to commit, and the transaction is aborted, i.e., leaves the network at this point. With probability $1 - q$ a customer changes to class 5 and continues to the *commit* stage of the two-phase commit process. Once the commit is completed the customer leaves the network, i.e., this represents a successful completion of a transaction. See section 5.1 for an example of a detailed solution of this queueing network model.

In order to manage a collection of facets that describe a single system, an instance of class *System* is used. Similar, to a domain object, an instance of the system class, is never accessed or viewed directly, so it is implemented as a base class. Operations for iterating through the facets, as well as selecting one of a given class and parent domain are defined.

This implementation scheme allows for a consistent and efficient handling of most situations, i.e., most of the functionality embedded in the Unidraw classes is utilized, and only exceptions must be re-implemented.

5 Model Solution Techniques

As mentioned in section 3, domains provide model solution techniques such as *inheritance* and *dynamic classification*. The more traditional of the two, inheritance, allows for a general, default behavior to be defined in the higher levels of the hierarchy, while optimizations and specialized knowledge can be concisely developed in the lower levels. For example, the Product Form Queueing Networks (PF QN) domain, shown in Figure 3, can inherit (instead of redefining) all the queueing components defined by its parent class, Extended Queueing Networks (Ext QN) domain, while defining its own, with improved efficiency in comparison to Ext QN, solution techniques, such as MVA [REIS80]. Dynamic classification provides a complementary effect. Instead of delegating authority up the hierarchy and allowing a parent to perform an action requested of its child, it allows a parent to delegate the responsibility of performing an action by re-classifying the request to be more specialized, i.e. belonging to one of its children. Since the child is more specialized than the parent, the re-classification allows us to put more constraints on the problem and narrows down the solution space and hence gain in efficiency. For instance (going back to the queueing network example), given a model in the Ext QN domain and a query regarding the average queue length at one of the queueing centers, a general approach would be to simulate the network and measure the queue length. On the other hand, if the network happened to be product form, we could re-classify the model as PF QN and use MVA to compute the queue length, in a much more efficient manner.

There is at least one other technique that can be employed to answer requests posted

now time to pose a query on our database system and use all the knowledge accumulated in domains and all the facets built so far to compute its solution.

We will begin by constructing the query. Suppose we were to enquire about the response time of successful transactions of type TPy. An appropriate place to pose this query would be the mapping facet (see section 4.2). A proper way to ask the query is to: a) attach one port of a dual-ported gauge object, which is a graphical query facility discussed in Section 4.3, to the input port of the *parse* task and attach the other port of the gauge to the output port of the commit task, and b) attach an attribute to the gauge object representing the actual query, namely resp_time=. Once the query construction is completed, the user can choose the *Solve* command and await the end of the computation process.

Execution of the *Solve* command involves three actions: understanding/parsing the query, searching for a solution technique, and invoking an appropriate solver to compute the answer to the query. Let's now follow the solution searching and computation process for this particular query. Since the mapping facet is defined in the DBM Mapping domain, it (the domain) has the first try at answering the query. Unfortunately, there is nothing in this domain that can answer the query. At this point our best strategy is to take advantage of the domain hierarchy and continue up the domain tree until we find an ancestor that (better) understands the question asked by the user. This search will bring us to the DBM domain which does have the keyword resp_time in its lexicon. However, another misfortune befalls us here; even though DBM understands the query, it has nothing in its solution library capable of computing an answer to it. Fortunately, DBM can classify this query as a performance issue and is therefore capable of dynamically re-classifying the facet as a DBM Performance model. The DBM Performance domain can now invoke the transformation routines, defined earlier, to construct a queueing network out of the mapping facet[5] and then delegate the computation to the Extended Queueing Networks domain. This concludes our search for a solution technique.

The next step is to find an appropriate solver and compute the answer, i.e., construct a solution facet in the Extended Queueing Networks domain. There are two basic options available to us at this point. Since there are no known analytic solution methods available

[5]This construction was illustrated at the end of section 4.2

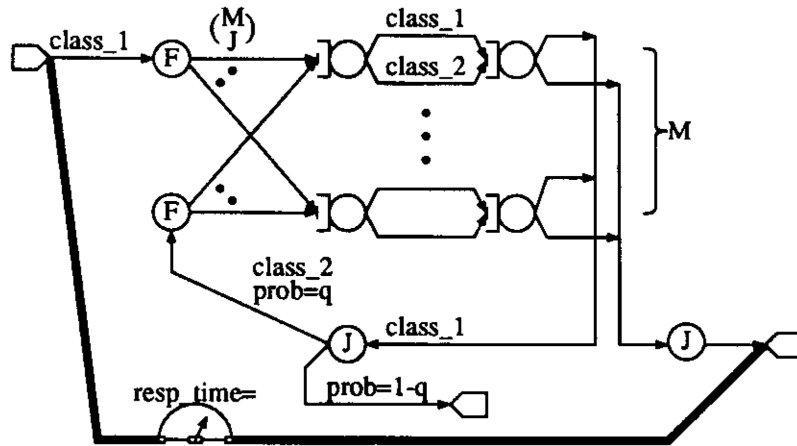

Figure 13: Fork/Join Queueing Model SM2.

- Solve SM2, in the Ext QN domain, using simulation to compute the total response time of *class_2* customers only (i.e., successful transactions).
- Compute the answer to the original query by summing the two values computed for SM1 and SM2.

The remainder of this process is simple. After obtaining the final result, control can go back to the DBM Performance domain. It can map the sum of the total response times of the queueing network submodels into the successful transactions response time, in the database system. This answer is returned to the DBM domain, which in turn returns it to the DBM Mapping domain, which can finally display the response as the solution to the original query.

5.2 Implementation

As previously mentioned (Section 3) domain objects are responsible for selecting the most appropriate solution technique and invoking the appropriate solver(s). In order to achieve some of the goals we have discussed earlier, our implementation should provide a general and consistent protocol for managing a heterogeneous collection of solution tools, which may include parameterized modules, other programs with complex invocation procedures, simulation packages, etc. In addition, the computational complexity of a particular solution

particular simulation facility. A wide variety of simulation packages are currently available, ranging from very general to very specialized. Our support for a simulation capability should be independent of a particular language/system used. Therefore, a particular simulation package is treated as an ordinary solver. A general *Simulation* domain class is part of the Tangram framework. This domain class is further specialized for whatever simulation package is to be used.

The actual simulation systems are abstracted by instances of the *Simulation Solver* subclasses, which generate the appropriate simulation code, perform the necessary compilation, instrumentation, and initiate the simulation. Application domains are then specialized to define application-specific simulation domains (e.g. the DBM simulation domain derived from the DBM domain). This allows the more advanced user to define and use model components and graphical query facilities that animate themselves as the simulation progresses. The framework also includes classes of graphical data display objects such as graphs and pie charts.

6 Conclusion

We started with the goal of creating a modeling environment which could accommodate a variety of analytic and simulation modeling techniques, be easily extensible with respect to both integrating new solution techniques and tailoring the system to specialized applications. In support of this goal we have made extensive use of the object oriented programming paradigm at all levels of the design. A major design decision was to introduce the notion of domains which are complex objects that encapsulate knowledge about a particular class of analytic models or applications. The idea is that a model and a query (of the proper type) can be sent to a domain as arguments and the domain object is responsible for finding the solution. To the extent that general classes of analytic models or applications can be defined and encapsulated in this manner two benefits accrue. One is that existing domains are available as resources upon which to extend the system without having to be concerned with the details of their implementation. Second, the encapsulation supports the extension of existing domains without modifying any clients that already use the domain. In addition, we have shown how domains can be *specialized* and how the inheritance hierarchy and dynamic

[GOYA86] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi, "The System Availability Estimator," *Proceedings of FTCS-16*, July 1986, 84-89.

[MAKA82] S. V. Makam and A. Avizienis "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault Tolerant Systems," *Proceedings of FTCS-12*, June 1982, 276-274.

[MARS84] A. M. Marsan, G. Conte, and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessors Systems," *ACM Transactions of Computer Systems*, May 1984, 93-122.

[MELA85] B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation," *IEEE Computer*, August 1985, 87-94.

[MOLL82] M. K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers*, Vol. C-31, No. 9, September 1982, 913-917.

[RAMA82] K. G. Ramakrishnan and D. Mitra, "An Overview of PANACEA, a Software Package for Analyzing Queueing Networks," *Bell System Technical Journal* Vol. 10, No. 10, 2849-2872, December 1982.

[REIS80] M. Reiser and S.S. Lavenberg, "Mean value Analysis of Closed Multichain Queueing Networks," *JACM* Vol. 27, 313-322, 1980.

[SAHN87] R. A. Sahner and K. S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Transactions on Reliability*, Vol. R-36, No. 2, June 1987, 186-193.

[SAUE81] C. H. Sauer, E. A. MacNair, and J. F. Kurose, "Computer Communication System Modeling with the Research Queueing Package Version 2," *IBM Technical Report RA-128*, November 1981.

[SAUE84] C. H. Sauer, E. A. MacNair, and J. F. Kurose, "Queueing Network Simulations of Computer Communication," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-2, No. 1, January 1984, 203-220.

[TAY85] Y.C. Tay, R. Suri and N. Goodman, "A Mean Value Performance Model for Locking in Databases," *JACM*, 1984, 618-651.

[WHIT83] W. Whitt, "The Queueing Network Analyzer," *Bell System Technical Journal*, Vol. 62, No. 9, November 1983, 2779-2815.

[VLIS89] J.M. Vlissides and M.A. Linton, "Unidraw: A Framework for Building Domain-specific Graphical Editors," *Proc. of the ACM SIGGRAPH/SIGCHI User Interface Software Technologies '89 Conf.*, November 1989, 157-167.