An Object-Oriented Modeling Environment*

Thomas W. Page, Jr., Steven E. Berson, William C. Cheng, Richard R. Muntz

Department of Computer Science
University of California Los Angeles

Abstract

Many tools and techniques exist for the modeling and analysis of computer and communication systems. These tools are often complex and tailored to a narrow range of problems. The system analysis task often requires coordinated use of multiple tools and techniques which is not supported by currently available systems. The Tangram project goal is to develop an environment which makes a large set of tools and techniques readily accessible and is easily tailored to specialized applications.

This system has been prototyped in an object-oriented extension to Prolog. The impact that these two paradigms, logic and objects, have had on the design is discussed. Several example applications are presented to illustrate the extensibility of the system.

1 Introduction

Will an increase in the discount rate drive this country into a recession? Is the weather going to be good next year for growing avocados? Is the satellite likely to remain operational for the entire mission time? Getting answers to this type of question can be of critical importance. Unfortunately, it is generally infeasible to conduct an experiment to answer these questions without committing to the very course of action whose outcome is in doubt. Instead, we try to capture the behavior of these types of systems in a *model*. Querying this model gives predictions of the future. These predictions can be anywhere from exact to wildly inaccurate depending on how well the model represents the essential behavior of the real world system. Experience with the model as to how well it explains previous behavior can give confidence in its predictions of the future.

While there have been tremendous advances in the mathematical techniques, the practice of modeling has advanced more slowly. Modeling typically requires specialized skills; knowledge of both the problem domain and a solution package. It is very labor intensive. The process involves an expert abstracting the application domain, selecting a solution method or solver, translating the abstraction into the input format of the solver, and then interpreting the numerical results of the solution. The solution tools typically have a very primitive user interface requiring input in a rigid format. An expert is often only highly proficient in one tool; "if you only have a hammer, everything looks like a nail." In addition, there is a shortage of experts and using the ones there are is costly.

The answer to these problems is a whole new generation of modeling tools. Driven by the personal computer revolution and by wider acceptance of primitive modeling tools such as spreadsheets and databases, the opportunity and the need for an advanced environment which can harness powerful modeling tools for non experts has never been greater. This paper proposes an architecture for such an advanced modeling environment. We are building Tangram, a prototype of such a system, using logic programming extended to support object-oriented programming [PAGE89,MUNT88].

1.1 Mathematical Modeling

Mathematical modeling spans a large variety of techniques; queueing theory, mathematical programming,

*This work was partially supported by a MICRO grant from the University of California and the Hughes Aircraft Company.

Markov chains, semi-Markov models, etc. In the same sense that this body of knowledge is organized into areas and subareas, one of our goals in Tangram is to organize modeling knowledge into modules that we call *domains*. A domain encompasses a class of models, a set of solution methods (solvers) and an interpreter for queries. The idea is that models are created "in a domain" and any such model must be a member of the class of model associated with that domain. The choice of what class of models a domain encompasses is a design decision. Basically, one would like to (a) incorporate a useful, general class of models and (b) subsume within the abstraction provided by the domain a significant amount of detail. As a simple example, suppose we construct a domain that deals with finite Markov chains. This domain may have a number of different solution techniques available; e.g. some may be appropriate for nearly decomposable chains, others for sparse chains, etc. The idea of the domain abstraction is that a client should not have to be concerned about details such as how to choose the best solution methods, but rather just ask for a solution and have the system choose the appropriate solver.

When the domain supports what is a classical mathematical abstraction (e.g. Markov chain, linear programming, queueing network, etc.) then it is perhaps more appropriate to call this a *modeling domain*. A specialization-generalization hierarchy of these domains occurs quite naturally. Further, as we shall demonstrate later in the paper, it is also convenient to form domains for particular applications, which we call *application domains*.

In addition to the central concept of domains, the following constitute the goals of the Tangram system which we believe are essential.

Extensibility: The system must be able to cope with a wide variety of application areas. In our view, an application domain will be customized by an expert for a non-expert to use. New applications may be created by specializing existing ones. New solution techniques must be incorporated in the system without modifying existing models or knowledge bases and be employed when appropriate.

User-friendliness: The user interface should make extensive use of graphics for defining models, queries, and expressing results. The form of the graphical communications should be customizable for each application. Applications should provide their own palette of objects customary to their domain.

Flexibility: It must handle both top-down and bottom-up modeling. In the top-down view, models are successively refined into more detailed sub-models. The bottom-up approach abstracts detailed low-level models into simpler representations.

Meta-modeling: The environment must function as a model-base management system for creating, storing, retrieving, updating, sharing, and querying models.

The following simple example is used to illustrate some of these features as they currently exist in Tangram.

1.2 Example Modeling Application

The Tangram system contains several base modeling domains (e.g. queueing networks, Markov chains) from which specialized problem-oriented environments may be created. We consider a trivial example of an environment for creating models of car-washes, similar to those in [GOLD83] and [MISR86], chosen to be easily understandable given space constraints. The car-wash domain is constructed (by an expert) as a specialization of the system's basic queueing network domain. It provides a set of objects from which to build car-wash models, consistency constraints, a query language, and rules for translating a car-wash model into one the system can solve.

The user first selects the application domain (from a pull-down menu). Graphical objects are then instantiated by the user from a palette of icons representing object classes provided by the creator of the domain. Figure 1 shows a simple model constructed in the car-wash domain using Tangram's graphical interface. Model objects include the attendants, car washing facilities (two shown), and a hand waxing station. In this particular example the user specifies that 10 percent of customers want their cars waxed (modeled by the branching probabilities at the exits of the car washing facilities), and that customers arrive at the attendant at a rate of 20 per hour. In this case, the very primitive query language consists of a generic query object (icon at upper left corner) with the attached key word avg_wait= requesting the average customer waiting time.

When the user selects solve from the pull-down menu (not shown), the model is sent the solve message. The rules provided by the domain expert can be viewed as an expert system which examines the model and the query and generates appropriate sub-models.

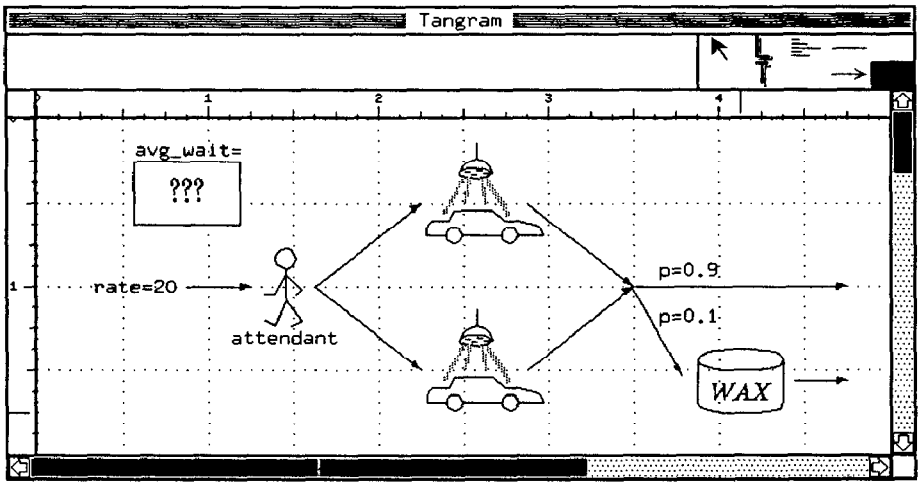

Figure 1: A Model in the Car-Wash Domain

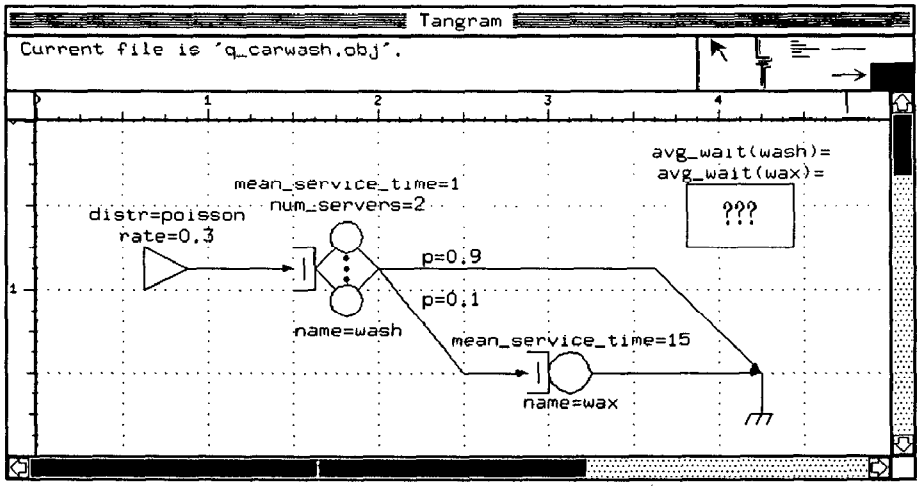

Figure 2: Queueing Model of the Car-Wash

In this case, the system simply derives an equivalent queueing network model as shown in Figure 2. [1]

In this queueing model, the customer arrival is modeled as a "source" with Poisson arrival with a rate of 0.3 per minute (20 per hour). The attendant and car washing facilities are aggregated and modeled as a multi-server (whose name is wash) with mean service time of 1 minute. The hand waxing facility is modeled as a first-in-first-out server with mean service time 15. The query to the queueing model becomes a function of the average waiting time at the 'wash' queue and the 'wax' queue.

The basic queueing network domain is able to solve the derived queueing model. The queueing theory expert knowledge base embedded in the queueing domain is exploited in the car-wash domain by transforming the original problem into a queueing problem. The answer to the original query is computed from the answers to the sub-model queries; in this case:

$$avg\_wait = avg\_wait(wash) + .1(avg\_wait(wax)).$$

To illustrate the flexibility required of the modeling system, suppose we are now interested in the availability of the car-wash; car washing machinery may break down and the car waxer may call in sick. In this domain's simple query language, the user attaches availability= to the query object and re-solves the model. The domain expert system reinterprets the car-wash model in the reliability domain generating an instance of a reliability model (graphical representation omitted due to space constraints). The reliability domain, already provided by the base system,

[1] Tangram does not currently generate the graphical representation of the derived queueing model; it exists only internally. Such a representation could be generated and would be useful for model debugging and explanation.

solves the model with Markov solution techniques.

Further, we can imagine extending the car-wash domain to handle profit and loss queries. We could model the effect of throughput on reliability (the waxing person becomes sick easier when he has to work above some threshold) or add a backup waxer who works slower, but is healthier. Solving complex models generally implies approximations and coordinated use of mathematical tools in problem dependent ways. An advanced modeling environment encourages experimentation with approximate analytic techniques such as decomposing the model and iteratively solving the sub-models (relaxation). In general, these sub-models are solved in different domains utilizing their own solution techniques, possibly using further decomposition. When the analysis procedure is validated through comparison with measurement data or simulation, it can be incorporated in the domain knowledge base.

1.3 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), all have the same problem. They provide a convenient interface for models that fit into the anticipated mode but no others. Tools for modeling queueing networks have similar problems (e.g. PAW [MELA85], QNA [WHIT83] and PANACEA [RAMA82] from Bell Labs, RESQ [SAUE81,SAUE84] from IBM and PAWS [BERR82] from University of Texas). RESQ offers both exact solution for a certain class of models and simulation for others, but the structure of the models is different depending on the analysis method. PANACEA and PAW go a step further and use the same language for several different analysis methods, but have no facilities for easily adding new modeling primitives or new analysis methods. At another extreme are tools such as petri nets [MOLL82,MARS84] that provide generality by using only the most primitive constructs. This places too much burden on the modeler. More recent efforts are starting to provide better tools. Structured Modeling [GEOF87] is attempting to provide a general high level interface for linear programming and other operations research tools. ANALYTICOL [ANAL85] attempts to provide a high level interface for statistical analysis. Finally, [BERS87] describes an environment for generating and analyzing Markov chains using an object-oriented approach.

A discussion of related work on object-oriented Prolog is beyond the scope of this paper. The interested reader is referred to [KAHN86,FUKU86, GULL85,MCCA87].

1.4 Organization of the Paper

This introduction has motivated the need for an advanced environment for modeling and has given an example using the prototype Tangram system. The following section presents our approach of encapsulating models and solvers in domains using object-oriented logic programming. Section three details our object-oriented Prolog language in which the Tangram system is built and discusses the relevance of this hybrid paradigm language for modeling. Section four describes the architecture and current state of our prototype and conclusions follow in section five.

2 Smart Models

The key to realizing the myriad design goals for a multi-domain modeling environment is the concept of *smart models*. A smart model must be able to respond to messages by performing high-level operations on itself: solve yourself, display yourself, suggest a solution technique, etc. We call the models "smart" because much of the burden of solving models is shifted off of the user onto the system. In order to create a model in an existing application domain, a user need only know what base objects the domain provides, how to connect them, and the query language for this type of model. The model has the intelligence (inherited from its domain) to select an appropriate solver, translate itself into the solver's input format, interpret the output of the solver, and answer the query.

The concept of smart models is realized in Tangram via the novel combination of several powerful ideas. The use of object-oriented modeling organized in domains, declarative programming, non-determinism, and a new variation on multiple inheritance called *semantic binding* combine to make possible a very flexible modeling environment.

2.1 Object-Oriented Modeling

Foremost for the success of the system is the object-oriented structure of the modeling environment. It is natural to represent entities in an application domain as objects which respond to a well defined set of

messages. For example, in a flexible manufacturing system model, domain objects might be tools, parts, and bins. New types of objects may be created by specializing existing ones. Complex subsystems can be modeled with composite objects (also called sub-models) and can be used in other models. A model as a whole is itself a composite object which responds to a set of messages.

Solvers in Tangram are also represented as objects. A solver must know what types of models it is capable of solving and be able to estimate its complexity and accuracy. The object-oriented approach to both models and solvers immediately obviates the monolithic nature of most modeling tools, naturally distributing model specifications and integrating multiple solvers.

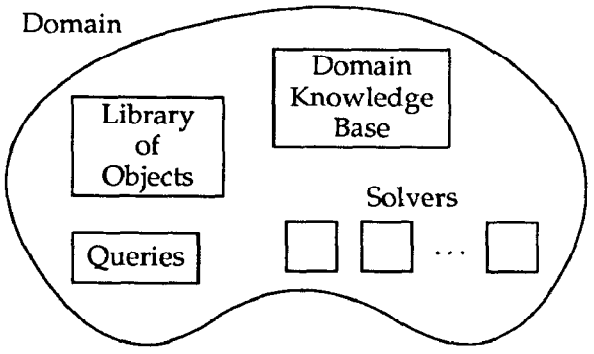

Figure 3: Conceptual Diagram of a Domain

Models are declared within *domains*. A domain is an environment which encapsulates the object definitions, query language, solvers, and heuristic knowledge base which combine to make up a customized package for models in a particular application. The knowledge base contains rules for checking the consistency (well-formedness) of a model, selecting a solver for a given model and query, and selecting an appropriate representation for results. Domains may be customized by expert users to create problem-specific domains for use by novices. For example, an environment for modeling the reliability characteristics of computer systems with repairable components has been created in Tangram as a specialization of a general Markov chain domain. Specializing a domain involves declaring rules for transforming a model and a query in the application domain into a sub-model and query in a more general domain.

Domain knowledge bases in Tangram are not implemented as separate software components. Rather, the knowledge is distributed among the objects in each domain. For example, solvers must "know" their own complexity; model objects must "know" how to display themselves. These objects must provide methods which query their portion of the domain knowledge. Details of how this is implemented are discussed in section four.

2.2 Semantic Binding

When an object receives a message, it is requested to perform a function, identified by the message's name, on itself. For example, a matrix may be sent the message "invert". However, there may be more than one implementation of the same logical function, e.g. one invert procedure for small matrices and another for large, sparse matrices. In our environment, we wish to be able to maintain an extensible tool kit of solvers; more than one solver may be applicable. In addition, the most appropriate solver is generally a function of the specific model structure, parameters, values to solve for, etc. Given an object and the name of a function, the system must dynamically bind to an implementation of the function.

In traditional object-oriented programming, given an object's class, a function name, and an inheritance lattice, the system performs the binding to an implementation of a method. To accommodate more than one implementation, either the class must be further specialized or the logic to select which version to use must be built into the method. The former is insufficient because it may not be known when the object is created which version is appropriate. If more than one is applicable, the best choice is often a non-trivial function of the object's state, not simply its class. The latter sacrifices the flexibility of adding new implementations of the function without changing existing methods.

In general, a program must run to choose the correct binding of a function name to an implementation. We term this process *semantic binding*. In Tangram, connecting a solver to a model for a given query involves running a domain specific expert system which queries the model and solvers and chooses the most appropriate binding. We call this binding program an expert system firstly because the choice of a solution technique for a particular model is an expert decision; a naive user, for example, need not be concerned with whether or not his queueing network is product-form. Secondly, the expert knowledge used to select a solver is most often best expressed in the form of rules. Such declarative knowledge representation is highly extensible; adding a new solver involves adding rules for that solver, not modifying rules. This is the programming style of an expert system.

Section four describes how semantic binding is implemented in Tangram using conventional object-oriented binding. For another view on varying object-

oriented name binding rules see [MINS87]. The next section presents the object-oriented extensions of Prolog upon which the Tangram environment is constructed.

3 Object-Oriented Prolog

It is crucial to the success of the modeling environment that we harness the advantages of both the logic and object-oriented paradigms. Declarative programming is essential to rapidly prototyping the behavior of model objects. Such rapid prototyping is necessary as one of the primary characteristics of a modeling environment is extreme flexibility. Further, the domain expertise is generally most conveniently and flexibly expressed as rules. To this end, we have specified and implemented an extended logic programming language, Object-Oriented Prolog (O-OP) and used this language to implement the Tangram modeling environment.[2] The remainder of this section presents details of this hybrid paradigm language.

3.1 Combining Prolog and Objects

There are two fundamental ways to modularize programs. Programs may be divided into packages of related functions usually called libraries (e.g. a math library, a statistics library, a strings library). Alternatively, we may group functions according to the set of objects they may be applied to. The same function name may bind to different code when applied to different objects. This is the object-oriented approach.

Central to object-oriented programming is the equation *Module* ≡ *Class* [MEYE86]. Object-oriented programming in Prolog amounts to using modules to encapsulate objects, plus a name binding facility for inheritance. Sending a message is interpreted as proving a goal in a module. While the two paradigms may be freely mixed in O-OP, it is natural to organize high levels of a program in an object-oriented (procedural) framework, while employing a more declarative style for the small, tightly circumscribed methods which implement object behavior. In this hybrid paradigm each style may be used to its best advantage: Prolog for programming in-the-small, and objects for programming in-the-large.

[2] While our implementation is based on our own addition of a modules facility to the Warren Abstract Machine[WARR83], the O-OP interpreter may be quickly adapted for any Prolog with a basic modules facility.

3.2 The Object-Oriented Prolog Language

Object-oriented Prolog is a super-set of standard Prolog. The infix predicate send/2 may be freely embedded in a program. Informally, the semantics of a goal send(Object, Message(Args)), are "prove the goal Message with arguments Args in the context associated with Object." For each object instance, there is a module of Prolog code which contains the instance variables of that object, including the object identifier of its class (the "isa" pointer). For each class, there is also a module containing both instance variables (in particular a pointer to its super class) and method code.

When Prolog tries to prove the goal send(Object, Message(Args)), it first locates the module associated with Object and uses its "isa" pointer to locate the module associated with that object's class. If an implementation for the Message(Args) occurs in that class, it is used to attempt to prove the goal and may succeed (and thus bind some of the arguments) or fail causing backtracking. If none is found, the interpreter looks for the method in the super class. The search continues up the hierarchy until an implementation is found or the root object is reached, causing the send goal to fail, prompting backtracking.

Multiple inheritance is supported via backtracking. There may be more than one "isa" or "super" instance variable in an object. The interpreter backtracks through the multiple inheritance paths if more than one exists, the ordering of clauses determining the order in which results are found.

New objects are created via a message to a class. While each class may specialize the new_object method, in its general form it requires in its arguments the object identifier of the new object and a list of its initial instance variables. If the object identifier is uninstantiated, an internally generated unique identifier is used. New classes are created by sending a message to the "class" object specifying code for the class's methods. Instance variables of specific object instances are accessed from methods inherited from ancestor classes running on the object's behalf via the goal inst(VariableName(Args)). Instance variables may be inherited from ancestor classes allowing default values. Changes in object states are accomplished by methods which assert or retract instance variables in the object's module.

Objects may be made persistent via the save message which causes a summary of their state to be recorded on disk. Dormant (disk resident) objects are addressed just as active, main-memory objects using their object identifier. If a dormant object is sent a

message, the system transparently locates the object on disk and creates a main-memory representation. Thus the environment incorporates a very primitive object-oriented database.

Prolog's powerful knowledge representation and knowledge base querying capabilities are used in Tangram by specifying object behaviors via Prolog rules. The built-in *unification* pattern matching in Prolog allows very general rules to be expressed quite simply. Solution packages written in other languages may be encapsulated within Prolog procedures using the foreign function interface. By combining rule-based specification with object-oriented structuring and inheritance, O-OP is an ideal language for building a smart modeling environment. The next section describes the architecture of Tangram and the current status of the system.

4 The Tangram Modeling System

A prototype of the Tangram object-oriented modeling system is operational on SUN 3/60s. The user can construct models, define new objects, and query models graphically. Our immediate applications are in computer systems performance modeling and our first domains were chosen to support this area's queueing network models and Markov chain analysis. In the queueing network domain, we have incorporated several exact and approximate analytic solvers. In addition, animated simulation is available. The Markov chain domain uses Prolog's backtracking to generate a set of reachable states of the model and the state transition rate matrix. Several numeric solvers for Markov chains are present. A specialized reliability analysis domain for modeling repairable computer systems is operational on top of the Markov chain domain. Each domain was built in a very short time frame (2 to 3 man-weeks).

4.1 The Architecture of Tangram

Figure 4 shows the basic components of the Tangram modeling system. Rectangular boxes are software components and ovals are classes and object instances inside the Object-Oriented Prolog language.

In the *Graphical Front-End*, models are represented by a collection of icons (graphical representations of objects) and lines (relationships among icons) with attributes (graphical instance variables) attached to them as in Figure 1. Models are constructed with a MacDraw[3]-like user interface (all figures in this paper are generated with this front-end tool), with object orientation extensions, entirely implemented in C running in the X Window System[4]. The graphics interface also supports model hierarchies in which sub-models may be represented by icons and used in higher level models.

In our prototype implementation, after the graphical representation of the model is specified, commands in O-OP to create the objects are generated by a translator and batched together to be sent to the object system. In the future, the front-end will be implemented in O-OP; instantiation at the front-end will cause immediate instantiation in the object system.

4.2 Sample Interaction

We again use the car-wash example to illustrate the features of the Tangram prototype. When the user selects solve from the pull-down menu after composing the model shown in Figure 1, the translator generates the corresponding model and associated objects (Figure 6) in the object system. The model is sent the avg_wait query.

The class of all models in the car-wash domain contains a query method which is inherited by this model instance. The query method invokes the car-wash domain expert system. From the constituents of the model and the query posed, the domain expert system generates O-OP code as shown in Figure 5 to create a queueing model and sends it a list of queries. Figure 6 shows part of the object hierarchy after the queueing model is created. Ovals represent classes and boxes are object instances with instance variables. The small tabs on top of the objects depict object IDs; internally generated IDs are shown with single quotes. Each new_object message in Figure 5 causes an instance of the specified class to be created. The first argument of new_object is bound to the object ID of the newly created instance and the second argument specifies a list of initial instance variables. The add_center message registers a list of queue objects with the model object, and add_routing specifies a list of routes between queues with associated branching probabilities. Finally, the query(Queries,Results) message causes M1 to solve itself, answering the queries specified in the first argument.

M1 inherits the query method from the class of all queueing models in the queueing network domain. The method invokes the queueing domain expert system which deduces that this model can be solved

[3]MacDraw is a trademark of Apple Computer Inc.

[4]X Window System is a trademark of the Massachusetts Institute of Technology.

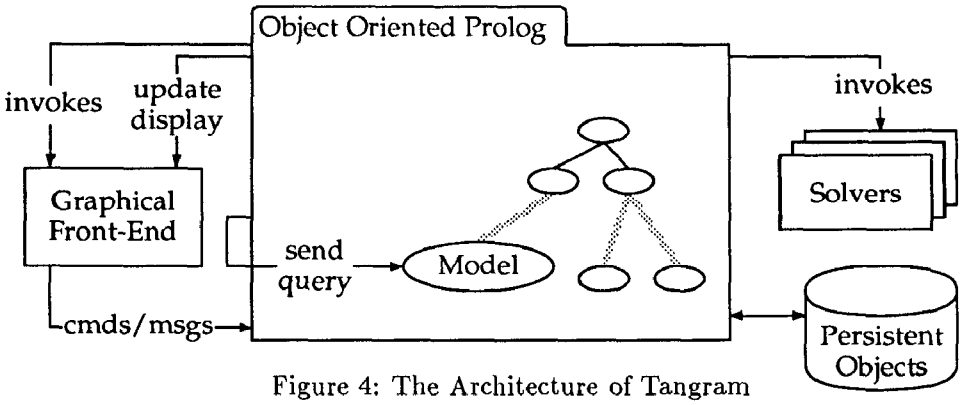

Figure 4: The Architecture of Tangram

```
queueing_model send new_object(M1, []).
source send new_object(SrcObj, [distr(poisson), rate(0.3)]).
ms send new_object(MsObj, [mean_service_time(1),num_servers(2)]).
fifo send new_object(FifoObj, [mean_service_time(15)]).
sink send new_object(SinkObj, []).
M1 send add_center([SrcObj,MsObj,FifoObj,SinkObj]).
M1 send add_routing([route(SrcObj,MsObj,1.0), route(MsObj,FifoObj,0.1),
        route(MsObj,SinkObj,0.9), route(FifoObj,SinkObj,1.0)]).
M1 send query([avg_wait(MsObj),avg_wait(FifoObj)], Results).

Results = [1.023, 27.273]
```

Figure 5: Sample O-OP Code

directly using numerical solvers. Each solver implements a query(Model,Queries,Results) method which binds Results to the list of numerical values of the answers to Queries for Model. The domain expert system tries to semantically bind the query message sent to the M1 with the query methods of the numerical solvers in the domain.

To accomplish this, the domain expert system asks each element of the domain's list of candidate solvers (implemented as the collection object "q_solvers" in Figure 6) to estimate the complexity of answering the query. The solver with the smallest complexity measure is selected and its query method is *semantically bound* to the query message sent to M1. The selected solver is sent the message query(M1,Queries,Results) which will bind Results to a list containing the average waiting time for the car wash and the car wax facilities.

If the queueing model is more complicated and can not be solved directly with numerical solvers, more sophisticated techniques such as decomposition can be invoked to solve the model. Once the queueing model has solved itself, the car-wash domain expert system uses the results to compute the answer to the original query and updates the display in the front-end.

5 Conclusions

We began with the goal of creating a modeling system which could accommodate a variety of analytic and simulation modeling techniques and would be easily extensible with respect to both integrating new solution techniques and tailoring the system to specialized applications. In support of these goals we developed a design philosophy that combines features from both the object-oriented and the logic programming paradigms. We introduced the notion of "smart models" which allows us to think of models which are not merely passive but rather can respond to high level queries to solve themselves, suggest solution methods, etc. This is accomplished by creating models in a "modeling domain" from which a model instance inherits knowledge of how to solve itself, etc.

A prototype of the system exists and is being used. It currently features modeling domains for queueing networks and Markov chains. Several specialized do-

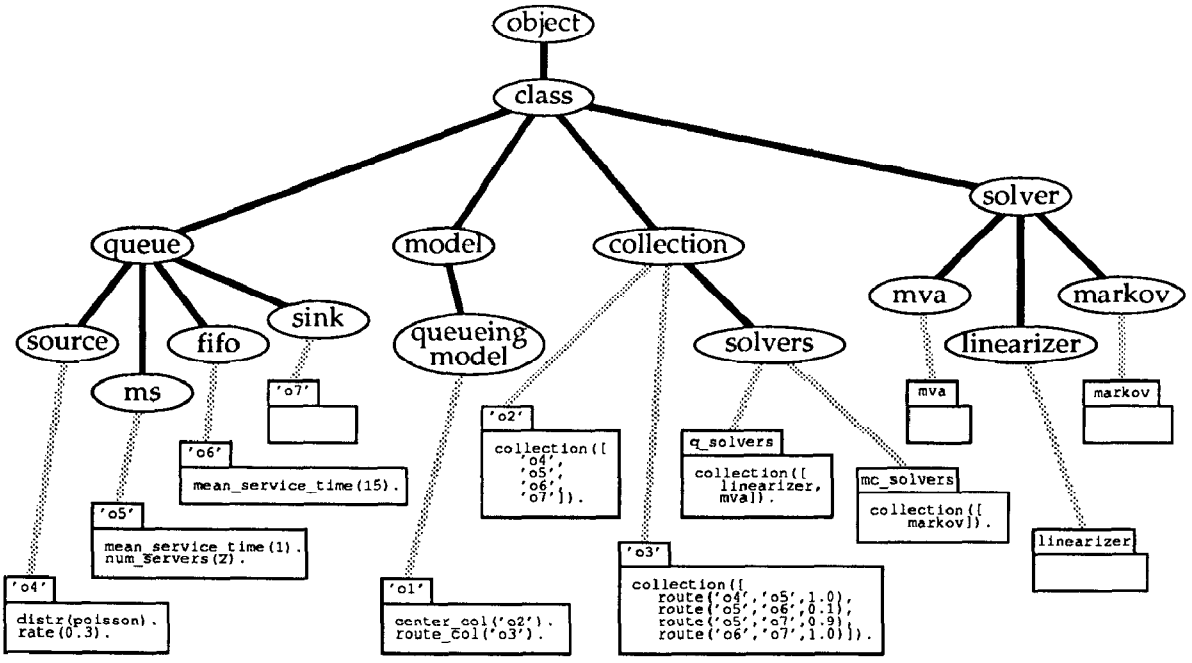

Figure 6: Part of Object Hierarchy Containing A Queueing Model

mains (e.g. reliability models) have been built on the basic system. We have found it very easy to add new solution modules to existing domains, create new domains, or specializing existing domains. In the near future we expect the system to expand quickly. We will be adding domains for analysis of distributed algorithms, load balancing, etc. We also expect a quickly expanding set of users from outside the implementation group. The expanding user base and range of applications will test our goals of providing a sufficiently flexible and powerful system satisfying diverse needs. While this remains to be verified, our experience thus far has been quite positive.

Acknowledgements

The authors wish to acknowledge the contributions of Gary Rozenblat, Leana Golubchik, and Jon Edwards to the implementation of the Tangram modeling system.

References

[ANAL85] "ANALYTICOL - An Analytical Computing Environment," *AT&T Technical Journal*, Vol. 64, No. 9, November 1985.

[BAVU87] S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann, and W. E. Smith, "Analysis of Typical Fault-Tolerant Architectures using HARP," *IEEE Transactions on Reliability*, Vol. R-36, No. 1, June 1987, 176-185.

[BERR82] R. Berry, K. M. Chandy, J. Misra, and D. M. Neuse, "Paws 2.0: Performance Analyst's Workbench Modeling Methodology and User's Manual," *Information Research Associates*, Austin, Texas, 1980.

[BERS87] S. Berson, E. de Souza e Silva, and R. R. Muntz, "An Object Oriented Methodology for the Specification of Markov Models," *UCLA Technical Report CSD-870030*, July 1987.

[COST81] A. Costes, J. E. Doucet, C. Landrault, and J. C. Laprie, "SURF: A Program for Dependability Evaluation of Complex Fault-Tolerant Computing Systems," *Proceedings of FTCS-11*, June 1981, 72-78.

[FUKU86] K. Fukunaga and S. Hirose, "An Experience with a Prolog-based Object-Oriented Language," *Proceedings OOPSLA '86*, September 29 - October 2, 1986, 224-231.

[GEOF87] A. M. Geoffrion, "An Introduction to Structured Modeling," *Management Science*, Vol. 34, No. 5, May 1987, 547-588.

[GOLD83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[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.

[GULL85] E. Gullichsen, "Biggertalk: Object-Oriented Prolog," *MCC Technical Report STP-125-85* Austin, Texas, November 1985.

[KAHN86] K. Kahn, E. Tribble, M. Miller, D. Bobrow, "Vulcan: Logical Concurrent Objects," *Proceedings OOPSLA '86*, September 29 - October 2, 1986, 580-618.

[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.

[MCCA87] F. G. McCabe, "Logic and Objects," *Imperial College Department of Computing Tech. Report 86/9*, London, England, 14 May 1987.

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

[MEYE86] B. Meyer, "Genericity Versus Inheritance," *Proceedings OOPSLA '86*, September 1986, 391-405.

[MINS87] N. H. Minsky and D. Rozenshtein, "A Law-Based Approach to Object-Oriented Programming," *Proceedings OOPSLA '87*, October 1987, 482-493.

[MISR86] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, March 1986, 39-65.

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

[MUNT88] R. R. Muntz and D. S. Parker, "Tangram: Project Overview," *UCLA Technical Report CSD-880032*, April 1988.

[PAGE89] T. W. Page, Jr., "Object-Oriented Prolog," *Ph.D. Dissertation, UCLA Department of Computer Science*, Los Angeles, CA, September 1989.

[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.

[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.

[WARR83] D. H. D. Warren, "An Abstract Prolog Instruction Set," *SRI Technical Report 309*, October 1983.

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