Compositional Development of Performance Models in POEMS

J. C. Browne and E. Berger
Department of Computer Science
University of Texas
Austin, TX 78712
{browne,emery}@cs.utexas.edu

A. Dube
Microsoft Corporation
Redmond, Washington 98052
adube@Exchange.Microsoft.com

Abstract

Performance models are software systems where the components implement abstractions of the behavior of a total system. This paper describes a capability for semi-automatic development of performance models of computer systems spanning applications, operating systems and hardware by composition from a library of components. Compositional development of performance models is a domain specific instance of the general problem of software component reuse or design reuse. The concepts enabling compositional development of performance models in POEMS are encapsulation of analysis level objects with associative interfaces and hierarchical dynamic data flow graphs as a structuring model. Objects with associative interfaces will be called compositional objects. Compositional objects and hierarchical dynamic data flow graphs provide a framework for development of performance models which incorporate multiple modes of evaluation, span multiple semantic domains, span multiple levels of abstraction and parallel implementation. Algorithms for composition through associative interfaces with automatic generation of parallel executables for the performance models will be defined.

1. Introduction

The goal of the Performance Oriented End-to-end Modeling System (POEMS) project is to create and experimentally evaluate a problem solving environment for end-to-end performance modeling of complex parallel/distributed systems, spanning application software, runtime and operating system software, and hardware architecture. The POEMS project combines innovations from communication models, data mediation, parallel programming, performance modeling, software engineering, and CAD/CAE to realize this goal. The motivation for POEMS is that determining the end-to-end performance of large-scale parallel and distributed systems across all stages of design enables more effective development of these complex software/hardware systems. But if performance modeling is to be useful then the cost and effort of developing performance models must be a small fraction of the cost of system development. Compositional development of performance models from libraries of models of components is a necessary ingredient of cost/effort minimization for development of performance models.

The POEMS compositional process has been sketched in [1]. This paper extends the overview given in [1] and adds definition and description of the algorithms for automated composition and generation of executable parallel programs for performance models.

1.1 Definition of Terms

The terms in which performance models are defined in this paper are:

Component - A component is a logical or physical element of a system. An example of a component is a cache in a model of the system level architecture of a processor/memory system.

Component Model - A component model is a representation of a component which resolves some or all of the behavior of the component. A component may be represented by many different instances of component models.

Transaction - A transaction is a unit of work to be executed. A transaction has an identity and a state which may persist across execution sites. Instances of transactions cross interfaces to generate interactions among model instances.

Interaction - A simple interaction is the flow of a transaction across component interfaces. An interaction may be simple or compound. A compound interaction is a sequence of simple interactions governed by a protocol.

Interface – An interface is an encapsulation of a component model. An interface specifies the interactions in which a component model can participate both as invoker and invokee. The interactions in which a component model can participate are specified in the interface by an associative characterization of its behavior. (Section 2.3)

1.2 Performance Models as Software Systems

Performance models are software systems with characteristics as follows:

(a) Performance models span three (or more) semantic domains:
(i) An application domain,
(ii) The operating system and the runtime systems used by the application,
(iii) The hardware architecture of the execution environment.
Any given model may or may not explicitly incorporate all three domains.
(b) Components are frequently defined at multiple levels of resolution in a performance model. Transactions may have to be translated across levels of resolution of component models.
(c) Multiple methods of evaluation (different simulators and/or a mixture of procedural and analytic evaluation methods) may be applied to components of a given system model.
(d) Performance models must maintain a logical clock.
The result of these characteristics is that end-to-end performance models of computer systems are software systems which are challenging to develop. But the individual

components of performance-oriented models of computer systems typically have relatively simple and precise semantics.

1.3 Overview

Section 2 defines and describes the conceptual framework for automated composition of performance models, Section 3 gives the algorithms for composition and translation. Section 4 gives the current status of the project. Section 5 briefly surveys related research.

2. Conceptual Framework.

The conceptual elements of the POEMS process for automated composition of performance models from components are:

(a) Abstraction hierarchies.
(b) Domain and object models [16, 14, 5].
(c) Associative interfaces [3] which are used to encapsulate standard objects to create compositional objects.
(d) Dynamic hierarchical dependence graphs [12] as a uniform structuring mechanism for the software systems implementing performance models.
The familiar concepts, abstraction hierarchies, standard object models and data flow graphs, will be treated only briefly to sketch their use in the compositional process.

2.1 Abstraction Hierarchies

Abstraction hierarchies decompose a system into layers of coherent semantic elements. The obvious abstraction hierarchy for representation of end-to-end models of computer systems is shown in Figure 1. Each layer of the abstraction hierarchy defines a semantic domain for the development of an object model of components and component models for that domain.

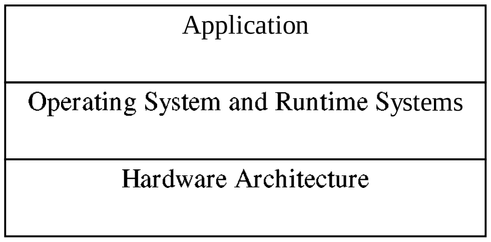


Figure 1 - Abstraction Hierarchy for Performance Models of Computer Systems

Each layer has quite different functionality and different models of computation. Typically software is represented in an asynchronous model of computation which may be either sequential or parallel. Operating systems are typically defined in an asynchronous parallel model of computation. The operations of software systems are often of substantial semantic complexity. Applications such as matrix computations are composed from operations which are very different from the operations such as scheduling and queue management which are characteristic of operating systems. Hardware is typically represented in a synchronous parallel model of computation in which the operations have simple semantics.

Components in lower levels of the abstraction hierarchy are referred to as being in the implementing domains for the higher level objects.

2.2 Domain and Object Models

Development of object models [16, 14, 5] for software systems begins with partitioning of systems into coherent semantic domains. This partitioning we accomplish for computer system models with definition of the abstraction hierarchy of Figure 1.

The first step in development of an object model for a domain is to identify the objects (in our case components) which occur in the domain. Identification of objects and thus components in the application domain of our abstraction hierarchy follows the conventional process [16, 14, 5]. Identification of objects (components) in the operating system and hardware architecture domains is determined by the systems being modeled and by the level of resolution at which the models are defined.

The properties of each object are defined in terms of a set of attributes which characterize the states and behaviors of the objects. Each attribute is a simple or enumerated type where the value set is determined by domain experts as a part of domain analysis. Objects which are to represent component models for performance models of computer systems require additional attributes including the level of resolution of the models for the components and the mode of evaluation of the component models. For example, a "resolution" attribute will specify a measure of the faithfulness with which a component model represents the actual component.

The third step in definition of standard objects is usually the definition of relationships among objects. The associative interface carries the information captured in relationships. The compositional process implicitly determines the relationships among objects.

We use a simplified model of hardware architecture at the memory system level to illustrate associative interfaces and compositional objects. The components (=objects) of a hardware architecture at the memory system level include caches and memories. A cache object might have as attributes: size, mapping algorithm, and line width. A memory object might have as attributes: size, line width and latency. The simple system model given following couples a cache directly to a memory. This simplification corresponds to a system where bus delay is ignored. This simple example does not require the full capabilities of compositional objects but it demonstrates the fundamentals of associative interfaces. An example of a segment of a generic object model for a memory architecture with cache and memory components is seen in Figure 2.
A set of component models for the memory system architecture domain might include four or five instances of the cache component, each with a different set of attribute values.

2.3 Associative Interfaces and Compositional Objects

Associative interfaces encapsulate standard objects (components) with an interface which specifies all of the interactions in which a component model can participate and which specifies the behaviors it implements in terms of the attributes which define the behavior and the states of the standard objects. An associative interface has two elements: an *accepts* interface and a *requests* interface.

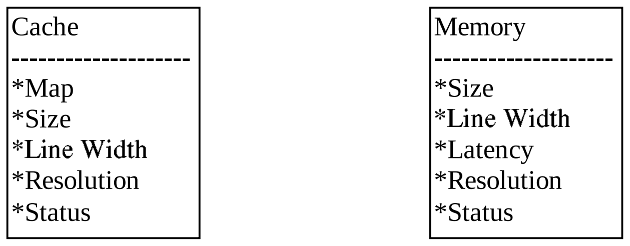


Figure 2 – Object Definitions of Cache and Memory Components

The *accepts* interface for a component is a set of three tuples (profile, transaction, protocol).

A profile is a set of attribute/value pairs.

A transaction specifies the functionality and the parameters of the units of work which are executed on a given interaction by this instance of the component.

A protocol defines the sequence of simple interactions necessary to complete the interaction specified by this accepts interface entry. The two allowed values of protocol are “call return” and “data flow.”

The *requests* interface is a set of three-tuples (selector, transaction, protocol).
A selector is a conditional expression over the attributes of the objects in its domain and the other domains with which it has interactions. The conditional expression of a selector is a template with slots for attribute name/value pairs specified in the profiles of other object instances. A selector is said to match a profile whenever the conditional expression of the selector evaluates to true when the attribute values from the profile are inserted into or compared with the slots in the template of the conditional expression. Figure 3 shows an instance of a cache object encapsulated with accept and request interfaces.

The informal syntax of *accepts* and *requests* interface tuples is

Accepts <Component name> {profile([attribute/value pairs];
transaction= transaction specification;
protocol = protocol specification}

Requests <Component name> {selector(conditional expression over attributes);
transaction = transaction specification;
protocol = protocol specification}

A compositional object may have multiple *accepts* and *requests* in its associative interface. Multiple accepts arise when a component model implements more than one behavior. Multiple requests arise from requirements for parallel execution and/or requests targeting different domains. A component model may have a request instance for a service from an implementing domain and a request instance for continuation of execution in its own domain.

The compositional object specification of a cache component model given in Figure 3 accepts transaction "return-value" for the value associated with a given address and either "returns" the value or invokes a component model of a memory to "fetch" the value if necessary. The "Resolution = trace" attribute selection in the profile stipulates that this instance of a cache model is capable of accepting actual address traces. This selection implies (but does not guarantee) that this component model of a cache maintains its storage state. The requests interface has a "Resolution = any" specification which implies that the memory model need not resolve addresses. That is, the memory model may simply return a "value" after a time delay. The memory model also has a "return-value" transaction.

```
Accepts Cache-CPU
{profile (Map = direct, Resolution = trace, Size = 256K, Line Width = 32B,
Status = accept-address);
transaction = return-value(address);
protocol = call return}

                Cache
                -----------------
                *Map
                *Size
                *Line Width
                *Resolution
                *Status

Requests Cache-Memory
{selector = (Memory, Resolution = any, Line Width = 32B, latency = 0.5,
Status = accept-address);
transaction = return-value(address);
protocol = (call return)}
```

Figure 3 – Compositional Object for a Cache Component

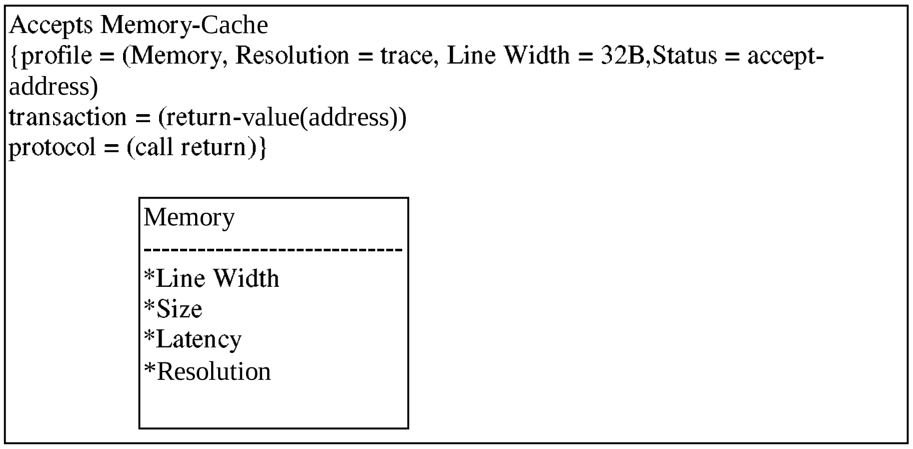
In general components require entries in the Accepts and Requests interfaces for every interaction. However, the Cache component model does not require a Requests interface entry for returning a value to the CPU component model since the Cache component model is invoked through a synchronous protocol. Similarly the Memory

Component model does not require a Requests interface entry for returning a value to the Cache component since it is invoked only through a call return protocol.

Associative interfaces, together with a runtime system carry the information necessary: to enable matching of component models, to invoke runtime system procedures, to implement translation of transactions across levels of abstraction and to sequence interactions to implement protocols required by the matchings between component models. Translation of transactions across levels of abstraction is a generalization of the compiler mechanism of type coercion. In this case, since the cache model transaction sends an address, compilation of the model will drop the argument "address" if the available memory model does not use the address.
An instance of a memory component model with an accepts interface which will match the requests interface of the cache component model of Figure 3 is given in Figure 4.

2.4 Specification of Object Behaviors

In most variants of the standard object model the behavior of objects are defined by some type of state machines which transition among the set of states defined by the allowed values of the attributes of the object. The operations which implement the actions associated with a state transition are initiated by the arrival of an event. Selector-transaction pairs in POEMS correspond closely to events in the standard object model. The composition process described following binds events (selector/transaction/protocol tuples) to actions (profile/transaction/protocol tuples).


Accepts Memory-Cache
{profile = (Memory, Resolution = trace, Line Width = 32B,Status = accept-address)
transaction = (return-value(address))
protocol = (call return)}

Memory
----
*Line Width
*Size
*Latency
*Resolution

Figure 4 – Compositional Object Model for a Memory Component

The behavior of a component model will be described by either of:
(i) one or more programs written in some simulation language, Maisie [2], SES/Workbench [17] or SimpleScalar [6]
(ii) one or more programs which supply a value determined by an analytical solution of a behavioral specification or
(iii) the actual source or executable code of the component model.

The arrival of a selector/transaction pair will cause the invocation of the operation of the transaction in the selected component.

2.5 Program Structure

A PSL (POEMS Specification Language) program consists of a set of component models and a set of attribute/value pairs from each domain. The set of component models consists of a mandatory Start node, a set of component models which the analyst specifically wants to include in the performance model and a mandatory Stop node. A Start node has only a requests interface. A Stop node has only an accepts interface. The set of component models included in the program can range from the complete set of component models which are to be composed to the empty set. Composition is initiated by the requests interfaces of the Start node and terminates with the accepts interfaces of the Stop node.

The component models for each domain are kept in a database which is accessible to analysts and to the compilers and translators. The database will include definitions of transaction types and definitions of type coercion routines for mapping among transaction types. The database will also include definitions of the protocols for implementation of multi-step interactions.

The database will be populated by analysts constructing components in conformance to the object model and entering them into the domain model database. Analysts will be able to extend the domain models by defining new objects, attributes and relationships as required.

3. Compositional and Translation of PSL Programs into System Models

3.1 Composition Algorithm

Composition of component models is accomplished through *matching* of component model interfaces. A selector expression is a conditional expression over the values of the attributes of the domains of the system model. When evaluated using the attribute values of a profile, a selector is said to match the profile whenever it evaluates to true. A match that causes the selector to evaluate to true selects an object as the target of an interaction. The match is completed and the component models are bound only if the transactions of the requests interface entry and the accepts interface entry are conformable and if the protocol specifications of the requests interface entry and the accepts interface entry are identical. Transactions are conformable if the signatures either match or the POEMS component model library includes a routine to translate the parameters of the transaction in the *requests* interface instance to the parameters of the transaction in the matched *accepts* interface and vice versa.

The composition algorithm is as follows:

a) The compositional compiler attempts to find matches for the entries in the requests interface of the Start node. Search begins with the set of component models in the initial set provided by the analyst and proceeds to the component data base if necessary. Type coercion of transaction parameters is done where necessary.

b) The compiler then attempts to match the entries in the requests interfaces of the component models bound to the start node in step a to the accepts interfaces of the component models in the analyst supplied component model set or in the component model data base.
c) The matching process continues until the requests interfaces of all of the selected component models terminate on the accepts interfaces of the Stop node.
d) If any entry in the requests interface of any component model which has been bound to the system model cannot be matched the compiler generates a message notifying the analyst that system model composition cannot be completed.

The result of a successful application of the composition algorithm is a directed graph program structure where the bound components define an end-to-end performance model. The next step is to instantiate this program structure into an executable parallel program.

3.2 Translation to Executable

The executable program is realized by instantiating the PSL data flow graph program as a program in the CODE [12] parallel programming system.

The source code for CODE is a generalized hierarchical dynamic dependence graph (HDGDGs) [12]. Each node in a CODE HDGDG is either a type declaration for a node which executes an atomic computation or a type definition of a subgraph. Each node consists of a set of input ports, firing rule, a computation, a routing rule and a set of output ports. (Information on CODE can be found at the url http://www.cs.utexas.edu/users/code.) Ports are typed containers for values. Firing rules are predicates over the states of the input ports. The computation is a C function or procedure. Routing rules are predicates over the state of the computation and the input ports which map values to output ports. Each arc in the HDGDFG carries data from an output port of one instance of a node type to an input port of some instance of some node type. Arcs are conceptually unbounded FIFO queues. The connectivity of the graph is determined by the matches among the associative interfaces of the component models. The number of instances of each node is specified as a parameter which can be varied at runtime.

Translation of the nodes and arcs of the data flow graph generated by the composition algorithm to the nodes and arcs of a CODE graph is straightforward. The translation converts each node in the PSL data flow graph to a node in the CODE HDDFG format. An input port is generated for each accepts transaction which has been matched. An output port is generated for each requests transaction which has been matched. The firing rule for the CODE nodes is either an "and" over all of the input ports or an "exclusive or" over all of the input ports. Each binding of a requests interface entry to an accepts interface entry is converted into a CODE arc binding an output port to an input port. Call return protocols generate arc pairs. Data flow protocols generate a single arc.

3.4 Data Flow Graph Based Management of Simulation Time

Execution of discrete event simulation models in general and performance models with components which use discrete event simulation as an evaluation mechanism requires explicit management of simulation time. Parallel and distributed implementations of discrete event simulations have been and are a major research problem. A recent survey of research in parallel and distributed simulation is given by Low, et. al. [11]. We sketch briefly here the data flow graph based mechanism used for management of simulation time in the parallel/distributed discrete event programs generated by the PSL compilation system.

a) The simulation time is set to zero by the Start node.
b) Each node maintains a local simulation time clock.
c) Each source/sink node pair which is connected by a data arc also has a clock arc which sends the local time at which the atomic execution of the source node terminated to the sink node.
d) The set of local clocks are updated by the Lamport algorithm [10].

This simulation clock maintains causality consistent with some *parallel* execution of the system represented by the performance model. A paper describing this mechanism for management of simulation time is in preparation.

4. Status and Summary

A specification language for performance models based on compositional objects as described in this paper has been defined [7]. A feasibility demonstration compiler which maps programs written in this specification language to the CODE [12] parallel programming system has been written [7]. Simple models have been translated to CODE. The models and the translations can be found in [7]. A more robust compiler is under development.

5. Related Research

Space limitations preclude detailed discussion of the great volume of conceptually related work. Associative interfaces have been adapted from Bayerdorffer's [3, 4] associative broadcast model of communication. The most directly related work is the Linda [9] programming system. Linda and associative interactions are both instances of a communication model derived from the top classification of a taxonomy of naming models [3]. (Java Spaces [18] is a distributed form of Linda.) Concepts equivalent to associative interfaces have long been used in research on knowledge based systems [8]. Associative interfaces are also generalizations of typed ports which have a long history in both operating systems and programming systems. Object brokers, for example Corba [13], and COM/DCOM [15] implement composition through some type of mediation interface.

6. Acknowledgements

This work was supported by DARPA/ITO under Contract N66001-97-C-8533, "End-to-End Performance Modeling of Large Heterogeneous Adaptive Parallel/Distributed Computer/Communication Systems." Formulation of the composition, translation and time management algorithms owes much to the members (V. Adve, R. Bagrodia, E. Deelman, E. Houstis, P. Teller, J. Rice and M. Vernon) of the POEMS project.

7. References

[1] Adve, V., R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. To appear in IEEE Trans. on Software Engineering (special issue on Software and Performance).

[2] Bagrodia, R., and W. Liao. Maisie: A Language for Design of Efficient Discrete-Event Simulations. IEEE Transactions on Software Engineering, Apr. 1994.

[3] Bayerdorffer, B., Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems, Ph.D. Dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin. Dec. 1993.

[4] Bayerdorffer, B., "Distributed Programming with Associative Broadcast", Proceedings of the Twenty-eighth Hawaii International Conference on System Sciences, January 1995,pp. 25-35.

[5] Booch, G., J. Rumbaugh and I. Jacobson The Unified Modeling Language User Guide (Addison Wesley Longmans, Reading, MA. 1999)

[6] Burger, D., and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, UW CS Tech. Rept. 1442, Madison: University of Wisconsin-Madison, June 1997.

[7] Dube, A. 1998 A Language for Compositional Development of Performance Models and its Translation. Masters Thesis, Department of Computer Science, University of Texas at Austin.

[8] Falknerhainer, B., and K. Forbus, Compositional Modeling: Finding the Right Model for the Job" Artificial Intelligence 51, 95-153 (1991).

[9] Gelernter, D., N. Carriero, N. Chandran and S. Chang, Parallel Programming in Linda Proceedings of the International Conference on Parallel Processing (St. Charles, IL, August 1985) pp. 255-263.

[10] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 17:8, 558-565 (1978).

[11] Low, Y-H. *et. al.* Survey of Languages and Runtime Libraries for Parallel Discrete Event Simulation Simulation 72:3,170-186(1999)

[12] Newton, P., and J. C. Browne The CODE 2.0 Graphical Parallel Programming Language, Proceedings of the 1992 International Conference on Supercomputing, Washington, DC, July 1992, pp. 177-177.

[13] The Object Management Group, The Common Object Request Broker: Architecture and Specification. Revision 2 (OMG, June 1995)

[14] Rumbaugh, J., *et. al.* Object-Oriented Modeling and Design (Prentice-Hall, Englewood Cliffs, NJ, 1991)

[15] Sessions, R., COM and DCOM: Microsoft's Vision for Distributed Objects_(John Wiley,1997)

[16] Shlaer, S. and S. Mellor Object Lifecycles: Modeling the World in States (Yourdon Press, New York, 1992)
[17] SES. 1996. SES/Workbench 3.1 Users Reference Manual. SES, Austin, Texas.
[18] Sun Microsystems The Java Spaces Specification. [online] Available: (http://www.sun.com/jini/specs/js-spec.html)