Systems of Systems as Communicating Structures

Vadim Kotov
Computer Systems Laboratory
HPL-97-124
October, 1997

E-mail: kotov@hpl.hp.com

concurrent distributed systems, systems of systems, Communicating Structures, object-oriented modeling, visualization

By **Systems of Systems (SoS)** we mean large-scale concurrent and distributed systems the components of which are complex systems themselves (e.g. enterprise intranets). **Communicating Structures** are hierarchical structures that represent SoS in a uniform, systematic way as composition of a small number of basic system objects.

Communicating Structures are focused on modeling of SoS the performance of which largely depends on **communication**, **data traffic** and **data placement**. The systems components are represented as **nodes**. The nodes have **memory** that may contain **items**. **Nets** are sets of **links** that connect the nodes. The items move from a node to a node along links. All these objects may have hierarchical structure.

CSL is a C++ based library and an object-oriented core environment for the modeling and analysis of SoS in the framework of Communicating Structures. It includes both simulation and analytical (queueing analysis) options as well as GUI for the model construction and visualization tools for analysis of the modeling results.

A case study in which CSL was used to analyze a global distributed computing environment is presented.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1997

1 INTRODUCTION

Hardware, software, network, and application systems are merging into *integrated information systems.* As a result, the variety of feasible integrated system architectures and their complexity is rapidly increasing. Such systems become very large and very complex systems which are, in fact, *Systems of Systems* as their components are themselves complex systems.

Some examples of today's Systems of Systems (SoS) are

- multiprocessor servers and clusters;
- enterprise intranets supporting common business processes;
- distributed global mission-critical applications;
- distributed control systems;
- distributed design/manufacturing systems;
- World-Wide Web;
- their combinations.

Typical SoS have to satisfy many strict requirements, among which are cost effectiveness (SoS are often unique and very expensive); responsiveness; throughput; scalability and flexibility; availability; maintainability; reliability, fault tolerance, and recoverability; data and application integrity; security.

Many of these requirements are conflicting. Only modeling can help to find the best compromise solutions. The competitive market does not give much time for experimenting with prototypes. There is a clear need for design methods, techniques, and tools that allow designers to construct and analyze quickly and reliably various architectural hypothesis and evaluate them against a wide spectrum of desired system properties.

However, SoS represent a challenge for the modeling and analysis as the solution space is huge and complex. Modeling methods and tools typically used in object-oriented analysis and design (for example, UML [Fow97], statecharts [Har97]) are biased to the specification aspects as their main goal is to support the rigorous and efficient design and development process. For SoS, the main problem is to identify satisfactory solutions among a sea of solutions. SoS are also too complex and divers to fit into formal semantics Procrustean frames.

To meet this challenge, the SoS models should be as simple as possible without, of course, loosing those features which are important for the system validation. The best way to simplify the models is to identify

- objectives of the modeling and those global system features which are relevant to those objectives;
- a small number of concepts that are common for most of the SoS and in which terms the modeling objectives can be adequately specified.

2 COMMUNICATING STRUCTURES

Here we propose to view large information systems in a uniform and systematic way as *Communicating Structures* in which main activities are related to coordination of *data traffic* and *data placement.* The modeling objectives are

- evaluation of the system performance in terms of average latencies, throughput, utilization, sensitivity to variation of the system and workload parameters;
- identification of congestions, bottlenecks, non-fairness and unpremeditated behavior.

Such a view emphasizes those system features and components that generate and manage the data traffic.

This paper presents the *Communicating Structures Library (CSL)*, an implementation of the Communicating Structures as a core environment for the modeling and synthesis of SoS. In its most general form, a Communicating Structure is a *hierarchical* and *concurrent* structure that represents the SoS components and communication between them. The system components are represented simply as *nodes.* The nodes have *memory* that may contain *items. Nets* are sets of *links* that connect the nodes. The items are generated in some nodes and move from node to node along links concurrently and with some delay. Items may be modified within nodes. The item traffic models the data traffic in the system which is represented as a Communicating Structure.

Items, nodes, memories, and nets may be elementary or may have some structure. For example, items may represent simple data such as frames, packets, as well as complex messages and large chunks of data. The nodes may represent relatively small units such processors, memories, I/O and storage units, or systems such as multiprocessors, computer clusters, servers, networks, etc. Nets may represent simple point-to-point links as well as busses, crossbars, interconnects, cascaded switches, communication lines and other data transfer facilities.

Thus, the items, nodes, memories and nets are Communicating Structures *objects* which are either simple or hierarchical. The objects may be assigned different attributes (numbers, variables, functions, and processes) which

- define quantitative parameters such as the number of subobjects in an object; the memory capacity, delays, the current number of items in a memory, time constraints, etc.;
- locate an object in the model hierarchy such as, the object's name, its relative address in the hierarchy tree;
- generate and control the item traffic;
- change behavior of objects;
- provide an input data for objects and register their behavior and for output and further analysis;
- provide data and functions for analytical modeling.

SoS are too complex and divers to fit into a unique formal semantics Procrustean frame. Instead, Communicating Structures provide default semantics for basic objects, functions and processes which may be changed by a user. This means that Communicating Structures provide a common base for future more specific and, if required, more rigorous "dialects" adjusted for particular tasks and level of detail of SoS analysis and construction.

In general, any information that is relevant to a specific study of a modeled system may be easily added to a Communicating Structure describing the system. Communicating Structures allow easy abstraction/refinement modifications in order to be used at different levels of the specification and modeling detail.

CSL is accumulating generic or parameterized CSL objects, functions, and processes that may be quickly assembled into a particular model and tuned for a specific case study using input parameters.

3 C++/CSIM CONSTRUCTS

The CSL hierarchy is based on C++ classes and CSL concurrency uses the main structures of C++/CSIM, a process-oriented discrete-event simulation package [Sch95].

A set of modified CSIM structures is introduced to generate and coordinate concurrent processes (see Figure 1). A *process* is a C++ procedure which executes a *create* statement. This statement invokes a new thread that proceeds concurrently with the process that has invoked it.

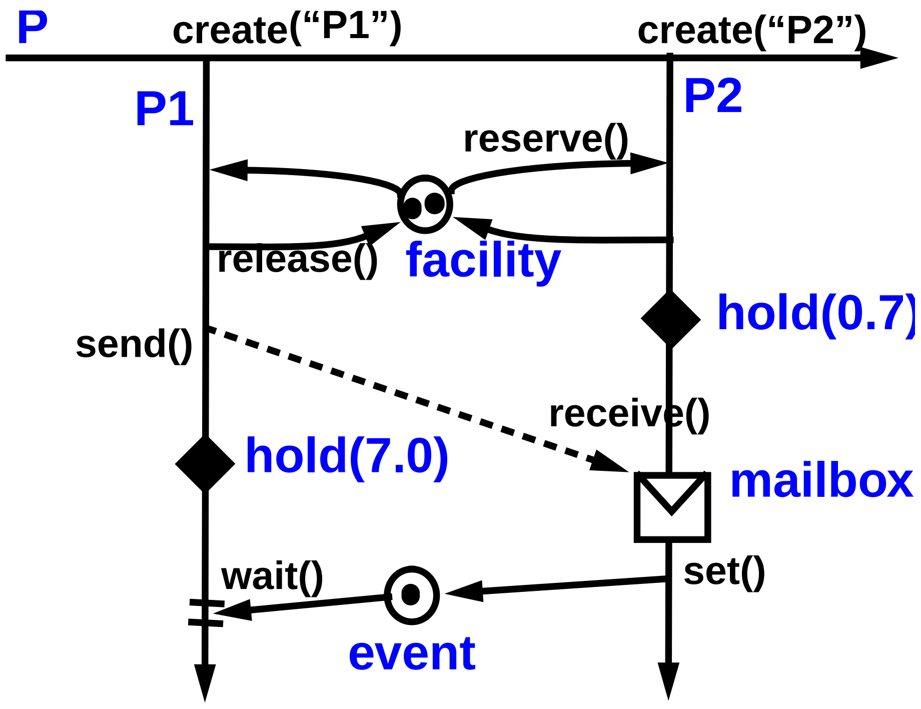


Figure 1: CSIM concurrency constructs

There can be several simultaneously active instances of the same process each of which has its own runtime environment. A process can be in one of four stages: *passive* (and ready to start), *active*, *holding* (allowing a modeling time to pass), and *waiting* (for permission to continue after it has been interrupted).

The mechanisms to organize the interaction between the processes are *mailboxes*, *facilities*, and *events*.

The CSL class *Mailbox* is used for interprocess communication. A process can *send* a message to a mailbox and *receive* a message from a mailbox. If a process does a receive operation on an empty mailbox, it automatically waits until a message is sent to this mailbox. The CSL Mailbox is augmented by an additional operation em send_with_delay that makes it possible to send a message to a mailbox with some time delay.

The class *Facility* models a resource. It contains a single queue and several servers. Only one process can at a time hold a server after it executes the *reserve* statement. If there is no available server, the process waits in the queue until one of the servers is *released* and there is no waiting process in the facility queue ahead of this process.

Events are used to synchronize processes. An event is a state variable with two states, *occurred* and *not occurred*, and two queues for waiting processes. One of these queues is for processes that have executed the *wait* statement (and are in a waiting stage) and another is for processes that have executed the *queue* statement (and also are in the waiting stage). When the event occurs, by executing the *set* statement, all waiting processes and only one of the queued processes are allowed to proceed.

4 THE COMMUNICATING STRUCTURE OBJECTS

Four basic elements of Communicating Structures are *items*, *memory*, *nets* and *nodes*.

Nodes typically generate, receive, store, forward, and, perhaps, modify data abstractly presented as items. They store and retrieve items in the node's memory. Items are "dynamic" objects which are born, travel and perish. Nets connect the nodes into a Communicating Structure in which the items travel from source nodes to destination nodes.

All these elements are derived from the common CSL class *Object*. As all CSL objects may have a hierarchical structure, the class Object represents tree-like hierarchies with the class members and functions which help to handle the hierarchy, for example, to select subtrees and sets of subtrees, to check some properties of trees, to apply functions to objects-subtrees, etc.

The class Object is derived also from the class Facility. This makes the object to be a resource for a competition among concurrent processes. The number of servers in the object is set during the object construction.

A CSL *Memory* is an object to store items. In the general case, the Memory is a hierarchy of (sub)memories with the ability to store items at different levels of the hierarchy. The top memory of the hierarchy is contained in a node. At the bottom of this hierarchy are *Locations*, "elementary" memories which hold pointers to stored items. One location is capable of storing exactly one item.

In the general case, the net inherits a multilevel hierarchy from the class Object. So, a net may consist of subnets. A "top" net enters into a "docking" node for which it defines communication links among the node's subnodes. So, the constructor of the net contains a docking node as a potential argument.

At the bottom of this hierarchy are *Link*s, "elementary" nets each of which connects just a pair of nodes. Each link delivers items from a *from-node* to a *to-node* with *link delay* which is either a constant or a function of some of that item's attributes. The from- and to-nodes are identified by their pointers. Being derived from the class Object, the link is a resource with some number of servers which define the maximal number of transfers that may occur along the link simultaneously.

The main building block of Communicating Structures and CSL models is a node. It contains a memory and, possibly, subnodes as well as a net with its links connecting subnodes. The whole model itself is a top level CSL node. The nodes are assigned different functions and processes which originate and control the items movement in the Communicating Structures simulation runs. Most of the basic node member functions and processes are virtual and may be customized by user for specific purposes. The default definitions of these functions provide some "generic" item traffic which is generated in some subset of nodes and destined for some subset of nodes with shortest path routing on the way.

To apply the queueing analysis, the CSL nodes is supplied by classes that implement queue models such as *M/M/m* or *G/G/m*. The analytical model of a communicating structure is built using a network of queue models, this network being derived from the topology of the strucutre.

5 THE COMMUNICATING PROCESSES

With each node, a *main process* and a *generation process* are associated.

When the generation process generates an item, the item is stored in the node's memory, and a message is sent to the node's mailbox in order to activate the node's main process. This message contains a pointer to the address of the location in which the item has been stored.

The *main_process* (see Figure 2) prescribes the node functionality and behavior. (A rectangle represents a function (procedure), a rounded rectangle represents a CSIM process, a rhombus is a condition, and a circle is a loop condition.) The process is awakened by a message to the node mailbox and starts with finding a location in the memory to work on. It may be either the location indicated in the message or any other location prescribed by the memory access function.

Then the main process analyzes the destination path of the item stored in this location. If the destination path is empty, the process completes its work doing actually nothing. Otherwise, the head of the path is extracted. If it is a pointer to this node and it is the only element of the path, the *transformation* procedure is initiated. This function may make some changes to the item. In particular, the function may change the item's destination, or make clones of this item for subsequent spawning into the communicating structure. The transformation function will almost always be customized, as it actually defines the node's functionality. The default version of transformation is an "empty action".

After the transformation, the main process either terminates or the *transfer* procedure is initiated. The decision to terminate or to transfer is made at the end of the transformation

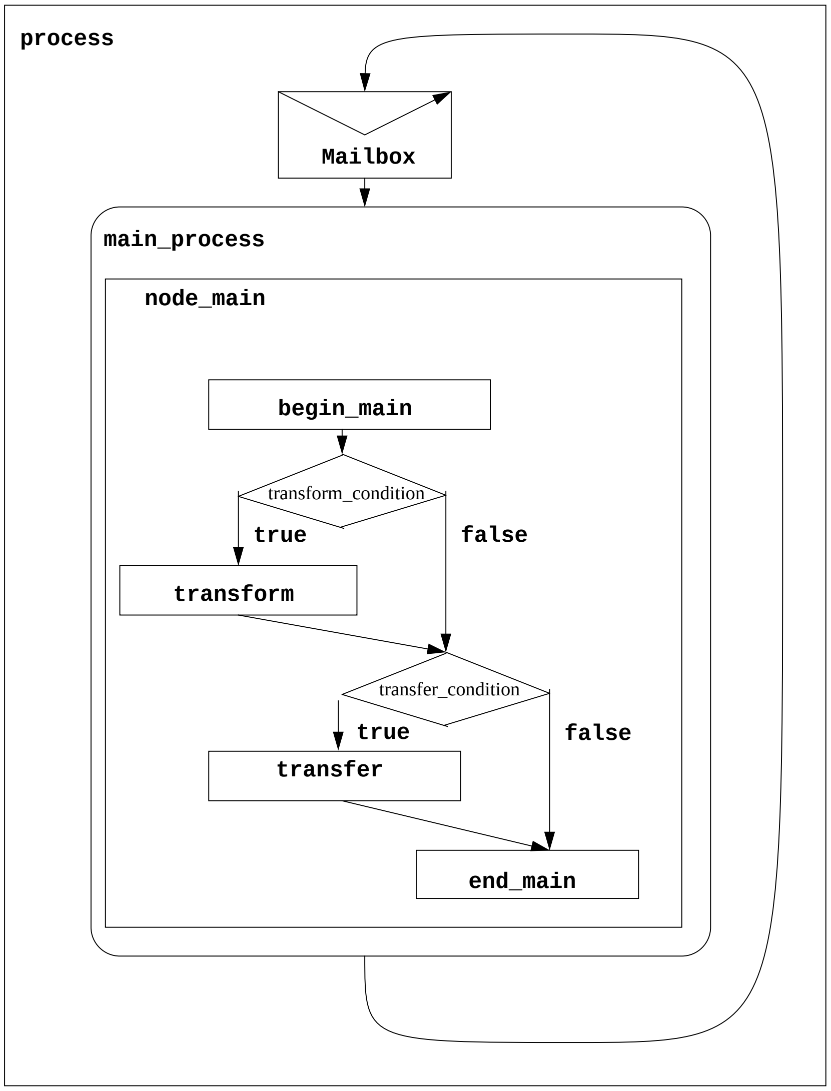


Figure 2: Main node process

and is signaled in some way. For example by using the memory pointer as in the the default version of the main process.

The transfer function organizes the transfer of the item (or the item's copies) to other nodes. In the default definition, it analyzes the item's destination path and selects one of the possible transfer modes: a monotransfer or a multicast, a synchronous or an asynchronous transfer.

6 CSL PARTS KIT

The system parts kit contains sublibraries which accumulate often used system structures, functions, and processes. Some of these parts are generic, that is, they are used quite often but are not basic CSL objects or functions. Others may be specialized and often used in domain specific models.

For example, specific types of memories which are derived from the class *Memory* are introduced in the KIT.

The class *Buffer* contains some number of locations that are addressed by an integer index. One can put an item into a location only if the location is vacant, that is, it does not contain any item. One can get an item from a location only if the location is not empty, that is it contains an item. In similar way the classes *FIFO*, *Stack* and *PriorityQueue* are introduced. These memories are often serve as "control memories" which help to control implicitly the traffic in Communicating Structures.

In many cases it is convenient to have a node memory with two submemories each of which hosts a part of the traffic going through the node. For example, one submemory may take care of the ingoing traffic and another of the outgoing traffic. (In this way one can avoid deadlock situations.) To support such types of memory, the classes *DoubleBuffer* and *DoubleFIFO*, *DoublePriorityQueue* are provided.

Often used topologies are accumulated in the "Nets" part of the CSL PARTS KIT.

Let us consider two sets of nodes which we will refer to as *input* nodes A and *output* nodes B. The input nodes will represent the from-nodes for net links and the output nodes will represent the to-nodes for net links. These sets may intersect or even be identical. In the last case, one set A will represent both sides of the transferring activities. Let also N be the number of input nodes and M be the number of output nodes.

It is convenient to define the topology of connections defined by a simple net using auxiliary *connectivity functions*. The connectivity functions are predicates which are valid for some subsets of integer pair. The first element of each pair is in the range $[0, N]$; the second element is in the range $[0, M]$. There is a link between the i-th and j-th nodes in a connection defined by some connectivity function if and only if the value the function is **true** for the pair (i,j).

For example, the function *alwaysconnected* is the predicate which has the value **true** for any pair of integers which is in the range. The function *parallelconnected* is the predicate which has the value **true** only for pairs of the type (i,i).

Suppose we want to connect the node sets A and B by links that lead from every node of A to every node of B. This type of connection is represented by the *OneWayMultiBus* simple net. It is formed of the set of links that connect each input node with each output node (a bipartite graph constructed with the help of the connectivity function *alwaysconnected*). All links have the same basic delay. If the the node sets A and B coincide, the net *MultiBus* is derived.

The number of servers with which we supply the nets *OneWayMultiBus* or *MultiBus* will define the restrictions on how many items are allowed to be transferred concurrently between these nodes. If the number of servers is equal to the number of links (this is the default number of servers), there is no restriction on parallel traffic among the nodes. If, however, we supply the net with only one server, only one item transfer at a time may occur in the net.

The latter case is represented by the simple net *Bus* which is derived from the net *MultiBus* simply by fixing the number of servers equal to 1. Thus the *Bus* net is a Communicating Structures abstraction of real bus-type nets. This abstraction captures the two basic properties of simple busses: (1) any input point is connected to any output point, and (2) only one item at a time may be transmitted.

The simple net *RightLoop* connects nodes in a loop by unidirectional links in such a way that the i-th node is connected to the $((i-1) \bmod N)$-th node where N is the total number of nodes. This type of connection is built with the help of the connectivity function *rightcyclicshift*.

The number of servers in the net defines the number of transfers that may occur simultaneously in the *RightLoop*. In the similar way, the simple net *LeftLoop* is constructed.

Restricting the number of servers to 1, the loop nets may be transformed into ring nets in which only one transfer may occur simultaneously.

The combination of the *RightLoop* and *LeftLoop* makes the *Loop* which connects any node with its both left and right neighbors.

Quite complicated restrictions imposed on the item traffic in systems can be expressed through hierarchy of nets and the default scheme of reserving links and subnets which contains the links.

6.1 A SoS CASE STUDY

A project of a world-wide distributed system which represents a decentralized computing environment for a global transportation company has been analyzed using the Communicating Structures methodology and CSL.

The system does

- packages tracing and monitoring (hundreds of millions of transactions per day);
- statistics, billing data processing, and decision support ;
- customer services (including WWW) ;
- common enterprise business applications.

Three-level hierarchical network of three-tiered computing centers (see Figure 3):

- global Data Centers (DC), several of them;
- regional Processing Centers (PC), tens of them;
- local Operations Centers (OC), tens of thousands of them;
- mission critical, “almost real-time” computing environment;
- cost effectiveness is the dominant requirement;
- a client/server computing model;
- a publish/subscribe data distribution model.

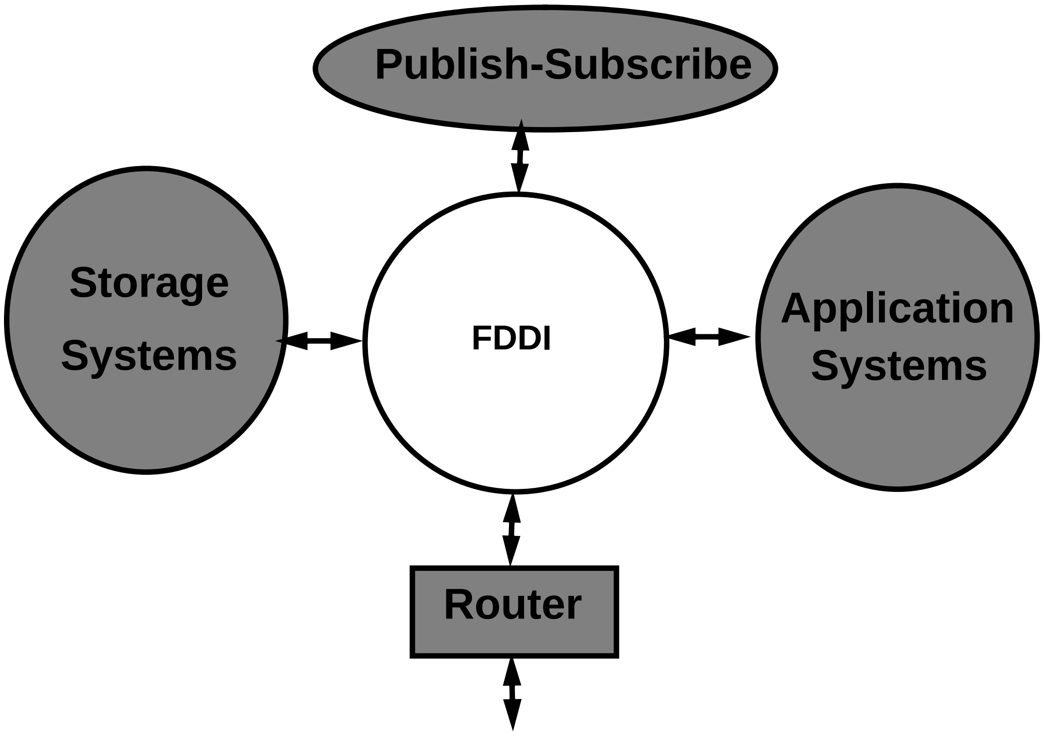


Figure 3: Three-Tiered computing Center

So, this is a typical SoS and it is specified in the Communicating Structures terms in a natural way (see Figure 4). The computing centers are CSL nodes of different levels of hierarchy, the communication lines are links, the functionality of computing centers is modeled by the functionality of CSL nodes.

Information is distributed and exchanged among centers according to the *Publish-Subscribe* paradigm: applications publish data for potential use by other applications and are subscribers for data published by others.

Point-to-Point dispatch and *data brokerage* are two alternative models for the implementation of the Publish-Subscribe methodology (see Figure 5).

The first case represents a spider web of point-to-point interfaces which are "hard-coded" with specific languages, platforms, application and data formats. Applications maintain unique relationships between themselves.

In the second case, point-to-point links are replaced by the publication of common messages usually in a standard format which are sent to *Data Brokers*. The latters have the tables of subscribers for each type of the messages and forward the messages to subscribers.

The task was to evaluate the project with the emphasis on comparison of the two Publish-Subscribe models.

The constructed CSL model presents the project as a Communicating Structure which contains those and only those system features that influence the message traffic and are important

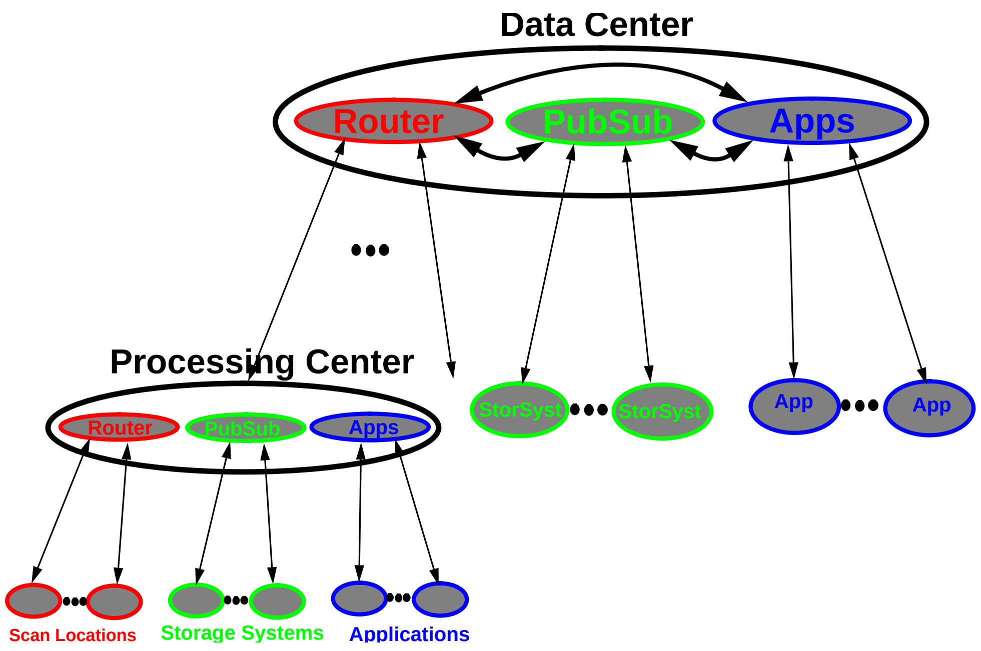


Figure 4: Three Level Network of Processing Center

for the system requirements were satisfied. The model helped to identify bottlenecks and the system sensitivity to changing parameters (the number of processing centers, bandwidth in local and global networks, message packaging principles, etc.).

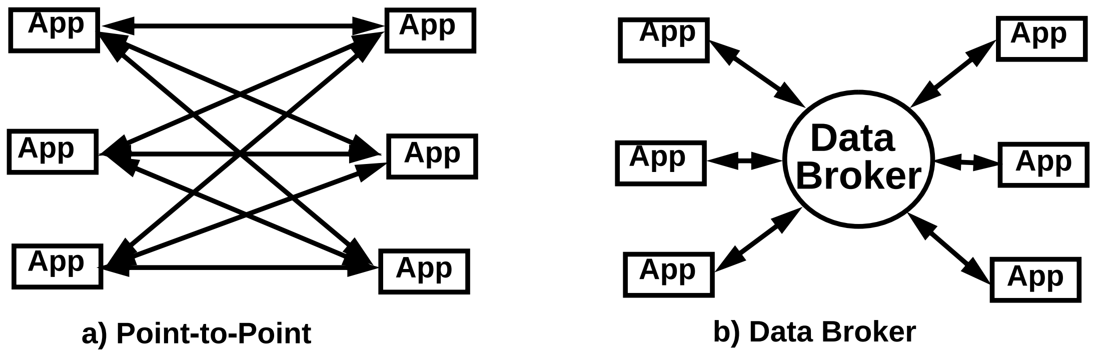
The sensitivity analysis included:

- system response on bursty workload;
- system scalability;
- Data Broker overhead;
- message packaging strategy.

Utilization analysis:

- utilization of Data Brokers as a function of workload;
- networks utilization.

The main result of the project validation was the reduction of the proposed three-level system architecture to two-level architecture. The CSL analysis of the traffic in the system has shown that if the functions of the second level are redistributed between the top level of Data Centers and the low level of the Operation Centers then the global traffic becomes less congested, response time is improving, basic requirements to the system are satisfied and the overall cost is, of course, dramatically reduced (see Figure 4).



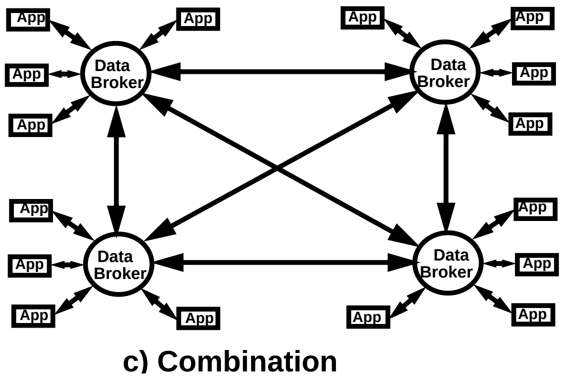


Figure 5: Publish-Subscribe Models

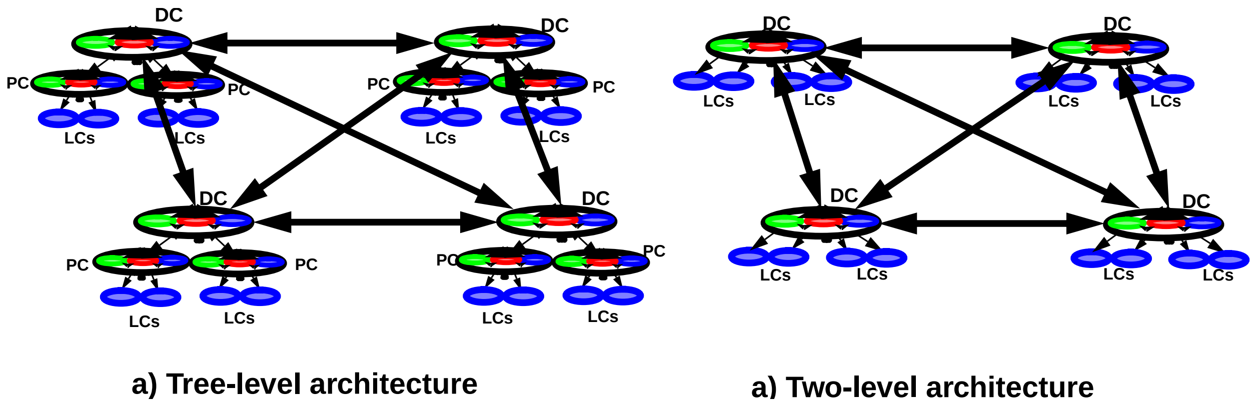


Figure 6: Two-level versus three-level System Architectures

6.2 VISUALIZATION

The huge analysis and design space of SoS requires a special instrumentation to deal with data collection, workload and test data generation, results collection and analysis, etc. Especially useful is visualization of the model behavior, visual analysis of results, visual support of the model debugging and validation. Figure 7 shows a "hot spots" picture of a SoS model with .

Another visualization tool SIMON allows to see and verify the communication between concurrent processes (see Figure 7).

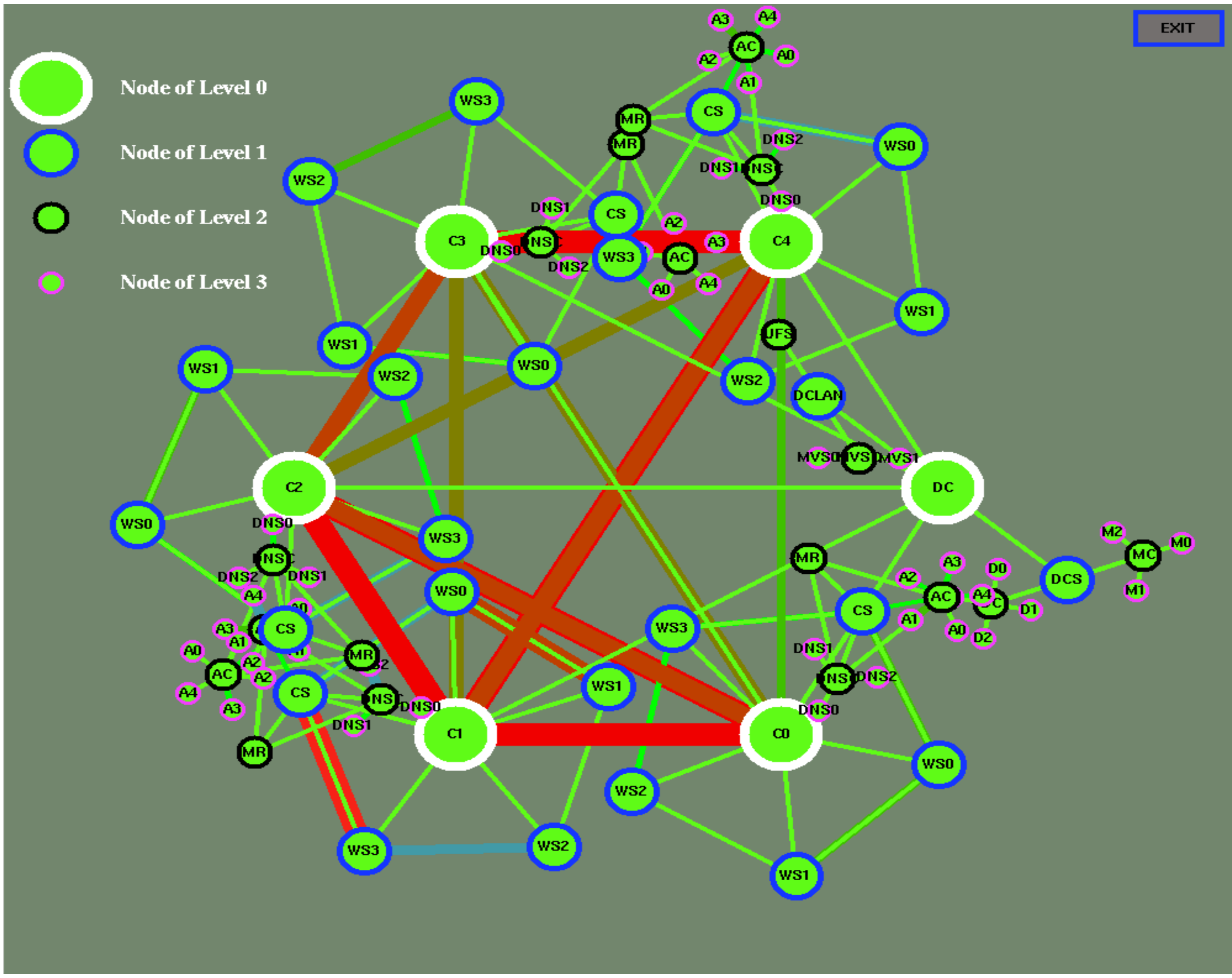


Figure 7: System Hot Spots

6.3 CONCLUSION

Communicating Structures reduces the complexity of the SoS modeling and analysis, in particular:

- to simplify construction of SoS models of different levels of detail by using abstraction/refinement mechanisms;
- to describe parallel processes and their interaction in an object-oriented way speeding-up the model debugging and increasing the trustworthiness of models;
- to speed up simulation of a large number of concurrent processes;
- to accumulate and reuse prefabricated general-purpose and domain specific modules ("parts kit");

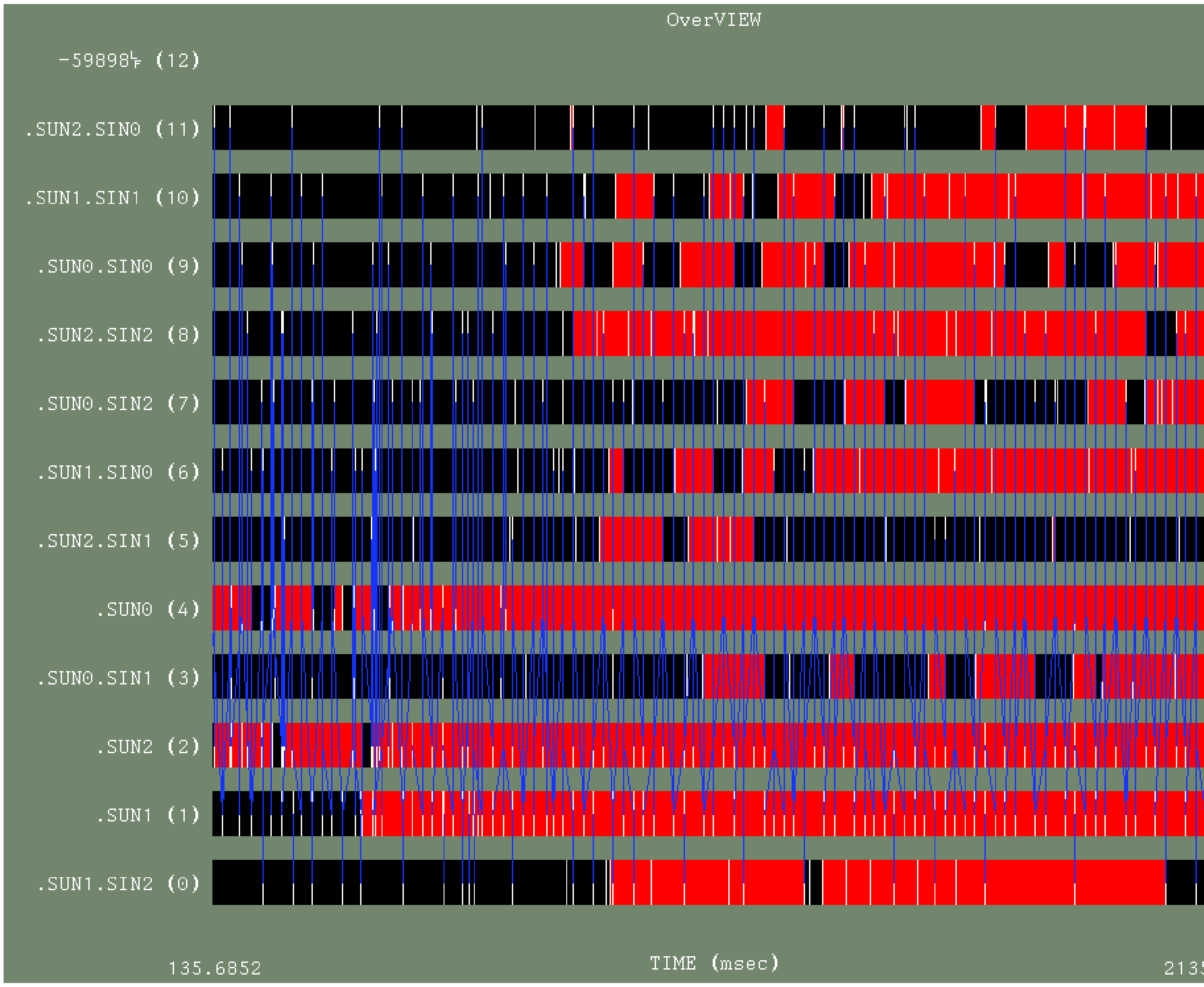


Figure 8: Communication between Processes

- to generate and analys a larger number of the system configurations and behaviors;
- to provide friendly programming and modeling infrastructure (data generation, collection, analysis, visualization, etc.).

The current version of CSL has been mostly used for the simulation of SoS because analytical modeling methods were unapplicable to the SoS under consideration. However, the analytical methods, if they work for particular types of SoS, may complement the simulation using the queueing analysis classes associated with the CSL nodes and a network of queues derived from the topology of a model.

The most interesting extension of CSL that we are going to build is related to the intelligent browsing of the huge solution spaces for SoS. The goal is not to miss good architectural solutions. This is a sort of system synthesis which relies on combining simulation, analytical methods, and formal methods.

6.4 Acknoledgement

Lucy Cherkasova and Tom Rokicki coauthored and help to shape the idea of Communicating Structures. Tom help to implement CSL by contributing his elegant code to the CSL base. Lucy was the first user of the first version of CSL and feedback from her modeling efforts drove the further CSL progress. The visualization tool SIMON was developed by Sekhar Sarukkai.

The author would like also to thank Denny Georg and Rajiv Gupta for sharing ideas, encouraging discussions, and support.

7 References

[Fow97] UML Distilled. Addison-Wesley, 1997, 180 pp.

[Har97] Harel D., Gery E. Executablre Object Modeling with Statecharts. In *Computer* , vol. 30, No. 7, July 1997, p. 31-42.

[Sch95] Schwetman, H. Object-oriented simulation modeling with C++/CSIM17. In *Proceedings of the 1995 Winter Simulation Conference*, Washington, D.C.. ed. C. Alexopoulos, K. Kang, W. Lilegdon, D. Goldsman, 1995, p. 529 - 533, Washington, D.C.