CSL:
COMMUNICATING STRUCTURES LIBRARY FOR SYSTEM MODELING AND ANALYSIS

Vadim Kotov, Tomas Rokicki, Ludmila Cherkasova
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
e-mail: {kotov,rokicki,cherkasova}@hpl.hp.com

ABSTRACT

CSL is a C++-based library and a core environment for the modeling and analysis of **Systems of Systems (SoS)**, *large-scale distributed systems with components that are complex systems themselves.*

To be able to model SoS, the whole diversity of their components and structures is represented as **Communicating Structures**, *uniform and systematic compositions of a small number of basic concepts that describe* **communication**, **data traffic**, *and* **data placement** *inside SoS. SoS are represented as hierarchical networks of* **nodes**. *Each node has* **memory** *that may contain* **items**. **Nets** *are sets of* **links** *that connect the nodes. The items are generated at some nodes and move from node to node along links, with some delay. The item generation, transformation, and movement is controlled by the parallel processes running on the nodes. Items, nodes, memories, and nets may be elementary or aggregated objects.*

CSL uses modified structures of C++/CSIM, a process-oriented discrete-event simulation package [Sch95, Mes96], to generate and coordinate concurrent processes as well as GUI and visualization tools for the model construction, providing input parameters, and analysis of the modeling results.

Key Words: concurrent distributed systems, hierarchical systems, Communicating Structures, system modeling, simulation.

Contents

1 INTRODUCTION **4**

2 THE CSL STRUCTURE **8**

3 THE CSL BASE **9**

3.1 **Data Structures** 9

3.2 **C++/CSIM Constructs** 9

3.3 **Names, Parameters, Attribures, Trees** 11

4 THE CSL OBJECTS **13**

4.1 **CSL Object** 13

4.2 **CSL Item** 14

4.3 **CSL Memory** 15

4.4 **CSL Net** 17

4.5 **CSL Node** 19

4.6 **Registry and Registers** 20

5 THE CSL BASICS **21**

5.1 **Process** 21

5.2 **Simulation Node** 22

5.3 **Queue Node** 27

5.4 **Queue Net** 27

6 CSL PARTS KIT **28**

6.1 **Memories** 28

6.2 **Nets** 29

6.3 **Nodes** 31

7 SIMPLE MODEL EXAMPLES **31**

7.1 **The First Example** 31

7.2 **The Second Example** 37

7.3 **The Third Example** 39

8 GRAPHICAL INTERFACE FOR CSL **40**

8.1 **Programs and Files** 40

8.2 **Start** 40

8.3 **Subnodes Information** 41

8.4 **Canvas** 42

8.5 **Adding a New Node** 42

8.6 **Editing Node Information** 42

8.7 **Add a New Link** 45

8.8 **Creating a New Node (Net, Memory) Type** 47

8.9 **Setting and Running the Model** 49

8.10 **SIMON** 50

8.11 **Hotspots** 51

8.12 **Multiruns** 53

9 CONCLUSION **53**

10 CONTRIBUTIONS AND ACKNOWLEDGMENTS **55**

11 References **55**

1 INTRODUCTION

Standardization and unification of processors, memories, storages, and I/O devices and the increasing use of commodity off-the-shelf components has raised the level of computer design. The computers themselves have become components of various types of clusters, systems, and networks. Networks of computers are combined into multilevel distributed systems of diverse topologies. The integration interfaces and protocols are exploding in number and complexity.

Individual software components and software systems are also integrated into large distributed software systems with specific *software architectures*. Specification, verification, and analysis of software architecture is becoming more and more important in the industrial software environment.

A growing number of today's applications have become distributed. Different stand-alone applications, both proprietary and commercial, have been integrated into large heterogeneous application systems with specific *application architectures*.

Finally, hardware, netware, software, middleware, and application architectures have combined into *integrated system architectures* and, as a result, the variety of feasible system architectures is rapidly increasing. The complexity of these systems is increasing significantly because they contain large numbers of subsystems and components built by diverse vendors and often represent "black boxes" for the system integrators. We call such systems *Systems of Systems (SoS)*.

Some examples of today's SoS are:

- clusters of multiprocessor computers;
- multi-tiered processing centers;
- enterprise intranets supporting common business processes;
- distributed mission-critical applications;
- distributed control systems;
- distributed design/manufacturing systems;
- the World-Wide Web;

- and combinations of the above.

Typical SoS have to satisfy many strict requirements (many of which are conflicting), among which are

- cost effectiveness (SoS are often unique and expensive);
- responsiveness;
- throughput;
- scalability and flexibility;
- availability;
- maintainability;
- reliability, fault tolerance, and recoverability;
- data and application integrity;
- security.

Concurrency and non-trivial distributed communication are major challenges to system designers. More and more objective and elaborate methods and tools are required for the evaluation of system architectures. For all this, the competitive market does not allow much time to experiment with prototypes. There is a clear need for design methods, techniques, and tools that allow designers to construct and analyze quickly and reliably various hypotheses and evaluate them against both a wide spectrum of desired system properties and against various applications.

However, SoS represent a challenge for the modeling and analysis, as their solution space is huge and complex. Modeling methods and tools typically used in object-oriented analysis and design (for example, UML [Fow97] and 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 the satisfactory solutions among a sea of solutions. SoS are also too complex and diverse to fit into the Procrustean framework of a formal semantics.

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

- identify the few main modeling objectives and those system features and parameters that are relevant to the objectives;
- use the minimal number of concepts that are common to most of the systems and with which the modeling objectives can be adequately specified;
- develop convenient modeling data handlers, statistics, and analysis libraries, as well as visualization-animation tools.

Here we propose to view computer and information systems as *communicating systems* in which the main activities are related to the coordination of *data traffic* and *data placement*. The modeling objectives are:

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

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

This report presents the *Communicating Structures Library (CSL)*, a core environment for the modeling and analysis of large communicating systems. CSL is intended to meet the need for a global system modeling methodology that handles both complex behavior and high performance. The key issue is to find a way to model systems at different levels of abstraction in order to support both top-down and bottom-up design.

In its most general form, *Communicating Structure* is a *hierarchical* and *concurrent* structure that represents the system components and the communications between them in a uniform and systematic way.

The system components are represented simply as *nodes*. Each node has *memory* that may contain *items*. *Nets* are sets of *links* that connect the nodes. The items are generated at some

nodes and move from node to node along links, with some delay. Items may be modified by 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, an item may represent simple data such as a byte, a word, a frame, a packet, as well as a complex message and large chunks of data. The nodes may represent relatively small units such as registers, caches, and functional units, or larger computer components such as processors and memories, or subsystems such as multiprocessors, large storage units, computer clusters, subnets, etc. Nets may represent simple point-to-point links as well as busses, crossbars, interconnects, cascaded switches, LANS, communication lines, and other data transfer facilities.

Thus, the items, nodes, memories and nets are *CSL objects* that are either simple or aggregated. The CSL objects may be assigned different attributes (numbers, variables, functions, and processes) that:

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

A model in CSL is a hierarchy of nodes with one top node that has no parent. Nodes are assigned processes that are invoked to generate items, receive/send items from/to other nodes, and to transform the items, if necessary.

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

In the case of simulation, CSL accumulates generic or parameterized CSL objects, functions, and processes that may be quickly assembled into a particular simulation model and tuned to a specific case study as defined by user-entered input parameters (see 3.3).

CSL is a system analysis package, not a universal modeling language that makes emphasis on the precise specification of system structure or behavior. It supports a programming, not a pictorial, style of modeling (though it has a graphical interface and visualization support for the analysis of the modeling results). The CSL hierarchy is based on C++ classes; CSL concurrency uses the main structures of C++/CSIM, a process-oriented discrete-event simulation package [Sch95, Mes96]. CSL is transparent in that sense that the user can use not only the CSL constructs but also everything below them: C++/CSIM, C++, and plain C.

2 THE CSL STRUCTURE

There are several levels of the modeling primitives and constructs in the CSL environment.

The lowest level, CSL BASE, is formed by data structures, classes, and algorithms that allow us to comfortably define, construct, and modify the components of Communicating Structures.

The second level, CSL OBJECTS, contains the basic components of Communicating Structures and instruments to assemble them.

The third level, CSL BASICS, provides means to build simulation or analytical models out of components of the previous level.

The next level contains the generic CSL PARTS KIT, which serves to customize and refine Communicating Structures to make them adequate for the specification and evaluation of particular types of systems. It also accumulates mathematical and statistics libraries.

Finally, CSL DOMAIN LIBRARIES are built by users and provide the means to make the construction and analysis of models in different domain-specific areas fast and reliable.

The CSL GUI provides the graphical and visualization means for the easy construction, running and analysis of the CSL models, particularly during prototyping and debugging stages.

3 THE CSL BASE

The CSL BASE is a base sublibrary of appropriate templates and auxiliary classes underlying CSL in order to

- form high-level abstract data structures (array, list, tree, etc.);
- provide the control and synchronization of parallel processes; and
- organize the connections the between model's structural attributes and its external input and output parameters.

3.1 Data Structures

For example, the template class *Array*<T> declares an array that can expand to accommodate new elements and can be pruned by deleting some positions (together with the elements in those positions). The positions and the elements can be addressed both by an integer index and by using an array *iterator* as in C++ STL (Standard Template Library [Muss96]).

Other examples of such structures are the template classes *List*<T>, *PriorityQueue*<T> and *AssocMatrix*<Key, Val>. In general, the usage of the abstract structures and algorithms notations and semantics in CSL follows the STL line. However, STL is not included in CSL as a supporting library, as most of the STL potential is not used by CSL.

3.2 C++/CSIM Constructs

A set of modified CSIM structures is introduced to generate and coordinate concurrent processes. A C++/CSIM *process* is a C++ procedure that executes a *create* statement. This statement invokes a new thread that proceeds concurrently with the process that has invoked it. So, a typical process looks like:

```
process(arg1, ..., argn){
create(''PROCESS'');
.....................;
}
```

where *process* is the process identifier and *PROCESS* is the process's external name.

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 simulation time to pass), and *waiting* (for permission to continue after it has been interrupted).

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

The CSL class *Mailbox* is derived from the CSIM *mailbox* and all CSIM mailbox operations and functions are valid in CSL. Mailboxes are used for interprocess communication. A process can *send* a message to a mailbox and *receive* a message from a mailbox. When 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 *send_with_delay* that makes it possible to send a message to a mailbox with some time delay.

The basic mailbox statements are:

```
MAILBOX.send(int message);
MAILBOX.send_with_delay(int message, TIME delay) ;
MAILBOX.receive(int *message);
```

The semantics of the CSL class *Facility_ms* is similar to that of CSIM *facility_ms* but it is implemented via a *Mailbox*. This was done in order to avoid the CSIM restrictions on the *release* operation. Facility_ms models a resource. It contains a single queue and several servers. Only one process at a time can hold a server after it executed 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 process waiting in the facility queue ahead of this process.

So, the basic facility statements are:

```
FACILITY.reserve();
FACILITY.release();
FACILITY.use(TIME hold_time);
```

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 state) and another is for processes that have executed the *queue* statement (and also are in the waiting state). When the event

occurs, by executing the *set* statement, all waiting processes and only one of the queued processes are allowed to proceed. The statement *clear* resets the event to the not-occurred state.

The basic event statements are:

```
EVENT.wait();
EVENT.queue();
EVENT.set();
EVENT.clear();
```

3.3 Names, Parameters, Attribures, Trees

The class *CSL_Name* provides a convenient way to construct compound names, which are useful for naming hierarchical objects. Such a compound or "full" name is, in fact, a "multistring", a string that consists of substrings delimited by a special delimiter (the default is "."). Each substring represents the name of a predecessor of the object in the hierarchy to which it belongs. For example, a subobject of an object may be given a "full name" which contain the name of this object as a prefix and the subobject's name as its "first name". An example of a name is *net1.computer5.CPU2.cache*.

To easily parameterize the CSL models, the classes *Parameter* and *Attribute* are introduced. The external parameter makes it possible to add to an input file a named input string and convert it into a value of a simple type (integer, double, string), into a list of values, into a list of lists, etc. For example, the standard CSL parameter *EndTime*, which sets the time when a simulation must stop, is defined as follows:

```
Parameter EndTime("EndTime", "Stops simulation by time", "1000000") ;
```

The default time is 1000000 time units. A user can change it by adding the following line into an input file:

```
EndTime = 500000
```

The class *Attribute* provides the connection between structural attributes and external parameters through regular expression-matching between the structural name and the external parameter. Each *Attribute* has a name associated with it. It may be, for example, the name of an object or the name of another element related to the object. Each *Attribute* also has a match,

which is a pointer to a *Parameter* that is the most specific matching parameter. For instance, **.InterArrivalTime* is a potential match that is not very specific. *Node1.*.InterarrivalTime* is a more specific match. After all of the model parameters are defined and all of the attributes are constructed, the matches are found and can then be used.

The classes *Observe* and *Utilize* provide external visibility for statistical and other computed values. Each observable has a name and a value, which keep track of time-valued observables. These are values such as utilization for which the average value over time, rather than just the average of a number of observations, is critical.

As all CSL objects may have a hierarchical structure, the class *Tree* is introduced. It represents tree-like hierarchies with the class members and functions that 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.

A CSL object may be *simple*, in which case it has no internal structure. It may also be a *structured* object, i.e., a tree that consists of other objects, which are the object's direct subobjects, or *children*. (The containing object is, naturally, the *father* for its children.) The children may be, in turn, hierarchical objects. A structured object is the *top* object if it has no father.

CSL objects may have external hierarchical names. Such a name consists of the "first name", which is preceded by the name of the object's father and the names of all its other ancestors. The object also has its "local address", which is the object's position number in the array of its siblings, i.e all the children of its father. So, the position of an object in a hierarchy of objects may be indicated both by its compound name and by a list of integers that are "local addresses" of all the object's ancestors plus the position number of the object as the last element of the list.

Here is a simplified structure of the class *Tree* (without member functions and some auxiliary class members):

```
class Tree{
public:
Tree *father;            // the tree supertree (father)
Tree *top;               // the top tree in the tree's hierarchy
int number;              // the ordinal number of the object as a child
CSL_Name name;           //  the tree external name
```

```
Array<Tree*> *children;   // pointer to the direct subtrees (children)

 Tree( Tree *_father, char *_name );

};
```

4 THE CSL OBJECTS

The 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. Nets connect the nodes into a communicating structure in which the items travel from source nodes to destination nodes. These elements are derived from the common CSL class *Object*.

As all CSL objects may have a hierarchical structure, the class *Object* is derived from the class *Tree*, which represents tree-like hierarchies with the class members and functions that help handle the hierarchy, for example, selecting subtrees and sets of subtrees, checking some properties of trees, applying functions to subobjects, etc.

The class *Object* is also derived from the class *Facility_ms* and has an *attribute* and a *utilize* associated with it. This makes the object a resource for which concurrent processes may compete and provides "hooks" for supplying input data to objects and collecting statistical information.

4.1 CSL Object

The class *Object* is derived from the classes *Tree* and *Facility_ms*. The first derivation makes possible the construction of structured hierarchical objects. The second derivation makes the object a resource for competing concurrent processes. The number of servers in the object is set during the object construction. The object member functions make it possible to reserve and to release subobjects of the object. In particular, the reservation of a chain of nested hierarchical objects may be used in many different ways for organizing special accesses to the internal hierarchy of structured objects.

An auxiliary class *Context* combines into one argument three typical arguments for object constructors: a pointer to the object's father, the object (last) name, as well as a *Parameter*

that may be used to provide constructors with external information about the object structure and parameters.

```
class Object: public Tree, public Facility_ms{
public:
 Attribute attr;
 Utilize use;

Object( Context C );

}
```

An object also has an *Attribute* and a *Utilize* associated with it. The object attribute *attr* will match any structural parameter name that is a string ending with the suffix ".AT". The value of such a parameter is a list of pairs. The first element of such a pair is a "keyword" identifying the value that is the second element of the pair. For example, the list *(SERV 1)* provides one server for the object as a facility. The Utilize *use* keeps track of the time during which the object was used for some specific activities.

4.2 CSL Item

An *Item* represents an entity that migrates in a communicating system. In CSL, an *Item* is, in fact, a *Tree*, not an *Object*, so it has no facility, attribute or utilize. This is because items are "dynamic" entities that are born, travel and perish. They cannot be reserved and it is meaningless to provide values to them via attributes and matching.

In addition to the common *Tree* members, the class *Item* has special members. The item has a unique (for its life cycle) *id*, a numerical identifier created at the time when the item is generated and released at the time when the item ceases to exist (to be used by newly born items). Items are generated in nodes of Communicating Structures that are represented in CSL by the class *Node* described below. Each item carries with it a pointer *send* to the sender-node that is its birthplace and a pointer *last* to the last-visited node. It also carries a destination path *dest*, which defines the item potential route leading to its destination (maybe just to some intermediate destination). The path represents a sequence of pointers to nodes which the item intends to visit. Not all nodes that the item will actually pass need to be listed in the path. The routes between subsequent points of the path are optional and subject to some chosen routing. The original destination path may be modified on the way or, after the

item reaches its original destination node, it may be assigned a new destination. The item has an integer *length* that is typically treated as a message (packet, etc.) length in some user's units (bits, bytes, words, etc.). The item length is used in CSL BASICS only in a default virtual function that computes the speed of transfer of items along links. However, any other user interpretation of *length* is possible.

If an item has subitems, then these subitems may be spawned into a set of items that are issued when the item has reached its destination and is ready to disappear. This topic will be discussed in more detail later on when the transfer of items in Communicating Structures is described.

Each item may be assigned a special *ItemTag* that represents the item type and serves to distinguish between different sorts of items. Here is a simplified declaration for the class *Item*.

```
class Item: public Tree{
public:
int id;                  // the item identifier
Node *send;              // the item sender
Path dest;               // the item destination path
Node *last ;             // the last visited node
int length ;             // the item length
const ItemTag *tag ;  // the item type tag

Item(Node* send_node, const ItemTag &_tag, int _length, char *_name, Item *_father);

};
```

The class *Item* member functions serve to modify items and to handle time and space attributes, for example, to mark time stamps, to change the item destination, the item length, or to change its tag.

4.3 CSL Memory

A CSL *Memory* is an *Object* that stores *Items*. In the general case, a 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 simple memories *Buffers*, which are arrays of *locations* holding pointers to stored items.

The class *Memory* has members (the memory size, the current number of stored items, the number of items waiting to be stored, and the last-visited submemories or locations) that help to monitor and control the availability of items and storage space in the memory. As a CSL *Object*, the memory can be a resource that allows us to prevent noncontrolled nondeterministic concurrent access to it.

An *Address* is a pair that represents the address of an item in a *Memory*. Its first element is a pointer to the memory or its submemory. Its second element is an integer. If the pointer points at a structured memory, then the integer identifies its submemory. (If the integer is negative, then this is interpreted as an arbitrary submemory.) If the pointer points at a *Buffer*, then the integer identifies a location in this *Buffer*. (If the integer is negative, then it is interpreted as an arbitrary location.)

These are simplified *Memory* and *Buffer* class declarations:

```
typedef Pair<Memory*,int>  Address ;

class Memory:  public Object{
public:
int memsize;       // the (maximal) capacity of  the memory
int numitems;      // the  current number of items in the memory
int waiting;       //  the number of items waiting to be stored
int last_get;      // the number of the last-visited submemory or location
                   //to get an item
int last_put;      // the number of the last-visited submemory or location
                   //to put an item
event GET;         // the event that signals retrieval of an item
event PUT;         // the event that signals storing an item

Memory( Context C );

virtual Item* get(Address address);
                         // extracts an item stored at the address
virtual Address *put(Item *item, Address address );
                         // stores an item at the address

};

class Buffer: public Memory, public Array<Item*>{
```

```
 public:

Buffer( Context C ) ;

Item* get_item(int n);
                  // extracts an item from the {\em n}-th location
void put_item(Item *item, int n);
                  // stores the item into the  {\em n}-th location

};
```

The virtual access functions *get* and *put* first find and reserve requested or available locations; then they retrieve or store an item. For this purpose, these functions use the virtual functions

```
 virtual Address OKget() ;
 virtual Address OKput() ;
```

The functions serve to define specific patterns of the sequential access to a submemory (including a location) when its exact address is not specified. The functions use information about the ordinal number of the last-visited submemory or location (*last_get* or *last_put*). Then they calculate a new position taking into account the previous access position. One can put an item in a location only if the location is empty, that is, it does not contain an item. One can get an item from a location only if the location contains an item. That enables the creation of special memory access strategies, for example, those used in FIFO or pushdown (stack) memories. This possibility will be exploited in the the CSL PARTS KIT section.

4.4 CSL Net

The CSL *Net* is an *Object* that makes connections between the nodes. In the general case, the net inherits a multilevel hierarchy from the class *Object*.

```
class Net: public Object{
public:

Net ( Array<Net*> *subnets, Context C ) ;

};
```

The "top" net, that is, a net with no father-net, is a part of a *Node* definition for which it

provides communication links among the node's subnodes. At the bottom of this hierarchy are *Links*, "elementary" nets, each of which connects just a pair of nodes.

The default number of servers for a net is chosen to be the sum of the number of servers of the net's children. In case of a "simple" net (its subnets are links), a net is just an array (set) of links defining some connection pattern for a set of nodes. The default number of servers for a simple net is the sum of the number of servers of the net's links. That means that we assume that the net is a priori just a set of links or a set of sets of links and does not impose any additional restrictions on the traffic of items along its links. If a user decreases the number of servers for the net, some restrictions may occur, as the reservation of a link requires also reservations of all nets containing the link.

Each link delivers items from a *from-node* to a *to-node* with delay which is a function of the link *bandwidth* and the transferred item length (or some other item 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 that define the maximal number of transfers that may occur along the link simultaneously.

This is a simplified class declaration for the CSL *Link*.

```
class Link: public Net{
 public:
 Node *from;            // from-node
 Node *to;              // to-node
 double bandwidth;      // the link bandwidth (in some user's units)

 Link(Node *_from, Node *_to, double _bandwidth, int numservers);

};
```

The net hierarchy may be treated as a hierarchy of sets of links and their subsets. This makes possible the use of structured nets to model switches and interconnects at an abstract level, as the logic of switching is conveniently expressed in set theory terms. Some examples of this approach will be presented in the section CSL PARTS KIT.

4.5 CSL Node

The main building block of Communicating Structures and CSL models is the *node*.

The class *Node* serves to define the static structure (topology) of CSL models. Such a node is an object that represents a hierarchical graph; its subobjects (subnodes) are its vertices and its directed arcs are the links connecting its subnodes. Both the node and its subnodes may contain memory. The class *Node* also contains *Memory* as its member. The whole CSL model itself is a top-level CSL node.

The node's internal links (if it is an aggregated node) are clustered into a *Net*. The member *net* contains all the links that connect the node with its subnodes (both ways) and the node's subnodes among themselves. The net fully defines the node structure. To be able to analyze and modify the structure with ease, two useful lists of links are added to the class *Node*:

- the list *inlinks* contains the links that enter the node;

- the list *outlinks* contains the links that leave the node.

The class *Node*'s member functions serve to construct the node communication structure, identify specific groups of links, find paths in the internal structure of the node, and modify this structure.

```
class Node: public Object{
public:
Memory *memory;             // the node's memory
Net *net;                   // the node's net
List<Link*> *inlinks;       // the list of in-links
List<Link*> *outlinks;      // the list of out-links

Node(int memory_size, char *memory_type, char *subnode_type,
int num_subnodes, char *subnodes_name,
char *net_type, double net_bandwidth, Context C);

Node( Memory *_memory, Array<Node*> *subnodes, Context C );

Node(Context C);

};
```

When given two subnodes of a node, one of the member functions, namely *path(Node *from, Node *to)* , finds the shortest local path in the node's internal structure connecting these two subnodes.

4.6 Registry and Registers

Most of the CSL basic objects such as *Memory*, *Net*, *Node*, as well as new types of objects that a user may introduce as derivative classes of the basic objects, may be constructed in two ways. They may be built with the help of constructors that explicitly define and initialize all the subobjects and class members in the constructor body's code. A more flexible and convenient way is provided by a constructor with an argument that is a CSL *Parameter* whose value can be defined and modified with a model input file. However, if we want to inform the object constructor about the types of its subobjects, we need mechanisms that convert an input string specifying the type into an identifier of the object type that can be used in the constructor.

The template class *Registry* is a "metaconstructor" that constructs an object of a specific type using a string name for this type and *Context*, which contains a *Parameter* describing the structure, parameters, and properties of the constructed object.

The object type is a string that has been registered by *Register*. All registered classes should also have a constructor of the form

```
Class(Context) ;
```

For example, let the class *Memory* contain a constructor

```
Memory(Context C) ;
```

This class must have been registered by *Register* in the following way:

```
Register<Object, Memory> MemoryObject("Memory");
```

The string *"Memory"* identifies the registered name of the class *Memory*.

Now we can use the function:

```
Memory *make_memory(char *object_type, Context C) ;
```

which in the call *make_memory("Buffer", context) ;* builds a *Buffer* and returns a pointer to it. This function actually calls the *Registry* member function *buildOne* which selects an appropriate class constructor to build a memory of the indicated type.

Using object constructors with *Parameter*, a user can completely transfer the description of the structure of the whole model into the *ModelStructure* parameter in the input file.

5 THE CSL BASICS

The basic CSL objects form a conceptual CSL kernel that is augmented by classes that convert the kernel into a CSL model of specific type: simulation model, or queueing model, or (stochastic) Petri Net model, etc. These classes are currently collected in the CSL BASICS sublibrary.

5.1 Process

The class *Process* introduces main generic processes that are associated with the simulation node (see the next section 5.2).

The *startup* process makes the node active using the *generation_process*, which initializes item traffic from the node (if the node is allowed to generate as indicated by the Boolean tag *generator*), and starts the main node procedure *process* that generates the default or user-defined processes that transfer items to and from the node and allocate them in the node's memory.

Figure 1 shows the structure of the *startup* process. (In this and subsequent figures, a rectangle represents a function (procedure), a rounded rectangle represents a CSIM process, a rhombus is a condition, and a circle is loop condition.)

When a CSL model is initiated, each model's node with a non-zero tag *generator* starts its *generation_process*. This process is recursive: it may generate an item or it may initiate another next-level generation process. Several levels of generation are useful when there is a hierarchy of generated items: messages consisting of frames, sessions consisting of messages, etc. The value of the tag *generator* specifies the number of the generation levels.

The virtual function *generation* defines the generation procedure for each level of generation. The default function is "empty". The virtual *timing* function is used to define the interarrival

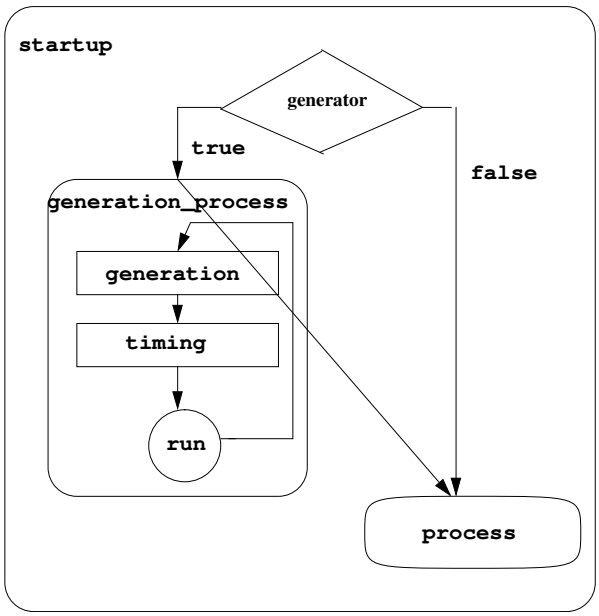


Figure 1: Startup Process

time for the generated items.

5.2 Simulation Node

The class *SimNode* represents a simulation node and is derived from the class *Node* and from the class *Process*, which is derived from the class *Mailbox*.

The node's processes generate and control item traffic and change and register the node's behavior. Most of the basic simulation node member functions and processes are virtual and may be customized for specific purposes by user. The default definitions of these functions provide "generic" item traffic that is generated in one subset of nodes and destined for another subset of nodes using the shortest path routing.

This is a simplified class declaration for the node with main member processes and functions:

```
class SimNode: public Node, public Process{
public:

SimNode(int memory_size, char *memory_type, char *subnode_type,
int num_subnodes, char *subnodes_name,
char *net_type, double net_bandwidth, Context C);
```

```
SimNode( Memory *_memory, Array<Node*> *subnodes, Context C );

SimNode(Context C);

virtual void initialize();                  // the initializer function
void generation_process(int level,          // the item generation process
                        Item *item);
void main_process(Address *location);      // the main node's process
virtual void transform(Item *item);        // the item transform function
virtual void terminal_point(Item *item);   // the termination function
virtual void transfer(Item *item);        // the item transfer function

};
```

The virtual function *generation* generates an item (with the help of the virtual functions *make_item* and *timing*) and then stores the item in the node memory with the help of the *store* function (see Figure 6).

In the default definition, the function *make_item* generates an item with the generating node as sender. A *destination path* is defined with the help of the virtual function *destination*. The default function *destination* produces a destination path that consists of one randomly selected node. An identifier is attached to the newborn item. Any customized definition of the function *make_item* may replace the default one.

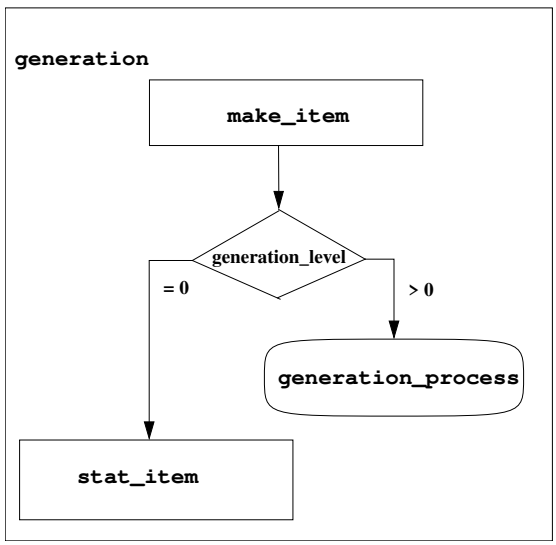


Figure 2: Generation

After the generated item is stored in the node's memory, the virtual function *push_process* sends a message to the node's mailbox in order to activate the node's *main_process*, which was initiated by the *Process* function *process*, and then waits for a message to arrive at the mailbox. This message contains a pointer to the address of the location in which the item was stored.

The *main_process* (see Figure 3) prescribes the node behavior. The *node_main* function executed in the process is virtual. It is defined as a superposition of several virtual functions discussed below. Hence, the *node_main* function may be either completely redefined in derived classes or it may be partially customized by changing the definitions of some of the constituent functions while leaving others unchanged.

The *node_main* argument is a memory location pointer that arrives as a message to the node mailbox. The function starts with the *begin_main* function, which reserves the node and its memory or submemories and returns a memory address from which an item will be extracted for processing in the node. This returned address may differ from the address received by the mailbox).

Then the *node_main* extracts the item and analyzes its destination path. If the destination path is empty, the process completes its work without actually doing anything. Otherwise, the head of the path is studied. If it is the pointer to this node, it is deleted from the path. If the item is simple and its remaining path is empty then the *terminal_point* function is initiated. The *terminal_point* terminates travel for the item in the communicating structure and collects statistics related to the item.

Otherwise, the *transform* function starts. This function may make some changes to the item. In particular, the function may change the item destination or make clones of the item for subsequent spawning into the communicating structure. The *transform* function will almost always be customized, as it actually defines the node's functionality. The default version of *transform* is an "empty action".

After the transformation, the main process either terminates, if the item is simple and received no new destination, or the *transfer* function is initiated (see Figure 4).

The transfer function organizes the transfer of the item or the item copies in the case of multicast to other nodes. In the default definition, it analyzes the item destination path and selects one of the possible transfer modes: monotransfer or multicast; synchronous or

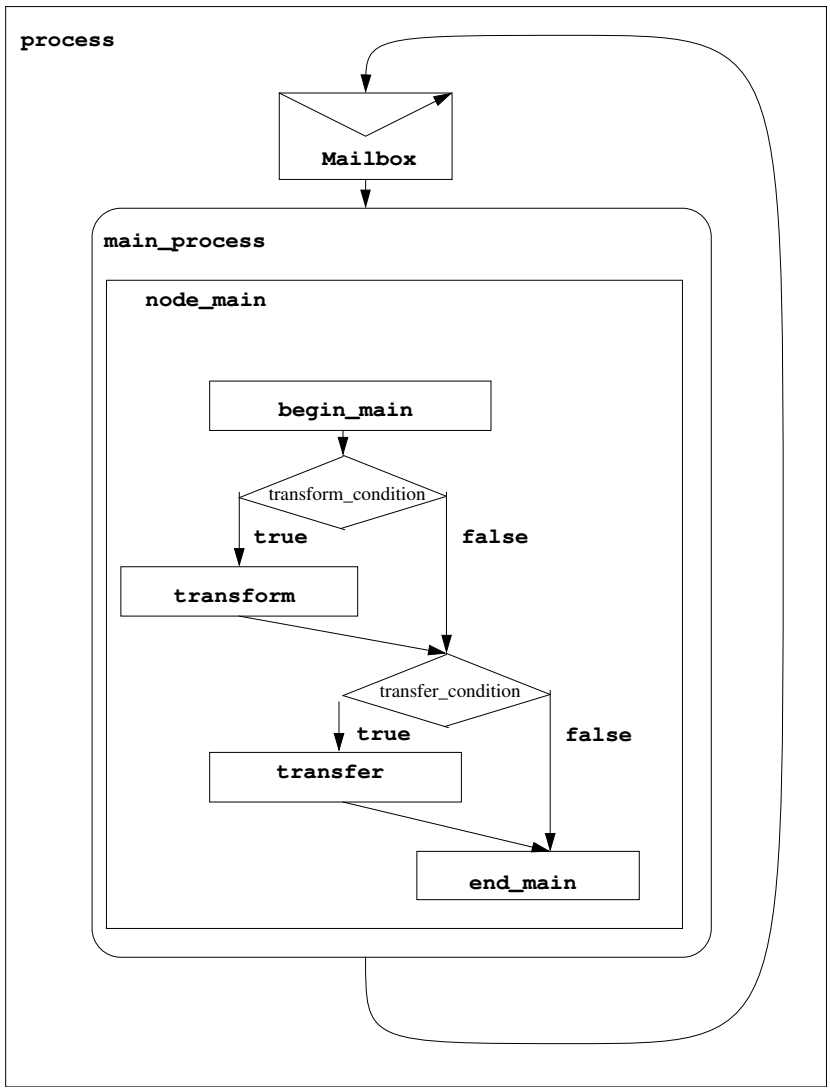


Figure 3: Main Node Process

asynchronous transfer.

In the monotransfer synchronous mode, the *transfer_function* is directly executed. The function starts with the *routing*. An adjacent node is chosen to which the transferred item is sent. The link connecting these nodes is reserved, the counter of items waiting to be stored in the accepting memory is incremented, and space for the item in this memory is reserved. If there is no space, the link is released and the process waits for the GET event to occur to repeat its transfer attempt. If the attempt is successful, the item is stored in the accepting memory after the transfer delay. The link is released, and the waiting counter is decremented. The transfer of the item is complete.

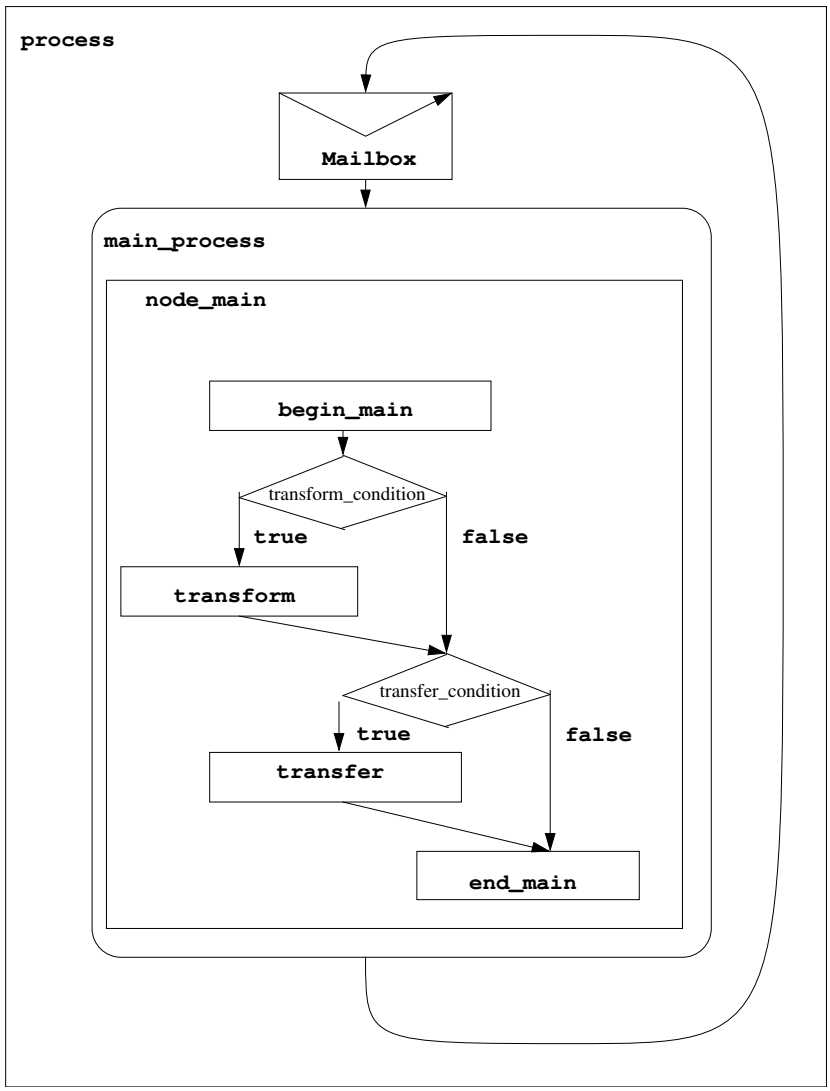


Figure 4: Transfer

The monotransfer asynchronous mode differs from the synchronous one in that while the main process proceeds the item is copied and a process concurrent with the transfer function is invoked to transfer the copy.

The multicast occurs if the item has an empty *destination path* and is a composite item, that is, it has subitems. These children are transferred and the item itself ceases to exist. For each child an individual transfer is organized. If the multicast is asynchronous, the main process spawns the individual transfer processes and proceeds. Otherwise, it waits until all spawned processes terminate. To determine when all the individual transfer processes have terminated, a counter of spawned processes *num_proc* and an event TRANS are introduced. The counter

is set to the number of transfers; each completed transfer decrements the counter; the event TRANS occurs when the counter becomes equal to 0.

The main process is ended by the virtual function *end_main* which, for example, may release the resources reserved by the *begin_main* function.

5.3 Queue Node

The current version of CSL allow us to construct simple queueing networks [Tan95]. A CSL node and a CSL net (more often, a link) can be presented as a "service center" or a "queue node" (we will use the second term, though the first term is often used when discussing queueing networks).

Such a queue node is modeled by a *QueueModel*, a class that constructs and executes a queueing model that is associated with the queue node, that is, with a node or a link. The type of the model is defined by *arrival_type* that defines the arrival time distribution, by the *service_type* that defines the service time distribution, and by the number of servers *num_servers*. We use Kendall's notation from queueing theory, where for instance "M" stands for the exponential distribution, and "G" means a general distribution. Thus, the with "MG10" type denotes a queueing model with the exponential distribution arrival time distribution, the general service time distribution, and 10 servers.

The input data for the queueing model are: an interarrival rate *rate* and a mean service time *service_time*. In the case of the general distribution, the squared coefficients of variation are required for both arrival time and service time distributions.

The model returns the the average waiting time (different approximations: heavy traffic, Allen-Cunneen, upper bound), the average time spent in a queue node, the average number of items in the node, the average number of waiting items, and their standard deviations.

The different constructors *QueueModel(Node *node)* and *QueueModel(Link *link)* are used to build queueing models for nodes and for links.

5.4 Queue Net

The class *QueueNet* makes possible to describe a Jackson network of queues. The network is constructed with the help of function *make_net*. In the default version of this function, each

CSL node and each CSL link is assigned a queueing model, which is introduced using the class *QueueModel*.

Given the number of the network queue nodes and, for each queue node, the arrival rate from outside the network, the probability that an item goes from this node to another given node, the service time, the number of servers, the *QueueNet* returns for each node: the average time spent in a queue node, the average number of items in the node, the average number of waiting items, and their standard deviations.

6 CSL PARTS KIT

The CSL PARTS KIT contains sublibraries that accumulate those system templates (structures, functions, and processes) that are frequently used. 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 are often used in domain-specific models. We consider some examples of useful system parts in this section. We expect the CSL PARTS KIT to continue to grow and to be subdivided into specific domain-oriented kits.

6.1 Memories

Some specific types of memories that are derived from the class *Memory* are introduced in the "Memories" part of the KIT.

The classes *FIFO*, *Stack* and *PriorityQueue* are introduced as classes derived from the class *Buffer*. These memories are often serve as "control memories" that help to implicitly control the traffic in Communicating Structures.

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

More sophisticated memory is exemplified by the class *InOrderFIFO*, which is derived from the *FIFO* class. All items stored in the memory are divided into two groups, *frame items* and *data items*. For each frame item, there is exactly one data item with the same identifier.

When a frame item arrives, it is stored according the FIFO rules. When a data item arrives, it finds its matching frame and replaces it. It is possible to get an item from the memory head only if the item is a data item. This memory may be used to organize in-order traffic in Communicating Structures.

6.2 Nets

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

First of all, the class *SimpleNet* represents non-structured nets that are just arrays of links. Those connections between nodes in which it is not necessary to distinguish between subsets of links may be represented as simple nets.

Let us consider two sets of nodes that we will refer to as *input* nodes A and *output* nodes B. The input nodes 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 represent both sides of the transfer activities. Let N be the number of input nodes and M be the number of output nodes.

It is convenient to define the topology of the connections defined by a simple net using auxiliary *connectivity functions*. The connectivity functions are predicates that are valid for some subsets of integer pairs. 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 function is **true** for the pair (i,j).

For example, the function *alwaysconnected* is the predicate that is **true** for any pair of integers in the range. The function *parallelconnected* is the predicate that is **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 using a set of links that connects each input node with each output node (a bipartite graph constructed with the help of the connectivity function *alwaysconnected*). All links have the same bandwidth. If the node sets A and B coincide, the net *MultiBus* is derived.

The number of servers supplied to the nets *OneWayMultiBus* or *MultiBus* defines how many

items can 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 can be transferred at a time.

The latter case is represented by the simple net *Bus* which is derived from the net *MultiBus* simply by setting the number of servers to one. Thus, the *Bus* net is a communicating structure 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 similar way, the simple net *LeftLoop* is constructed.

Restricting the number of servers to one, transforms the loop nets into ring nets in which only one transfer at a time may occur.

The combination of *RightLoop* and *LeftLoop* makes a *Loop* that connects a node with both its left and right neighbors. This is made with a special SimpleNet constructor that takes two SimpleNets as arguments, builds a union of their link sets, and uses the resulting link set to make the *Loop*. This is an example of compositional construction of CSL objects. This approach is especially sound for nets as they represent sets of links for which set operations may be used directly as compositional operations.

The classes *RightRing*, *LeftRing*, and *Ring* are derived from the classes *RightLoop*, *LeftLoop*, and *Loop* by imposing the restriction that only one transfer may occur at a time in each of these nets.

In a net from the class *StarOut*, a central node is connected by N links to a set of N nodes-receivers.

Similarly, each node in a set of N nodes-senders is connected by a link to the central receiver-node in a net from the class *StarIn*.

The combination of *Starout* and *StarIn* results in the net class *Star* which provides bydirectional traffic among a central node and a set of "peripheral nodes".

Other more complex nets introduced in CSL PARTS KIT, such as *XBar* (crossbar) and *Hammock*, show that quite complicated restrictions imposed on item traffic in interconnects can be expressed through hierarchies of nets and schemes of reserving subnets and links.

6.3 Nodes

The CSL node implements the central notion of Communicating Structures and is the basic building block of any CSL model. Hence, the part of the CSL PARTS KIT that provides special cases of CSL nodes is the central part of the kit.

We introduce some derived classes of nodes with some special types of memories: *Buffer*, *FIFO*, *DoubleBuffer*, *DoubleFIFO*, etc. These memories help to control item traffic implicitly. Some *Node* and *SimNode* virtual functions are redefined to support implicit traffic control.

The derived classes can be also parameterized with relation to the types of their subnodes and types of their nets using *Parameters* and *Registry* mechanisms.

7 SIMPLE MODEL EXAMPLES

7.1 The First Example

This program is a simple model of a communicating structure. It consists of *exm1.h* and *exm1.C* files:

exm1.h file:

```
#ifndef EXM1_H
#include "User.h"
#include "CSL_Basics.h"
#include "Nodes.h"

extern Parameter NumSuperNodes ;
extern Parameter SuperMemType ;
extern Parameter SuperMemSize ;
extern Parameter SuperNodeDelay ;
extern Parameter GlobalNetType ;
```

```
extern Parameter GlobalBandwidth ;

extern Parameter NumSimpleNodes ;
extern Parameter FIFOSize ;
extern Parameter SimpleNodeDelay ;
extern Parameter LocalNetType ;
extern Parameter LocalBandwidth ;

class SimpleNode: public DoubleFIFONode{
public:
  SimpleNode(Context C = Context::C0) :
    DoubleFIFONode(FIFOSize.intVal(), 0, C){
      generator = 1;
      delay = SimpleNodeDelay.realVal() ;
    }
  virtual ~SimpleNode() ;
  virtual void destination(Item *item) ;
};

class SuperNode: public SimNode{
public:
  SuperNode(Context C = Context::C0) :
    SimNode(SuperMemSize.intVal(), SuperMemType.charVal(),
         "SimpleNode",NumSimpleNodes.intVal(), "SIN",
         LocalNetType.charVal(),
         LocalBandwidth.realVal(), 0, C) {
      delay = SuperNodeDelay.realVal() ;
    }
  virtual ~SuperNode() ;
} ;

class Model: public Node{
 public:
  Model(): Node(0, "Memory", "SuperNode",
                NumSuperNodes.intVal(), "SUN",
                GlobalNetType.charVal(),
                GlobalBandwidth.realVal(), 1) {}
} ;
#define EXM1_H
#endif
```

exm1.C file:

```
#include "exm1.h"

Parameter NumSuperNodes("NumSuperNodes", "The number of supernodes", "1");
Parameter SuperMemType("SuperMemType", "The type of a memory in a supernode", "1");
Parameter SuperMemSize("SuperMemSize", "The size of a memory in a supernode", "1");
Parameter SuperNodeDelay("SuperNodeDelay", "The SuperNode delay", "0") ;
Parameter GlobalNetType("GlobalNetType", "Global et type","Bus") ;
Parameter GlobalBandwidth("GlobalBandwidth", "Link bandwidth in global net", "0.0");

Parameter NumSimpleNodes("NumSimpleNodes",
                            "The number of simple nodes in a supernode", "1");
Parameter FIFOSize("FIFOSize", "The size of a FIFO in a node", "1");
Parameter SimpleNodeDelay("SimpleNodeDelay", "The SimpleNode delay", "0") ;
Parameter LocalNetType("LocalNetType", "Local net type","Star") ;
Parameter LocalBandwidth("LocalBandwidth", "Link bandwidth in local net", "0.0");

Register<Object,SimpleNode> SimpleNodeObject("SimpleNode") ;
Register<Object,SuperNode> SuperNodeObject("SuperNode") ;

void SimpleNode::destination(Item *item) {
  Array<Tree*> allleaves = TOP->all_leaves() ;
 item->dest = Path(1, (Node*)(TOP->some(&allleaves))) ;
}

SimpleNode::~SimpleNode() {} ;
SuperNode::~SuperNode() {};

void user_main(void){
  _dummy_nets() ;
  TOP = new Model() ;
}

void user_termination(){}
void user_watch(void){}
void user_special_main(void){}
```

This model represents an object of the class *Model*, which is a top node in the model hierarchy of nodes. It consists of some number of structured nodes that are called *SuperNodes* and each of which consists of the same number of simple nodes *SimpleNodes*.

The class *SimpleNode* is derived from the CSL PARTS KIT class *DoubleFIFONode*. Each simple node has *DoubleFIFO* as its memory. This memory consists of two *FIFO* submemories, each having a maximum size defined by the parameter *FIFOSize*. This and other parameters serve to supply the model with user-defined information that is provided in some *input file* (the file *exm1.in* in our case).

The class *SuperNode* is derived from the class *SimNode*, and subnodes of the supernode are of type *SimpleNode*. Each supernode also has a memory of a type defined by the parameter *SuperMemType* and a size defined by the parameter *SuperMemSize*.

The supernodes are interconnected in the model by a net of a type defined by the user with the parameter *GlobalNetType*. It may be, for example, the net *Bus*, which is defined in the CSL PARTS KIT. All links of the global net have the same bandwidth defined by the parameter *GlobalBandwidth*.

Similarly, the simple nodes are interconnected by a net of a type defined by the user with the parameter *LocalNetType*. It may be, for example, the net *Star* which is defined in the CSL PARTS KIT. All links of the local net have the same bandwidth defined by the parameter *Local Bandwidth*.

The parameters *SuperNodeDelay* and *SimpleNodeDelay* define delays in the supernodes and simple nodes.

Items are generated in simple nodes and are sent also to simple nodes. The customized virtual function *destination* generates a random destination node for each item. The generated items are stored in the second *FIFO* of the node's *DoubleFIFO* memory.

The simple-node *transform* function determines which of its two *FIFOs* contains the item that initiated the transformation. If it is *FIFO* number one, the item was just generated in this node and no transformation need be made. If it is *FIFO* number zero, the item has arrived at this node as its final destination, and the standard node function *terminal_point* is executed.

The function *user_main* contains the execution part of the model. It is called from the *CSL_main.C* file of CSL BASICS which contains the main code that runs the model in the CSL environment. Note that both node classes are registered in the *exm1.C* file:

```
Register<Object,SimpleNode> SimpleNodeObject("SimpleNode") ;
Register<Object,SuperNode> SuperNodeObject("SuperNode") ;
```

This allows us to use strings "SuperNode" and "SimpleNode" as arguments in class constructors.

The values of the parameters mentioned above are provided by the user in the input file *exm1.in*:

```
ModelName = exm1

NumSuperNodes = 4
SuperMemSize = DoubleFIFO
SuperMemSize = 6
SuperNodeDelay = 10
GlobalNetType = Bus
GlobalBandwidth = 50

NumSimpleNodes = 4
FIFOSize = 2
SimpleNodeDelay = 5
LocalNetType = Star
LocalBandwidth = 100

InFile = csl.in
```

This input file also contains a "reference" *InFile* to another input file *csl.in*, which is the default for "standard" CSL parameters introduced in CSL BASE. These parameters are collected in a separate file for convenience.

```
StartTime = 0
EndTime = 2500
PrintStatistics = true
PrintStructure = true
Debug = true
ShowMemory = false
Watch = true
WatchUtilization = true
WatchInterval = 250
Simon = false
Commun = false
SimonList = ()
```

```
SimonNonList = ()
UserSpecialMain = false
BreakName = ""
BreakTime = 0
FullName = false
MeanInterarrive = 40
AttributeResolution := false
ListAttributes := false
CSIMEntityCount = 1000000
CheckMemLeaks = false
WatchMemAllocs = false
CSIMTraceOn = false
LogUtilization = false
```

Here the parameter *EndTime* sets the modeling termination time. The parameter *PrintStructure* prints the model structure. The parameter *Debug* prints the states of nodes and items as the items arrive at the nodes and are transformed there. The parameter *PrintStatistics* prints when its value is the final statistics. And so on.

A typical command line appears as follows:

```
exm1 InputFile=exm1.in OutputFile=exm1.log
```

It may be abbreviated as:

```
exm1 IF=exm1.in OF=exm1.log
```

One can modify the value of a parameter defined in the input file by adding the name of the parameter and its new value to the command line:

```
exm1 IF=exm1.in OF=exm1.log NumSimpleNodes=10 NumSuperNodes=10
```

The model file *exm1.C* should contain include CSL files *User.h*, *Basics.h*, and *Nodes.h*, to be executed in the CSL environment.

So the Makefile should look like this:

```
CSLBASEDIR = /common/csl/base
```

```
CSLOBJDIR = /common/csl/objects
CSLBASICSDIR = /common/csl/basics
CSLKITDIR = /common/csl/kit
CSIMDIR = /common/csim
SIMONDIR = /common/simon
CXXFLAGS = -I$(CSLBASEDIR) -I$(CSLOBJDIR) -I$(CSLBASICSDIR) -I$(CSLKITDIR) \
           -I$(CSIMDIR) -I$(SIMONDIR) -g -v

default: exm1
OBJS = exm1.o
exm1: $(OBJS)
        CC $(CXXFLAGS) -o exm1  $(OBJS) $(CSLKITDIR)/kit.a\
         $(CSLBASICSDIR)/csl.a $(CSLOBJDIR)/obj.a $(CSLBASEDIR)/base.a \
         -L$(SIMONDIR) -lmon -lgcsim -lm
```

7.2 The Second Example

In the first example, the structure of the model was defined inside the code with the help of "detailed" constructors of nodes *Model*, *SuperNode*, and *SimpleNode*. Parameters were used just to define numbers and strings in the lists of arguments.

The second example presents the same model, for which both the model structure and numerical parameters are defined in an input file using the standard CSL BASE parameter *ModelStructure*. *ModelStructure* represents a nested list of strings, some of which serve as keys that are recognized by the node constructors of type *Node(Context C)* (in which *Context C* contains a parameter *C.param*). These keys are followed by strings that are treated as input data, for example, the parameter substring:

```
net (type Bus band 50)
```

This parameter informs a node constructor that its net will be of type *Bus* with a link bandwidth of *50*.

In this case the file *exm2.h* looks like:

```
#ifndef EXM2_H
#include "User.h"
#include "CSL_Basics.h"
#include "Nodes.h"
```

```
class SimpleNode: public DoubleFIFONode{
public:
  SimpleNode(Context C): DoubleFIFONode(C){
    generator = 1;
  }
  virtual ~SimpleNode() ;

  virtual void destination(Item *item);
};

class SuperNode: public SimNode{
public:
   SuperNode(Context C): SimNode(C) {}
   virtual ~SuperNode() ;
} ;

#define EXM2_H
#endif
```

The file *exm2.C* follows:

```
#include "exm2.h"

Register<Object,SimpleNode> SimpleNodeObject("SimpleNode") ;
Register<Object,SuperNode> SuperNodeObject("SuperNode") ;

void SimpleNode::destination(Item *item) {
  Array<Tree*> allleaves = TOP->all_leaves() ;
 item->dest = Path(1, (Node*)(TOP->some(&allleaves))) ;
}

void user_main(void){
 _dummy_nets() ;
}

void user_termination(){}
void user_watch(void){}
void user_special_main(void){}
```

The input file *exm2.in* follows:

```
ModelStructure = \
(name exm2 net (type Bus band 50.0 sep 1) \
      numsubs 4 subs ( (name SUN type SuperNode memory \
                           ( type DoubleFIFO size 6 ) node_delay 10.0 \
                        net ( type Star band 100.0 ) \
                  numsubs 4 subs ( (name SIN type SimpleNode memory \
                                   ( type DoubleFIFO size 2 ) \
                                     node_delay 5.0) ) )))

InFile = csl.in
```

7.3 The Third Example

When a user builds a model using only predefined CSL objects from the CSL PARTS KIT or a CSL DOMAIN LIBRARY, the user may use the following simple *.C* file:

```
#include "CSL_Basics.h"
#include "PartsKit.h"
#include "User.h"

void user_main(){
_dummy_nodes();
_dummy_memories() ;
_dummy_nets() ;

}

void user_watch(){}
void print_statistics(){}
void user_termination(){}
void user_special_main(){}
```

The user may also use the input file *exm3.in* with parameter *ModelStructure* similar to that used in the *exm2.in*.

```
ModelStructure = \
(name exm3 net (type Bus band 50.0 sep 1) \
      numsubs 4 subs ( (name SUN  memory ( type DoubleFIFO size 2 ) \
      node_delay 10.0 \
                        net ( type Star band 100.0 ) \
```

```
                numsubs 4 subs ( (name SIN  memory ( type DoubleFIFO size 2 ) \
                                 node_delay 5.0 generator 1) ) )))

InFile = csl.in
```

8 GRAPHICAL INTERFACE FOR CSL

The CSL GUI provides means to easily construct, run and analyze CSL models. It includes a collection of programs that are accessed through the main GUI program *csl*.

The CSL GUI is implemented under Tcl/Tk (version 7.6/4.1).

8.1 Programs and Files

```
csl             //the main Tcl/Tk program for constructing and running models
csl.in          //the input file that contains the CSL BASE parameters
set_and_run     //the Tcl/Tk program that sets parameters and runs models
slice           //an auxiliary shell program processing parameters
template.C      //the template C++ code for the construction of  a new model
template.class  //the template C++ code for a new node(net, memory) class
template.funcs  //stors all signatures of virtual functions for new classes
template.make   //the template to generate a makefile for a constructed model
simon           //the shell command invoking the SIMON program (by Sekhar Sarrukai)
hotspots        //the Tcl/Tk visualization program showing hot spots in models
plotaid         //the program for tailoring resulting graphs (by Tom Rokicki)
xgraph          //the graph plotting program
```

8.2 Start

The GUI command line is:

```
csl [ModelName]
```

When the program starts, a prompt Start Window (Figure 5) appears with the model name, if one has been specified in the command line. One can enter a new model name or change the old model name in the entry. The Start Window has two buttons: "New" begins construction of a new model; "Set & Run" supplies the model (either built at the previous step or constructed independently) with parameters and runs it.

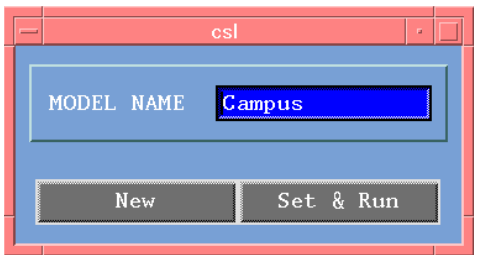
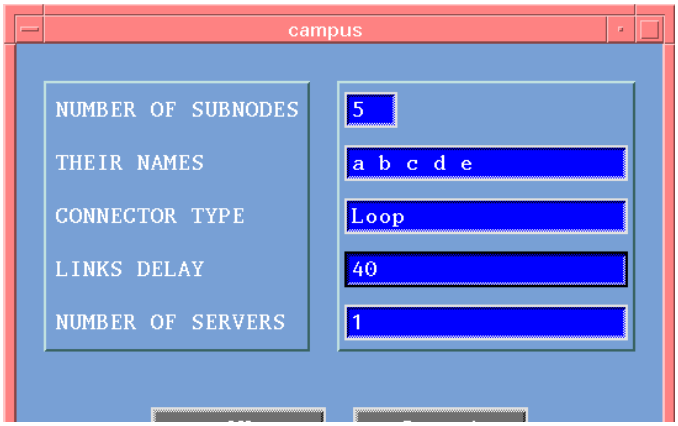


Figure 5: Start Window

8.3 Subnodes Information

After the button "New" is clicked, a Subnodes Dialog Window (Figure 6) requesting information about the structure of the model's top node appears on the screen. This window also pops up when one clicks on the "SubNodes" button in the Node Dialog Window (see Figure 9). The name of the parent node is on the title bar. There are several entries in this dialog