Schemer

A Hierarchical Software Architecture for Real Time Problem Solving

Sharad Srivastava

A research paper submitted to
The Department of Computer Science
Oregon State University
in partial fulfillment of the requirements
for the degree of
Master of Science
in
Computer Science
September 26, 1988

Acknowledgements

I take this opportunity to thank all the people who made my program at Oregon State University educative, challenging, enjoyable and possible.

In particular:

Dr. Bruce D'Ambrosio, my major professor, for his guidance, accessibility and readiness to get involved in details,

Jim Edwards, who has been an indispensable friend, for the long discussions that helped clarify many confusing concepts and abstractions of Computer Science,

My dearest wife Ila, who made all the sacrifices.

Contents

1. Overview 1

2. Background 2
   2.1. EMYCIN 2
   2.2. OPS5 2
   2.3 Blackboard Systems 3
   2.4 BB1 4
   2.5 SOAR 6

3. The need for Schemer 7
   3.1. The need to perform tasks simultaneously 7
   3.2. The need for timely response 7
      3.2.1. Communication 7
      3.2.2. Interruptability 8
      3.2.3. Opportunistic deliberation 8

4. Schemer 9
   4.1 Model of Communication 9
      4.1.1. CONNECT 9
      4.1.2. SEND 9
      4.1.3. RECEIVE 10
      4.1.4. PUT 10
      4.1.5. GET 10
      4.1.6. Examples 11
         4.6.1.1. Synchronous Operations 11
         4.6.1.2. Asynchronous Operations 12
   4.2. The Schemer Organization 14
      4.2.1. Blackbox view of KHs 14
         4.2.1.1. Synchronous Ports 15
         4.2.1.2. Asynchronous Ports 15
      4.2.2. Constituent view of a KH 16
         4.2.2.1. Internal structure of a FKH 16
         4.2.2.2 Internal structure of a PKH 17
   4.3. The Schemer Operation 17
      4.3.1. Basic Operation 17
      4.3.2. Independent vs. Supervised Handlers 18
      4.3.3. The Managers 19
         4.3.3.1. Communication Manager (CM) 19

4.3.3.2. Task Manager (TM) 19
4.3.3.3. Schedule Manager (SM) 20
4.3.3.4. Event Manager (EM) 20

5 Schemer Implementation 21
5.1. The Implementation Environment 21
5.1.1. PCL 21
5.1.2. Stack Groups 22
5.1.3. Processes and Multiprocessing 23
5.2. Structure of Ports 24
5.2.1. Properties of the class PORT 24
5.2.1. Properties of the class SYNC 24
5.2.2. Properties of the class ASYNC 24
5.3. Structure of Knowledge Handlers (KHs) 25
5.3.1. Properties of the class GKH 25
5.3.2. Properties of the class FKH 25
5.3.3. Properties of the class PKH 30
5.4. Structure of a Carrier 31
5.5. Structure of a Message 32
5.6. How the user specifies his Application 33
5.6.1. Static and Run time Structures 33
5.6.2. Giving Names to Knowledge Handlers 35
5.6.3. Running multiple copies of a KH 35
5.6.4. Customizing the KHs 35
5.7. The Managers 36
5.7.1. The Communication Manager (CM) 36
5.7.2. The Task Manager (TM) 38
5.7.3. The Schedule Manager (SM) 39
5.7.4. The Event Manager (EM) 39
5.8. Communication Handle of a PKH 40

6. Issues for Investigation 41
6.1. Enhancing the Schedule Manager 41
6.2. Running Multiple copies of a KH 41
6.3. Meshing 41
6.4. Parallel Processing 41

7. References 43

1. Overview

This paper describes Schemer, a hierarchical software architecture for Real Time problem solving. Real time applications must be able to react to critical events quickly, and be able to explore simultaneous solutions in response to, or, in anticipation of such events.

Schemer, which is a Blackboard-like architecture, addresses the aforementioned objectives by

A. Allowing hierarchical and recurs partitioning of problem solving modules.

B. Allowing interruption and activation at any depth in the hierarchy while still operating under the umbrella of hierarchical control.

C. Providing a model for direct communication between modules, and

D. Allowing multiple modules to be running simultaneously.

This paper is organized as follows. First, in the Background section, extant architectures are explored. Then, the Schemer architecture is described in detail, followed by the discussion of an implementation and implementation issues for future research.

2. Background

Some Artificial Intelligence based Problem Solving architectures are discussed.

2.1. EMYCIN

EMYCIN (1) is an early example of a backward chaining rule based architecture. It evolved out of MYCIN and was an attempt to make MYCIN less medicine specific so that new domains could be tried.

A backward chaining control structure is imposed by EMYCIN. Control is implicit in the order of clauses on the Left Hand Sides of rules and on the order of the rules themselves.

2.2. OPS5

OPS5 (5), an example of a forward chaining rule based system evolved as a member of programming languages supporting the Production systems architecture. A production system is composed of rules called productions, that reside in a Production memory. These productions operate on assertions stored in a Working memory. The architecture operates in a Recognize-Act cycle that iterates through three steps. The first step, Match, evaluates the Left hand side of productions to determine which are satisfied given current contents of working memory, and adds those satisfied along with the instantiations to a Conflict-set. The second step, Conflict-resolution, selects one production with a satisfied LHS. The third step, ACT, performs actions specified on the RHS of the selected productions. Actions may add new productions to Production memory and more commonly, add, modify or delete working memory elements.

Control in production systems is the selection of the one production-instantiation from the conflict set. The domain independent control heuristics are rigid and prefer recent instantiations and productions with more specific left hand sides.

User control can be achieved by having control clauses in antecedents.

2.3. BLACKBOARD SYSTEMS

Comparing Blackboard Architectures (10) to rule based systems, the rules scale up to knowledge sources, the working memory elements of production systems and the objects of EMYCIN scale up to Blackboard data structures and most importantly, the implicit control in rule based systems is replaced by an explicit agenda.

Blackboards offer the following advantages over rule based systems:

a. A larger grain of complex computation. It is difficult to do computation in a single rule.

b. Solutions that are complex structured objects such as identification of Koala bear locations, atomic structure of proteins, a sentence structure or a determination of vehicular movements.

c. Explicit and flexible control. An agenda allows the separation of the Match (a determination that a module can execute) from the select and execute. It allows domain dependent search in deciding what should execute next. That is, it is possible to reason about what to do next.

Because Schemer (4, Conceived by Michael Fehling) is Blackboard based, a somewhat detailed discussion of Blackboard systems follows.

The Blackboard model consists of three major components. The knowledge sources (KSes), a global database called the blackboard, and a control component that is not specified.

External sources post events onto the blackboard that maintains the data state. KSes react to changes on the blackboard, and produce changes to the blackboard that may trigger other KSes. "Communication and interaction among the Knowledge sources take place solely through the Blackboard" (10).

Blackboard architectures produce solutions that are structured objects (7, 8, 10). Each knowledge source posts a solution element onto the Blackboard. The Y-axis of the Blackboard is usually split into several

layers representing a structured abstraction with the solution being in the top most layer and sensory inputs from external sources occupying the bottom most layer. The X-axis usually represents a multidimensional space.

Thus a Blackboard system that identifies Koala bears in a foliage, could have the following layers in the structured hierarchy. Koala, Head/Torso, Limbs, Regions, Lines. Each layer would haves its own, objects and vocabulary, and knowledge sources would typically trigger on data patterns in a layer and would post changes in the same layer or a layer above. The X dimension would be the locations of the sensory input (lines) and partial and final solutions.

Normally blackboards run in a data-driven fashion, but it is possible to have goal directed behavior, by posting goals onto the blackboard and have KS actions trigger KSes in the layer below.

GBB (2), a generic blackboard development system illustrated the use of Blackboard systems in producing solutions that are structured objects. GBB concentrated on efficient updating and retrieval of structured blackboard objects via multidimensional indexing.

All that the control component specified was that 'knowledge sources respond opportunistically to changes in the blackboard' (pieces of knowledge are applied either forward or backward at the most 'opportune' time) (10).

2.4. BB1

BB1 (7, 8) was an implementation of a blackboard architecture with explicit treatment of control. BB1 provided explicit domain and control blackboards as well as domain and control knowledge sources.

Like the domain blackboard, the control blackboard was split into layers. The layers, from top to bottom, were:

a. The Problem
b. Strategies
c. Focus
d. Heuristics

e. Feasible actions
f. Chosen actions

The control knowledge sources modified objects in these layers and operated in both data-driven and goal-driven fashion.

Problem solving took place in a three step basic control loop. Each loop iteration was a deliberate and act cycle (referred to as a solution interval) and served as the X dimension of the control blackboard.

There were three basic types of control knowledge sources corresponding to the three steps of the solution interval. These knowledge sources had no control knowledge of their own, but they adapted themselves to the objects on the control blackboard. Problem solving proceeded as follows.

Step 1. Update the feasible actions layer which is a To-do-set of pending knowledge source activation records (KSARs). A KSAR is a knowledge source coupled with the place on the blackboard where it should be executed.

Step 2. Select which one to execute, based on the objects in the Focus and Heuristics layer. Only one KSAR is selected in each solution interval. This could correspond to a control or a domain knowledge source. This is the 'deliberate' step.

Basically all executable KSARs are rated against how well they match the current focus.

Step 3. Execute the selected KSAR. This is the 'act' step.

The BB1 system entailed high computational costs, making it impractical for real time applications. Because a lot of time was spent deciding what to run next, the grain size of each computation had to be large.

An alternative to the 'deliberate in each loop' approach of BB1 is the 'deliberate when you impasse' approach of SOAR.

2.5. SOAR

SOAR (9) is a problem solving architecture in which every task of attaining a goal is formulated as finding a desired state in a problem space. A problem space is a space of operators and states.

A subgoal is set up to make any decision for which immediate knowledge is insufficient. Whenever a subgoal is encountered in solving a problem, the problem solver begins at some initial state in the new problem space.

Decision situations that give rise to subgoaling are varied. Examples are: which of the many applicable operators to use, how to perform an operator, how to satisfy its preconditions, how to evaluate a state.

Effective control in SOAR depends on the quality of the search functions that select problem spaces, states and operators.

3. The need for Schemer

3.1. The need to perform tasks simultaneously

Real time applications operate in a world of uncertain and incomplete information. They must be able to explore simultaneous solutions, in response to, or in anticipation of critical events so they can weigh the relative costs and benefits of each solution. They also need to be able to actively indulge in activities that help them improve their stored internal model. New and relevant information not only guides solution development, but is invaluable as feedback during solution execution. The determination of what is critical and what is relevant is in itself a complex activity (4).

Neither SOAR nor Blackboard implementations provide support for simultaneous development of multiple solutions. In BB1, there is only one schedule, and only one KSAR is executed in a solution interval. There is only one Focus, which may change. Although the Blackboard Model has the knowledge sources doing their actions in parallel, there is only one Blackboard. When pursuing simultaneous solutions, different solutions and recursively, their sub-solutions may require different Blackboard dimensions and abstraction layers.

3.2. The need for a timely response

Real time applications must be able to respond quickly.

3.2.1. Communication

"Communication and interaction among the Knowledge sources take place solely through the Blackboard". In the SOAR OPS implementation, this is the underlying working memory.

To aid in the timeliness of the response, a communication model is needed for direct message passing among the knowledge sources. Thus in addition to event driven activation, the knowledge sources can invoke each other.

3.2.2. Interruptability

In order to enhance interruptability in a resource bounded system, the application must be able to partition a task into arbitrary grain size subtasks according to it's needs. In blackboard systems, the blackboard is global and portions cannot be cleanly encapsulated.

Partitioning aids in timeliness of response in another way. Computer systems that interact with a dynamic external environment, do so with incomplete and uncertain information. They make assumptions that need to be retracted later. Thus they must provide for management of uncertainty. An ability to partition the underlying truth maintenance system will reduce it's computational complexity. Related research on partitioning the ATMS is ongoing.

3.2.3. Opportunistic deliberation

The BB1 architecture deliberates in each loop. This entails high computational costs. The SOAR architecture deliberates only when it impasses. This may result in poor response to changing conditions.

A real time application should deliberate whenever it is appropriate to do so.

Schemer, whose features go beyond the Blackboard paradigm, addresses these issues.

4. Schemer

4.1. Model of Communication

The communication model is described first because it's constructs are subsequently referred to.

The model used is based on the one described by Reid (11).

The basic mechanism of communication is ports. Ports can be classified in two ways, by type (Synchronous or Asynchronous) and by direction (Input or Output).

The operations on ports are CREATE, DESTROY, CONNECT, DISCONNECT, SEND, RECEIVE, GET and PUT. Some of these are briefly discussed.

4.1.1. CONNECT

Only ports that are compatible (same type and opposite directions) can be connected. Thus a Synchronous Output Port can be connected to a Synchronous Input Port, and an Asynchronous Output Port can be connected to an Asynchronous Input Port.

A port can be connected to multiple compatible ports at the same time.

4.1.2. SEND

The send operation is used to write a message to a synchronous output port. A send can be classified in two ways. It can be a tightly coupled send or a loosely coupled send, and it can be a conjunctive send or a disjunctive send.

In a tightly coupled send, the sender blocks until the message is read via a receive operation on one(all) connected input port(s). In a loosely coupled send, the sender does not block.

In a conjunctive send the message is sent to all connected input ports. In a disjunctive send, the message is sent to only one connected

input port.

Synchronously sent messages are never lost or overwritten. They are always received, and may have to be queued.

4.1.3. RECEIVE

The receive operation is used to read a message from a synchronous input port. The message is removed after it is read.

The receive operation blocks if there is no message to be read.

4.1.4 PUT

The put operation is used to write a message to an Asynchronous output port. A put can be classified as 'Retained' or 'Transmitted'.

A put operation with 'Retained' semantics, destroys all messages at all connected input ports and places the new message at the output port, overwriting any existing message.

A put operation with 'Transmitted' semantics, puts the new message in all connected input ports, overwriting any existing message. Any message stored at the output port is destroyed.

The put operation never blocks. Asynchronously sent messages are never queued. A message may be destroyed even before it is read.

4.1.5 GET

The get operation is used to read a message from an Asynchronous input port. A get can be classified as 'Destructive' or 'Non destructive'.

A destructive read destroys a message after reading it. A non destructive read doesn't.

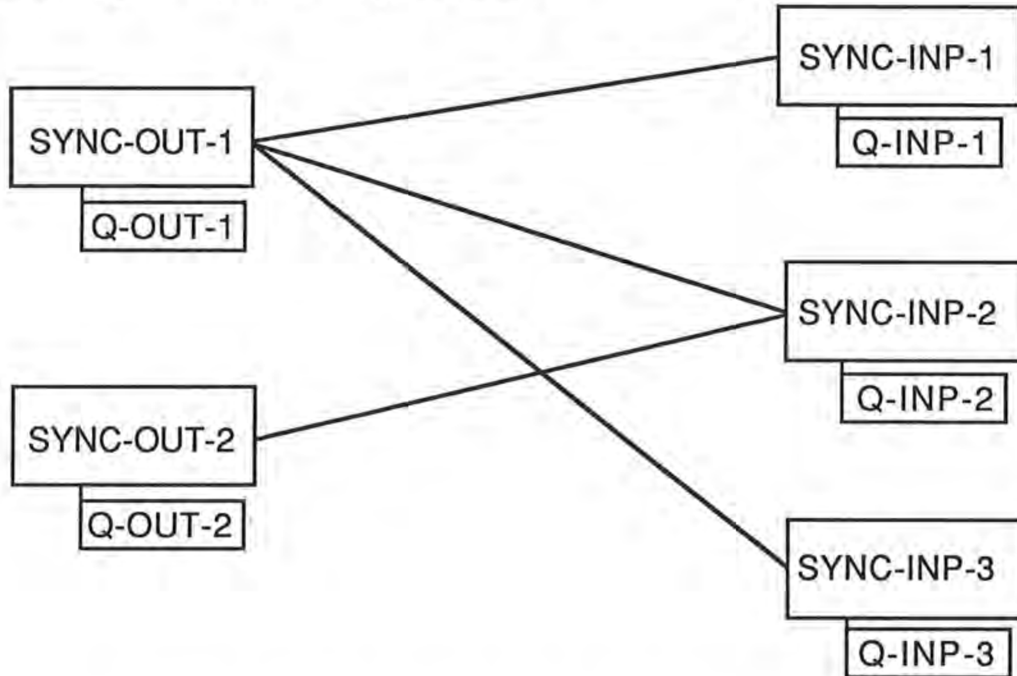
The get operation first checks for a message is stored at the input port. If not, the connected output ports are checked.

The get operation never blocks. The same message may be read many

times.

4.1.6. Examples

4.6.1.1. Synchronous Operations



In the above diagram, the larger boxes represent synchronous ports, each small box represents a queue attached to it's corresponding port, and a line between two ports indicate that the two ports are connected.

A Conjunctive SEND on SYNC-OUT-1 queues the message in Q-INP-1, Q-INP-2 and Q-INP-3.

A Conjunctive SEND on SYNC-OUT-2 queues the message in Q-INP-2.

A Disjunctive SEND on SYNC-OUT-1 queues the message in Q-OUT-1.

A Disjunctive SEND on SYNC-OUT-2 queues the message in Q-OUT-2.

A RECEIVE on SYNC-INP-1 looks for a message in Q-INP-1 or in Q-OUT-1. The order in which the queues are searched is implementation dependent.

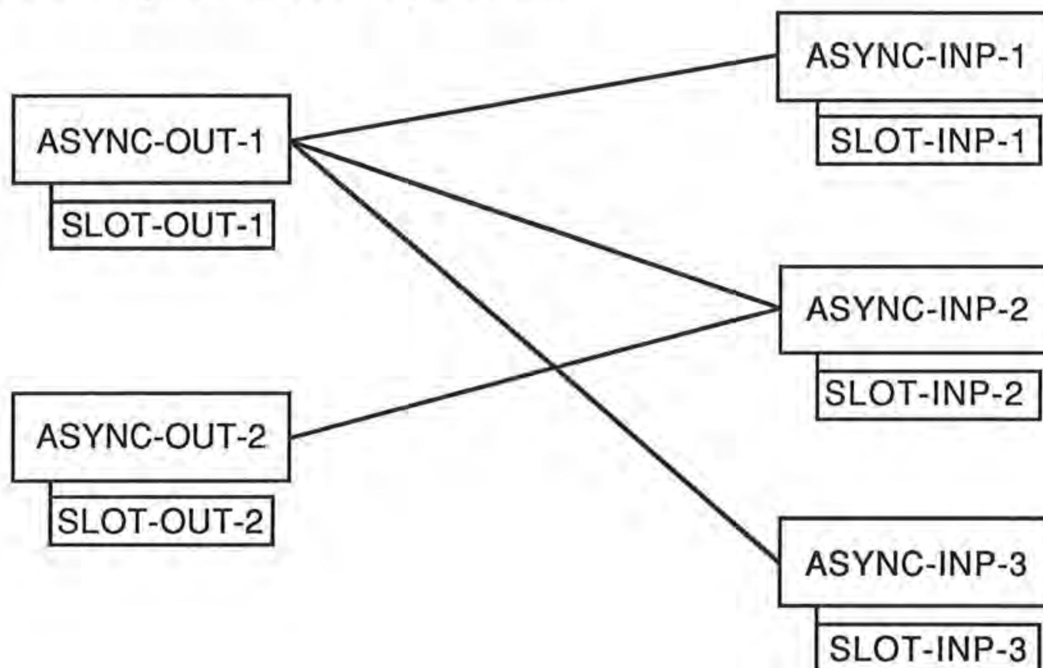
A RECEIVE on SYNC-INP-2 looks for a message in Q-INP-2 or in Q-OUT-1 or in Q-OUT-2. The order in which the queues are searched is implementation

dependent.

A RECEIVE on SYNC-INP-3 looks for a message in Q-INP-3 or in Q-OUT-1. The order in which the queues are searched is implementation dependent.

In a RECEIVE, the message just read is removed from the queue from which it was read.

4.6.1.2. Asynchronous Operations



In the above diagram, the larger boxes represent asynchronous ports, each small box represents a slot attached to it's corresponding port, and a line between two ports indicate that the two ports are connected. A slot can only hold one message, so putting a message in a slot overwrites any existing message.

A transmitted PUT on ASYNC-OUT-1 destroys any message in SLOT-OUT-1 and puts the message in SLOT-INP-1, SLOT-INP-2 and SLOT-INP-3.

A transmitted PUT on ASYNC-OUT-2 destroys any message in SLOT-OUT-2 and puts the message in SLOT-INP-2.

A retained PUT on ASYNC-OUT-1 destroys any message in SLOT-INP-1,

SLOT-INP-2 and SLOT-INP-3 and puts the message in SLOT-OUT-1.

A retained PUT on ASYNC-OUT-2 destroys any message in SLOT-INP-2 and puts the message in SLOT-OUT-2.

A GET on ASYNC-INP-1 retrieves a message from SLOT-INP-1 or SLOT-OUT 1. The order in which the slots are searched is implementation dependent.

A GET on ASYNC-INP-2 retrieves a message form SLOT-INP-2 or SLOT-OUT 1 or SLOT-OUT-2. The order in which the slots are searched is implementation dependent.

A GET on ASYNC-INP-3 retrieves a message from SLOT-INP-3 or SLOT-OUT 1. The order in which the slots are searched is implementation dependent.

If the GET is destructive, the message just read is removed from the slot from which it was read.

4.2. The Schemer Organization

The components of a Schemer system are

1. A set of ports.

2. A top level that controls the system operation. The top level consists of 4 elements called the Communication Manager (CM), the Task Manager (TM), the Schedule Manager (SM) and the Event Manager (EM).

3. A global store called the Knowledge Space (K-SPACE) that houses objects called Knowledge Handlers (KHs).

4. The KHs on the K-SPACE, which provide a uniform representation corresponding to both the blackboard objects and the knowledge sources of the Blackboard model.

   Structurally, there are two types of Knowledge Handlers

a. Full Knowledge Handlers (FKHs)
b. Primitive Knowledge Handlers (PKHs)

The KHs are described next, first from the Blackbox view (how they look from the outside) and then from Constituent view (what are they made up of).

4.2.1. Blackbox view of KHs

Knowledge Handlers provide a uniform representation for problem solving data and problem solving knowledge. A KH can represent either or both.

Regardless of their type (FKH or PKH) or what they represent (data, knowledge or both), all KHs look the same from the outside. That is, they present the same interface to their embedding environments. Each KH has a set of synchronous ports and a set of asynchronous ports.

4.2.1.1. Synchronous Ports

Synchronous ports are used for messages that must not be overwritten or destroyed before they are received. In a tightly coupled send, the sender blocks till the receiver receives the message. On any input port, a receiver can only receive the message once.

Synchronous input ports are for messages that come from the outside world or other KHs. Similarly the Synchronous output ports are for messages that go to the outside world or other KHs.

Any implementation can specially designate ports for messages to/from the outside world.

4.2.1.2. Asynchronous Ports

Each KH makes data available to the outside world through numerous Asynchronous output ports. Data is put on these ports using the 'Retain' semantics and read via connected input ports using the 'non-destructive' semantics.

These semantics are appropriate for problem solving data that may be read multiple times, or be changed many times without being read.

Although the KH is an object, the difference from a strictly object-oriented view is that any data in these Asynchronous Output Ports is freely available to any other object that is properly linked. So the owning KH need not be entered and the requestor does not lose control.

The Asynchronous output ports can contain any arbitrary objects. Examples are a) stimulus frames b) response frames c) preconditions d) a default priority e) an indication of the resources a typical task done by the KH takes f) a Schedule to be used by the SM g) a trigger table to be used by the EM h) a History i) the result of a trial run j) a partial or a complete plan k) a goal or a subgoal to be achieved and so on and so forth.

Thus a knowledge handler that constructs a plan could be incrementally building it on one of its Asynchronous output ports.

4.2.2. Constituent view of a KH

This section describes the internal structure of Full Knowledge Handlers (FKHs) and Primitive Knowledge Handlers (PKH).

4.2.2.1. Internal structure of a FKH

Each FKH is itself a Schemer system. It is internally composed of a set of internal ports, a top level consisting of the four managers (CM, TM, SM and EM) and a K-SPACE that houses KHs.

Schemer is therefore recursively composed. All KHs on the K-SPACE are siblings and are 'Embedded in the owning FKH'.

The internal components of a FKH are:

a. Synchronous Input Ports

These ports are to receive messages from embedded KHs.

b. Embedded KHs

Since they can house data, the embedded KHs allow the embedding KH to have a local persistent data state. The asynchronous output ports of some of these hold special data structures like a schedule of pending tasks or a history of completed tasks.

The schedule is a prioritized queue of carriers. A carrier corresponds to the KSAR of BB1. It allows an embedded KH to be multiply scheduled with different arguments. The schedule allows for flexible control and an interrupt capability.

c. Communication Manager (CM)

The CM manages the port structure of the FKH and the flow of information between its K-SPACE and the outside world.

d. Task Manager (TM)

The TM runs the tasks on the schedule and performs process

management functions.

e. Schedule Manager (SM)

The SM manages and prioritizes the schedule.

f. Event Manager (EM)

The EM monitors the data state changes on the K-space and informs the SM about potentially runnable handlers. Roughly, the EM, SM and TM perform the deliberate-act loop of the BB1 architecture.

A fixed interruptable algorithm could be implemented as a degenerate case of a FKH which has no event triggered activity at all, but an embedded handler representing a fixed schedule of other embedded handlers.

4.2.2.2. Internal structure of a PKH

As mentioned earlier, Schemer is recursively composed. Primitive knowledge handlers bottom out this recursion.

A PKH must specify a communication handle which acts as a top-level to an uninterruptable algorithm. The execution of a PKH cannot be interrupted.

A PKH can have additional components including those with local persistent data states.

4.3. The Schemer Operation

4.3.1. Basic Operation

A data change takes place in the K-SPACE either as a result of external conditions, processing of new inputs by the CM or by the TM's execution of a KH from the schedule.

The EM notices the data change and determines that the new data state matches the trigger conditions of some embedded KHs. The EM forms a carrier and passes it to the SM.

The SM places the carrier on the Schedule.

The TM responds to the new schedule by running/resuming the "n" most important tasks. Any previously running task that is not among the "n" most important is suspended. Suspending a FKH means suspending all its (recursively downwards) running KHs. Each KH reports to its embedding TM (recursively upwards) the amount of resources used by it.

Conceptually, the four managers run in parallel. So the distance from the triggering to the need to change the control state, to changing it is zero. Any implementation must minimize this distance.

4.3.2. Independent vs. Supervised Handlers

Normally a KH runs under the direct supervision of its embedding TM. It is possible for a KH to attain an independent status (i.e. not under the direct control of the embedding TM). However, this independent status will usually be for a brief period, because the embedding TM reasserts its control as soon as it can. The outermost FKH is always independent, because it does not have an embedding TM.

Why have independent handlers ? Consider the following situations.

A KH that is suspended or terminated may receive a critical message from another KH or from the outside world. The sender of the message may feel that the message is critical and resume or start the KH. In either case, the status of the KH becomes independent, until the embedding TM changes it to supervised.

A KH might get new data that changes the data state of the embedding KH in a manner that triggers some other KHs. If the embedding KH is suspended or terminated, then it has to be resumed or started. The embedding KH becomes independent.

Thus Schemer allows for interrupts and activation at any depth in the hierarchy while still operating under the umbrella of hierarchical control.

4.3.3. The Managers

4.3.3.1. Communication Manager (CM)

The CM receives and processes messages that arrive at the input ports of the KH.

These messages could be control messages for activation, suspension, resumption and termination. They could be non-control messages that are processed by the CM or forwarded to the input ports of other KHs.

The connections between ports of different KHs are either predefined by the structure of the application, or are new connections that are established via messages sent over the predefined connections.

It is the job of the CM to manage/authorize new ports and connections that connect ports internal to the KH to the ports outside.

The CM can connect a handler's ports to external ports. It is these connections that give global environments.

4.3.3.2. Task Manager (TM)

Takes the "n" available processors and allocates it to the "n" most important tasks on the schedule. It also brings under its control any KH that is running with an independent status.

The TM should be able to initiate, suspend, resume and terminate tasks. That is, it should be able to perform basic process management functions.

When a task is suspended, it must be done in a (application specific) way that consistency problems are taken care of.

Because KHs have local persistent data states, the TM must be able to duplicate a KH, for one might want to suspend a KH and run another carrier corresponding to the same KH.

4.3.3.3. Schedule Manager (SM)

The schedule at any instant is indicative of potential future actions.

The SM maintains the schedule. Together, the TM and the SM do the base level of control reasoning in the system. The carriers that make up the schedule can contain some default scheduling information (like priorities).

The schedule itself is available like other problem solving data, which provides an ability to distribute control via certain handlers called control handlers. These handlers can be triggered by changes to the schedule, and/or directly manipulate the schedule. Control handlers will usually be high priority, so that they run first.

4.3.3.4. Event Manager (EM)

The EM watches out for data state changes in the K-SPACE for patterns that match the trigger conditions of the embedded handlers.

There would be no need for an EM in a system if every embedded KH could trigger itself.

One of the things in an implementation is to compile away as much overhead as possible. This amounts to placing each embedded KH's trigger conditions in a trigger-table, a structure which like the schedule is available like other problem solving data. The main user of this structure is the EM of the embedding KH.

5. Schemer Implementation

The managers of any FKH could be specialized and encode application specific control knowledge. The implementation supports such specialization. The implementation also provides generic managers. These managers have the ability to receive and process application specific messages. The evolution of these generic managers can best take place through experimentation.

5.1. The Implementation Environment

Schemer is implemented in Allegro Common Lisp on a Tektronix workstation running the UTek operating system. A port to Lucid on a SUN workstation running SunOS is in progress.

The implementation uses the PCL and the MULTIPROCESSING packages.

5.1.1. PCL

PCL provides the programmer interface for object-oriented programming in the Common Lisp Object System (CLOS).

Using PCL, it is possible to setup a class hierarchy with property inheritance. PCL provides the ability to specify multiple methods for a function, each method corresponding to a unique specialization of the function's parameters. Default accessor functions and functions to create class instances are also provided.

The following code fragment illustrates the specification of a class hierarchy of ports and the use of methods.

```
(defclass PORT ()
    ((name) (owner) (connected-to :type LIST :initform nil)))

(defclass SYNC-PORT (PORT)
    ((message-list :type LIST :initform nil)))

(defclass SYNC-INP-PORT (SYNC-PORT) ())
(defclass SYNC-OUT-PORT (SYNC-PORT) ())
```

```
(defclass ASYNC-PORT (PORT)
    ((message :initform nil)))

(defclass ASYNC-INP-PORT (ASYNC-PORT) ())
(defclass ASYNC-OUT-PORT (ASYNC-PORT) ())
```

A uniform interface for sending messages on both types of output ports can be achieved as follows:

```
(defmethod SEND-TO-PORT ((port SYNC-OUT-PORT) (msg t)) ..........................)
(defmethod SEND-TO-PORT ((port ASYNC-OUT-PORT) (msg t)) .......................)
```

A uniform interface for retrieving the owner for all types of ports could be:

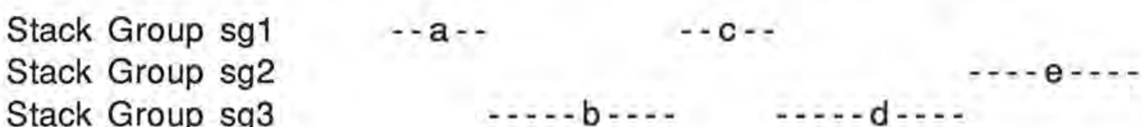
```
(defmethod RETRIEVE-OWNER ((port PORT)) ..........................................................)
```

5.1.2. Stack Groups

Stack groups are briefly described here because multiprocessing is implemented on top of stack groups.

"A stack-group represents the state of a Lisp computation. It most closely corresponds to the notion of a co-routine. The only thing a stack-group can do (aside from normal Lisp computation) is to resume some other stack-group, and perhaps be itself resumed later on" (12).

In the following illustrative diagram, the X-axis is time.



a. sg1 is resumed the first time.
b. sg1 resumes sg3 the first time.
c. sg3 resumes sg1.
d. sg1 resumes sg3.
e. sg3 resumes sg2 the first time.

5.1.3. Processes and Multiprocessing

"A process corresponds to the usual operating system notion of a process. The process mechanism provides a convenient programming interface to stack-groups. Each process has its own stack-group. A scheduler process is responsible for managing all other processes" (12).

The scheduler process cannot interrupt a process that is executing code wrapped in a **without-scheduling** macro. A process can voluntarily give control to the scheduler process via the **process-allow-schedule** construct and optionally direct the scheduler to resume a specified process.

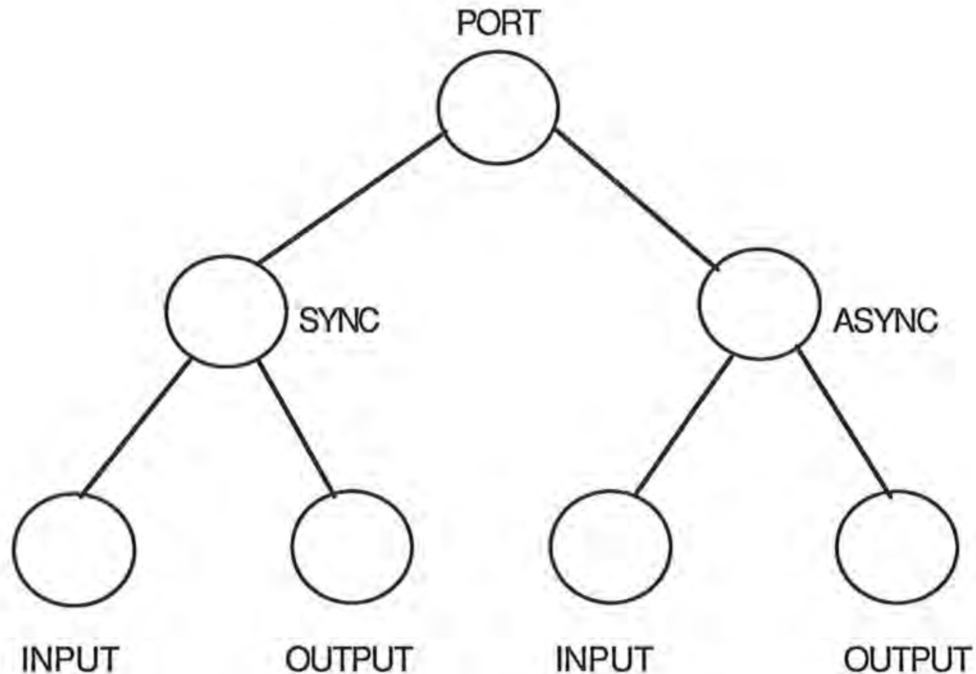
Each process is an object, and has associated with itself among other things, a list called **run-reasons**, a list called **arrest-reasons** and a **property list**.

For a process to run, the **run-reasons** list must not be nil, and the the **arrest-reasons** list must be nil. Disabling a process makes both these lists nil. Enabling a process puts the token `:enable` in the **run-reasons** list for the process.

The managers are implemented as processes. A property called **:action** is defined whose value can be one of `resume`, `suspend` or `kill`. Processes are required to check the value of their **:action** property regularly, which is a directive.

5.2. Structure of Ports

Ports are PCL objects. The class hierarchy of ports is as follows.



5.2.1. Properties of the class PORT

a. **p-name**: Name of the port.

b. **p-owner**: Owner of the port, which is a KH.

c. **p-conn**: List of compatible ports, a port is connected to.

5.2.2. Properties of the class SYNC

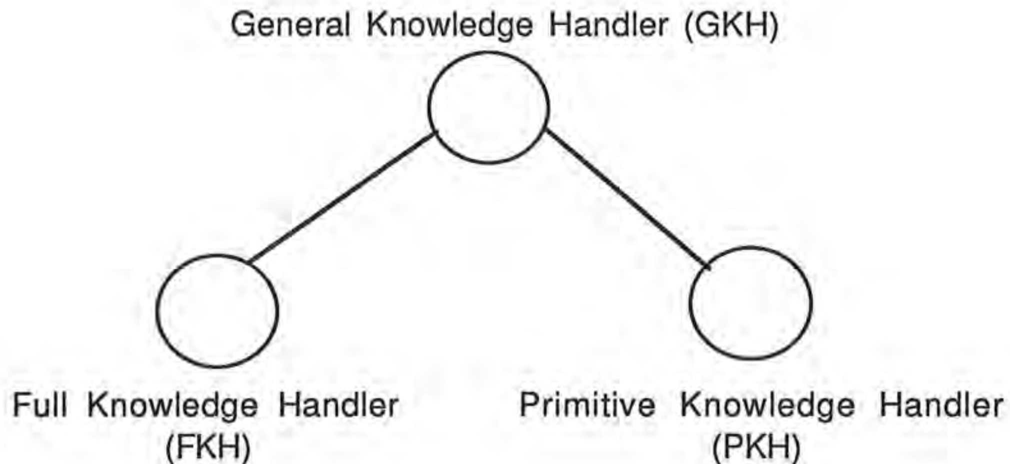
a. **p-msg-lst**: List of messages.

5.2.3. Properties of the class ASYNC

b. **p-msg**: A message.

5.3. Structure of Knowledge Handlers (KHs)

Knowledge Handlers are PCL objects. The class hierarchy of KHs is as follows.



5.3.1. Properties of the class GKH

The class GKH corresponds to the Blackbox view of knowledge handlers.

a. name (NAME)

Each KH has a name. The name is unique. Construction of the name is explained later.

b. External field (EXT-FLD)

This field is **t** if the KH is currently executing in the independent mode, and **nil**, if it is in the supervised mode.

c. Main Input Port (INP-P)

This is a Synchronous Input Port. Each message sent to this port is read by the Communication Manager in order.

d. Interrupt Port (INT-P)

This is a Synchronous input Port. This Port behaves like the Main Input Port, but is given the top priority among all ports.

It is also possible for the outside world to have direct access to this port, no matter how deeply the KH is embedded.

e. Direct Input Port (DIR-INP-P)

This is a Synchronous input port and like the INP-P and INT-P ports, each message sent to this port is read in order. These ports are connected to the DIR-OUP-P ports of other KHs.

f. List of Direct Output Ports (DIR-OUP-P-LST)

This is a list of Synchronous output ports. A message sent on the DIR-OUP-P port of a KH is conjunctively sent to all connected DIR -INP-P ports.

g. List of Output Data Ports (DATA-P-LST)

This is a list of Asynchronous output Ports. These ports can contain arbitrarily complex objects.

h. List of Input Data Ports (INP-DATA-P-LIST)

These are Asynchronous Input Ports that are connected to the Output Data Ports of other KHs.

i. Trigger Port (TRIG)

This port is implemented as a slot. It contains the trigger conditions of the KH, in the form of a predicate that returns true, when conditions are ripe for the KH to be triggered.

The predicate takes as arguments Data Ports of the KH and of sibling KH's.

j. Response Port (RESP)

This port is implemented as a slot. It contains an indication of what the KH will do/achieve. It is useful for scheduling purposes.

k. Priority Port (PRIOR)

This port is implemented as a slot. It containes a priority and is useful for scheduling purposes.

l. Grain Port

This port is implemented as a slot. It gives a rough estimation of the time the KH takes to execute. It is used for both process management and scheduling purposes.

m. Initial Flag (init-fl)

This flag when **t**, causes the SM of the embedding KH to put this KH on the schedule on start up.

n. Control Knowledge Handler Flag (ckh-fl)

This flag when **t** indicates that this KH may modify the schedule.

5.3.2. Properties of the class FKH

a. Communication Manager code (cm)

This slot contains the communication manager code.

b. Communication Manager Process (cm-proc)

This slot contains the process corresponding to the executing CM.

c. Task Manager code (tm)

This slot contains the task manager code.

d. Task Manager Process (tm-proc)

This slot contains the process corresponding to the executing TM.

e. Schedule Manager code (cm)

This slot contains the schedule manager code.

f. Schedule Manager Process (sm-proc)

This slot contains the process corresponding to the executing SM.

g. Event Manager code (em)

This slot contains the event manager code.

h. Event Manager Process (em-proc)

This slot contains the process corresponding to the executing EM.

i. Internal Input Port for the CM (cm-p)

This Synchronous Input Port is for messages from the embedded KHs to the CM.

j. Internal Input Port for the TM (tm-p)

This Synchronous Input Port is for messages from the embedded KHs to the TM.

k. Internal Input Port for the SM (sm-p)

This Synchronous Input Port is for messages from the embedded KHs to the SM.

l. Internal Input Port for the EM (em-p)

This Synchronous Input Port is for messages from the embedded KHs to the EM.

m. Special port for the SM (SM-EM-P)

This special Synchronous Input Port is read by the SM. It is meant for messages from the EM.

n. Special port for the EM (EM-TRIG-P)

This special Synchronous Input Port is read by the EM. It is used for messages that the EM uses to index into a trigger-table.

o. Schedule

The schedule which is a list of carriers is implemented as a slot.

p. Schedule Flag (sched-fl)

This flag when t indicates that the schedule has been initialized.

q. Schedule Lock (sched-lock)

A lock that can only be held by one process at a time.

r. Trigger Hash (trig-hash)

The Trigger Table which is built from the information in the trigger ports of embedded KHs.

Asynchronous output data ports of embedded KHs hash into this table to retrieve a list of KHs they can potentially trigger.

s. Quiesce Flag (quiesce)

When a KH terminates, all the manager processes terminate. This results in the associated stack-groups being destroyed. When this flag is **t**, instead of terminating the stack-groups, the processes are disabled and marked as quiescing.

The advantage is that new processes need not be created for the next execution of the KH. Only the processes need be enabled.

t. List of embedded KHs (kh-lst)

u. Hash table of embedded KHs (kh-hash)

v. List of embedded KHs whose init-fl is t

w. List of embedded KHs whose ckh-fl is t

5.3.2. Properties of the class PKH

a. Communication Manager code (cm)

This slot contains the communication handle which acts as a top level to an uninterruptable algorithm.

b. Any other properties.

5.4. Structure of a Carrier

A carrier which is a defstruct object has the following structure:

priority:
The Task Manager uses this priority to sort the schedule.

kh:
Contains the ID of the Knowledge Handler that will be or is being executed.

proc:
Contains the process object associated with the CM of the KH.

alloc:
Contains an indication of the number of execution units the handler should take to run to completion. Could also contain symbolic values like `:quick`, `:no-hurry` etc.

slice:
Contains the execution units after which the TM may give the processor to some other carrier.

The value of this slot is transferred to the :slice property of the CM-PROC and is used for time slicing among carriers.

new-slice:
Indicates whether the **slice** slot contains a new value.

suspend:
Indicates whether the KH is to be suspended. Carriers whose KHers are executing will be suspended if this field is **t**. Carriers whose KHers are not executing will not be started if this field is **t**.

This is a primitive mechanism for run-time dependencies among Handlers. (A must execute before B etc. etc.).

ph-fl:
t if the KH is a primitive kh. **nil** otherwise.

arg:

A list whose contents will be put into the INP-P port of the KH before it is started. Currently not used.

ind-fl:

The independent status of the carrier. It is **t** if the associated carrier was invoked by a process other than the Task Manager of the embedding KH. It becomes nil when the Task manager of the embedding KH takes control.

Meant to support direct invocation from the outside.

5.5. Structure of a Message

Messages that arrive at the non-special input ports of a FKH have the following structure:

path-remaining:

The CM uses this field to determine the next KH to forward this message to. If **nil**, the CM executes the message.

path-sofar:

This field contains the path taken by the message so far.

orig-kh:

Contains the originating KH of the message or **t** (If the message came from the outside world).

task:

This field contains a function. Executing this message means executing this function.

args:

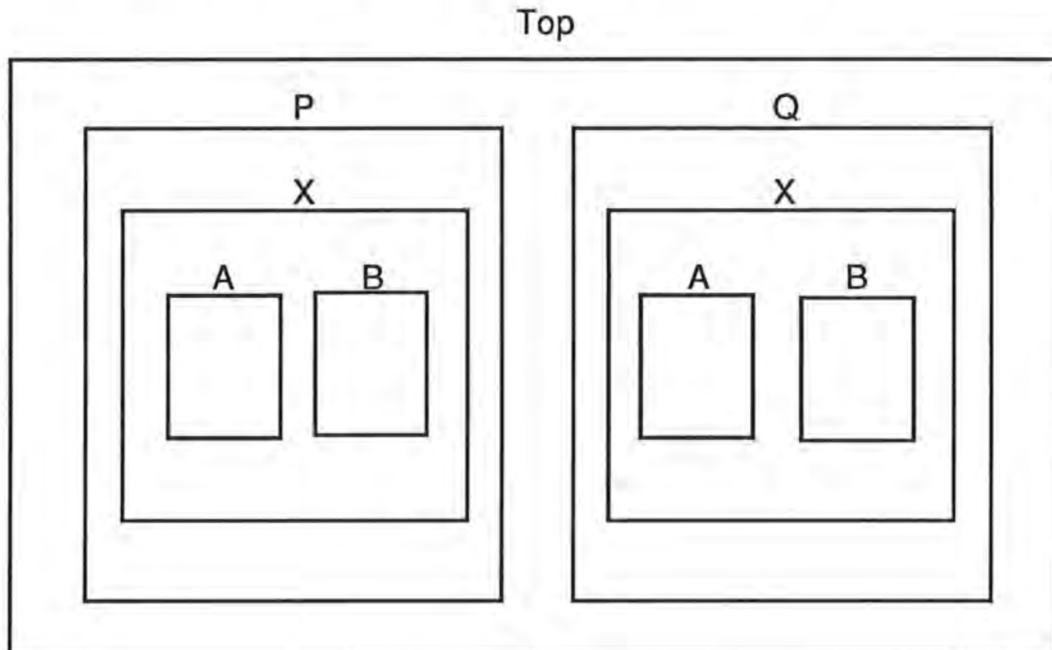
This is a list of arguments. The executing KH is consed onto this list which is applied to the function.

5.6. How the User specifies his Application

When the user specifies his application, he specifies what is termed as the static structure. From this, a run time structure is generated, which can change during run time.

5.6.1. Static and Run time Structures

In the following diagram, each box represents a seperate KH. The label on top of each box, is the user defined class of the KH.



The user can specify the above configuration by the following:

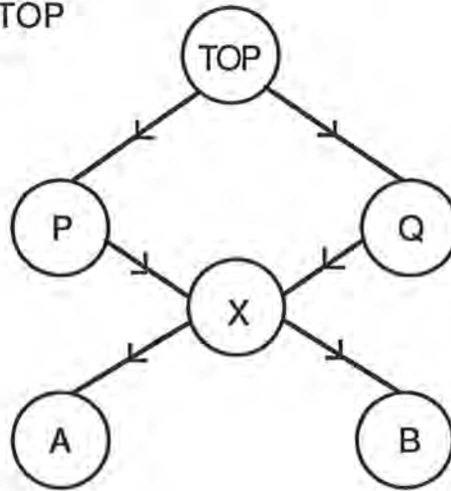
(def-fkh-class TOP (P Q) . . . .
(def-fkh-class P (X) . . . .
(def-fkh-class Q (X) . . . .
(def-fkh-class X (A B) . . . .

(def-pkh-class A . . . .
(def-pkh-class B . . . .

TOP, P, Q and X correspond to FKHs. A and B correspond to PKHs.

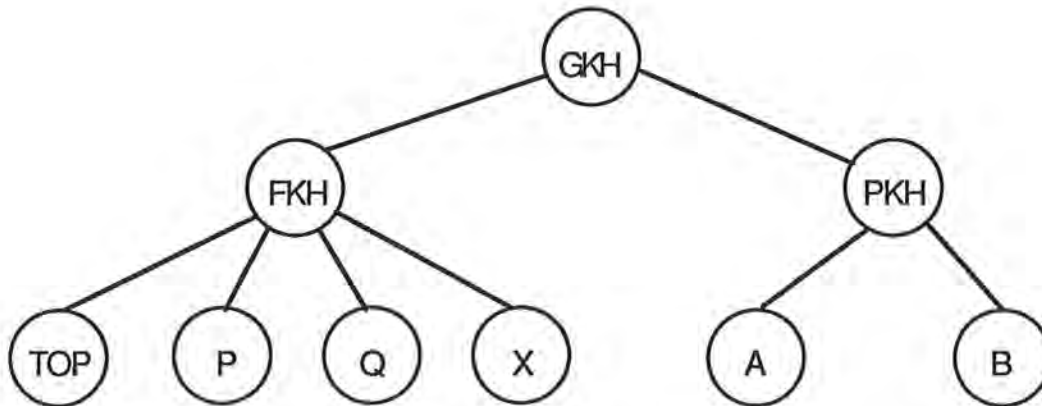
As a result of executing the def-fkh-class and def-pkh-class forms, two static structures are generated.

a. A *dag* rooted at TOP



Preorder is: TOP (P (X (A B)) (Q (X (A B))))

b. An extension to the class-subclass hierarchy of KHs as follows.



The run time structure can be created by traversing the *dag* in preorder, creating the corresponding KH instance at each node, and linking it to the KH instance that is its parent. The leaves of the *dag* generate PKHs. The other nodes generate FKHs.

Thus X, A and B are defined only once, but their instances appear twice in the run time structure. The def-fkh-class and def-pkh-class macros define PCL classes.

5.6.2. Giving Names to Knowledge Handlers

Each knowledge handler has a unique name. The name is dotted pair of the form (**<class> . <int>**) where **<class>** is the user defined class and **<int>** is a unique integer for each instance of that class (incremented by one on every time an instance is created).

Thus if the run-time structure of the above examples were traversed in a pre-order fashion, the KH names would print out as follows:

(TOP . 1) [(P . 1) [(X . 1) [(A . 1) (B . 1)]] [(Q . 1) [(X . 2) [(A . 2) (B . 2)]]]]

This naming convention uniquely identifies each KH, and makes it possible to send interrupt messages to all instances of a user defined class of handlers. The structure *slot* indexes instances by the user defined class.

5.6.3. Running multiple copies of a KH

The way the user specifies his application and the KH naming convention makes it possible to have multiple copies of a KH embedded in the same FKH, even though each KH has a local persistent data state.

Although it is not implemented, a handler can be duplicated at run time by traversing the sub *dag* rooted at the node that represents the class of the handler to be duplicated. The mechanism is the same that is used to create the initial run time structure from the static structure.

5.6.4. Customizing the KHs

Both the def-fkh-class and def-pkh-class forms allow the user to specify initial values for the relevant properties of the GKH class.

The def-fkh-class form allows the user to specify initial values for the relevant properties of the FKH class.

The def-pkh-class form allows the user to specify the value for the CM-P property (the Communication Handle). In addition it allows the user to add new properties and specify their initial values.

If the value for any of the CM, TM, SM or EM slots is not specified with the def-fkh-form, then that slot defaults to the corresponding generic manager. For any user defined class, any generic manager can be replaced by an application specific counterpart.

Examples:

```
(def-fkh-class  X (A B C)  :CM #'my-cm :PRIOR 9
                           :TRIG (volt-fluctuatingp (volt history)))

(def-pkh-class  A :GRAIN 10 :CM #'my-function)
```

The trigger table of the embedding KH is constructed from the :TRIG values.

5.7. The Managers

5.7.1. The Communication Manager (CM)

The communication manager code which is in the CM slot of the FKH must be executed to start up the FKH. This code runs as a process and the process object is stored in the CM-PROC slot of the FKH.

The CM-PROC process has the following properties: **:action**, **:refresh**, **:quiescing**, **:used**, **:slice**, **:independent** and **:numproc**.

:action, which can be `kill`, `resume` or `suspend`, serves as a directive for the CM and the TM.

:refresh, when `t`, indicates that the four managers must execute their initialization code.

:quiescing, when `t`, means that the four managers have logically terminated, but physically the associated processes exist in a disabled state (helps speed up the next initialization).

:used holds the amount of processing units used by the KH since it's value was last initialized. The values of the grain ports of executed PKHs propagate up the run time structure and contribute to this value.

:slice holds the amount of processing-slice given by the TM of the embedding KH. If **:used** exceeds **:slice**, **:used** is initialized and the KH is suspended (and recursively so are all the executing KHs on the schedule) until the TM of the embedding KH, resumes it with another slice.

:independent, when `t`, indicates that the KH is running in an independent state (as opposed to supervised state), and,

:numproc, which is an integer, holds the number of processes the TM can execute simultaneously.

As part of it's initialization the CM creates the TM, SM and the EM processes. It then passes direct control to the TM via the **process-allow-schedule** construct.

When resumed again, the CM goes into it's basic loop (as do the other managers) disabling itself at the end of each loop if it has nothing to do in the next iteration. The basic loop operation is as follows:

If the KH is dormant (the schedule and all the synchronous input ports are empty), the CM kills all the managers (including itself). If the quiesce flag of the FKH is t, the kill is logical and the four managers start quiescing. Otherwise, the associated stack-groups are destroyed.

The values of **:used** and **:slice** are compared. If **:used** is greater than **:slice**, then **:action** is set to `suspend`.

The CM then looks at is it's **:action** property. If the action is `kill` (or `suspend`) it kills (or disables) the SM and the EM and passes direct control to the TM.

If the **:action** is `resume`, it first enables the other managers and then processes all the messages in it's INT-P port and a maximum of one message from each of it's other synchronous input ports.

Each message has a **path-remaining** component. If it is null, the CM executes the actual message. Otherwise the message is forwarded to the next KH in the path. Another component, **path-sofar**, contains the path traced by the message so far.

The **task** component of the message is application specific (see the section 5.5. 'Structure of a Message'). Examples of generic tasks are: Connect a port, Install a KH).

5.7.2. The Task Manager (TM)

The task manager code is in the TM slot of the FKH. This code runs as a process and the process object is stored in the TM-PROC slot of the FKH.

As part of its initialization the TM passes direct control to the SM via the **process-allow-schedule** construct.

When resumed again, the TM goes into it's basic loop, disabling itself at the end of each loop if it has nothing to do in the next iteration. The basic loop operation is as follows:

The TM goes through all the carriers on the schedule, transferring to the **:used** quantity from the corresponding KHs to the **:used** property of it's associated CM-PROC.

The TM then looks at **:action** property of CM-PROC. If the action is `kill` (or `suspend`), it passes this directive to the CMs of all the running embedded KHs (excepting those running with an independent status) before killing (or disabling) the CM and itself.

The Task Manager then goes through the first **:numproc** carriers on the schedule (excluding those that have **:suspend = t**), essentially running or resuming them and suspending all the carriers after the first **:numproc**.

If any carrier in the first **:numproc** entries has used up its time slice (i.e. the **:used** property of the associated CM is greater than the **:slice** property), then the associated KH is suspended. If all of the first **:numproc** carriers have used up their time slices, then each is given a fresh slice and resumed. Thus the processing resource allocation among concurrently running KHs is distributed in the ratio of their **:slices**.

Any resumed KH that had an independent status becomes supervised (**:independent** is set to `nil`). A KH gains independent status when started by someone other than it's embedding TM.

The TM then executes messages in it's TM-P input port.

5.7.3. The Schedule Manager (SM)

The schedule manager code is in the SM slot of the FKH. This code runs as a process and the process object is stored in the SM-PROC slot of the FKH.

As part of it's initialization the SM initializes the schedule and creates carriers for the embedded KHs whose init-fl is t. It then passes direct control to the EM via the **process-allow-schedule** construct.

When resumed again, the SM goes into it's basic loop, passing direct control to the TM at the end of each loop. The basic operation is as follows:

The SM reads the special SM-EM-P input port for any KHs that have been triggered as a result of changes in the K-SPACE. Information in this port comes from the EM. The SM then forms carriers and places them on the schedule. Most of the information that is put in the carrier comes from the asynchronous output ports of the KH.

The SM then executes messages in its SM-P input port.

5.7.4. The Event Manager (EM)

The event manager code is in the EM slot of the FKH. This code runs as a process and the process object is stored in the EM-PROC slot of the FKH.

The EM performs some initialization and relinquishes control.

When resumed again, the EM goes into it's basic loop, passing direct control to the SM at the end of each loop. Hence the distance from EM to SM to TM is reduced as much as possible. The basic operation is as follows:

The EM reads the special em-trig-p input port for the asynchronous data ports of any embedded KHs. Whenever any KH writes to an asynchronous output port, and that port has an index in the trigger table, a

pointer to the port gets queued as a message in the em-trig-p port.

The EM then indexes trig-hash using each asynchronous port and retrieves all the associated KHs. These KHs have the asynchronous port as one of the arguments to their trigger predicate. The trigger predicates of all the retrieved KHs are then executed by the EM and the ones that return `nil` are discarded.

Each of the triggered KHs are next used to index into trig-hash again. These retrieve the control handlers.

All the retrieved handlers are passed to the SM via the SM-EM-P special port.

The EM then executes messages in it's EM-P port.

5.8. Communication Handle of a PKH

The CM of a PKH is the top level to an uninterruptable algorithm. The TM of the embedding KH executes a PKH by funcalling it's CM. The value of the grain slot of the PKH is added to the **:used** property of the CM-PROC of the embedding system.

6. Issues for Investigation.

6.1. Enhancing the Schedule Manager

The generic managers are extremely simple. They have the ability to receive application specific messages and execute them.

Although control is distributed via control knowledge handlers, control handlers themselves have to be put on the schedule first.

Efficiency could be gained by having the ability to send messages to the SM that can permanently change it's generic behavior (Use a different sort predicate to sort the Schedule !)

6.2. Running Multiple copies of a KH

Since KHs can have a local persistent data state, they are not reentrant. The implementation does provides a mechanism to recursively duplicate a FKH, but it does not allow the copying of any local data at all.

PCL provides an ability to specify that the storage for a slot (property) not be located in each instance of the class but in the class object itself, making it possible to share data amongst all instances of the class.

6.3. Meshing

A KH is meshed (3, 6) if it is embedded in more than one FKH. Meshing gives a facility to specify global data, and can facilitate better performance (3). However the meshed KH may force the serialization of two concurrent FKHs (6).

Meshing is straight forward to implement when the meshed KH is a PKH with no local persistent data state.

6.4. Parallel Processing

The Schemer architecture is a natural for a parallel processing. The communication model provides the mechanism where the flow of data is separated from the flow of control (6). The data messages are queued in

the synchronous input ports while the control tokens are the value of the **:action** property (`kill`, `resume`, `suspend`).

7. References

1. Buchanan, B.G. and Shortliffe, E.H., Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (Addison-Wesley, Reading, MA, 1984).

2. Corkill, Daniel D., Gallagher, Kevin Q., and Murray, Kelly E., GBB : A Generic Blackboard Development System., Proceeding AAAI-86 (1008-1014) 1986.

3. Erman, L.D., Lark, J.S., and Hayes-Roth, F., ABE: An Environment for Engineering Intelligent Systems, Teknowledge, Inc., Report No. TTR-ISE-87-106, Nov., 1987.

4. Fehling Michael R., and Breese, John S., A Computational Model for Decision-Theoritic[sic] Control of Problem Solving under Uncertainty., Technical Memorandum No. 837-88-5 APR 1988.

5. Forgy, C.L., OPS5 user's manual, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1981.

6. Forrest, Stephanie., and Lark, Jay S., Parallel and Distributed Processing in ABE, Teknowledge Inc., Report No. TTR-ISE-88-101, Jan., 1988.

7. Hayes-Roth, Barbara., BB1: An architecture for blackboard systems that control, explain, and learn about their own behavior, Heuristic Programming Project, Report No. 84-16, December, 1984.

8. Hayes-Roth, Barbara., Garvey Alan., Johnson, M. Vaughan Jr., and Hewett M., A Layered Environment for Reasoning about Action, Knowledge Systems Laboratory, Report No. KSL 86-38, August 1986.

9. Laird J.E et al., SOAR: Architecture for General Intelligence, Artificial Intelligence 33 (1987).

10. Nii, H. Penny., Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, AI Magazine, Summer 1986.

11. Reid, Loretta G., Control and Communication in Programmed Systems, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1980.

12. Tek Common Lisp User Manual, December 1987