THE DESIGN AND AN EXAMPLE USE OF HEARSAY-MI

Lee D. Erman
USC/Information Sciences Inst.
Marina del Rey. CA 90291

Philip E. London
USC/Information Sciences Inst.
Marina del Rey. CA 90291

Stephen F. Fickas
USC/Information Sciences Inst.
Marina del Rey. CA 90291
and
UC Irvine
Irvine. CA 92717

Abstract

Hearsay-III provides a framework for constructing knowledge-based expert systems. While Hearsay-III makes no commitment to any particular application domain, it does supply a variety of generally applicable facilities. These include representation primitives and an interpreter for large-grained, flexibly schedulable production rules called knowledge sources. A detailed overview of the motivations behind Hearsay-III and the facilities it provides are presented. Finally, an application of Hearsay-III is described.

1. Introduction

Hearsay-III is a domain-independent framework for knowledge-based expert systems. That is. rather than addressing the problems in a specific application domain Hearsay-III provides a "bare" architecture in which to cast an expert problem solver for a chosen domain. In this sense, it is similar in spirit to EMYCIN [van Melie 79] and AGE [Nii 79], and other "expert-system-building systems". However. Hearsay-III differs substantially in the specific representation and control regimes it makes available to the expert-system builder.

Although Hearsay-III is specifically *not a speech-understanding system* (and we know of no one who expects to use it for building a speech understanding system), it draws strongly on the architectures of the Hearsay! [Reddy 73] and Hearsay-II [Erman 80] speech-understanding systems As was intended by the choice of its name, Hearsay-III can be viewed as an extension along some dimensions of the Hearsay-II architectural style, and as a generalization of it along others. The concepts of large grained, modular *knowledge sources* and system-wide communication via a structured global *blackboard* were attractive to us because they provide a major first step toward achieving our design goals for Hearsay-III.

This paper presents the motivations behind the design of Hearsay-III, a detailed overview of its architecture and facilities. and illustrations, via examples, of use of its features Although we concentrate on the novel aspects of Hearsay-III. we do not attempt to classify each feature as being new or from HearsayII: (Balzer 80a) presents an overview of Hearsay-MI with such an orientation.

This research was supported by Defense Advanced Research Projects Agency contract DAHC15 72 C 0308 Views and conclusions contained inthis document are those of the authors and should not be interpreted as repiesenting the otf<oai opinion or policy of DARPA. the U S Government or any other person oi 2;:~:. connected "ith them

The overall design goal for Hearsay-III is the development of representation and control facilities with which a user can construct an expert system for his chosen domain. The specific attributes we want our system-ouilding system to embody include:

— Facilities to support codification of diverse sources of knowledge. We have avoided building into Hearsay-III any commitment to a class of application domains (such as medical diagnosis) which might allow some specificity in the language for describing sources of knowledge Instead, we attempt to provide as much generality as possible in the types of knowledge that might be brought to bear on a problem from the chosen application domain.

— Facilities to support application of these diverse sources of knowledge. Beyond mere application of the knowledge sources, an important design goal is to allow flexible coordination of the knowledge sources in their pursuit of an acceptable solution.

— Facilities to represent and manipulate competing solutions which are *incrementally* constructed. This aspect of the Hearsay-III architecture distinguishes it from the "diagnosis-system-building system", such as KAS[Duda 78]. EMYCIN [van Melle 79], and EXPERT [Weiss 79].

—Facilities for reasoning about partial solutions. That is. not only does Hearsay-III allow for incremental construction of competing solutions, but it also supports in a straightforward way the ability to reason about and manipulate those solutions during the various stages of their construction.

-Facilities for describing and applying domain dependent consistency constraints to the competing partial solutions Thus, the system supports application of knowledge globally so as to aid in reducing the search for a solution

- Support for long term, large system development, and in particular, experimentation with varying knowledge for the application domain, and varying schemes for applying that knowledge.

In summary, our goal for Hearsay-III is to develop, debug, and experiment with theories of domain expertice. One important area we do not emphasize as a goal for the Hearsay-III design is performance of the application system. It is intended that one use Hearsay-III to gain an understanding of the problem-solving principles of a chosen domain •• to study the domain. Later, it may be necessary to use a more efficient formalism to construct a performance system for the domain.

2. The Architecture of Hearsay-III

2.1. The Underlying Relational Database

Hearsay-III is built on a foundation consisting of a relational database system and its corresponding control facilities. The database language is called AP3 (Goldman 78] and is embedded in Interlisp [Teitelman 78]. As will be seen in subsequent sections Hearsay-III relies critically on the facilities provided within AP3.

The AP3 database is similar in structure to those available in the PLANNER-like languages [Hewitt 72]. but it also includes strong typing on assertion, retrieval and parameter passing in function calls. The type facility in AP3 is available to a Hearsay-III user for application domain modeling in addition to being used to advantage within the Hearsay-III system itself. The Hearsay-III blackboard (Sec. 2.2} and all publicly accessible Hearsay-III data structures are represented in the AP3 database. Additional annotations required by the application knowledge sources may also be placed in the AP3 database. Because knowledge source triggers are implemented as uniformly represented AP3 demons, modification to the database gives rise to knowledge-source activity (as described in Sec 2.3).

AP3 also makes available to Hearsay-III applications a context mechanism similar to those found in AI programming languages such as QA4[Rulifson 72] and CONNIVER [McDermott 74]. Hearsay-III supports contexts in such a way as to make them an integrated part of the reasoning mechanisms made available to an application. This feature is somewhat unique among expert-system writing systems. The context mechanism supported in Hearsay-III allows reasoning along independent paths which may arise both from a choice among competing knowledge sources and from a choice among competing partial solutions.

The AP3 database system also provides facilities for inference rules and constraints. These facilities, in addition to being used in the implementation of Hearsay-III itself, are also available to the user for encoding global domain-dependent relationships. The interaction of constraints and contexts is supported by Hearsay ill in that reasoning in a context that produces a constraint violation results in the context being flagged as *poisoned* (see Sec 2.3)

2.2. Blackboard Structure

The *blackboard* is the central communication medium provided by Hearsay-III. It is used by an application program as a repository for a domain model, for representation of partial solutions, and for representation of pending activities Hearsay-Ill supports the representation on the blackboard of graph structures consisting of structured nodes called *units* and labeled arcs called *roles.* The blackboard is segmented into two: the *domain blackboard and* the *scheduling blackboard.* The domain blackboard is intended as the site for competence reasoning, while the scheduling blackboard is intended as the site for performance reasoning. The application writer can further subdivide each of these blackboards.

2.2.1. Units

Blackboard units are the fundamental components of the representations built by application programs in Hearsay-III. Units are typed AP3 objects: their types are called *unit-classes.* In fact, the segmentation of the reasoning space into distinct blackboards is accomplished simply as the decomposition of the unit-class *Unit* into several distinct subclasses. Thus, the domain blackboard consists solely of units of class *Domain-Unit* (and its subclasses): the scheduling blackboard consists solely of units of class *Scheduling-unit,* When desired, access can be restricted to a given blackboard simply by using type-restricted AP3 database retrievals

2.2.2. Choice Sets

Units have structure in addition to their types One interesting feature of units is that they can be augmented to explicitly represent unresolved decisions Such units are called *choice sets* Associated with a choice-set unit is a set of alternatives or a generator of alternatives (or both). A choice set can be viewed as a partial elaboration of a decision point; the alternatives represent still further elaborations (and they themselves might be choice sets) Thus, competing problem solutions may be represented with a single locus. Furthermore, structure common to all alternatives may be factored out and associated with the choice-set unit itself. The choice-set representation allows for the representation of decisions to be data about which the system can reason.

Hearsay-III provides two mechanisms for resolving the ambiguity represented by a choice set. These mechanisms interact in an integrated fashion with the context mechanism of AP3. The first mechanism is called a *deduce-mode Choose* of the choice set. An application program may perform a deduce-mode Choose when it has conclusive evidence that one alternative is the correct solution for the problem represented by the choice set and that there will be no desire to retract that choice based on further evidence In this case, the choice set is replaced by the alternative (i.e.. their properties are merged) in the context in which the choice is made In this context, all evidence that the choice set ever existed is eliminated and the blackboard structure appears as if this choice set was never there

The second choice mechanism is called an *assume-mode Choose.* An assume-mode Choose also replaces the choice set with a unit which represents a merge of the properties of the choice set and the chosen alternative. However, an assume-mode Choose makes these changes in a newly created context that is a child of the one in which the choice was made. The appearance of the blackboard structure in the new context is identical to that resulting from a deduce-mode Choose. The choice-set unit still exists in the parent context with structure modified only to eliminate the alternative just chosen. Thus, if subsequent reasoning indicates this alternative may not be best, it is possible to return to the original context and select a different alternative

2.2.3. Acceptance

Units have associated with the a further attribute called *acceptance.* Acceptance can be thought of as the process of assimilating a unit into larger structure and verifying that it is appropriate in that structure. More simply. Hearsay-III allows the application writer to associate with each unit class a collection of procedurally defined predicates, called V*audator, CanonicaHzer. Uniqueness-Determiner. Conflict-Determiner,* and *integrator.* Each time a unit is created or is marked unaccepted by a KS. the acceptance routines defined for the unit s class are run. If all succeed, the unit is marked as *accepted.* If any fail, the unit is marked as being *unacceptable:* this usually results in the currently active context being poisoned (see Sec 2.3) Until a unit has been accepted. Hearsay-III prevents KSs from triggering on it

"The names are merely intended to be suggestive of no., they are to be used

2.2.4. Component Roles

As mentioned earlier. Hearsay-III supports the construction of labeled graphs on the blackboard Units are the nodes in those graphs The labeled arcs are called *component roies* (or simply *roles),* and are represented as typed relations connecting two units The typing of roles is of significant convenience, because it allows the use of type-restricted AP3 retrievals to simplify searching the structure. Roles, in addition to being typed, are also placed in classes called ro/e *sets.* Role sets are used for two purposes. First, they define distinct component hierarchies in which units are related by the transitive closure of the roles in a given role set This allows the supression of detail along chosen dimensions when examining the blackboard structure. The second use of role sets relates to consumption, discussed next.

2.2.5. Consumption

Hearsay-III supports a facility for describing mutual exclusion of units in an aggregated blackboard structure This is accomplished by prohibiting any structure in which two units are both components of a third unit (by transitive closure over a role set), while at the same time those two units are declared to *consume* a fourth This facility allows a convenient form for expressing the undesirability of using the same partial solution or interpretation for different purposes in an overall solution

2.3. Knowledge Sources

The domain specific knowledge for an application built in Hearsay-Ill is embodied in *knowledge sources* (KSs) Each KS can be thought of as a large-grained production rule: it reacts to blackboard changes produced by other KS executions and in turn produces new changes.

To define a KS, the user provides a *triggering pattern, immediate code,* and a body. Whenever the pattern is matchabte on the blackboard. Hearsay-Ill creates an *activation record* unit for the blackboard and runs the immediate code At some later time, the activation record may be selected (see Sec. 2 4 about scheduling) and *executed,* i.e.. the body, which is arbitrary Lisp code, is run. In more detail:

- The triggering pattern is expressed as an AP3 pattern As such it is a predicate whose primitives can be AP3 fact templates and arbitrary Lisp predicates composed with AND and OR operators. Whenever the AP3 database (which includes the Hearsay III blackboard i.e.. the units and roles) is modified such that any of the AP3 templates in the pattern is matched, the entire pattern is evaluated If the entire pattern matches an activation record is created and has stored in it the KS s name, the AP3 context in which the pattern matched[3] (called the *triggering context).* and the values of the variables instantiated by the match.

- At the point the activation record is created the immediate code of the KS is executed This code which also is arbitrary Lisp code, may associate information with the activation record that may be of value later in deciding when to select this activation for execution In addition, the immediate code must return as its value the name of some unit class of the scheduling blackboard. The activation record is then placed on the blackboard as a unit of that class The immediate code is executed in the triggering context and haa available to it the instantiated pattern variables.

- At some subsequent time, the systems base scheduler (see below) may call the Hearsay-Ill *Execute* action on the activation record. The usual result of this is for the body of the KS to be run in the triggering context and with the pattern variables instantiated If, however, at the point of execution, the triggering context of the activation is poisoned and the KS haa not been marked as a *poison handier,* the body is not *run:* rather, the activation record is marked aa *awaiting unpoisomng.* and wm have its status reverted to *ready* if the poieon status of the context is ever removed.

Each KS execution ia indivisible it runs to completion and is not interrupted for the execution of any other KS activation This insulates the KS execution and simplifies the coding of the body: there need be no concern that during a KS execution anything on the blackboard is modified except as effected by the KS itself.

2.4. Scheduling

Hearsay-III is intended for use in domains in which scheduling schemes are likely to be complex. Also, the application writer is not expected to have a good a priori notion how to accomplish the scheduling. Thus he will need to be able to experiment freely with various schemes. Since we view the scheduling problem itself as having characteristics similar to the domain problem, we feel the Hearsay-Ill blackboard-oriented knowledge-based approach is appropriate for its solution aa well and thus supply the same mechanisms for its solution.

Because of the indivisibility of KS execution, the scheduling problem in Hearsay-III can be stated as follows: At the end of each KS execution, determine, from the state of the system, the KS activation to execute next. To help solve this problem, several concepts features, and mechanisms are useful:

- As described above, the time of execution of a KS body is delayed arbitrarily long from its triggering with the activation record unit, on the scheduling blackboard as the mechanism for representing the activation Also the immediate code of the KS is run on creation of the activation record, allowing KS specific scheduling information to be added to the activation record

— Some knowledge-sources, termed *scneauimg rSs.* may make additional changes to the scheduling blackboard to facilitate the selection of activation records. Scheduling KSs may respond to changes both on the domain blackboards and on the scheduling blackboard including the creation of activation records. The actions they may take include associating information with activation records (e.g.. assigning and modifying priorities) and creating new units to represent meta-information about the domain blackboards (e.g.. pointers to the current highest-rated unite on the domain blackboard). The scheduling blackboard is the database for solving the scheduling problem

• The application writer provides s *base scheduler* procedure that is called by Hearsay-III after startup and actually calls the primitive Execute operation for executing each selected KS activation. We intend the base scheduler to be very simple: most of the knowledge about scheduling should be embodied in the scheduling KSs For example, the base scheduler might consist simply of a loop that removes the first element from a queue, maintained by scheduling KSs. and calls for its execution. If the Queue is

[3] in AP3 the content in which a • pattern matches is defined to oe the least general context which each of the pattern pa'ti has mstched

ever empty, the base scheduler simply terminates marking the end of system execution

Hearsay-III provides a default base scheduler it is comoosed of two functions, either of which can be replaced by the application writer. The default outer *base scheduler* repeatedly calla the default *inner base scheduler* and expects it to return a list of activation record units. (When the inner base scheduler returns the empty list, the outer base scheduler exits, hatting system execution.) The outer base scheduler executes each activation record in turn If the KS executed is a scheduling KS and its execution returns a list of (non-scheduling) activation records, the outer base scheduler immediately executes each of thoee activation records in turn. Each time the default inner base scheduler is called it non-deterministically chooses one ready scheduling KS activation record, or. if there are none, one non scheduling KS activation record. The default inner base scheduler is particularly trivial and is expected to be replaced in any serious application: the default outer base scheduler is likely to be a reasonable skeleton for many applications

3. An Example of Use

To illustrate the use of Hearsay-III as an implementation language for expert systems we describe here the implementation of the Jitterer problem-solving system (Fickas 60]. The problem addressed by the Jitterer is the automatic transformation of program parse trees The Jitterer maps a given parse tree (initial state) into a new parse tree (the goal state) by the application of a sequence of equivalence-preserving transformations The initial parse tree and the intermediate and final parse trees generated by the transformation sequence are called *program development states.* The description of the goal state is supplied by the user.

The Jitterer is one component of a *Transformational implementation* system [Balzer 80b] which allows a user to semi automatically refine and optimize a high-level program specification into an efficient implementation. An example of an optimization step a Transformational Implementation user might attempt is the merging of two set enumeration loops. Before actually executing the merge step the user might call on the Jitterer to reach a (goal) atate in which the two loops a) are adjacent, b) generate the same sets, and c) do not rely upon or affect the enumeration order of the set elements. If the Jitterer is successful, the uaer can execute the merge step and achieve the desired optimization.

Each transformation is composed of 1) a *left-hand-aide pattern (or* simply *LHS)* that must match a portion of the current program development state. 2) zero or more *enabling conditions* that must hold in the LHS match context, and 3) a set of *actions* to perform when 1 and 2 have been satisfied. The application of a transformation generates a new. semantically equivalent, program development state.

The Jittering system faces several interesting problems:

- Many transformations the catalog may be applicable in a given development atate. Further, many transformations have a corresponding inverse transformation, allowing infinite sequences.

-Establishing the enabling conditions of a transformation may be coatly. in both machine time and user effort.

— Each Jittering problem has in general more than one solution (sequence of transformation applications leading to the goal state) Metrics must be identified for ordering competing solutions [4]

3.1. Design of the Jitterer

The Jitterer s basic problem solving mechanism is a backward chaining, best first search. This choice helps alleviate the problems associated with the large transformation fan-out. the potentially high cost of establishing enabling conditions, infinite paths and solution ordering

The Jitterer makes two types of control decisions a selection from among the competing partial solution paths of the next path to extend, and a selection from among competing transformations of a transformation for continuing the chosen path In order to limit search, rules are used to guide both K-nds of decisions A" example of a path selection rule is "if a path s ieng:r< exceeds PathThreshoid. suspend it", where PathThreshoid has been determined experimentally. An example of a transformation selection rule is "If a transformation has side-effect X lower (raise) its desirability'

These selection rules reference features extracted from the current state as well as features predicted about the effects of possible selections Features referenced by the path selection rules include current path cost predicted cost to solution number of transformations applied predicted number of total transformations needed current status (dead suspended alive complete solution) and solution compatability For transformation selection, features of interest include transformation side-effects and predicted transformation cost, as computed in both machine time and uaer effort Feature information can be computed on demand or stored and maintained explicitly: the latter approach waa chosen because of perceived recomputation costs

A transformation can be applied to a program development atate only after its LHS has been matched and its enabling conditions have been established A straightforward approach to establishing these conditions for a single transformation application would lump all teats into a single scheduiable activity Given the potentially high cost of establishing enabling conditions this approach is too inflexible It may be more efficient to order the establishing of the enabling conditions: an attempt to establish one enabling condition may provide information which will lead to the suspension or abandonment of the transformation application Thus we require that the establishment of each enabling condition be a separate scheduiable activity

3.2. Hearsay-III Implementation of the Jitterer

In this section, we describe how each component of the Jitterer is implemented in Hearsay-III.

3.2.1. State/Space Representation

The Jitterer design requires two collateral spaces: the program development space, generated by transformation applications and representing various program development states, and the reasoning space, generated by the best-first and backward chaining search and representing partial solution paths and goal/

One Solution metric it ho* **II a Solution fits in ..»tn the uftt' S "*?<$ g'rr* development ttrateg> For example an> Jitterer produced solution ••'*•: i undoes a previous optimization o< prevents a future optimization must be given ic.\ pno» t> To compute this metric the Jitterer must be able tc analyze past dev*iopment steps and predict future development steps, the latter pressing ©t :..s problems

subgoal relationships. By using Hearsay-IIIs unit-class mechanism, the class of Domain-Units can be subdivided into *Reasoning-Units* and *Development-Units,* and thus we implement the two spaces as a segmentation of the domain blackboard. Although the reasoning space references units in the development space, the two spaces are essentially independent.

A state in the reasoning space is an AND/OR goal tree. The goal tree is built from Reasoning-Units *{goa'unit, transformation-unit)* and component roles *{sub-gcai. achieves).* An OR node represents the choice among competing transformations. An AND node represents the set of goals (transformation applicability checks) that must be satisfied in order to apply a particular transformation. The Hearsay-in choice-set mechanism (see Sec. 2.2.2) provides a framewor* both for structuring the set of competing transformations and for managing child contexts assooated with the choice. An assume-mode choose is used. spawning new reasoning states (contexts) when a transformation is chosen.

A state in the program development space represents the entire program parse tree at a particular stage of development. The parse tree is built from Development-Units (e.g.. *loop-unit, assignment-unit)* and component roles (e.g.. *predicate, then-clause, loop-body).* In the development space, there is no notion of a choice set, rather, simply a recording of various program development paths. The application of a transformation generates a new Hearsay- III/AP3 context. Note that there is no need to copy the program development state (i.e.. the syntax tree) into the new context; the Jitterer relies on the context inheritance mechanism and thus needs to represent explicitly only those portions of the structure that are new or modified

3.2.2. Transformation Representation

Each transformation is implemented as a domain KS (henceforth, *transformation KS).* Because of the Jitterer s backward-chaining control, the trigger of a transformation KS corresponds to the action portion (translated so to match the goals of the reasoning space) of the corresponding transformation. The immediate code of a transformation KS is responsible for setting up as subgoals the LHS pattern to be matched and the enabling conditions to be established: we describe this further in tne next section The body. when executed, creates a new context (program development state) and makes the appropriate modifications

3.2.3. Control Knowledge

As described in Sec 3.1. the Jitterer uses rule-based selection knowledge to control search Each selection rule is implemented as a scheduling KS (see Sec. *2.4).* For example, one selection rule treats the desirability of a transformation as a function of the side-effects it produces Figure 3 1 shows the scheduling KS form for one instance of this rule namely that a transformation that unfolds a function in-line has the deleterious side-effect of flattening the program structure.[5]

```
(Declare-SKS structural-flattening (Op)
   Trigger: (AND (Competi ngOperator OP)
                 (SideEffect Op UnfoldsFunction))
   immediate Code: Operator-orderinglevel
   Body: (Dec reaseDes i rabilityOP))
```

Figure 3-1: A transformation selection rule

'The actual AP3'Hearsay-III syntax has been modifiec here for da'it.

Earlier we mentioned a rule that checks for a path growing beyond a PathThreshold. Figure 3-2 shows the scheduling KS form of this rule Note that the evaluation of the immediate code of the two scheduling KSs places their corresponding activation records on separate scheduling levels. The Jitterers base scheduler gives Path-state-change-level priority over Operator-ordering-level and hence path suspension is attempted before transformation ordering.

```
(Declare-SKS lengthy-oath (Path)
   Trigger:
      (AND
         (CompetingPath Path)
         (> (CurrentPathLength Path) PathThreshold))
   immediate Code: Path-state-change-level
   Body: (MerkAsSuspended Path))
```

Figure 3-2: A path selection rule

As discussed in Sec. 3.1. the Jitterer's selection rules reference certain computed problem-solving features This information is stored as *auxiliary reasoning structures* attached to the relevant units on the scheduling and domain blackboards. For example, an auxiliary reasoning structure for path selection is attached to each goal-unit in the reasoning space[6] The scheduling KS in Fig. 3-2 makes reference to the path length of Path. Current path length information is stored in and retrieved from the auxiliary structure associated with Path s frontier goal.

To descibe the auxiliary reasoning structure used for transformation selection, we must look more closely at the implementation of transformations as KSs. Hearsay-III divides a KS application between triggering and execution. As described Sec. 2.4. once a KS is triggered, an activation record is created on the scheduling blackboard where it resides until executed oy the application's base scheduler. The Jitterer selects from among the set of activation records of triggered transformation KSs. Thus, this set that must be ordered. The immediate code of each transformation KS is responsible for attaching an auxiliary reasoning structure to the corresponding activation record. For example, the scheduling KS in Fig 3-1 makes reference to the side-effects of a transformation. These side-effects are among the information stored in the auxiliary reasoning structure attached to the corresponding activation record. The immediate code is also responsible for adding the activation record to the choice set of the appropriate OR goal in the reasoning space.

3.2.4. The Scheduling of Enabling Conditions

The Jitterer design calls for the separate scheduling of each enabling condition. This is implemented in Hearsay-III in the same manner described for auxiliary reasoning structures in the previous section: the immediate code of a transformation KS augments the activation record with the set of enabling conditions The scheduling KSs and the base scheduler order the set and determine when to attempt establishment of the individual conditions In some cases it may be undesirable to execute an activation record even though all enabling conditions have been established A few scheduling KSs look for these cases ard flag the activation record accordingly in general the ability to divide problem solving into such fine-grained activities has been helpful for the Jitterer

'Each goal is ne.-.ed as the frontier of a pair, from the 'oot goat

3.2.5. The Scheduling of Scheduling KSs

The Jitterer's selection rules, implemented as scheduling KSs. help order the path and transformation search space. However, we are left with the problem of scheduling the scheduling KSs The Jitterer's scheduling blackboard is divided into a set of mutually exclusive, prioritized scheduling levels. Each scheduling KS is assigned to a single level by its immediate code The Jitterer's base scheduler returns, for execution, an activation record from the highest level on which activation records reside

For many levels, intra-level scheduling consists simply of executing activation records in arbitrary order until none remain on the particular level Operator-ordering level, referenced in Fig. 3-1. is scheduled in this way. However, some scheduling levels provide structures for ordering their activation records One example is the level on which the activation records of competing transformations are placed; the transformation selection rules maintain an ordered list. Another level. *Report-solution-level,* discussed below, provides a queue for recording the order of activation record appearance. When an activation record is placed on this level, a "queue maintenance" scheduling KS adds it to the end of the queue. Given the best-first search the final queue will contain spokesmen for all solutions found by the Jitterer in their order of preference

We have previously seen in Fig. 3-1 and Fig 3 2, two of the defined scheduling levels Another example is the scheduling blackboards highest priority level.[7] *Report-solution level.* Because the KS that detects complete solutions is assigned to this level by its immediate code, its activation records are executed immediately following any KS activation that satisfies its triggering pattern. Thus, the Jitterer reports a solution to the user as soon as it is found. If instead the Jitterer was to find all solutions to a problem before reporting any Report solution level should be made the lowest priority scheduling level.

3.2.6. Use of the Acceptance Routines

The Jitterer's scheduling KSs normally determine the difficulty of achieving a particular goal, attaching appropriate information to the goals auxiliary structure However, detecting Jittering goals that are inherently impossible is performed by a Validator acceptance routine. When a new Jittering goal is posted in the reasoning space, the appropriate Validator determines whether it falls into this special class. If so. the goal is marked as impossible before being considered by the rest of the system. Currently, only easily determined impossible goals are handled by the Validator routines. Thus, more sophisticated tests about impossible goal states are not included in the Validator routines because we want the system to be able to schedule these costly activities. Once a goal unit is created, all Validator routines pertaining to that goal are run.

The Jitterer applies a set of normalization and simplification rules each time a new program development state is generated (i.e.. whenever a parse tree is changed).[8] This cleanup process has been implemented in Hearsay-III through the Canonicalizer acceptors: each node type (unit-class) of a parse tree (e.g.. loop-unit, assignment-unit, conditional-unit) has an associated Canonicalizer: each Canonicalizer embodies the set of clean-up rules for its node-type It is the responsibility of a transformation changing the parse tree to mark the relevant nodes (units) for (re-Acceptance (see Sec. 2.2 3)

4. Conclusion

Hearsay-III was exercised initially on two small test cases: a cryptarithmatic problem and a cryptogram decoding problem In addition to the Jitterer. two major implementation efforts are currently underway The first of these is the reimplementation of SAFE, a system for constructing formal specifications of programs from informal specifications [Balzer 78]. Second. Hearsay-III is being used as the basis for a system for producing natural language descriptions of expert system data structures [Mann 79].

In some problem domains, the major implementation effort will be the encoding of the competency portion of the problem-solving system in the form of KSs. The performance portion of the system may not require sophisticated scheduling techniques sufficing on a reasonably tunable set of hardwired scheduling regimes, such as the AGE system provides In these cases, the Hearsay-III system may seem less useful since the user will have to build-up all but the most primitive control structures from scratch. Although such is currently the case, as more and more projects use Hearsay-III. the stock of different application schedulers will grow. It seems reasonable to assume that with a little worn these schedulers can be generalized to provide a new user with a library of Hearsay-III schedulers from which to choose. A new problem domain may be able to use an existing scheduler directly or as the foundation for a more application specific scheduler

Our experience to date supports our belief that the Hearsay-III architecture is a helpful one The separation of competence knowledge from performance knowledge helps in rapidly formulating the expert knowledge required for a solution. The flexibility that the Hearsay-III architecture gives toward developing scheduling algorithms will undoubtably go a long way toward simplifying this difficult aspect of the overall problem-solving process

Acknowledgments

Hearsay-III was originally designed was by Bob Balzer. Lee Erman. and Chuck Williams, with contributions by Jeff Barnett. Mark Fox. and Bill Mann Subsequently. Phil London and Neil Goldman contributed significant design modifications Lee Erman and Phil London implemented and maintain Hearsay-III. AP3 was designed, implemented and maintained by Neil Goldman. Steve Fickas designed and implemented the Jitterer Neil Goldman. Bill Mann. Jim Moore, and Dave Wile have also served as helpful and patient initial users of the Hearsay-III system

This is actually not quite the highest level even higher are those used by the scheduling KSs that do mtra-ievel structuring

[8] While these rotes also make changes to the parse tree they do not cat:-: •** generation oi ne.. development states

References

[Balzer 78] Balzer. R_M N. Goldman, and D. Wile. "Informality in Program Specifications." *IEEE Trans. Software Eng.* SE-4. (2). March 1978.

[Balzer 80a] Balzer, R.. L D. Erman. P. London, and C. Williams, "Hearsay-III: A Domain-Independent Framework for Expert Systems." in *Proc. 1st National Conf. on Artificial intelligence,* pp. 108-110. Stanford. CA. August 1980.

[Balzer 80b] Balzer. R.. N.. "Transformational Implementation: An Example." *IEEE Trans. Software Eng.*. November 1980. (Also appeared as Technical Report. USC/Information Sciences Institute. RR-79-79)

[Duda 78] Duda, R O.. P. E. Hart N J. Nilsson. and G. L. Southerland. "Semantic Network Representation in Rule based Inference Systems." in D. A. Waterman and F. Hayes-Roth (eds.). *Pattern-Directed Inference Systems,* pp. 203-222. Academic Press. New York. 1978

[Erman 80] Erman. L. D.. F. Hayes-Roth. V. R. Lesser, and D R Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty " *Computing Surveys* 12. (2). June 1980. 213-253.

[Fickas80] Fickas. S.. "Automatic Goal-Directed Program Transformation." in *Proc. 1st National Conf. on Artificial Intelligence,* pp. 68-70. Palo Alto. CA. August 1980.

[Goldman 78] Goldman. N . AP3 User's Guide 1978 Unpublished Memorandum. USC/Information Sciences Institute.

[Hewitt 72] Hewitt. C. E.. *Description and'Theoreticai Analysis (Using Schemata) of PLANNER A Language for Proving Theorems and Manipulating Models m a Robot.* MIT Al Laboratory. Technical Report TR258 1972

[Mann 79) Mann. W. C. and J. A Moore. *Computer as Author •• Results and Prospects.* USC/Information Sciences Institute. Technical Report RR-79-82. 1979

[McDermott 74] McDermott. D.. and G. J. Sussman. *The CONNiVER Reference Manual.* MIT Al Laboratory. Memo 259a. 1974

[Nil 79] Nn. H. P.. and N Aiello. "AGE (Attempt to Generalize): A Knowledge-based Program for Building Knowledge-based Programs." in *Proc 6tn int. Joint Conf. on Artificial Intelligence,* pp. 645-655. Tokyo. 1979

[Reddy 73] Reddy. D. R.. L. D. Erman. R. D. Fennel!, and R. B. Neely. "The Hearsay Speech Understanding System: An Example of the Recognition Process," in Proc. *3rd int. Joint Conf. on Artificial intelligence,* pp. 185-193. Stanford. CA. 1973.

[Rulifson 72] Rulifson, J. F., R. J. Waldinger, and J. A. Derksen. "A Language for Writing Problem-Solving Programs." in *IFIP 71,* pp. 201-205, North-Holland, Amsterdam. 1972.

[Teitelman 78] Teitelman, W.. *interiisp Reference Manual,* Xerox Palo Alto Research Center. 1978.

[van Melle 79] van Melle. W.. "A Domain-independent Production-rule System for Consultation Programs." in *Proc. 6th int Joint Conf on Artificial Intelligence,* pp. 923-925. Tokyo. 1979.

[Weiss 79] Weiss. S. M and C. A Kulikowski "EXPERT: A System for Developing Consultation Models." in *Proc. 6th int joint Conf on Artificial intelligence* pp 942-947. Tokyo 1979