PLOT UNIT RECOGNITION FOR NARRATIVES

Wendy Lehnert
Cynthia Loiselle

COINS Technical Report 83-39

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

January 1984

This research was supported in part by the National Science Foundation under contract IST-8217502 and in part by the Office of Naval Research under contract N00014-83-K-0580.

ABSTRACT

Plot units present a memory representation particularly well suited to the task of narrative summarization. The detection of "top level" plot units and their connectivity from an "affect state map" enables us to identify the central ideas of a story. A computer program, PUGG (the Plot Unit Graph Generator) has been implemented which allows us to derive this information from the affective structure of the story.

Beginning with a brief introduction to the underlying memory representation, we procede to describe the mechanisms that enable PUGG to produce a listing of top level plot units and their connections to one another. Specifically, we explain how PUGG uses a read-integrate-predict loop to generate predictions for subsequent units based on the affective structure previously encountered. These predictions take the form of demons anticipating the structure of the unread portion of the affect state map. Plot units are recognized through a process of demon "movement". Additional features such as flexible plot unit definitions and causal link transitivity increase the power of the program enabling the recognition of more units displaying greater connectivity. Finally a "micro" version of the program is presented followed by a complete listing of the set of plot units currently in use.

TABLE OF CONTENTS

Section one: Memory Representation and Plot Units

I. MEMORY REPRESENTATION

Memory representation has always been a central concern for theories of text comprehension. While it is widely conceded that the conceptual concept of a text must be stored in memory in some form other than the original input sentences, there is no general agreement on exactly what representational techniques must be used. A variety of approaches have been proposed, ranging from predicate calculus formalisms [Kintsch 1974, Woods 1970], to case grammars [Rumelhart & Norman 1975, Fillmore 1968, Winograd 1972, Graesser 1981], to systems of primitive decomposition [Schank 1975, Wilks 1978, Lehnert 1979]. Each technique continues to attract a following of active practicioners, although there seems to be psychological evidence for primitive decomposition over more lexically-oriented representations [Gentner 1981].

A major impetus in recent research concerns the organization of large memory representations. A narrative is more than the sum of its individual sentences: readers supply their own inferences [Rieger 1975, Schank & Abelson 1977, Wilensky 1980], idiosyncratic interpretations, and personal belief systems [Carbonell 1978] during the understanding process. It has been estimated that the ratio of implicit information derived from a narrative to explicit information present in that text is something like 8:1 [Graesser 1981]. If a text understanding system (human or otherwise) is not generating these implicit assumptions, it is difficult to say in what sense the text has been understood.

As the problems of inference began to demand serious attention, the notion of an internal memory representation moved further and further away from sentence-driven propositions. While sentences continued to provide a necessary starting point, the amount of conceptual representation generated in response to any given sentence began to look less like a function of that sentence per se, and more like a function of causal connectivity, goal-oriented behavior, and expectation-driven knowledge structures. Large memory representations could be generated in response to simple 3-sentence stories [Cullingford 1978, Wilensky 1980], and computational models began to struggle with the organization of multi-layered representations, where various levels of memory representation specialize in describing physical event sequences, inferred goals for active characters, likely plans acting in the service of those goals, and affective reactions for important characters [Dyer 1983].

In the course of investigating complex strategies for multi-layered memory representations, we developed a level of representation that appears to be particularly well-suited for narrative text summarization. When one is confronted with the task of summarizing a narrative, it is possible to imagine two general strategies for memory retrieval. On the one hand, we can design some sort of "brute force" algorithm that examines all the information available in memory and then procedes to filter that information according to some extensive set of deletion rules and/or principles of focus. On the other hand, we might strive to design a memory representation that somehow makes it very easy to distinguish the important central concepts from the peripheral and secondary concepts.

With such a memory representation, the retrieval algorithms we need can be direct and simple. This is the approach we have taken in formulating the representational system of plot units.

II. AFFECT STATES AND PLOT UNITS

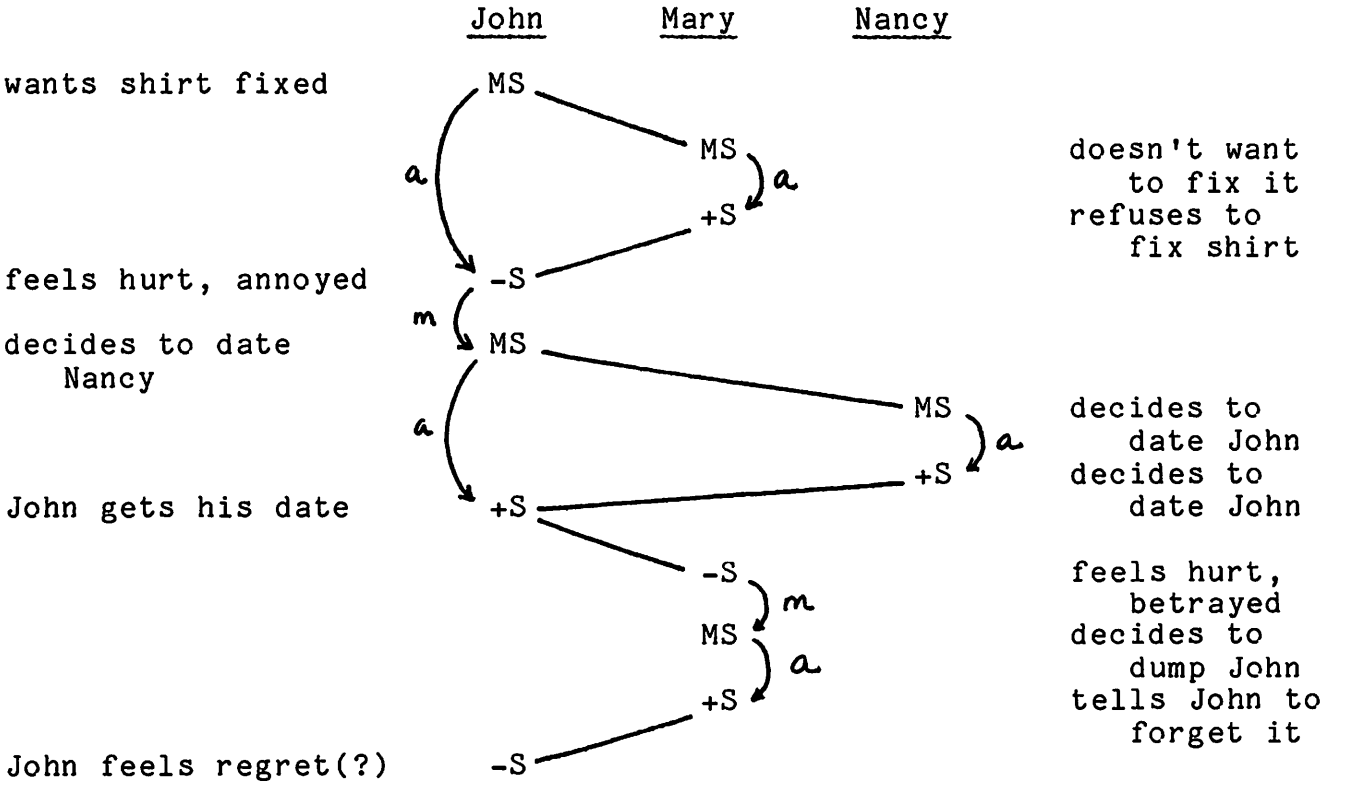
A plot unit is a simple configuration built up of three primitive affect states. In this system we need only distinguish generally positive reactions, negative reactions, and the more neutral mental states associated with desires, goals and plans. These three affect states will be referred to as (1) +S (positive states), (2) -S (negative states), and (3) MS (neutral mental states). For each character of a story, we create an affect state map for that character, where we record a chronological sequence of affect states experienced by that character. We expect that much of this information must be inferred by the reader, so we build this level of representation "on top of" lower levels of representation that specialize in various inference strategies. The BORIS system provides a good picture of what these lower levels of memory representation must entail [Dyer 1983].

An affect state map is structured in part by the chronology of the affect states experienced, but additional structure is present in the form of causal links joining pairs of affect states. There are four causal links used to join affect states, and each link type is restricted by a syntax that dictates in what ways two states can be connected. The links are (1) motivation links, (2) actualization links, (3) termination links, and (4) equivalence links (usually abbreviated as m-links, a-links, t-links and e-links.) A motivation

link can only be used to "explain" an MS in the sense that some other affect state has motivated the MS. So we can have a motivational link pointing from a +S to an MS to indicate that a positive state enabled the mental state. We can alternatively have a motivational link pointing from an -S to an MS whenever some negative reaction is driving a goal or plan. And we can also have an MS motivating another MS when we want to express a subgoal relationship or a goal-plan relationship. We will not describe the full syntax of our causal links here, although it is important to note that there are only 15 syntactically correct combinations of state-link-state triples. For a complete description of the syntax see [Lehnert 1982].

In addition to these four links which are used to connect affect states for a single character, we also use a cross-character link (c-link) to connect affect states experienced by two different characters. In this way, a story containing 5 major characters can be represented by five corresponding affect state maps, with causal connections between them wherever two characters interact or respond to one another. The best way to get a feel for the system is by looking at an example.

Suppose we have a story about John and Mary, in which John asks Mary to fix a rip in his shirt, and Mary says no. John then procedes to ask Mary's best friend Nancy for a date. Nancy agrees, and when Mary finds out about it, she dumps John altogether. Here we have a story involving three characters, and a number of cross-character interactions:

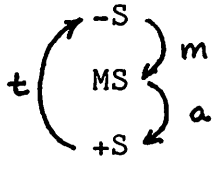


This story involves a number of actualized goals as well as a failed goal when John initially fails to get Mary to fix his shirt. We see two instances of goal motivation: the first occurs when John decides to date Nancy, the second when Mary decides to dump John. We also see some symmetry in the interactions between John and Mary. At first, Mary does something (refuses to fix his shirt) which affects John negatively. His reaction then motivates him to actualize a goal (dating Nancy) which will affect Mary negatively (she feels hurt). The same pattern is reciprocated when John does something (dates Nancy) which affects Mary negatively. Her reaction then motivates her to actualize a goal (dumping John) which will affect John negatively (he feels regret). These repeated patterns represent instantiations of retaliatory actions. Since it would be reasonable to summarize this story as one where Mary got even with John for something he did

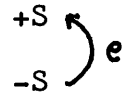
to her, it is crucial for us to recognize these instances of retaliation.

Affect state maps provide us with a level of representation which is strictly instrumental to the level we really want to access; we cannot design a retrieval mechanism for the summarization task at the level of an affect state map alone. To effectively summarize our narrative, we must derive a new level of representation from the affect state map. This higher derivative is called a plot unit graph, and the remainder of this paper describes the derivational process that allows us to extract plot unit graphs from affect state maps.

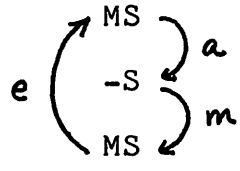
Within any given affect state map, we will see configurations of affect states and causal links that appear with some frequency across a variety of stories. These configurations correspond to intuitively recognizable situations, such as intentional problem resolutions, mixed blessings, perseverence after failure, and so on and so forth. Some of these configurations require only one character while others involve two characters (for example, an honored request or a reneged promise). A few of these are listed below to illustrate their affect-state level encodings:



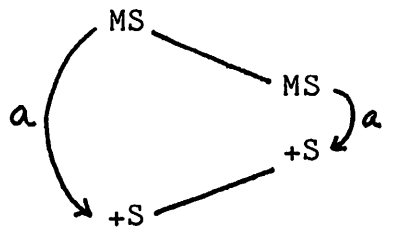
Intentional
Problem-Resolution



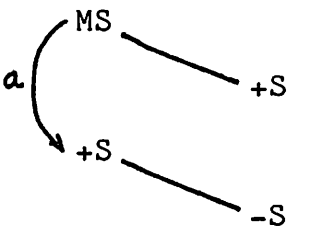
Mixed
Blessing



Perseverence
After-Failure



Honored-Request



Reneged-Promise

Configurations like these are called plot units. We have identified about 50 plot units, ranging from the 15 "primitive" plot units consisting of only state-link-state combinations to more complicated structures involving many affect states and connecting causal links. These units will typically overlap with one another at shared affect states, and we often see large plot units "subsume" smaller unit components when the larger unit contains all the affect states and link types present in the smaller unit. For the purposes of the plot unit graph, we are only interested in the "top-level" plot units, that is, those units that have not been subsumed by larger units.

To form a plot unit graph, we create a node in the graph for each top-level plot unit present in the affect state map, and we join two nodes in the graph whenever their corresponding plot units intersect at one or more affect states. The structure of this plot unit graph then reflects the essential connectivity of the top-level plot units found in the affect state map. We can readily examine the graph structure to see which units are strongly connected (and therefore central to the narrative), and which units are weakly connected (and peripheral to the core story). The actual indentification of central plot units depends on structural features of the graph [Lehnert 1983],

and may be somewhat more involved than finding the unit or units with maximal degree. But we will not need to go into those issues here.

The identification of top-level plot units in an affect state map is an interesting problem in its own right, and one that has been realized by a computer program, PUGG (the Plot Unit Graph Generator) which runs in linear time. We will now describe the processing strategies used by PUGG to derive plot unit graphs from affect state maps.

Section two: The Plot Unit Graph Generator

The Plot Unit Graph Generator takes as input a chronologically ordered list of affect states and their links (an affect state map.) PUGG processes this memory representation and returns a listing of all the top-level units and their relationships with one another. This provides us with the information we need to derive the plot unit graph structure of the source narrative.

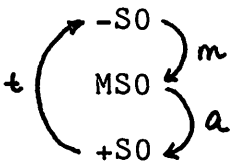
PUGG consists of two independent modules. The recognition module contains the central core of the program, where the identification of plot units occurs within a Read-Integrate-Predict (RIP) loop. This routine loops continuously, (1) reading a new affect state (2) integrating this information into what is already known, and (3) predicting the occurrence of more plot units later in the text. When all the input data has been read in and the plot unit identification is completed, PUGG enters its second half, the report module. This section prints summary statistics for the set of plot units just

identified, including a listing of the top level plot units and their relationships to each other.

I. RECOGNITION MODULE

A. READING PHASE

The recognition module constructs a record for each affect state as it is read in. This record contains the affect state's name (a unique subscripted identifier, i.e. MS2, +S0), the character experiencing the affect state, a listing of any links to or from other affect states and a short comment describing the state. States are read in chronologically so only previously defined affect states can be linked to the current state. Let us take the INTENTIONAL-PROBLEM-RESOLUTION unit as an example:



When formatted as input to PUGG, the above affect state map becomes:

more states?	state type	char	in-links?	from state?	link type	more in links?	out-links?	to state?	link type	more out links?
	-s	Ted	n				n			
y	ms	Ted	y	-s0	m	n	n			
y	+s	Ted	y	ms0	a	n	y	-s0	t	n
n										

(the description fields have been left off due to space limitations)

We read the information contained in the affect state map in the following manner: First, the initial affect state is entered (-S), followed by the name for the character experiencing that state (for this example let us suppose the character is named Ted.) Each state is given a unique name by the reading functions, so that the -S states entered will be (consecutively) named -S0, -S1, -S2, etc. Links into or out of this state are the next items requested. While there is a link out of -S0 pointing to MS0, this information will not be entered at this point because MS0 has not yet been defined. We will be able to establish this link shortly, however.

After any links have been entered, the read loop asks if any additional affect states are waiting to be read in. A "yes" begins the process anew with the next affect state. We read in the affect state (MS) and character's name (Ted) and again request links into or out of this affect state. At this point we pick up the M-link into MS0 from -S0. The A-link from MS0 to +S0 must wait for the next round since +S0 has not been created by the recognition module. On the third and final iteration, the state type (+S) and character (Ted) are encountered and a new state named +S0 is created. The A-link into +S0 from MS0 can now be recorded. We can also read in the T-link out of +S0 to -S0 since -S0 has already been created. When all the data for one affect state has been entered, PUGG prints a message informing the user that this state has been created along with whatever associated information is recorded for that state. The recognition module's trace for the INTENTIONAL-PROBLEM-RESOLUTION unit is included below:

Trace from Recognition Module:

Affect state -s0 has been created.

Name = -s0

Char = Ted

Links into -s0 from = nil

Links out of -s0 into = nil

Description = (problem)

Affect state ms0 has been created.

Name = ms0

Char = Ted

Links into ms0 from = ((m . -s0))

Links out of ms0 into = nil

Description = (idea)

Creating a new plot unit:

Name = unit0

Type = problem

Role(s) = (Ted)

Affect States = (ms0 -s0)

unit0 SUBSUMES nil

Creating demons looking for:

success-born-of-adversity

fortuitous-problem-resolution

```
Affect state +s0 has been created.

     Name = +s0
     Char = Ted
     Links into +s0 from = ((a . ms0))
     Links out of +s0 into = ((t . -s0))
     Description = (success)

Creating a new plot unit:

     Name = unit1
     Type = resolution
     Role(s) = (Ted)
     Affect States = (+s0 -s0)

   unit1 SUBSUMES nil

Creating a new plot unit:

     Name = unit2
     Type = success
     Role(s) = (Ted)
     Affect States = (+s0 ms0)

   unit2 SUBSUMES nil

Creating demons looking for:
                              fleeting-success
                              competition
                              killing-two-birds
```

Creating a new plot unit:

Name = unit3

Type = success-born-of-adversity

Role(s) = (Ted)

Affect States = (+s0 ms0 -s0)

unit3 SUBSUMES (unit2 unit0)

Creating demons looking for:

intentional-problem-resolution

Creating a new plot unit:

Name = unit4

Type = intentional-problem-resolution

Role(s) = (Ted)

Affect States = (+s0 -s0 ms0)

unit4 SUBSUMES (unit1 unit3)

B. INTEGRATION PHASE

As each state is read in it is integrated with those states previously defined. Cross-referencing links are generated for each link pointing from A to B so that a corresponding back-pointing link is created from B to A. Similarly, for each outgoing link coming from the current state, a corresponding incoming link is established. Thus for the example given above, when the M-link into MS0 from -S0 is read in, a backpointing M-link into -S0 from MS0 is created. And when the T-link out of +S0 to -S0 is entered, a forward-pointing T-link from -S0 to

+SO is established. These backpointers are not part of the representational system per se; they merely enable useful processing during the prediction phase. Having links pointing both up and down the affect state map allows for a process called "demon movement" to occur in both directions as well. (Demons and their movement will be discussed in the next section.)

C. PREDICTION PHASE

At each point in the input stream, the recognition module forms predictions for what it expects to see next. The affect state structure previously encountered determines the specific predictions formed. These predictions are instantiated as demons attached to their originating affect states. Demons use situation-action rules [Hayes-Roth and Waterman, 1978] utilizing structural information about the definitions of plot units. Since complex plot units are built up from more primitive units, the recognition of these smaller units enables us to anticipate more complex structures. So each time a primitive unit is identified, a set of predictions (demons) is spawned listing the structures we would need to encounter in order to recognize more complex units.

When we first read in the information defining a particular affect state, we spawn a set of demons looking for the primitive units based on that state type. Thus, encountering a -S affect state spawns a set of demons for those primitive units beginning with a -S state. In the case of a negative affect state, we create a demon looking for each of the following primitive plot units: PROBLEM, HIDDEN-BLESSING,

COMPLEX-NEGATIVE, RESOLUTION, NEGATIVE-TRADE-OFF, EXTERNAL-PROBLEM, NEGATIVE-REACTION and POSITIVE-REACTION (the last three units having a "wild-card" initial state: -S, +S or MS.) In the same way, once we've recognized the primitive unit SUCCESS, we generate a set of demons whose definitions begin with an instance of this unit. In the example trace above we see that the creation of a SUCCESS unit results in the formation of demons looking for the FLEETING-SUCCESS, COMPETITION and KILLING-TWO-BIRDS plot units. In other words, when we encounter a SUCCESS plot unit, we predict that we might also find one or more of these other units. By ordering our unit predictions hierarchically, we minimize the search time spent considering possible configurations. We will always consider the basic primitive predictions, but predictions for more involved configurations will only be loaded when we've seen enough structure to justify such expectations.

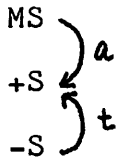
1. Demon Structure

a. Path Structure

Demons are records consisting of the name of the demon (unit-id), the character(s) concerned, a list of the affect states involved, any subsumed units and, most importantly, the demon's "path." A path is an ordered list of states and links which must be traversed in order for PUGG to recognize the plot unit in question. [Example: ((-S . ?X) (GIVES . M) (MS . ?X))] Any smaller units which form part of the definition for this unit are also included in the demon's path. Each of the items in a demon's path is called a step. Each step

describing a state or a subunit also indicates which character(s) must be experiencing the affect states specified in order for the definition to be realized.

An example should help clarify matters at this point. The definition for the plot unit FLEETING-SUCCESS is shown below.



One path for this unit is represented as:

((MS . ?X) (GIVES . A) (+S . ?X) (TAKES . T) (-S . ?X))

where the key words "gives" and "takes" are used to indicate the direction of the link involved. So here we have a path beginning at some mental state (MS) for a given character (?X). It then describes the A-link coming <u>from</u> the mental state to a positive state associated with ?X (the same character who experienced the mental state). Then we pick up a T-link coming <u>into</u> the positive state from a negative state, for the same character. If all these steps can be traversed, we have an instance of the FLEETING-SUCCESS plot unit.

Alternately, we could define the path for the unit SUCCESS to be:

((MS . ?X) (GIVES . A) (+S . ?X))

and use that in the path for FLEETING-SUCCESS as follows:

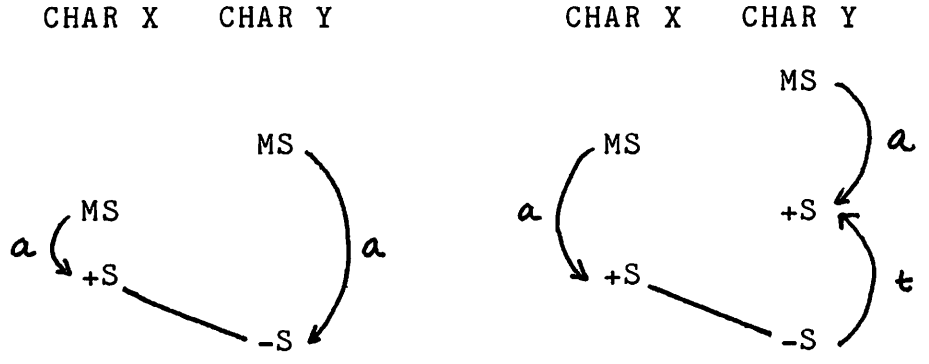
((SUBUNIT SUCCESS (?X)) (TAKES . T) (-S . ?X)).

Wherever possible, we use the subunit form of a path list. This

permits us to consider the demon predicting a FLEETING-SUCCESS unit only when a SUCCESS unit has been recognized, rather than whenever an MS state is encountered. We prefer this reduction because it (1) keeps demon propagation down to a minimum and (2) increases the probability of encountering the affect state structures needed to satisfy predicted paths.

b.) Flexible Plot Unit Definitions

Some plot units have more than one corresponding affect state map representation. For example, both maps below are encodings of the COMPETITION plot unit:



COMPETITION-1 COMPETITION-2

Whether the losing character experiences a temporary victory which is then lost (FLEETING-SUCCESS) or an immediate defeat (FAILURE), the concept of competition is maintained. In such a case we say that we have a "flexible definition" for the plot unit COMPETITION. We want either affect state structure to be recognized as an instance of the COMPETITION plot unit.

We need two demons for such flexibly defined plot units, one for each affect state map configuration [1]. The paths for these demons are listed below:

COMPETITION-1

```
((SUBUNIT SUCCESS (?X)) (GIVES . C) (-S . ?Y)
 (SUBUNIT FAILURE (?Y))
 (SUBUNIT NEGATIVE-REACTION (?Y . ?X)))
```

COMPETITION-2

```
((SUBUNIT SUCCESS (?X)) (GIVES . C) (-S . ?Y)
 (SUBUNIT FLEETING-SUCCESS (?Y))
 (SUBUNIT NEGATIVE-REACTION (?Y . ?X)))
```

By spawning both of these demons when a SUCCESS unit is identified, the recognition module forms predictions for both forms of this unit. Units with flexible definitions are often set up to have one definition as the main or default definition. We can ask the recognition module to look for plot units using either default definitions or flexible definitions.

2. Demon Management

All demons associated with a given affect state are stored on the property list for that state. Many demons are active at any point during the prediction phase of the RIP loop and each will be processed at least once for each iteration of the loop. When a demon successfully traverses a step in its path we say it "fires." And when a demon fires, it generates new information (such as the recognition of plot units) which may be needed by other demons. For example, it

---

[1] Or three demons if there are three possible configurations, etc.

may identify a primitive plot unit needed by another demon to recognize a complex unit. Therefore, if one demon fires, we cycle through the list of active demons again, reprocessing each to see if it might fire now that this new information is available.

The recognition module is responsible for updating active demons associated with affect states and reducing the demon's path to nil when all its predictions are met. The updating process is largely accomplished by moving active demons from one affect state to another. There are two forms of demon movement: progression and migration. Progression is the simpler of the two and will be described first, but before doing so we need to explain how we reduce a demon's path.

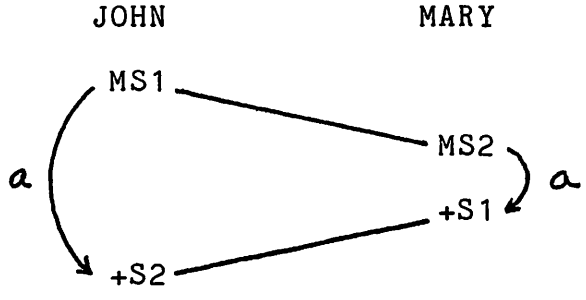
a.) Demon Path Reduction

As we read in each new affect state, the information it presents is compared with that given in each demon's path list. Using the example path for SUCCESS above for illustration, we note that once we have encountered an MS state for character JANE, the demon for the SUCCESS unit is spawned. The first step in this demon's path is (MS . ?X). Since this matches the information we have just read in, we can make a binding of JANE to ?X for this demon and the demon fires. We no longer need this step in the path list so we remove it leaving a path of just ((GIVES . A) (+S . JANE)), (i.e. we "pop" the path.) Note that the character binding occurs throughout the entire demon, rather than in the first step only. When we encounter a +S state for JANE with an incoming A-link from that previous mental state we can make another match against the path. We establish a match between the (GIVES . A) step in the path and the A-link we have just

read in, but before we remove it from the path we want to verify that the state it points to also matches the one expected by the demon. Finding that we do indeed have a +S state for JANE, we now fire the demon again and pop both steps off the path, leaving a path of nil.

Having reduced the path to nil, we instantiate a SUCCESS unit for JANE. PUGG maintains a list of all the units it identifies during a run (the all-unit list), so this unit is now added to that list. In addition, all demons for units based on the SUCCESS unit, such as the demon for FLEETING-SUCCESS, will now be spawned. These demons are attached to the +S state at the end of the SUCCESS unit for JANE.

Complex units are recognized in a similar fashion, but these often require additional steps that have not yet been discussed. One other step type is the subunit step, a notation for each smaller unit recognized in the process of indentifying a larger unit. The unit HONORED-REQUEST serves as a good example here. The affect state structure for this unit is represented below:



Note that this unit is made up of four subunits, namely EXTERNAL-MOTIVATION, POSITIVE-REACTION and two SUCCESS units. These subunit steps are included in the path list for the HONORED-REQUEST demon, shown below:

```
((SUBUNIT EXTERNAL-MOTIVATION (?X ?Y)) (GIVES . A) (+S . ?Y)
 (SUBUNIT SUCCESS (?Y)) (GIVES . C) (+S . ?X)
 (SUBUNIT SUCCESS (?X)) (SUBUNIT POSITIVE-REACTION (?X ?Y)))
```

b.) Demon Progression

Continuing our discussion of path reduction, we will now see how this proceeds hand in hand with demon progression, the movement of the demon from one state in the affect state map to another. The demon above is spawned and placed on the property list for MS2 when an EXTERNAL-MOTIVATION unit is found for JOHN and MARY. PUGG tests this path against the information it has so far about the affect state map and finds that there is a match so JOHN and MARY are bound to ?X and ?Y, respectively. The path is popped and the demon fires, but it does not progress, remaining attached to MS2. The resulting path appears as:

```
((GIVES . A) (+S . MARY) (SUBUNIT SUCCESS (MARY))
 (GIVES . C) (+S . JOHN) (SUBUNIT SUCCESS (JOHN))
 (SUBUNIT POSITIVE-REACTION (JOHN MARY)))
```

When +S1 and the corresponding A-link for MARY are read in, these are matched against the steps (GIVES . A) (+S . MARY) in the path. Everything agrees so the demon fires and the path is popped again. This demon is then deleted from the property list of state MS2 and added to the property list for state +S1. In this way, as a demon progresses through the affect state structure, it is always attached to the state corresponding to the last item in the path matched.

Meanwhile, other demons have been spawned for the new states read in. Encountering state MS2 under MARY spawned several demons, one of which was the demon for a SUCCESS unit. This unit was recognized when

state +S1 and the A-link were read in and "hooked" onto (placed on the property list of) its last state (+S1). Thus, when the HONORED-REQUEST demon's path has been reduced as above, the next step, (SUBUNIT SUCCESS (MARY)), may be matched with the SUCCESS unit just identified for MARY. This step is then removed from the path and the demon fires and progresses to state +S2. When the state +S2 and the corresponding C- and A-links are read in for JOHN, a similar process ensues. Units POSITIVE-REACTION (for JOHN and MARY) and SUCCESS (for JOHN) are identified. We then process the remainder of the demon's path, matching against John's positive state and verifying the subunits for John's success and John and Mary's positive reaction. Again the path is reduced to nil and the HONORED-REQUEST unit is recognized.

In our example above, the plot unit HONORED-REQUEST is said to "subsume" the smaller units it contains. These units will therefore not be included in the listing of top-level plot units produced by the report module. To keep track of subsumed units, we place each subunit in a "subsumed-list" for any demons that have successfully "picked up" the subunit [2]. These units are then marked with a "subsumed" flag (which is placed on the property list for that unit) once the larger unit has been recognized.

We refer to those steps in the demon's path describing an affect state type and character as state-bindings, and those steps describing a link and link type as link-bindings (or S-bindings and L-bindings.)

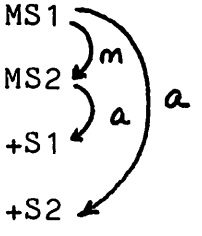
---

[2] The subsumed-list is simply another slot in the record structure for demons.

Thus (MS . ?Y) is an example of an S-binding and (GIVES . C) is and example of an L-binding. As can be seen from the discussion above, the various types of steps are processed differently. S-bindings are matched with data being read in whereas L-bindings and subunit steps are matched against information stored on the property list for the current affect state. Therefore PUGG must test each step in the path list to see whether it describes an L-binding step, an S-binding step, a subunit step or a migration step before attempting to make a comparison.

c.) Demon Migration

Our fourth and final step type describes demon migration. Migration occurs when a plot unit contains a "dead end," a structure that is necessary in defining the unit, but which fails to connect up with other elements of the plot unit. The unit NESTED-SUBGOALS provides a useful illustration of this phenomenon:



The demon for this path must reflect both the entire structure of the unit as well as the chronological order of the affect states. Starting with the first affect state (MS1), the next state encountered is MS2 which is connected to MS1 with an M-link. +S1 is next in line and this is joined to MS2 using an A-link. But when we encounter state +S2, we note that there is no connection from this state to +S1.

Rather, the connection exists back at the beginning of the unit, via an A-link from MS1. Since demons progress along the graph as they process the states involved, our demon for NESTED-SUBGOALS is now attached to state +S1 and unable to make any further matches with the states being read in. We need to move the demon from +S1 and attach it to state MS1 so that it can progress down the A-link to +S2.

To accomplish this task we "migrate" the demon from +S1 back to MS1. Migration is another path structure using L-bindings and S-bindings to specify how demon movement may occur, only it relies on previously encountered affect states instead of predicted states. The path for the NESTED-SUBGOALS unit shows how this feature is included:

```
((MS . ?X) (GIVES . M) (MS . ?X) (GIVES . A) (+S . ?X)
 (MIGRATING ((TAKES . A) (MS . ?X) (TAKES . M) (MS . ?X)))
 (GIVES . A) (+S . ?X))  [3]
```

Here we see that the MIGRATING step is also a record with a path structure of its own.

The demon processes the path for NESTED-SUBGOALS in the usual fashion, matching the S-bindings and L-bindings with the states and links read in. When the MIGRATING step is encountered, the cross-referencing links created during the integration phase are used to traverse the migration path, moving back up the graph. We therefore reverse the directionality of the links inside the migration path, resulting in the L-bindings (TAKES . A) and (TAKES . M) where

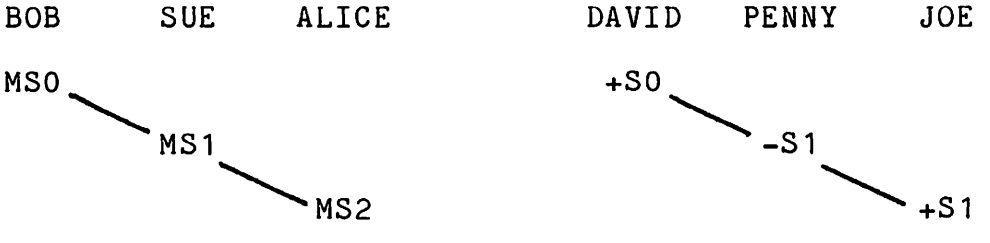
---

[3] This is a "simplified" version of the path for this demon. In actual practice, the path would begin with the MOTIVATED-SUCCESS subunit and be followed directly by the migrating step. The subunit SUCCESS would also be included in this path after the last step. The definition above was chosen to help clarify the process of migration.

previously we had (GIVES . M) and (GIVES . A). This path is processed in much the same way as for progression, with the demon firing and each step being popped (deleted) as matches occur. As the demon migrates back along the path, it is removed from the property list of each processed state and attached to the next state in the path. When the migration path has been completely traversed, the demon rests on the property list for MS1. This reduces the migration path to nil, and this step is deleted from the NESTED-SUBGOALS demon's path list. The demon now progresses as before traversing the path via progression, eventually recognizing the plot unit.

d.) Causal Link Transitivity

When two characters react to a given event or share a goal state we use c-links to connect the corresponding affect states. In the same manner, we can represent the shared reactions and goal states of three or more characters. In all cases where affect states are joined by c-links, these states occur in response to a single event. So if three characters share a goal state, the affect state map would show three MS states c-linked together. If two characters have a positive reaction to an event and a third has a negative reaction to the same event, the affect state map would show two +S states and one -S state c-linked across:



Shared goal state Reactions to shared event

In such cases, the apparent "chronological" ordering of these states is arbitrary and based only on the left to right ordering of the characters in the affect state map. All the states in each of the above graphs represent reactions to the same event.

Given a structure such as the one above representing reactions to a shared event, we want to establish not only the dyadic units NEGATIVE-REACTION (David and Penny) and POSITIVE-REACTION (Penny and Joe), but also the unit POSITIVE-REACTION for David and Joe. When such a configuration arises, we want to recognize plot units which exist between all pairs of characters, not merely those who happen to be adjacent to each other in the affect state map. We call this ability c-link transitivity. Let's see how we can accomplish this using the shared event map above for an example.

The recognition module reads in the information needed to create affect state +SO. It then forms predictions for the primitive plot units with an initial +S state by spawning demons for these units. Among these demons are those for POSITIVE-REACTION and NEGATIVE-REACTION, the two dyadic primitives which can begin with a +S state:

POSITIVE-REACTION:

((*?* . ?X) (GIVES . C) (+S . ?Y))

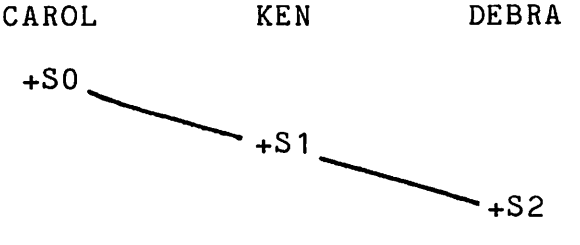
NEGATIVE-REACTION:

((*?* . ?X) (GIVES . C) (-S . ?Y))

(In the first step of the above paths, the symbol *?* indicates that any state can be the initial state for these units.)

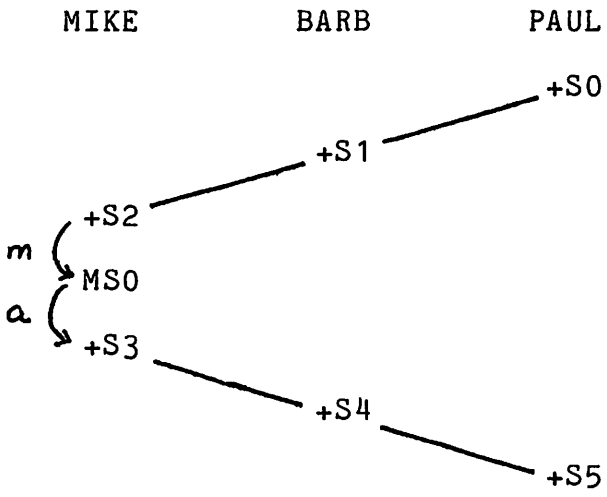
For each of the two paths we bind ?X to David, fire the demon and pop the path. Next we can match -S0 with the negative state predicted by the NEGATIVE-REACTION demon, instantiating that unit for David and Penny. But we cannot establish a match for the last step in the path of the POSITIVE-REACTION demon. Since only the demons attached to the current state and those states linked directly to it are active, once we read in state +S1 for Joe we will not be able to go back to the POSITIVE-REACTION demon we left hanging off of +S0. We therefore copy the demon from state +S0 to state -S0. No matches can be made, so we make no changes to the demon path. Since +S1 is directly linked to the state holding this demon, we are now able to recognize the POSITIVE-REACTION plot unit for David and Joe after state +S1 has been created.

The implementation must proceed in the following manner: When the recognition module creates an affect state which is c-linked to a previous state we (1) establish the initial character bindings for each of the demons attached to the antecedent affect state (which might already have been done in demons for complex units) (2) copy these demons from the preceding state to the current state and (3) process the demons as before. It is important that we only copy the demons from the antecedent state to the current state, leaving a copy still attached to the original state. To illustrate, consider the following affect state map:

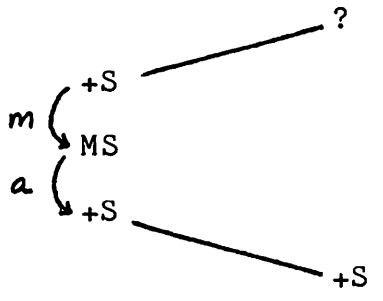


If we copy the demon predicting the POSITIVE-REACTION unit to +S1 but also remove it from +S0, we will still recognize the POSITIVE-REACTION unit for Carol and Debra involving states +S0 and +S2. (Recall the initial character binding of ?X to Carol has already been made when the demon is copied.) But we will fail to recognize the POSITIVE-REACTION unit for Carol and Ken since we moved the demon before its path was compared with the new information just read in (the c-link and state +S1). If we leave a copy behind with this state we can easily pick up the other instance of this unit.

Plot unit transitivity gives rise to one further complication in demon management. Consider the affect state map below:



Transitivity permits us to recognize the two POSITIVE-REACTION units between Mike and Paul in addition to those between Mike & Barb and Barb & Paul. In turn, these additional units allow us to identify two instances of the REWARD plot unit (Mike rewarding Barb and Mike rewarding Paul).



REWARD

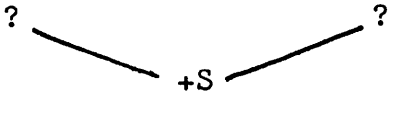
Demon path for REWARD:

```
((SUBUNIT POSITIVE-REACTION (?X . ?Y)) (GIVES . M) (MS . ?X)
 (GIVES . A) (+S . ?X) (SUBUNIT ENABLED-SUCCESS (?X))
 (GIVES . C) (+S . ?Y) (SUBUNIT POSITIVE-REACTION (?Y . ?X)))
```

This demon is spawned when a POSITIVE-REACTION unit is recognized. Recall that a subunit step is processed by searching for a match against all the plot units hooked to the current state (+S2). In this case we have two POSITIVE-REACTION units attached to +S2: POSITIVE-REACTION (Mike Paul) and POSITIVE-REACTION (Mike Barb). (Remember that plot units are always hooked to the final state in their configuration, after being identified).

The syntax constraints on causal links permit only one link of a given type to enter and leave an affect state. Therefore, if we disallow c-link transitivity, we can only have one POSITIVE-REACTION unit hooked to a +S state:

PERMITTED



** ILLEGAL **

This feature insures efficiency in processing demon paths. When we process a subunit step, we compare the information in that step

against each of the plot units hooked to the current state. If we find a match with one of these units we can stop our search and ignore the rest of the list. We are guaranteed that only one such unit can be hooked to this state.

Once we permit transitivity, however, we no longer have this guarantee. In the example above we have two POSITIVE-REACTION units that can match with the first step of our demon. Therefore, we must maintain a separate demon for each match encountered since each could potentially result in recognition of the REWARD unit. Once this initial generation of multiple demons is accomplished, the rest of the processing continues normally. We should note that this generation of multiple demons occurs for all demons spawned by encountering the POSITIVE-REACTION unit, not just for REWARD.

Since states joined by e-links represent multiple perspectives of an event or multiple instantiations of a goal state, we can consider plot unit transitivity across e-links as well. The implementation is precisely the same as for transitivity across c-links. Whenever a state is created that is e-linked to a previous state we make the appropriate initial bindings and copy over the demons to the next state. Transitivity across e-links brings up the same concerns and complications as were discussed for c-link transitivity. The mechanisms that permit us to handle these problems for c-links work identically in this case.

However, the general utility of e-link transitivity is not as clear as is the case for c-links. With c-links, we know that all the linked states are really referring to a single event, but e-linked

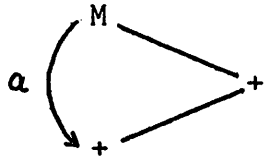
states may refer to different "surface" events arising from the same base event or goal state. For example, if you reapply to a college after having been rejected, you have actually performed two separate acts (application-1 and application-2). But the underlying goal state is the same - to go to college. This aspect raises questions concerning the application of e-link transitivity to all possible cases.

3. Conditionals

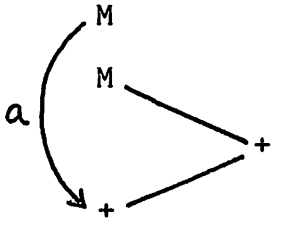
For many of the possible plot units, the corresponding demon's path does not specify a unique affect-state structure. For example, the plot unit PROMISE has the path:

```
((SUBUNIT EXTERNAL-MOTIVATION (?X . ?Y)) (GIVES . C) (+S . ?X)
 (SUBUNIT POSITIVE-REACTION (?X . ?Y) (SUBUNIT SUCCESS (?X)))
```

Both of the two affect-state maps shown below satisfy the description of this unit as given by the above path. The map for PROMISE-1 is the correct version. It is important that the recognition module not identify structures such as that for *PROMISE-2 as the plot unit PROMISE.



PROMISE-1

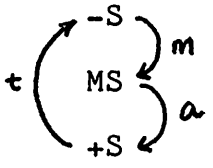


*PROMISE-2

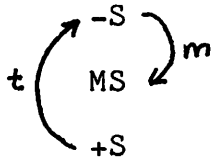
In this example, a simple check on the number of affect states present in the map is sufficient to preclude the bogus structure from being

identified as the unit PROMISE. When we stipulate that the map for this path must contain only three affect states, the only possible structure is the correct one shown for PROMISE-1 above. This test must therefore be included as a part of the demon for the PROMISE plot unit.

A few plot unit pairs differ from one another by only the presence or absence of a single link. Such is the case with the units INTENTIONAL-PROBLEM-RESOLUTION and FORTUITOUS-PROBLEM-RESOLUTION shown below.



INTENTIONAL-
PROBLEM-RESOLUTION



FORTUITOUS-
PROBLEM-RESOLUTION

Here, only the presence of the the A-link from the MS to the +S state distinguishes INTENTIONAL-PROBLEM-RESOLUTION from FORTUITOUS-PROBLEM-RESOLUTION. It is necessary, therefore, to insure that this A-link is not present before we identify the FORTUITOUS-PROBLEM-RESOLUTION plot unit. Again, a test specifying this condition is needed as part of the demon for FORTUITOUS-PROBLEM-RESOLUTION.

We call such a test a conditional and in all instances it is executed only after the demon's path has been reduced to nil. In cases where we must count the number of affect states making up a certain structure, this can only be known after the entire structure is traversed. And in the case of the FORTUITOUS-PROBLEM-RESOLUTION

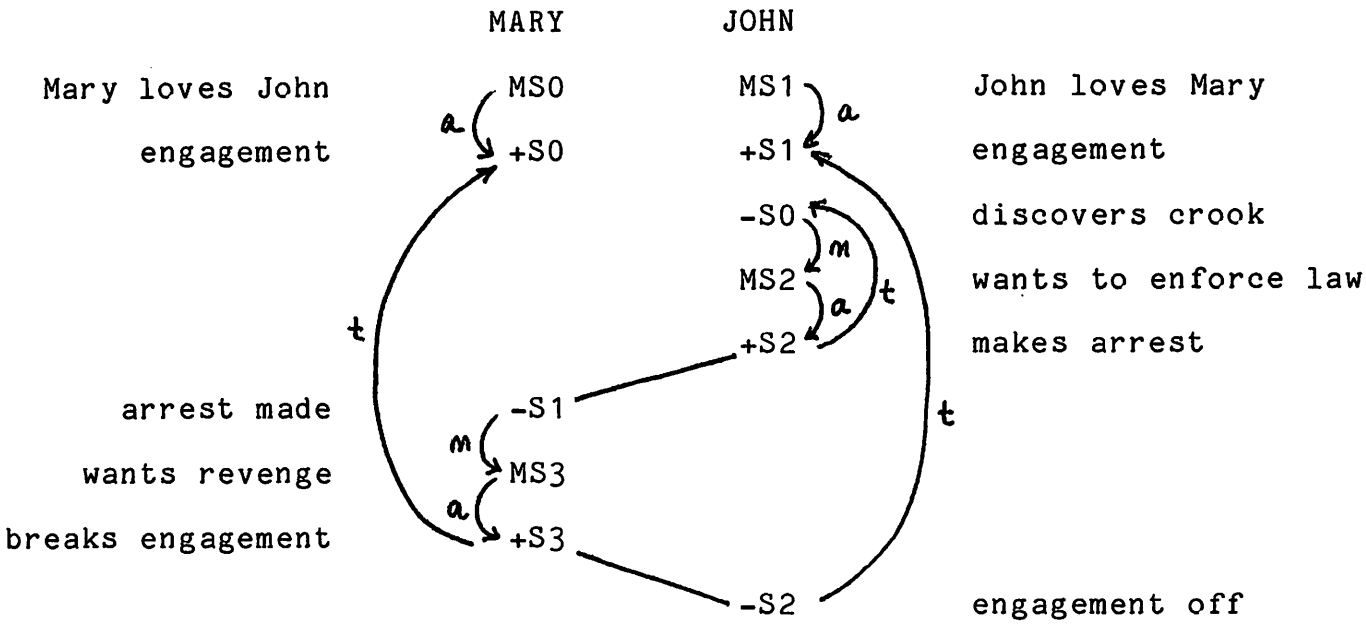
unit, the conditional must insure that there is no A-link entering the +S state, the last state in the unit. This test cannot be executed until this last state is recognized as being part of the plot unit, which does not occur until the last step for this demon's path has been processed and removed. Furthermore, there is no point in even making the test unless we know that all the other requirements for recognizing the unit have been met. This will only be true when the demon's path has been reduced to nil.

II. REPORT MODULE:

When every affect-state has been entered and all the plot units in the text have been recognized, PUGG shifts control to the report module. Here summary statistics on the units found and their relatives are produced. Below is a trace produced by the report module. The narrative and corresponding affect state map used for this trace are also shown.

> John was thrilled when Mary accepted his engagement ring. But when he found out about her father's illegal mail order business, he felt torn between his love for Mary and his responsibility as a policeman. When John finally arrested the old man, Mary called off the engagement.

Affect State Map



Trace from Report Module:

There are 5 top-level plot units:

1 unit0 = success (linda)
2 unit6 = intentional-problem-resolution (bill)
3 unit12 = sacrifice (linda)
4 unit15 = fleeting-success (bill)
5 unit16 = retaliation (linda bill)

The adjacency matrix:

```
o-o--
-o--o
o-o-o
---oo
-oooo
```

Name = unit0
Type = success
Role(s) = (linda)
Affect States = (+s0 ms0)

Name = unit6
Type = intentional-problem-resolution
Role(s) = (bill)
Affect States = (+s2 -s0 ms2)

```
Name = unit12
Type = sacrifice
Role(s) = (linda)
Affect States = (+s3 ms3 +s0)

Name = unit15
Type = fleeting-success
Role(s) = (bill)
Affect States = (-s2 +s1 ms1)

Name = unit16
Type = retaliation
Role(s) = (linda bill)
Affect States = (-s2 +s3 ms3 -s1 +s2)
```

Unit Families:

```
1 Degree = 1 Relatives = (3)
2 Degree = 1 Relatives = (5)
3 Degree = 2 Relatives = (1 5)
4 Degree = 1 Relatives = (5)
5 Degree = 3 Relatives = (2 3 4)
```

Masterlist:

```
((1 1 3) (2 2 5) (3 1 3 5) (4 4 5) (5 2 3 4 5))
```

Connected Subgraphs:

```
(1 3 5 4 2)
```

A. TOP LEVEL UNIT LIST

1. Identification of Top Level Units

The first step in this procedure is to identify all the top-level plot units from among all those that have been identified. Each unit which was subsumed by another plot unit was so marked by the recognition module with a "subsumed" flag placed on that unit's property list. Eliminating these units from the all-units list gives us the top-level unit list, a listing of those plot units not subsumed by any other unit.

2. Dyadic primitives

The second step is to check for the inclusion or exclusion of dyadic primitives at top level. These particular units (EXTERNAL--MOTIVATION, EXTERNAL-PROBLEM and EXTERNAL-ENABLEMENT) vary in their usefulness at the top level from story to story, therefore we may want to include them in some cases and exclude them in others. For narratives where the affect-state map yields many two character plot units they may add little to the structure of the top-level plot unit graph. For other stories, however, the interactions between characters may be important, but not structured in such a way as to form plot units. Here the inclusion of these dyadic primitives can help preserve the connectivity of the top-level unit graph. To indicate how we want these units treated for this particular case, we use a global variable to set a flag. Checking this flag, the report module either deletes these primitives from the top-level unit list or lets them remain. In general, we prefer to exclude the dyadic primitives from top level.

Referring to the trace produced by the report module above, we see that many different statistics are produced, starting with the number of top-level plot units. Each unit is then identified by listing its name (UNIT0, UNIT6), type (SUCCESS, SACRIFICE) and characters involved (Linda, Bill). Units, like demons, are record structures and each of these pieces of information is stored in a separate slot in the record. Additionally, a fourth slot lists those affect-states making up the unit in question.

B. ADJACENCY MATRIX

Following this, the adjacency matrix is generated. Using a brute force intersection search, the affect-state slots for all the units are compared. Each intersection (where two units share an affect-state) is noted and output as a circle. A dash represents no shared states. Naturally, each unit will intersect with itself, thus the adjacency matrix will always have a solid diagonal.

A more detailed listing of each top-level unit comes next. In addition to the information presented before, the affect states comprising each plot unit are listed. Again, this information is easily obtained by simply fetching it from the appropriate slot in the record for each unit.

C. UNIT FAMILIES

To generate the unit families, a "masterlist" of related units is constructed. Every unit has a separate "relatives-list" in the masterlist, and when a shared state is found, we include the intersecting unit in the appropriate relatives-list. Thus, referring again to the trace from the report module above, since unit 1 is related to itself and unit 3, both are included in the relatives-list for unit 1. Using the masterlist, the unit families information is easily produced. The degree of a particular unit is given by the number of elements in the relatives-list, not counting that unit. Listing these other elements gives us the relatives for that unit.

D. CONNECTED SUBGRAPHS

The list(s) of the connected subgraphs for a narrative (those portions of the top-level plot unit graph which have a path connecting all the units in that section) is produced last. This is generated by taking each top-level plot unit and checking its relatives-list to see who it is related to. These relatives are then checked for their relatives, and so on, until all the relatives for a particular unit are identified. By keeping track of those units we encounter during this search, the connected subgraphs lists are formed. In the example above, all the top-level units are related and so there is only one connected subgraph. This will not be the case for all narratives, however.

Section three: Conclusion

We have seen how PUGG (the Plot Unit Graph Generator) takes an affect state map and produces a listing of the top level plot units present and their inter-relationships. Plot units are identified by the recognition module using a Read-Integrate-Predict loop. Data on each affect state is read in and integrated with any previously defined states. Using this information, we can form predictions for expected states and units. These predictions take the form of demons anticipating the structure of the unread portion of the affect state map. Manipulating demons through progression and migration, we can recognize the plot units present in the narrative. Flexible unit definitions and transitivity of plot units across c- and e-links

increase the power of the recognition module and permit identification of units otherwise overlooked. When recognition of plot units is completed, the report module produces summary statistics for the set of plot units just identified. This listing of each of the top level plot units and how they are related and connected to each other is used to generate the top level plot unit graph structure.

PUGG was first implemented on a DEC-1060 at Yale University by Wendy Lehnert at the same time that Michael Dyer was developing the BORIS system [Dyer 1983]. BORIS was designed to integrate multiple knowledge structures that had been proposed by various researchers at Yale in the period from 1975 to 1980. While many computer programs had been implemented to investigate each of these knowledge structures in isolation, BORIS was the first attempt to bring them all together to study the control problems of complex inference systems. As a sideline, the BORIS project was also interested in the role of affect in narrative text comprehension. It was clear that a number of inferences required knowledge about human emotions, but no computational models had been proposed to account for this class of inference. Our desire to examine problems of inference relating to affect resulted in the representational techniques of affect state maps and plot units. Only after these levels of memory representation had been suggested, did we realize their importance for text summarization.

Shortly after the completion of the BORIS project, Lehnert moved to the University of Massachusetts, where PUGG was reinstated on a VAX-780 under VMS. At that time, Cynthia Loiselle began to embellish the existing code, making it possible for us to experiment with c-link

and e-link transitivity. Flexible unit definitions were also implemented at this time, and new plot unit definitions were added to our universal dictionary of proposed plot units. In its current implementation, PUGG reads its data from files containing affect state maps, although it can also be set up to allow interactive input. Through the use of signalling flags and varied data bases (which store the demon definitions) we can enable or disable PUGG's ability to handle any of c-link transitivity, e-link transitivity and flexible unit definitions.

At this time, a large library of affect state map encodings and resulting plot unit graphs is being established with the help of JoAnn Brooks and John Brolio. With this data we are testing various hypotheses about encoding conventions and plot unit recognition heuristics. In a related effort, Malcolm Cook is working on an interface between PUGG and MUMBLE [McDonald 1983], so we can follow through on the generation of English sentences to complete the full summarization process. We do not currently have the facilities to automate the generation of affect state maps at UMass (this would involve bringing up another system like BORIS), but we are nevertheless making significant progress with the problems that arise given the existance of an affect state map.

Section four: McPUGG - a Micro Version of PUGG

McPUGG was prepared in an effort to make more understandable the processing mechanisms used by PUGG, the Plot Unit Graph Generator. For that reason, we have chosen to keep that part of the program which we feel demonstrates its major source of computational power -- the subsumption of plot units which occurs in recognizing more complex units in order to ascertain the top-level plot units in a narrative. In order to retain this ability, many other features of the full program have been eliminated in this micro version.

Notably absent is the ability to process texts for more than one character. While most of the complex plot units are two-character units, we feel that processing power of PUGG can be amply demonstrated in narratives involving only one character. All link types (with the exception of c-links, naturally) can still be handled, so most of the primitive units are still able to be recognized (although they are not all included in this version of the program). Also absent is the ability to process multiple links in or out of a given affect state. To simplify control, only one link in and one link out are permitted per affect state. Again, while this does decrease the number of units that can be recognized, the remaining units are still sufficiently interesting.

The primitives and complex units we selected to go with McPUGG are only those necessary to process the demonstration text included below. One obvious and fairly easy exercise for

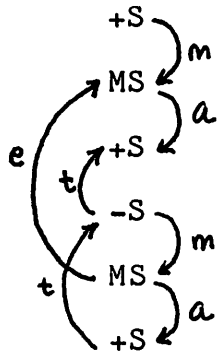
anyone working with this code would be to add new unit definitions to the ones already present. The given units provide enough complexity so that many aspects of PUGG's processing power can be demonstrated. Specifically, the ability to subsume complex units as well as primitives is shown in the run on this text, as well as PUGG's ability to handle all link and state types.

Other features missing from this version include migration, conditionals checking and c-link and e-link transtitivity. The latter would not be an easy feature to add as many changes would need to be made to the data structure manipulation and flow of control of the program. The other two would make reasonable projects, requiring varying amounts of work and changes.

Below is the story used for the demonstration run, its affect state map, the input encoding used by McPUGG, the trace of this run and the resulting top level plot unit graph. Following that is the program and support functions.

> John had some time off from work and wanted to have a great vacation. He had carefully saved his money and made all the arrangements for an exciting trip to Bermuda. But no sooner had he arrived at his hotel when he discovered that his pockets had been picked and he only had $50 left for his whole vacation! At first, John was terribly disappointed and thought he would be forced to return home, but he decided to make the best of things and found a store that would rent him a tent and sleeping bag. John spent the rest of his vacation camped out on the beach and had such a wonderful time he vowed he'd go camping again on his next vacation.

Affect state map for above story:



has time off work

wants a great vacation

goes to Bermuda

loses money

decides to camp out

has a great time

Input encoding for McPUGG:

```
   +s  n             n            (has time off work)      ; +s0
y  ms  y +s0 m       n            (wants great vacation)   ; ms0
y  +s  y ms0 a       n            (goes to Bermuda)        ; +s1
y  -s  n             y +s1 t      (loses money)            ; -s0
y  ms  y -s0 m       y ms0 e      (decides to camp out)    ; ms1
y  +s  y ms1 a       y -s0 t      (has a great time)       ; +s2
n
```

Trace of McPUGG on above input:

```
Affect state +s0 has been created.

     Name = +s0
     Links into +s0 from = nil
     Links out of +s0 into = nil
     Description = (has time off work)

Affect state ms0 has been created.

     Name = ms0
     Links into ms0 from = (m . +s0)
     Links out of ms0 into = nil
     Description = (wants great vacation)

Creating a new plot unit:

     Name = unit0
     Type = enablement
     Affect States = (ms0 +s0)

   unit0 SUBSUMES nil
```

```
Creating a demon looking for:
                                    enabled-success

Affect state +s1 has been created.

     Name = +s1
     Links into +s1 from = (a . ms0)
     Links out of +s1 into = nil
     Description = (goes to Bermuda)

Creating a new plot unit:

     Name = unit1
     Type = success
     Affect States = (+s1 ms0)

   unit1 SUBSUMES nil


Creating a demon looking for:
                                    fleeting-success

Creating a new plot unit:

     Name = unit2
     Type = enabled-success
     Affect States = (+s1 ms0 +s0)

   unit2 SUBSUMES (unit1 unit0)


Affect state -s0 has been created.

     Name = -s0
     Links into -s0 from = nil
     Links out of -s0 into = (t . +s1)
     Description = (loses money)

Creating a new plot unit:

     Name = unit3
     Type = loss
     Affect States = (-s0 +s1)

   unit3 SUBSUMES nil


Creating a new plot unit:

     Name = unit4
     Type = fleeting-success
     Affect States = (-s0 +s1 ms0)

   unit4 SUBSUMES (unit3 unit1)
```

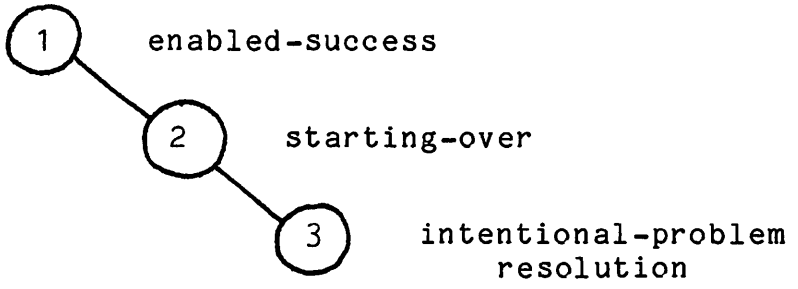
```
Creating a demon looking for:
                                starting-over

Affect state ms1 has been created.

     Name = ms1
     Links into ms1 from = (m . -s0)
     Links out of ms1 into = (e . ms0)
     Description = (decides to camp out)


Creating a new plot unit:

     Name = unit5
     Type = perseverence
     Affect States = (ms1 ms0)

   unit5 SUBSUMES nil


Creating a new plot unit:

     Name = unit6
     Type = problem
     Affect States = (ms1 -s0)

   unit6 SUBSUMES nil


Creating a demon looking for:
                                success-born-of-adversity

Creating a new plot unit:

     Name = unit7
     Type = starting-over
     Affect States = (ms1 ms0 -s0 +s1)

   unit7 SUBSUMES (unit5 unit6 unit4)


Affect state +s2 has been created.

     Name = +s2
     Links into +s2 from = (a . ms1)
     Links out of +s2 into = (t . -s0)
     Description = (has a great time)

Creating a new plot unit:

     Name = unit8
     Type = resolution
     Affect States = (+s2 -s0)

   unit8 SUBSUMES nil
```

```
Creating a new plot unit:

     Name = unit9
     Type = success
     Affect States = (+s2 ms1)

   unit9 SUBSUMES nil


Creating a demon looking for:
                                fleeting-success

Creating a new plot unit:

     Name = unit10
     Type = success-born-of-adversity
     Affect States = (+s2 ms1 -s0)

   unit10 SUBSUMES (unit9 unit6)


Creating a demon looking for:
                                intentional-problem-resolution

Creating a new plot unit:

     Name = unit11
     Type = intentional-problem-resolution
     Affect States = (+s2 -s0 ms1)

   unit11 SUBSUMES (unit8 unit10)


There are 3 top-level plot units:


     Number = 1
     Name = unit2
     Type = enabled-success
     Affect States = (+s1 ms0 +s0)

     Number = 2
     Name = unit7
     Type = starting-over
     Affect States = (ms1 ms0 -s0 +s1)

     Number = 3
     Name = unit11
     Type = intentional-problem-resolution
     Affect States = (+s2 -s0 ms1)
```

Unit Families:

1 Degree = 1 Relatives = (2)
2 Degree = 2 Relatives = (1 3)
3 Degree = 1 Relatives = (2)

Using this information we can then produce a top-level plot unit graph by hand:



We should say a few words about the support environment needed to run the following code for McPUGG. Many very useful but nonstandard functions and macros are used and they deserve a quick mention. Several references exist where the reader may obtain further information on their use and implementation. They are listed below.

1. [AI] Charniak, E., Riesbeck, C. and McDermott, D., Artificial Intelligence Programming, Lawrence Erlbaum Associates, 1980.

   An extensive library of programming tools commonly used by AI researchers.

2. [ICU] Schank, R. and Riesbeck, C., Inside Computer Understanding, Five Programs Plus Miniatures, Lawrence Erlbaum Associates, 1981.

   A good reference for the reader wishing to learn more of how natural language understanding systems work. The chapter on Lisp explains many of the support functions used here.

3. [UCI] Meehan, J., The New UCI Lisp Manual, Lawrence Erlbaum Associates, 1979.

A reference manual describing the PDP-10 Lisp 1.6 system developed at Stanford University, the University of California at Irvine and Rutgers University containing detailed explanations and definitions for many of the functions we use.

Additionally, a few functions and global variables specific to our dialect of Lisp (CLisp) are used. These are also noted below. The following is a list of these support utilities, a brief explanation of their use and a list of references which may be of help for the reader not familiar with them.

Name	References	Description
:=	AI	Assignment macro
*INFILE	(Clisp)	Global variable for the current input file
ASSOC	AI, ICU, UCI	See references
DEMON:UNIT-ID	AI	Functions of this form are access functions into record structures
DEX	AI	Expands macros at time of definition
FOR	AI, ICU	Iterative looping macro
INCR	UCI	Consecutively increments given variable
INTERSECTION	AI, UCI	Same as INTERSECT in AI
LET*	(CLisp)	LET with sequential rather than simultaneous variable binding
LOOP	AI, ICU	Iterative looping macro
MACRO	AI, ICU, UCI	Same as DM in these books
MSG	ICU, UCI	Useful printing macro

NEWSYM	AI, UCI	Creates unique atoms
OPENI-CHECK-LIB	(Clisp)	Opens a file as an input file
PROP		Identical to GET, but with reversed argument order
RECORD-TYPE	AI	Constructs a record data type. See text for description of records and related functions.
REMOVE-ELEMENT	ICU, UCI	Same as REMOVE in these books

The following section contains the code for McPUGG, some support functions and a data base of complex plot unit definitions.

```
; ****************************************************************
;                          MICRO PUGG
; ****************************************************************

; ******************** SUPPORT/HELPER FUNCTIONS *********************

; Searches through LST (a list of lists) until it finds a list with
; CAR VAR and returns the CDR of this list (the value bound to VAR)

(dex binding (var lst)
     (cdr (assoc var lst)))

; Used to CONS a new VALUE onto the PROPERTY list for ITEM

(dex add-to-plist (item property value)
     (:= (prop property item)
         (cons value *-*)))

; Retrieves the information stored on the DATA p-list for the given
; STATE.  This information is in the form of an A-STATE record.

(macro state-data (state)
       `(get ,(cadr state) (quote data)))

; The following three functions are particularly dependent upon the
; CLisp dialect of LISP.

; This version of NEWSYM evaluates its argument.

(defun newsym (sym)
   (let (count (add1 (or (get sym 'sym-count) -1)))
      (putprop sym count 'sym-count)
      (make-atom (string-append sym count))))

; A function to run McPUGG on its demo story:

(defun micro-run nil
  (reset)
  (driver "bermuda.dat"))     ; VMS file name

; RESET clears the property lists for the atoms representing the
; various affect states and units generated in the course of a
; recent PUGG run.  Such a function is almost mandatory when more
; than one run is to be made.

(dex reset nil
   (mapcar '(ms -s +s unit)
      (lambda (x)
         (loop (initial num (get x 'sym-count))
               (while (and (numberp num) (greaterp num -1)))
               (do (wipe (make-atom (string-append
                           (make-string x) (integer-string num))))
                   (setq num (sub1 num)))
               (result (putprop x -1 'sym-count))))))
```

```
; ****************************************************************
;                          RECOGNITION MODULE
; ****************************************************************


; ***************** AFFECT STATE MAP CONSTRUCTION *****************

; The information we need to maintain for each affect state is
; stored in a record structure with slots for the NAME of the
; affect-state, its TYPE, the LINKS coming INTO and OUT of that
; state and the states they link up to as well as a short
; DESCRIPTION of that state.

; IN-LINKS and OUT-LINKS are LINKAGE records, DESC is a list.

(record-type a-state aff-st (name type in-links out-links desc))
(record-type linkage nil (link-type . state))

; An output function, verifies the information just read in for each
; new affect state.

(dex dump-state (affect-state pointer)
     (msg t t "Affect state " pointer " has been created." t
          t 5 "Name = " (a-state:name affect-state)
          t 5 "Links into " (a-state:name affect-state)
                    " from = " (a-state:in-links affect-state)
          t 5 "Links out of " (a-state:name affect-state)
                    " into = " (a-state:out-links affect-state)
          t 5 "Description = " (a-state:desc affect-state)))

; ACCEPT-STATE gathers the information needed for each affect state
; then generates reverse pointing links for the links just read in.
; Any existing demons are processed and demons for new primitives
; are created.

(dex accept-state nil
   (let* (state-type (read)
          pointer (newsym state-type)
          links-in (collect-links)
          links-out (collect-links)
          description (read)
          affect-state
             (putprop pointer
                      (a-state pointer state-type
                               links-in links-out description)
                      'data))
      (dump-state affect-state pointer)
      (add-out-links (a-state:in-links affect-state) pointer)
      (add-in-links (a-state:out-links affect-state) pointer)
      (process-demons pointer)
      (spawn-startup-demons pointer)))
```

```
; COLLECT-LINKS checks first to see whether any links exist for this
; state and direction.  If so, the corresponding state and link type
; are read in and formatted into a LINKAGE record structure.

(dex collect-links () ; Only previously defined a-states can be used
   (let (any (read))
      (cond ((eq any 'y) (:= state (read))
                         (:= link (read))
                         (linkage link state)))))

; ADD-IN-LINKS and ADD-OUT-LINKS generate opposite-direction links
; for each link-state pair read in.

(dex add-in-links (linkage-list ptr)
     (cond (linkage-list
             (:= (a-state:in-links
                    (state-data (linkage:state linkage-list)))
                 (linkage (linkage:link-type linkage-list) ptr))
             (process-demons (linkage:state linkage-list)))))

(dex add-out-links (linkage-list ptr)
     (cond (linkage-list
             (:= (a-state:out-links
                    (state-data (linkage:state linkage-list)))
                 (linkage (linkage:link-type linkage-list) ptr))
             (process-demons (linkage:state linkage-list)))))

; DRIVER controls the operation of McPUGG.  New affect states are
; read in until the end of the file is reached.  As states are being
; created, units are identified.  When this processing is completed,
; the REPORT function prints the top-level units found and their
; relationships.  CLisp file handling conventions are used here.

(dex driver (file)
     (let (*infile (openi-check-lib file)
           all-units      nil
           all-unit-ids  nil)
        (loop (initial another nil)
              (do (accept-state)
                  (:= another (read)))
              (until (eq another 'n))
              (result nil))
        (report)))


; ************************* PLOT UNIT DEMONS **********************
;
; Demons are data-driven using record structures that progress
; across affect states. Any a-state may have one or more active
; demons associated with it, stored under the property DEMONS.

(record-type demon active (path unit-id a-states subsumed-units))
```

```
; a PATH in a DEMON is a list of S-BINDINGs, L-BINDINGs, and
; SUBUNITS

(record-type s-binding nil (s-type))
(record-type l-binding nil (dir . l-type))
(record-type subunit subunit (input-unit))

; Demons for the primitive units are generated using the following
; data structure.  This structure gives the PATH for the demon, its
; UNIT-ID, and NIL fillers for the A-STATES and SUBSUMED-UNITS slots.

(:= primitives
'((success . (((ms) (gives . a) (+s)) success nil nil))
  (problem . (((-s) (gives . m) (ms)) problem nil nil))
  (enables . (((+s) (gives . m) (ms)) enablement nil nil))
  (persev  . (((ms) (takes . e) (ms)) perseverence nil nil))
  (resolut . (((-s) (takes . t) (+s)) resolution nil nil))
  (loss    . (((+s) (takes . t) (-s)) loss nil nil))))

; The names of the primitives spawned for each affect state type
; are stored on the property list for that state type under
; SEED-DEMONS.

(defprop ms (success persev) seed-demons)
(defprop +s (enables loss) seed-demons)
(defprop -s (problem resolut) seed-demons)

; For each primitive unit on the SEED-DEMONS list above, we spawn a
; demon and place it on the DEMONS property list for that affect
; state.

(dex spawn-startup-demons (state)
     (let (type (a-state:type (state-data state)))
          (for (x in (get type 'seed-demons))
               (do (add-to-plist
                     state 'demons
                     (apply demon (binding x primitives)))))))

;  A unit (primitive or complex) is instantiated whenever the path
;  of a demon is reduced to NIL.  When instantiation is ready to
;  take place, the demon will be associated with its most recently
;  touched a-state which then becomes the "hook" for that unit
;  instantiation. The exhausted demon places a UNIT structure on
;  the HOOK p-list, may add one or more complex unit demons to the
;  DEMONS p-list, and is finally itself removed from the DEMONS
;  p-list. The UNIT record structure records a unit name generated
;  by NEWSYM, the unit type and an a-state list containing all
;  a-states within the unit instantiation.

(record-type unit nil (unit-name unit-type aff-states))
```

```
; When a unit is created, we invoke the following function
; to add new demons to the DEMON p-list:

(dex demon-gen (ptr unit-type)  ; PTR points to where UNIT is hooked
   (cond ((get unit-type 'spawns)
          (let (spawned-demon (apply demon (get unit-type 'spawns)))
               (msg t t "Creating a demon looking for: ")
               (msg t 30 (demon:unit-id spawned-demon))
               (add-to-plist ptr 'newdemons spawned-demon)))))

; DUMP-UNIT is an output function that gives us the name, unit type
; and included affect states for each new unit created.  This
; function is also used in the REPORT section of the program.

(dex dump-unit (unit)
     (msg t 5 "Name = " (unit:unit-name unit)
          t 5 "Type = " (unit:unit-type unit)
          t 5 "Affect States = " (unit:aff-states unit)))

; UNIT-GEN creates a unit instantiation when a demon is exhausted.
; To simplify the recognition of top-level units, UNIT-GEN marks a
; subsumption flag on the p-list of any units that are properly
; contained by the new unit.  These will be useful when we compute
; our top-level units at the end of the story representation.

(dex unit-gen (dem ptr)  ; PTR shows the a-state we're currently at
   (let (newunit (unit (newsym 'unit)
                       (demon:unit-id dem)
                       (noduples (demon:a-states dem))))
     (for (x in (demon:subsumed-units dem))
          (do (putprop x t 'subsumed)))
     (add-to-plist ptr 'hook newunit)
     (msg t t "Creating a new plot unit:" t)
     (dump-unit newunit)
     (msg t t 3 (unit:unit-name newunit) " SUBSUMES "
        (for (x in (demon:subsumed-units dem)) (filter x)) t)
     (demon-gen ptr (demon:unit-id dem))          ; add new demons
     (:= all-units (cons newunit *-*))
     (:= all-unit-ids (cons (unit:unit-name newunit) *-*))))

; PROCESS-DEMONS cycles through the list of demons attached to the
; given affect state.  If any of them fire then the entire list is
; re-processed.

(dex process-demons (ptr)
    (loop (result nil)
          (initial *p-flag* t)
          (while *p-flag*)
          (do (nullpaths ptr)           ; TEST-DEMON resets *P-FLAG*
              (:= *p-flag* nil)         ; if a demon goes off
              (:= (prop 'demons ptr )
                  (for (x in (get ptr 'demons))
                       (filter (test-demon x ptr)))))))
```

```
; The first step in the path of the given demon is checked to see
; if it is a SUBUNIT step (default is an L-BINDING/S-BINDING step
; type) and the appropriate action is taken.

(dex test-demon (dem ptr)    ; a nil return kills the demon
    (cond ((is-subunit (car (demon:path dem)))
           (testsub dem ptr))
          (t (testlink dem ptr))))

; TESTLINK is called when the first step of the demon path is either
; an S-BINDING or an L-BINDING.  The 1st case of the COND handles
; L-BINDINGs, the 2nd case handles S-BINDINGs.  In the 2nd case we
; assume we're at a legal a-state, so we only have to add the affect
; state to the list this demon is maintaining and pop the path.
; In the 1st case, we assume we have an L-BINDING/S-BINDING sequence.
; We check for the L-BINDING link-type,the S-BINDING state type, and
; then tackle the bindings to first check for consistency and then
; add new bindings if necessary.

(dex testlink (dem ptr)
 (let (path (demon:path dem) newstate nil)
   (cond
    ((or (and (eq (caar path) 'gives)                    ; case 1
              (eq (cdar path) (car (a-state:out-links
                                    (state-data ptr))))
              (:= newstate (cdr (a-state:out-links
                                 (state-data ptr)))))
         (and (eq (caar path) 'takes)
              (eq (cdar path) (car (a-state:in-links
                                    (state-data ptr))))
              (:= newstate (cdr (a-state:in-links
                                 (state-data ptr))))))
     (cond ((eq (a-state:type (state-data newstate)) (caadr path))
            (:= (demon:path dem) (cddr *-*))
            (:= (demon:a-states dem) (cons newstate *-*))
            (add-to-plist newstate 'demons dem)          ; progress to
            (:= *p-flag* t)                              ; next state
            (cond ((null (demon:path dem)) nil)
                  (t dem)))))
    ((memq (caar path) '(ms +s -s *?*))                  ; case 2
     (:= (demon:path dem) (cdr *-*))
     (:= (demon:a-states dem) (cons ptr *-*))
     (:= *p-flag* t)
     (cond ((null (demon:path dem)) nil)
           (t dem)))
    (t dem))))                                  ; no action taken
```

```
; When the first step in a demon's path is a SUBUNIT step, TESTSUB
; compares it to the list of units HOOKED to the given affect
; state.  If a match is found, the affect states for the matched
; unit are added to the demon's list, the unit is placed on the
; subsumed-unit list for the demon, and finally the demon's path
; is popped.  *P-FLAG* is reset so that we will cycle through all
; the demons again.

(dex testsub (dem ptr)
  (let (u-type (subunit:input-unit (car (demon:path dem))))
    (loop (result dem)
          (initial l (get ptr 'hook))
          (while l)
          (until (and (eq u-type (unit:unit-type (car l)))
                      (progn (:= (demon:a-states dem)
                                 (append
                                       (unit:aff-states (car l))
                                        *-*))
                             (:= (demon:subsumed-units dem)
                                 (cons (unit:unit-name (car l))
                                        *-*))
                             (:= (demon:path dem) (cdr *-*))
                             (:= *p-flag* t))))
          (next l (cdr l)))))

; NULLPATHS moves through a list of demons, processing those
; which have null paths, and then eliminating them from the
; p-list.

(dex nullpaths (ptr)
   (putprop ptr nil 'newdemons)  ; this p-list may be reset by
   (putprop ptr                  ; UNIT-GEN if EXHAUSTED is
            (for (x in (get ptr 'demons)) ; called inside ALIVE
                 (filter (alive x ptr)))
            'olddemons)
   (:= (get ptr 'demons) (append (get ptr 'newdemons)
                                 (get ptr 'olddemons))))

; ALIVE returns those demons to NULLPATHS which have not yet had
; their paths reduced to NIL.

(dex alive (dem ptr)
     (cond ((null dem) nil)
           ((exhausted dem ptr) (cleanup dem) nil)
           (t dem)))

; A demon is EXHAUSTED if its path has been reduced to NIL.
; When this occurs, a plot unit is created by UNIT-GEN.  This
; action may add demons for more complex plot units to the
; p-list for PTR.

(dex exhausted (dem ptr)
    (cond ((null (demon:path dem))
           (unit-gen dem ptr)
           t)
          (t nil)))
```

```
; When a demon is EXHAUSTED, we check over every affect state
; contained in the unit just recognized.  For each of these
; states we get rid of any demons with null paths.  This prevents
; duplication of plot units.

(dex cleanup (dem)
     (for (x in (noduples (demon:a-states dem)))
          (do (:= (prop 'demons x)
                  (for (y in (get x 'demons))
                       (filter (cond ((null (demon:path y)) nil)
                                     (t y))))))))


; ***************************************************************
;                          REPORT MODULE
; ***************************************************************

; The REPORT function employs two manners of presenting the data
; on the top level plot units recognized above.  First, each
; top-level unit is numbered and listed as before, then a list of
; the top-level units, their degree and their relatives is given.

(dex report nil
  (let (unit-list (reverse
                   (for (x in all-units) (y in all-unit-ids)
                        (filter (cond ((null (get y 'subsumed))
                                       x))))))
    (:= state-lst (for (x in unit-list)
                       (save (unit:aff-states x))))
    (msg t t "There are " (length unit-list)
             " top-level plot units:" t t)
    (let (n 0)
      (for (x in unit-list)
           (do (msg t "     Number = " (incr n)) (dump-unit x)
               (msg t))))
    (msg t t)
    (dump-info state-lst)
    (msg t t t)))

; Creates a list of numbered elements from the given list.

(dex make-list (lst)
  (let (n 0)
     (for (x in lst) (save (cons (incr n) x)))))

; This data structure holds a list of affect states (A-LIST) for a
; given top-level plot unit (specified by NUMB).

(record-type numbered-list nil (numb . a-list))
```

```
; The affect states for each top-level plot unit are listed in
; STATE-LIST, created above by REPORT.  By conductiong a simple
; intersection search, the related units (those which share
; affect states) can be identified.

(dex gather-relatives (lst)
   (for (x in lst)
        (save (cons (numbered-list:numb x)
                    (find-relatives x lst)))))

(dex find-relatives (x lst)
   (for (y in lst)
        (filter (cond ((intersection (numbered-list:a-list x)
                                     (numbered-list:a-list y))
                       (numbered-list:numb y))))))

; We cycle through the STATE-LIST outputting the top-level unit
; number, degree and relatives.

(dex dump-info (state-list)
       (msg t t "Unit Families: " t t)
       (loop (result nil)
             (initial master (gather-relatives
                              (make-list state-list)))
             (while master)
             (do (msg t (numbered-list:numb (car master))
                        " Degree = "
                        (sub1
                          (length (numbered-list:a-list
                                   (car master))))
                        " Relatives = "
                        (remove-element
                          (numbered-list:numb (car master))
                                    (numbered-list:a-list
                                      (car master)))))
             (next master (cdr master))))
```

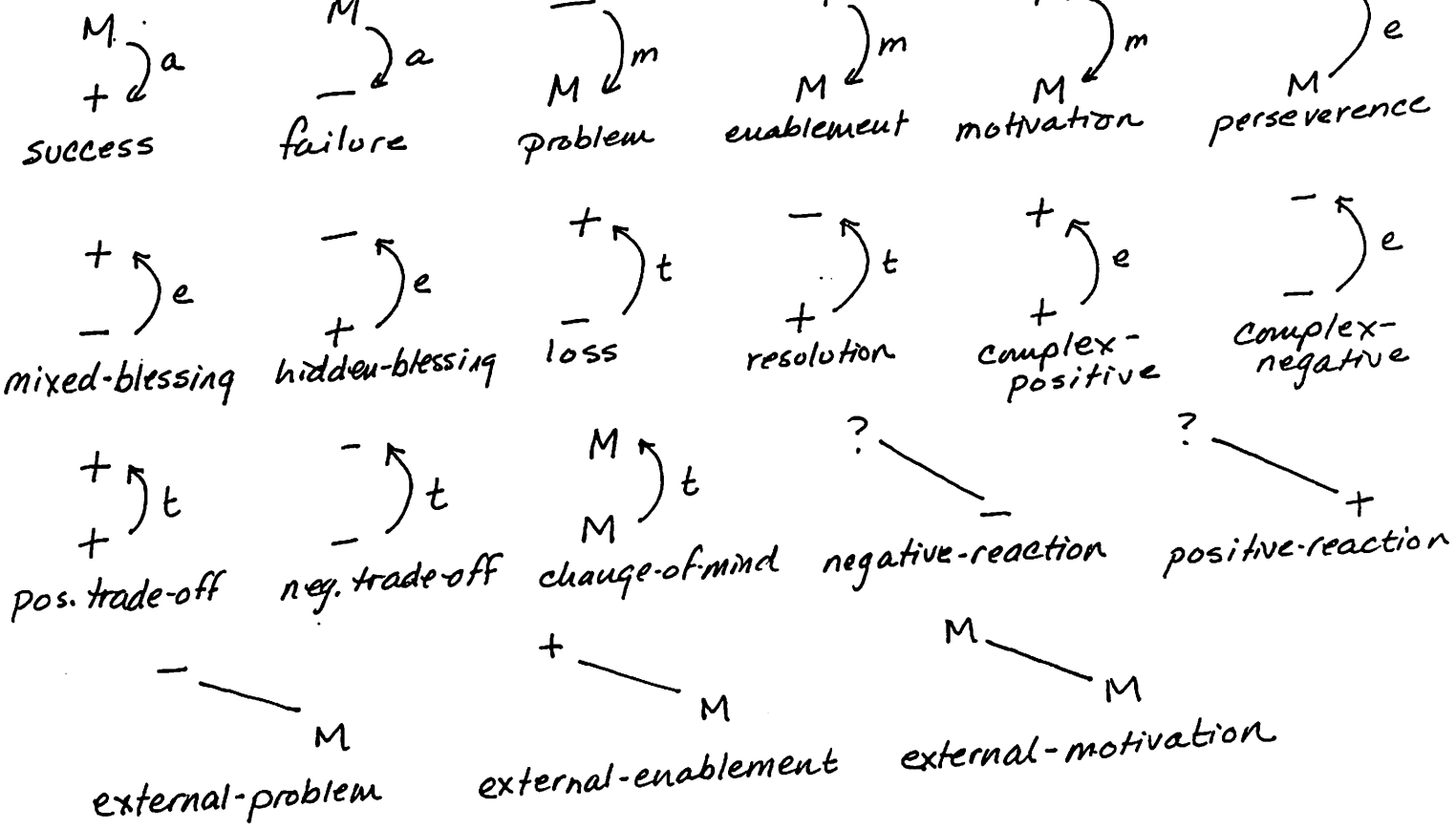
```
; ****************************************************************
;                COMPLEX PLOT UNIT SET FOR MICRO-PUGG
; ****************************************************************

; Each complex plot unit is based on a primitive or other complex
; plot unit.  These are given as the first SUBUNIT step for each
; PATH.  These "plot unit definitions" are demon structures that
; are stored on the SPAWNS p-list for the basic units (those given
; in the first SUBUNIT step.)  When the basic unit is recognized,
; these definitions are retrieved and placed on the DEMONS p-list
; for the current affect state.

(defprop success (
        ((subunit success) (takes . t) (-s) (subunit loss))
        fleeting-success nil nil)                              spawns)

(defprop fleeting-success (
        ((subunit fleeting-success) (gives . m) (ms)
         (subunit problem) (subunit perseverence))
        starting-over nil nil)                                 spawns)

(defprop enablement  (
        ((subunit enablement) (gives . a) (+s)
         (subunit success))
        enabled-success nil nil)                               spawns)

(defprop problem  (
        ((subunit problem) (gives . a) (+s) (subunit success))
        success-born-of-adversity nil nil)                     spawns)

(defprop success-born-of-adversity  (
        ((subunit success-born-of-adversity) (subunit resolution))
        intentional-problem-resolution nil nil)                spawns)
```

Plot Unit Definitions

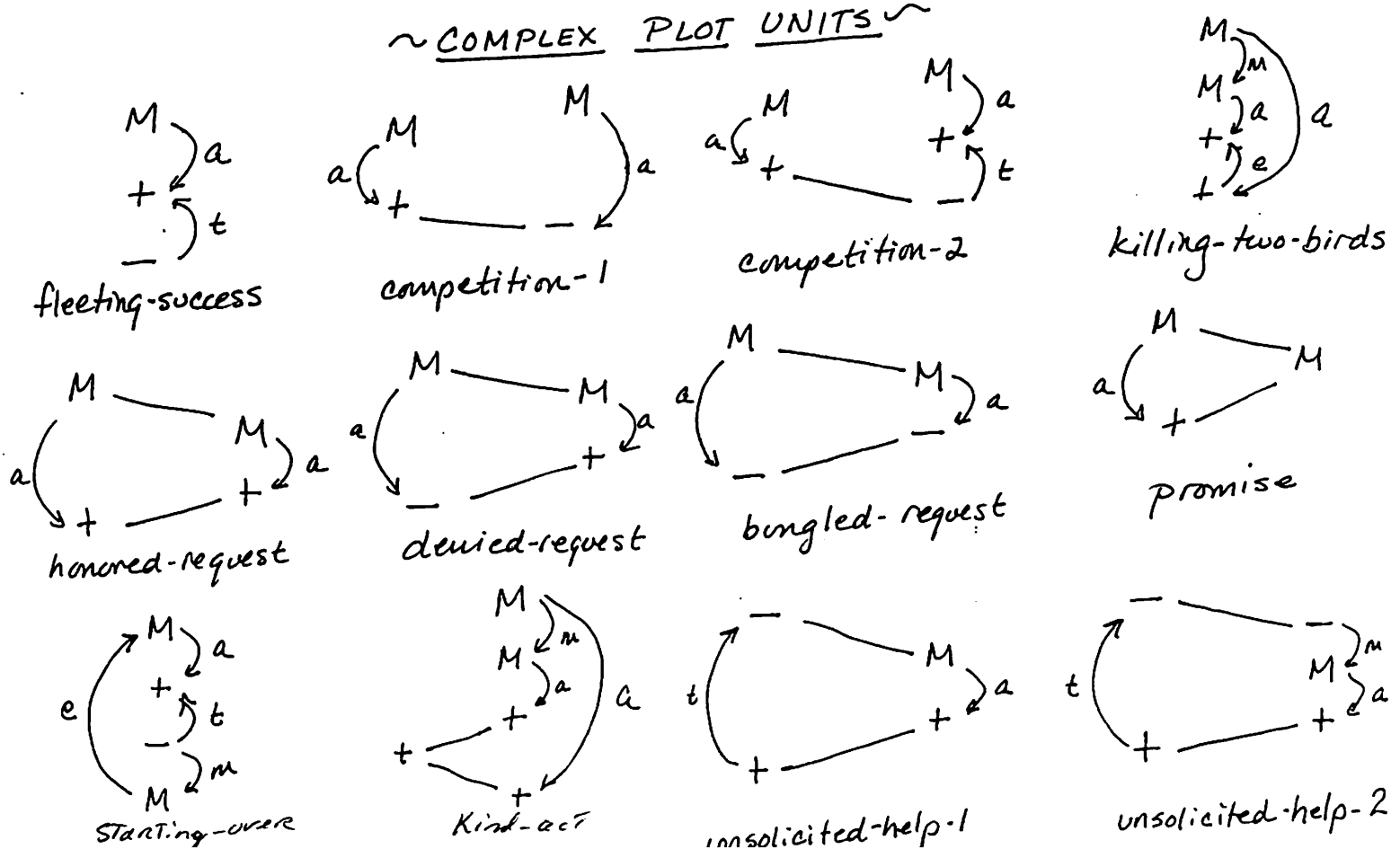
The following pages contain a listing of the set of plot units currently in use.

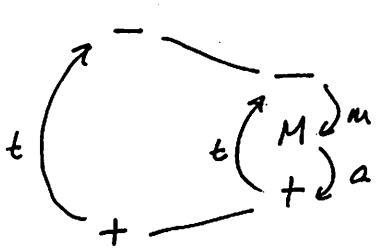
Plot Unit Set #2.2:

~PRIMITIVES~

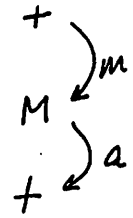


~COMPLEX PLOT UNITS~

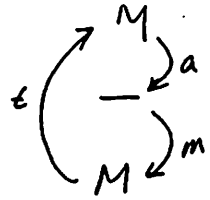




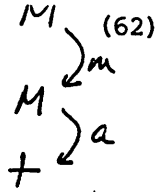
unsolicited-help-3



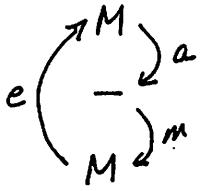
enabled-success



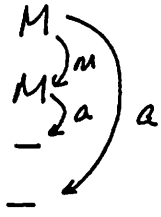
giving-up



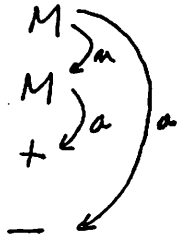
motivated-success



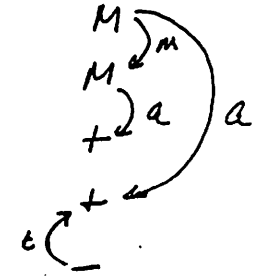
perseverence-after-failure



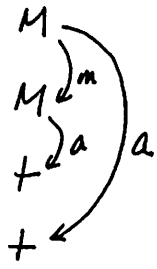
subgoal-failure



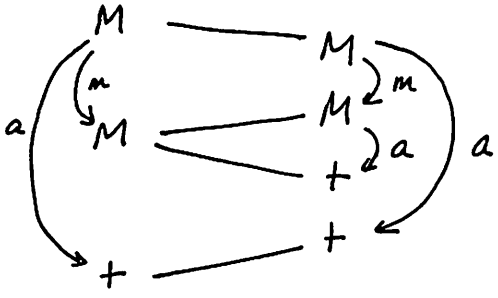
top-level-failure-1



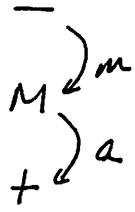
top-level-failure-2



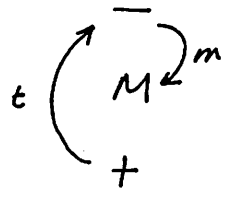
nested-subgoals-1



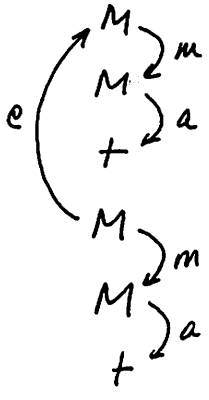
request-honored-after-conditional-promise



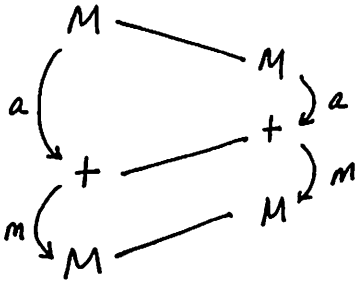
success-born-of-adversity



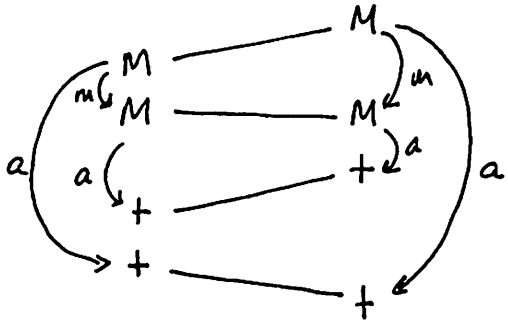
fortuitous-problem-resolutio



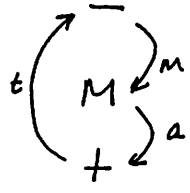
sequential-subgoals



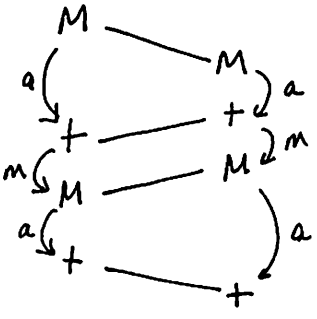
obligation



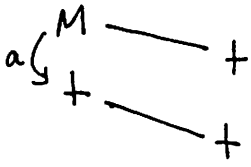
request-honored-after-conditional-request



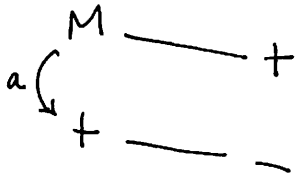
intentional-problem-resolutior



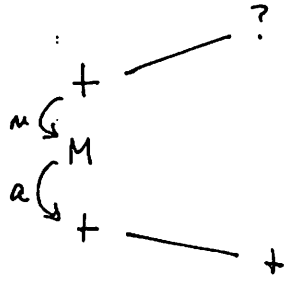
returned-favor



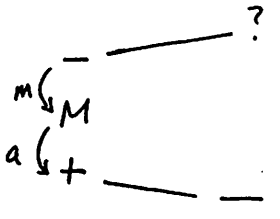
honored-promise



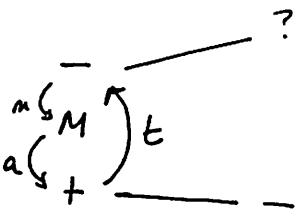
reneged-promise



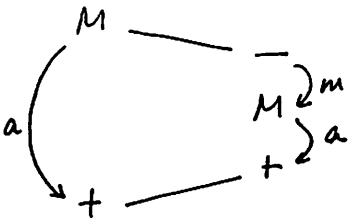
reward



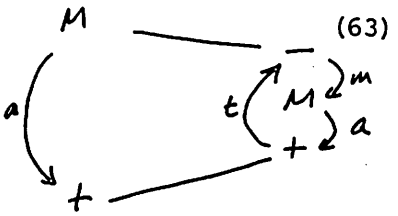
retaliation-1



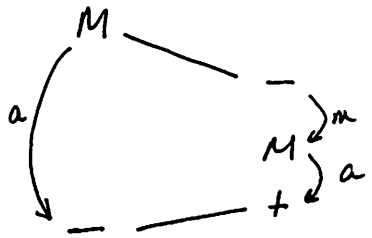
retaliation-2



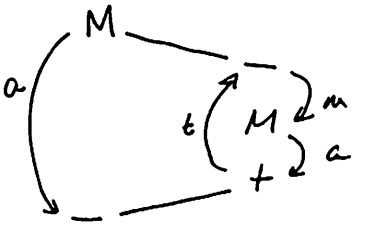
effective-coercion-1



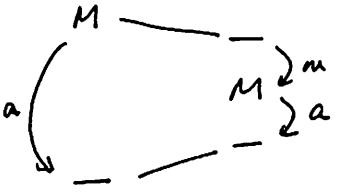
effective-coercion-2



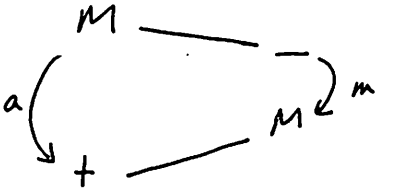
ineffective-coercion-1



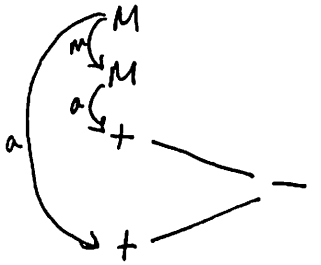
ineffective-coercion-2



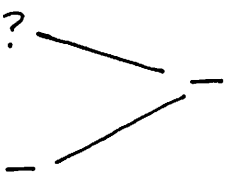
bungled-coercion



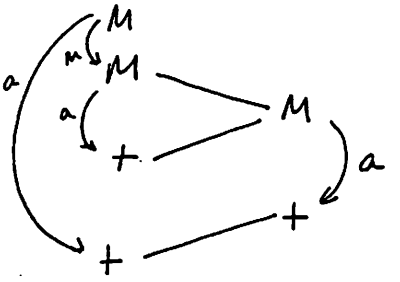
promise-under-duress



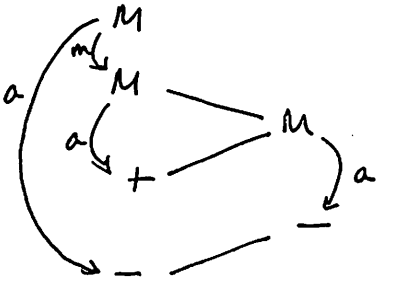
malicious-act



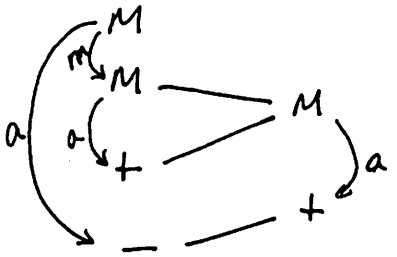
regrettable-mistake



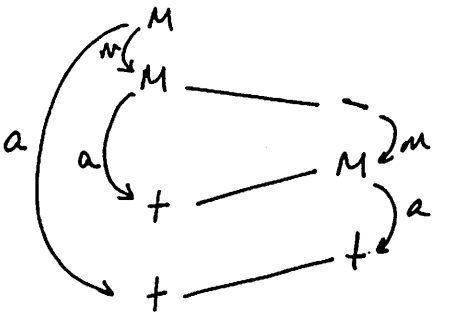
promised-request honored



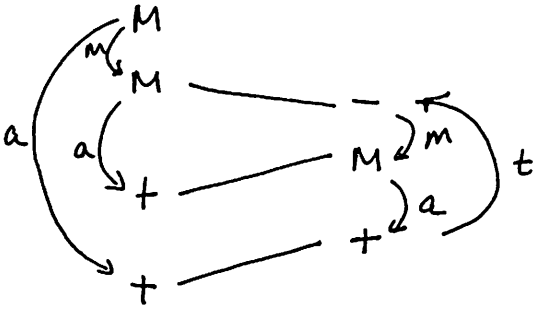
promised-request bungled



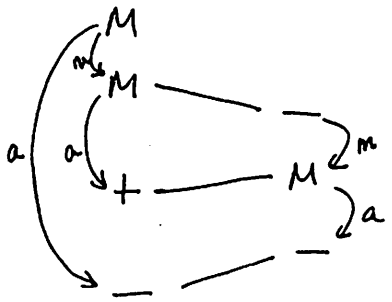
double-cross



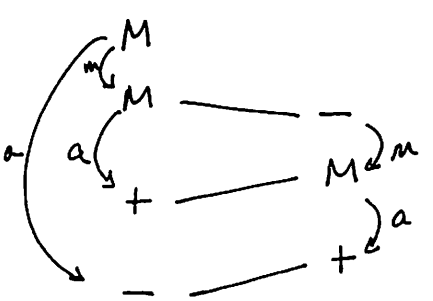
promise-honored-under-duress-1



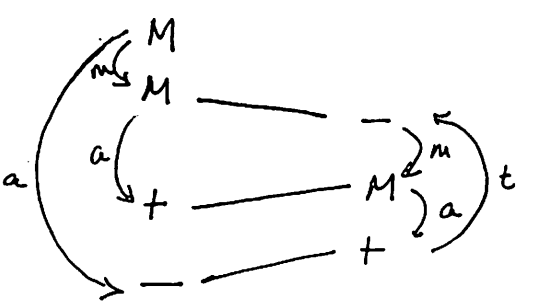
promise-honored-under-duress-2



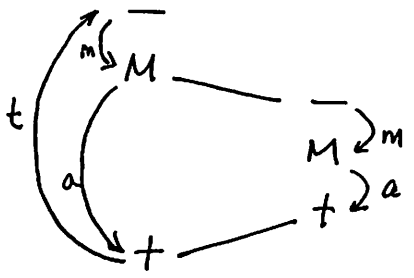
promise-bungled-under-duress



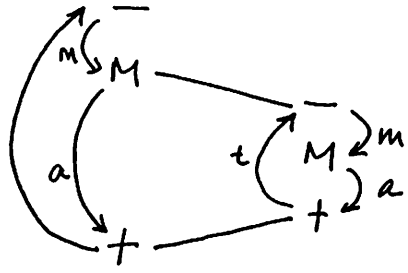
double-cross-under-duress-1



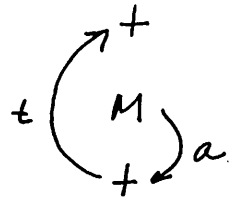
double-cross-under-duress-2



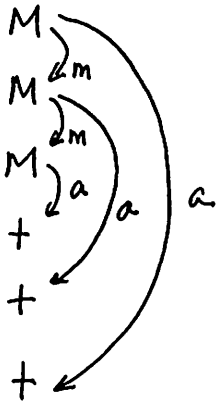
problem-resolution-by-coercion-1



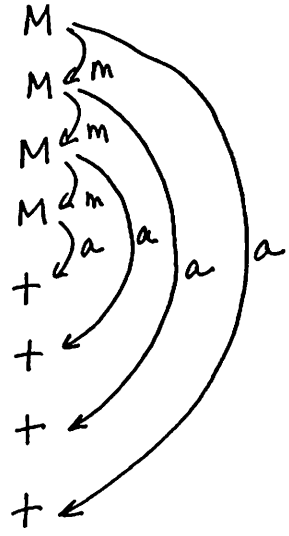
problem-resolution-by-coercion-2



sacrifice

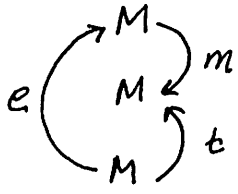


nested-subgoals-2

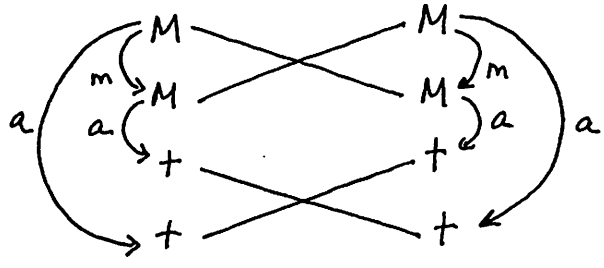


nested-subgoals-3

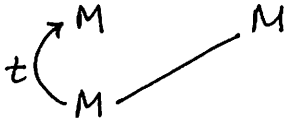
overriding-goal-conflict



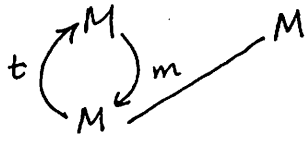
overridden-subgoal-conflict



simultaneous-exchange

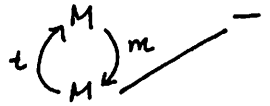


persuasion-1

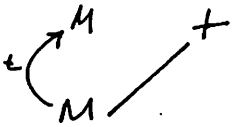


persuasion-2

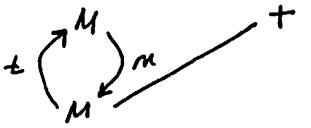
external-problem-change-of-mind.



external-problem-overriding-goal-conflict



external-enablement-change-of-mind



external-enablement-overriding-goal-conflict

References

Carbonell, J. G., Jr. (1978) POLITICS: Automated Ideological Reasoning, Cognitive Science, 2:1 pp 27-51.

Cullingford, R. (1978) Script Application: Computer Understanding of Newspaper Stories, Doctoral Thesis, Yale University, New Haven.

Dyer, M. (1983) In-depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension, Cambridge, Mass.: MIT Press.

Fillmore, C. (1968) The Case for Case, in Bach, E. and Harms, R. (Eds.), Universals in Linguistic Theory, New York: Holt Rhinehart & Winston.

Gentner, D. (1981) Verb Semantics and Sentence Memory, Cognitive Psychology, Vol. 13, New York: Academic Press.

Graesser, A. (1981) Prose Comprehension Beyond the Word, New York: Springer Verlag.

Hayes-Roth, Frederick and Waterman, D. A., (1978) eds. Pattern Directed Inference Systems, New York: Academic Press.

Kintsch, W. (1974) The Representation of Meaning in Memory, Hillsdale, N.J.: Lawrence Erlbaum Associates.

Lehnert, W. (1979) "Representing Physical Objects in Memory" in Philosophical Perspectives in A.I. (ed: Martin Ringle) Humanities Press. New Jersey.

Lehnert, W. (1982) Plot Units: A Narrative Summarization Strategy, in Lehnert, W. and Ringle, M. (Eds.), Strategies for Natural Language Processing, Hillsdale, N.J.: Lawrence Erlbaum Associates.

Lehnert, W. (1983) "Narrative Complexity Based on Summarization Algorithms," Proceedings of the Seventh International Joint Conference on Artifical Intelligence, Karlsruhe, Germany.

McDonald, D. (1983) "National Language Generation as a Computational Problem - an introduction" in Brady, M. and Berwick, R. (Eds.) Computational Models of Discourse, Cambridge, Mass.: MIT Press.

Rumelhart, D. and Norman, D. (Eds.) (1975) Explorations in Cognition, San Francisco: W.H. Freeman and Co.

Rieger, C. (1975) Conceptual Memory and Inference in Schank, R., Conceptual Information Processing, Amsterdam: North-Holland Publishing Co., Inc.

Schank, R. (1975) Conceptual Information Processing, Amsterdam: North-Holland Publishing Co., Inc.

Schank, R. and Abelson, R. (1977) Scripts, Plans, Goals and Understanding: an Inquiry into Human Knowledge Structures, Hillsdale, N.J.: Lawrence Erlbaum Associates.

Wilensky, R. (1980) Understanding Goal-based Stories, New York: Garland Publishers.

Wilks, Y. (1978) "Good and Bad Arguments for Semantic Primitives," Communication and Cognition 10, pp 181-221.

Woods, W. (1970) "Transition Network Grammars for Natural Language Analysis" Communications of the ACM 13:10.

Winograd, T. (1972) Understanding Natural Language, New York: Academic Press.