Draft Alma-Carne manual

K. Purang

April 1, 2001

Contents

Chapter 1

Introduction

Draft Alma manual

Active logics [2, 4] are have the following characteristics:

- they are situated in time
- they maintain a history
- they tolerate contradictions
- they enable meta-reasoning to be done

These characteristics make active logics suitable for use in various domains including time situated planning and execution [6, 9], reasoning about other agents' reasoning [3], discourse context updating [5], computation of Gricean implicatures [7], representation of meta and mixed-intiative dialog [8, 1].

For each of the domains above, it has typically been the case that when implemented, a new special purpose active logic had to be programmed. This document describes the implementation of a general-purpose active logic engine called Alma (Active Logic MAchine). The aim is that it should be possible to specify and execute any active logic using the language of Alma.

It is usually not sufficient to compute consequences of an active logic, we want the execution of the logic to be sensitive to the outside world and to have the logic affect the outside world. This is made possible by a procedure execution system called Carne that runs in conjunction with Alma. This document describes the Alma-Carne system.

After an overview of the Alma-Carne system, details of Alma are presented. First a description of Alma, then the method of running Alma and finally an account of the implementation. The same information is then presented for Carne. The next section describes some applications of the Alma-Carne system. The last two sections discuss the current bugs and the future development of the system. The appendices include a keyword reference and illustrations of running Alma-Carne.

Chapter 2

Alma-Carne

The Alma-Carne system consists of Alma which is an implementation of active logic and Carne which allows Alma to run prolog procedures asynchronously. Alma can cause programs to be executed in Carne through a reserved predicate in the Alma language. Carne can assert or delete formulas from the Alma database. This is the main mechanism used to get results of the procedures back from Carne and for Alma to observe get input from the world.

2.1 Characteristics of Alma-Carne

2.1.1 Asynchronicity

Alma and Carne run asynchronously. Alma makes a request to Carne and goes on doing the usual inference without waiting for a reply from Carne. Whenever an answer is available, it is asserted by Carne into the Alma database.

This allows the system to perform long computations without halting the inference. On the other hand, the formulas in Alma must be written to take this into account. Also there is no guarantee in general when an answer to a Carne query will return.

Similarly, inputs from outside the system are added to Alma whoever they occur so that Alma does not have to specifically wait for inputs.

2.1.2 State in Carne

Carne is meant to execute programs that run to completion once started. The procedures in Carne should ideally not have memory of past executions of that or of other procedures in Carne. In this way the only state information is kept in Alma and this can clarify the design of applications.

If Alma is meant to be connected to a running procedure that keeps its own state and needs to interact with Alma over a longer period of time, Carne can be used to transmit messages to and from that external system. In this way, Carne runs simple message transmission procedures that complete within a short time and do not preserve state. The dialog manager application of Alma-Carne (see later) uses Carne in this fashion. Carne builds and translates messages from several modules in the Trains system.

2.1.3 Carne serialization

Carne procedures execute serially. If a request is sent from Alma to Carne before a previous request has been completed, the new request will wait until Carne is done with the previous one. If this situation is likely to occur frequently, it may be more convenient to configure Carne as a message transmission system connected to servers that compute the requests.

2.1.4 Carne to Alma requests

It is not straightforward for a Carne procedure to make requests for information from Alma while they are running. This is consistent with the aim of running relatively simple procedures in Carne. Once again, if the procedure that one wishes to write needs to have extensive interaction with Alma, it is better written as an independent server.

2.1.5 Carne and eval_bound

Prolog procedures can be executed within the Alma process using the `eval_bound` operator (see later). This is meant to execute very short procedures within an Alma step. Doing so may slow the inference in Alma since it has to wait for the procedure to terminate to continue inference, on the other hand, the overheads of sending a message to Carne and getting one or more replies back is avoided.

2.1.6 What is Carne used for?

From the above, it can be seen that Carne is meant to be used for medium sized procedures that do not need state to be maintained. Smaller procedures can be implemented in Alma itself and larger procedures are better executed in external independent processes with Carne being used to transmit messages between Alma and the process.

Some examples of procedures used in Carne are:

- Input. In response to external events that Carne monitors, new information can be added to Alma.
- Output. Similarly, Alma can cause Carne to produce arbitrary outputs to the world.
- Computation. There are several kinds of computation whose results are needed by Alma but which are not conveniently or efficiently described in the logic. These can be executed in Carne. Since Carne is a separate process from Alma, this computation does not affect the performance of Alma– Alma can continue reasoning while computationally intensive work is done by Carne.
- Communication. Carne can be used to connect Alma to external processes. Carne procedures can translate from Alma formulas to the language accepted by the external processes and the converse.

2.2 Specifying the system

An application using Alma-Carne can be specified using three files:

- The Alma file. This is a file containing Alma formulas that specify the inference to be done by the system.
- The Alma procedure file. This contains the short procedures that are executed in Alma within a step.
- The Carne procedure file. This contains the procedures that can be called from Alma.

An important part of the design of an application is deciding what parts of the system are to be implemented in the logic and which as procedures. More flexibility can be gained by implementing more of the system in Alma, however the cost is lower efficiency.

Chapter 3

Alma

3.1 Overview of Alma

Alma is an executable active logic which proceeds in steps. At each step, the rules of inference are applied to the formulas in the database at that step to produce a set of new formulas. These are added to the database and the process repeats.

3.1.1 Time

The current step number is represented in the database and changes as the program executes. Formulas can therefore be written that are sensitive to the current time.

3.1.2 Referring to formulas

The formulas in the database have names which can be user specified or assigned by the logic. The names allows the user to assert properties of the formulas and to reason about these properties. One can for instance, assert that a particular formula is to be preferred to another, or record what the source of the formula is, or the probability of the formula being true and so on.

3.1.3 Properties of formulas

The logic maintains information about various properties of the formulas in the database, including the derivations of the formulas, their consequences and the time at which they were derived. These properties are available for reasoning through reserved predicates.

3.1.4 Inconsistency

In general the set of formulas in the database may be inconsistent. This eventually leads to the derivation of a literal and of its negation. This direct contradiction is detected by the logic and the contradictands as well as their consequences are made unavailable for further inference. The fact that these formulas are contradictory is also asserted in the database. This, together with the properties of the formulas can be used to resolve the contradiction. If either of the contradictands is thought to be true, that formula can then be reinstated in the database.

3.1.5 Procedures

Some computations that need to be done in the logic may be more easily, conveniently or efficiently done through procedures. To enable this, prolog programs can be specified as inputs to Alma. These can be invoked when needed through the formulas. An alternative for longer running procedures is Carne.

3.2 Syntax of the Alma Language

The following grammar specifies the syntax of the alma language.

$$
\begin{aligned}
ALMAFORMULA &\longrightarrow FORMULA \\
ALMAFORMULA &\longrightarrow FFORMULA \\
ALMAFORMULA &\longrightarrow BFORMULA \\
FORMULA &\longrightarrow and(FORMULA, FORMULA) \\
FORMULA &\longrightarrow or(FORMULA, FORMULA) \\
FORMULA &\longrightarrow if(FORMULA, FORMULA) \\
FFORMULA &\longrightarrow fif(CONJFORM, conclusion(POSLIT)) \\
BFORMULA &\longrightarrow bif(FORMULA, FORMULA) \\
FORMULA &\longrightarrow LITERAL \\
CONJFORM &\longrightarrow and(CONJFORM, CONJFORM) \\
CONJFORM &\longrightarrow LITERAL \\
LITERAL &\longrightarrow POSLIT \\
LITERAL &\longrightarrow NEGLIT \\
NEGLIT &\longrightarrow not(POSLIT) \\
POSLIT &\longrightarrow PREDNAME(LISTOFTERMS) \\
LISTOFTERMS &\longrightarrow TERM \\
LISTOFTERMS &\longrightarrow LISTOFTERMS, TERM \\
TERM &\longrightarrow CONSTANT \\
TERM &\longrightarrow FUNCNAME(LISTOFTERMS) \\
TERM &\longrightarrow VARIABLE \\
VARIABLE &\longrightarrow Prolog\ variables \\
CONSTANT &\longrightarrow Prolog\ constant \\
PREDNAME &\longrightarrow Prolog\ constant \\
FUNCNAME &\longrightarrow Prolog\ constant
\end{aligned}
$$

3.3 Intended meaning

The language is very similar to a first order language and the meaning of the connectives follows from that. The main differences are bif and fif:

- $bif(X, Y)$ means the same thing as $if(X, Y)$ except that it is only used in backward searches.

- $fif(X, conclusion(Y))$ is used as a rule of inference. If all the literals in the premise are in the database, then the conclusion is added. The behavior of fif is different from that of if in that like inference rules, we cannot derive new formulas from a fif except for the conclusion.

There are also a number of reserved predicates. These will be described in detail later.

3.4 Inference rules

There are a number of types of inference rules used: resolution; fif rules defined by the user; the contradiction detection rule; the clock rule; and rules interpreting the reserved predicates.

3.4.1 Resolution

The main inference rule is resolution. This is used in both forward can backward chaining procedures. If two formulas resolve, application of the rule causes the resolvent to be added to the database at the next step. Formulas with main connective `fif` are not used with resolution. There are also constraints to the used of `bif`.

3.4.2 User rules

Using fif, inference rules can be added to the logic. The rule consists of a conjunction as premise and a literal as conclusion. The conclusion is added to the database if all of the literals in the premise are in the database at the time the rule is invoked. This does not allow one to specify rule schemas.

`fif` formulas are similar to `if` except for the following:

- We cannot derive new `fif` formulas from old ones.
- The premises of the `fif` rule have to be in the database at the same time. In the case of `if`, the conclusion is still obtained even if not all the antecedents are in the database at the same time.

Consider a database containing `if(and(P, Q), R)`. If `P` but not `Q` is available, resolution results in `if(Q, R)`. `P` can then be removed from the database. This will not affect the derived formula. If `Q` is then added, `R` will be derived. At no time do we have both `P` and `Q` in the database, yet `R` is derived.

If instead of `if(and(P, Q), R)` we had `fif(and(P, Q), R)`, the presence of `P` alone cannot give rise to any new formula. If `P` is removed and `Q` added, nothing happens either. We will only get `R` if both `P` and `Q` are in the database at the same time.

3.4.3 Contradiction detection

The Alma database can become inconsistent and this leads to false formulas being derived. As the logic executes, more false formulas can be derived. At some point though the inconsistency must give rise to a direct contradiction and this can be detected by Alma. A minimum number of actions are taken to prevent further derivations of false formulas but that does not resolve the inconsistency. Alma however gives the user the possibility to write theories that will reason about the contradiction and gives the possibility to resolve the contradiction by removing or reinstating the appropriate formulas.

Detecting contradictions

The contradiction detection rule has as premises $\phi, \neg\phi$ where ϕ is a literal. The detection of the direct contradiction causes the following actions:

- A formula of the form `contra(N1, N2, T)` is added to the database where $N1$ and $N2$ are the names of the contradictory literals and T is the step number at which the contradiction has been detected.
- The contradictands and their consequences if any are removed from the set of formulas that can be used for further inference. However they remain in the database so that they can be reasoned about. These formulas are "distrusted".
- Formulas of the form `distrusted(N)` are added to the database where N is the name of a formula.

Once an inconsistency gives rise to a direct contradiction, the contradictands and their consequences are removed from consideration for further inferences. This prevents unknown formulas from being derived from the contradiction but the database may still be inconsistent and therefore generate new contradictions since this scheme does not determine the "root cause" of the contradiction.

The next step is to try and resolve the contradiction and perhaps the inconsistency. If while the contradiction has not been resolved any of the contradictands reappears through another derivation, a new contradiction is asserted, and that new formula is distrusted.

Resolving contradictions

Alma does not provide any automatic way to resolve contradictions. The implicit relations provide information that can be useful in deciding what the resolution of the contradiction should be. These include the derivations of formulas, the time at which they were derived, the consequences of the formulas and so on. Detection of a contradiction could also trigger a query to the user as to the solution. See elsewhere for applications of Alma that resolve contradictions.

3.4.4 Reserved predicates

The presence of the reserved predicates in formulas causes various actions to be performed. Details of these are presented below.

3.5 Inference procedures

3.5.1 Forward inference

The usual mode of operation for Alma is in the forward direction. The way this is seen in active logics is to apply all the rules of inference possible to all the formulas available in the current step to get the formulas in the database at the next step. One rule of inference that should be included in these is the inheritance rule.

Practical forward inference

However this is rather inefficient and this implementation first of all implicitly applies the inheritance rule to all the formulas and also applies the inference rules to the new formulas only.

The new formulas from the previous step are the ones resulting from applications of inference rules or additions to the database at that step.

All applicable inference rules are applied to the new formulas and between the new formulas and between the new formulas and the old ones. The resulting formulas and the new formulas added to the database at that step from external sources become the set of new formulas for the next step.

This account is not entirely accurate in the details. A more accurate view of the procedures is given in a later chapter.

Deviation from active logics

This change in the proof procedure causes some changes from active logics:

- Since all formulas are inherited by default, we need a special operation to delete formulas form the database. Such an operator is not needed if we do not inherit by default.
- Deletion of formulas or non-inheritance can cause changes in the database when these formulas are used in formulas with negative introspection. If we have an implication that depends on the absence of ϕ to assert ψ, then the non-inheritance of ϕ should cause ψ to be asserted if we apply all the inference rules to all the formulas at each step. This is not the case when we explicitly delete formulas.

3.5.2 Backward inference

Instead of forward chaining to an answer, Alma allows one to do backward chaining to find whether some specific formula is derivable. Since resolution is not complete, the ability to do backward chaining helps completeness.

How is it done

The backward chaining is done in the usual way as a proof by contradiction but in steps. We move back one step in the backward search tree at each Alma step. So, starting a backward search will result in the answer being asserted in the database at a later step. The presence of a contradiction while doing a backward chaining proof has to be treated differently from other contradictions.

`bs(X)` will start a proof by contradiction for X. The search is done in a breadth first manner and at each step the depth of the search is increased by one. This is to be contrasted to the depth first strategy of prolog. Here, we are guaranteed to find the shortest answer at the cost of space.

Results

If X is provable, it will eventually be asserted in the database. At that point, the search can be stopped or it can be continued if one wants to search for other solutions. A parameter (`delete_bs_trees`) is available to specify this behavior.

If X is not provable at the time the search was started, but becomes provable at a later time, by perhaps the addition of missing axioms to the database, X will then be proved and added to the database. The bs "waits" until X is provable.

Why backward search?

Since backward search also proceeds in steps and goes "at the same speed" as the usual forward search, the necessity of backward search may be questioned. Apart from the possibility of the forward search to miss some consequences because of the non-completeness of resolution, there is an efficiency concern also.

The use of `bif` connectives makes the formulas usable only in backward searches. Forward search will not use these formulas to resolve with other formulas. If these formulas generate very many consequences most of which are nor

useful, computing them in a forward direction can be inefficient. It can be better to have them as `bif` and to find their consequences when needed. Backward search also uses the `if` implications so there is no need to duplicate those as `bif`.

3.6 Interfaces

There are a few interfaces to Alma:

- **Interface to Carne.** If the machine name and port number of a Carne process is made known to Alma, it will attempt to connect to that Carne process. Then $call$ commands in Alma formulas are sent to Carne. The status of the $call$ is asserted in the Alma database. There are four possibilities: $doing$ if the call has been sent to Carne but there is no response yet; $done$ if Carne has completed the action successfully $failed$ if the action has failed in Carne. The other responses from Carne are sent through the same connection and the appropriate action: addition or deletion of formulas is made in the Alma database.
- **Interactive interface.** With a "keyboard true" argument, Alma runs in interactive mode. A prompt is displayed and the commands listed elsewhere are available.
- **The Alma files.** Alma files contain formulas in the Alma language that are loaded into the database at start-up time. Any ill-formed formulas are ignored.
- **The prolog files.** User defined computations to be run in Alma by $eval\_bound$ or $call$ are loaded as prolog files.
- **The history file.** The history file is written as computation proceeds contains a list of the additions and deletions made to the Alma database at each step together with the name of the formulas and its derivation. See the appendix for an illustration.

  From the history file one can recreate the evolution of the database which is useful for debugging the logic. There have been tools that do that automatically, but are unfortunately not working anymore.
- **The debug file.** There are three debug levels that can be specified and these provide information about the internal execution of Alma. This is useful for debugging Alma. See the appendix for an example.

3.7 Reserved predicates

The reserved predicates in Alma can be roughly divided into three groups:

- Predicates that are asserted by the system to reflect the internal state of the logic.
- Predicates that allow one to calculate certain relations that are not explicitly stored in the database.
- Predicates that trigger various actions not related to computing relations.

These predicates are summarized here. A more detailed account is given in the reference section.

3.7.1 Commands

These are commands that cause Alma to do a non-logical action. Some of them are meant to be used mainly at the interactive interface, others, in Alma formulas.

In Alma formulas

These predicates can only be used in Alma formulas. To use them at the keyboard, they have to first of all be asserted in the database.

- `alma(X)`
  X is a callable prolog term that is executed in the Alma process at the beginning of the next step.
- `bs(X)`
  This commands alma to start a proof by contradiction for X. If X is provable, it will be asserted in the database. There are other commands and options to control this proof.
- `call(X, A, Y)`
  X is a callable prolog term that is sent to Carne to be executed. Y is a unique identifier. A is a list of formulas that are to be used by carne in the call, as well as a special assertion that tells carne to use data from the messages it has passed to alma. (Not so sure what this last thing means).
- `cdebug(X)`
  X is the name of a directory. This is to be used with $alma$ and will cause the history and debugging directories to be written in X with names Alma and ADebug.
- `eval_bound(X, Y)`
  X is a callable prolog term and Y is a list of variables. When all the elements of Y are bound, X will be executed and the whole `eval_bound(X, Y)` literal will be replaced by X if it succeeds. This is useful for computing implicit relations.
- `gensym(X, Y, Z)`
  X is a list of variables, Y is a list of constants of the same length as X and Z is a list of formulas that contain the variables in X. The result of executing this operator is that the formulas in Z will be asserted in the database with the variables in X replaced by constants formed from the corresponding element of Y.
- `reinstate(N)`
  N is the name of a formula that is reinstated in the database. This is useful for reinstating formulas after a contradiction.

At the interface

These can be used at the interactive interface but can also be used in Alma formulas by wrapping them with $alma$ or $eval\_bound$. If they are used in Alma formulas without the wrappers, the clause will be asserted in the database without the action expected taking place.

- `af(X)`
  X is an Alma formula that is added to the database at the next step. The formula will appear in the database two steps later. For details, see the implementation section.
- `bs(X)`
  This commands alma to start a proof by contradiction for X. If X is provable, it will be asserted in the database. There are other commands and options to control this proof.
- `df(X)`
  This commands alma to delete all formulas that subsume X and are subsumed by X.
- `distrust(N)`
  This distrusts N which is the name of a formula.
- `distrust_descendants(N)`
  This distrusts all the descendants of N (including N itself) which is the name of a formula.
- `named(F, N)`
  F is an Alma formula and N is a name to be used to refer to that formula. This can be used instead of af and it can also be used in the Alma input file.

3.7.2 Internal state

These predicates reflect the internal state of the logic and are asserted and removed automatically. These should not be asserted or deleted by the user.

- `distrusted(N)`
  This is added to the database if the formula named N is either a contradictand in a direct contradiction or is a consequence of a contradictand.
- `doing(X, Y)`
  This is true iff $call(X, Y)$ has been asserted and Carne has been requested to execute X. Once Carne has succeeded or failed X, this is deleted from the database. So it is in the database only while X is being executed.
- `done(X, Y)`
  This is true iff $call(X, Y)$ has been asserted and Carne has successfully completed the execution of X. It is added at the same time as `doing(X, Y)` is removed.
- `error(X, Y)`
  This is true iff $call(X, Y)$ has been asserted and Carne has failed the execution of X. It is added at the same time as `doing(X, Y)` is removed.
- `now(t)`
  This is a predicate fluent where t is the step number of the step the logic is in. This is added at a step and deleted at the beginning of the next. It enables formulas to use the current step number.
- `contra(X, Y)`
  This is asserted when there is a direct contradiction between the formulas named by X and Y. See later for more detail on contradictions.

3.7.3 Implicit relations

These are relations that are not explicitly represented in the database but can be computed when they are needed using $eval\_bound$. They represent mainly meta-information that is kept track of at a lower level than Alma formulas.

- `form_to_name(F, N)`
  This relates a formula to the name of the formula in the database. We assume that the formula `F` results in just one formula (clause) in the database.
- `gather_all(F, L)`
  This relates a literal F to the list of literals L in the database that are subsumed by F.
- `gather_all_u(F, L)`
  This relates a literal F to the list of names of formulas, L, whose formulas unify with F.
- `gather_all(F, T, L)`
  This relates a literal (or clause) F to the list of literals (or clauses) L in the database and not distrusted at time T that are unifiable with F. If $T \geq NOW$, this fails. A clause is written as a prolog list of literals.
- `name_to_children(+Name,-Children)`
  This relates formula names to the names of the formulas that have been derived from them in a one-step derivation.
- `name_to_derivation(+Name, -Derivation)`
  This is the closure of name_to_parents. The derivations are of the form deriv(F1, ListofListofderivofparents).

- `name_to_deriv_fringe(+Name, -ListOfNodes)`
  the $ListOfNodes$ is the list of node names at the fringes of the derivation tree. These are the are the axioms or observations that led to $Name$.
- `name_to_descendants(+Name, -Descendants)`
  This relates the name of a formula to the list of formulas derived from it.
- `name_to_formula(+Name, -Form)`
  This relates the name of a formula to the formula.
- `name_to_parents(+Name, -ListofParents)`
  This associates the name of a formula, to a list of lists of names of formulas used to derive that formula. We need a list of lists for the parents because a formula can have several derivations. Each of the lists represents one derivation of the formula.
- `name_to_time(+Name, -Time)`
  This relates formula names to the time they were first derived.
- `pos_int(F)`
  This succeeds if a variant of the literal F is in the database at the time at which $pos\_int(F)$ is executed. It can be used for negative introspection as
  $+pos\_int(F)$.
- `pos_int_u(F)`
  This is similar to `pos_int(F)` except that instead of succeeding for variants, it succeeds for unifications.
- `pos_int(F, T)`
  This succeeds if a variant of the literal (or clause) F is in the database and not distrusted at time T. If $T \geq NOW$, this fails. A clause is written as a prolog list of literals.

Chapter 4

Running Alma

Alma can be run in different modes depending on the interactions with other systems and on whether the steps are executed automatically or manually.

These modes are listed in order of increasing control by the user:

- In the first case, Alma steps automatically at regular intervals and interacts only with Carne. Carne can in turn be connected to a larger system in which Alma is to be embedded.

  The user has no direct control over Alma and all decisions about what to do depend on the Alma formulas and the input to Carne from the external system.

  This mode is perhaps what the system is ultimately used for when it is debugged and provides inference services to the larger application. In this case, Alma is run as a stand-alone executable and does not need a terminal so it can be run in the background.
- In demo mode, Alma executes a user defined script that allows the user to step through a sequence of actions. This is useful for illustrating Alma execution in general, or a specific application. The user can only respond to the demo program and cannot affect the system in other ways.
- One can also run Alma connected to a larger system and running automatically, but with access by the user to the Alma database and commands. Here Alma runs with input and output going to a terminal from which the user can inspect and modify the database. This allows the user to modify Alma's behavior while it is running. This can be useful for relatively minor debugging.
- Instead of allowing Alma to run freely as above, the user can control the steps so that Alma only steps when requested by the user. This would be the mode to use when there is a lot of debugging to be done and the user needs to be certain that each step is being executed properly.
- The above debugging concerned debugging of the logic that is run in Alma and the associated Carne programs. If there is a need to debug the Alma code itself, Alma can be run within a prolog process. The Alma source code is loaded in the prolog and the usual prolog debugging facilities will be available.

In each case Alma can produce a history and debug file that record the execution of the logic.

The rest of this chapter describes the commands and variables that can be used to control the execution.

4.1 Controlling Alma

We can control the execution of Alma through the command line arguments on invocation; or a set of commands available when Alma is run interactively; and a number of variables whose values can be set at runtime and which affect various aspects of the execution.

4.1.1 Command line arguments

The following are the command line args:

- `alfile X`
  X is the file containing active logic formulas. There can be several `alfile X` argument pairs to load several files. Default: none.
- `debug N Fname`
  $Fname$ is the name of the file in which the debugging information will go. N is the debug level:
  0 nothing
  1 show step numbers, and at that step: added formulas, deleted formulas, tcp input, tcp output.
  2 in addition to the above, show the agenda
  3 in addition, show successful resolutions
  The debug files can get very large and can slow down the operation of Alma considerably.
- `delay X`
  X is a real number that is the time delay between successive steps if there is no message to Alma from Carne. default: 0.1
- `deletetrees X`
  If X is true, once a solution is found to a backward search, the backward search is stopped. If it is false, the backward search continues to find more solutions. Default: false.
- `demo F`
  F is the name of a demo file that is to be run in Alma. See a later section for more details.
- `history X`
  X is the name of the file in which the history is written.
- `histoket X`
  X is the tcp file that contains the machine name and the port number for a process that will read the history output.
- `keyboard X`
  X is true or false. If true, Alma will listen to standard input. This is useful in using Alma interactively. default: false
- `load X`
  X is the name of the prolog file to be loaded. This file is meant to contain user-defined prolog programs that are to be run using $eval\_bound$ or $alma$. Default: none.
- `memlimit M`
  M is the maximum memory that Alma will use. If it exceeds this amount, the process will be terminated. Default is 100000000 (bytes). The default does not effectively limit Alma.
- `run X`
  X is true or false. This determines whether Alma goes into free-run mode or whether it needs to be commanded to execute each step. This is useful for interactive use. Default: true.
- `sfile X`
  X is the name of the tcpfile written by Carne. This file contains the machine name and the port number of the Carne process. Default: none.
- `timelimit T`
  T is the maximum CPU time that Alma will use. If it exceeds this time, the process is terminated. Default is 6000000 (s). The default does not effectively limit Alma.

4.1.2 Run time commands

These commands are available to the user at run time when Alma is running interactively:

- `sr` commands alma to step once.
- `sdb` commands alma to show the contents of the (primary) database.
- `af(X)` this works as described above.
- `df(X)` this works as described above.
- `reset_alma` this resets all the alma data structures to the original state except for open files and sockets. The input files of alma are read in once again after the state has been initialized.
- `dump_kb(X)` this dumps the alma primary database, the secondary database, the primary index and the secondary index into the file X.
- `help` this displays some help. This help is a bit old though.
- `query(Form)` can be typed at the keyboard and if Form is in the database and not distrusted, it will say so. Form should be a literal. This is useful to figure if some formula is usable.
- `halt` this exits alma.

4.1.3 Internal variables

There are a number of parameters that control the workings of Alma but are not available at the command line. These should perhaps eventually be read from a parameter file. The parameters can be manipulated either at the keyboard in an interactive Alma session or by using $eval\_bound$ with the appropriate prolog commands.

- `agenda_time(T)` If this is non-zero, it determines the maximum time Alma is to spend in one step to process the actions on the agenda. Initialized to 0
- `agenda_number(T)` This determines the maximum number of agenda items Alma will process in one step. Initialized to 2000.
- `delete_trees(T)` If this is true, then on the first success of a backward search, the search is stopped and no more solutions will be found. If it is false, then backward searches run forever. Initialized to true. **Need to make this a parameter.**
- `contra_distrust_ddescendants(X)` If this is true, when a contradiction is found, the consequences of the contradictands are distrusted, otherwise they are not. Initialized to true.

4.2 Running Alma in prolog

If we want to debug the Alma code or some user defined code in Alma, it might be necessary to run Alma in prolog. Alma runs in Quintus prolog 3.2 (qui). This requires access to the source code.

To run Alma in qui,

- Load the file `toplevel.pl`.
- Initialize alma by executing `initialize.` at the prolog prompt.
- Set the parameters specified that you would specify in the command-line by `handle_args(X)` where `X` is a prolog list of the arguments you would have used had Alma been run as an executable.
- From then on, depending on the arguments, the usual Alma run-time commands are available (see above).

In this mode, the detailed workings of Alma can be inspected and debugged.

4.3 Demos

Demos are very simple scripts that are written by the user and that are meant to illustrate some the execution of Alma or a particular application. The user starts the demo which automatically executes the commands specified and then exits.

The scripts specify the commands one would type at the interface together with prolog predicates for printing messages and getting responses from the user. The commands are represented by `command(X)` where `X` is a command one would type at the interface.

See the appendix for an example.

Chapter 5

Alma implementation

5.1 Overall design

The main modules of Alma are:

- **The toplevel**. This loads appropriate files, initializes the datastrucutres, processes the command line arguments and starts the execution loop if necessary.
- **The database manager**. This does the indexing, the storing and the retrieval for formulas. It also stores formulas in the history and writes it out to the history file if needed.
- **The inference module.** This does the following:
  - Finds the possible applications of the rules of inference to a list of formulas.
  - Applies the inferential actions to the formulas.
  - Defines the inferential actions for:
    - Forward chaining resolution.
    - Backward chaining resolution.
    - User defined inference rules.
  - Defines some special commands like $eval\_bound$
  - Does the resolutions.
- **The interface module**. This consists of three parts:
  - The user interface. This defines the code for
    - the commands available to the user in interactive mode.
    - the processing of Alma files
    - the execution loop
  - The external interface. This is concerned with the interaction to and from Carne.
  - The demo module. This interprets demo files. It essentially reads the commands in the file and executes them.
- **The parser.** This module converts Alma formulas to CNF form which are asserted in the database together with some extra information.
- **The contradiction handling module**. The work this does is described elsewhere.
- **The help module**. This is meant to give on-line help but it is a bit outdated.
- **The data structures module** This defines the data structures needed.
- The miscellaneous module: various miscellaneous jobs.

The Alma system also comes with the manual, html documentation, demo files and other example files.

5.2 Data structures

The data structures module defines the data structures that are used to represent the formulas as well as other information.

5.2.1 Formulas

Formulas in the language are stored in data structures that represent the following:

1. The name of the formula. Every formula has a name. If a name is not provided by the user, an integer is used.
2. The formula in CNF. The formulas are represented in conjunctive normal form. The distinction between if or fif or bif are maintained by labels (se below).
3. The type of formula. The type is whether the formula is used for forward or backward chaining. Some formulas can only be used for backward chaining.
4. The assumptions on which the formula is based. In backward chaining in particular, one assumes the negation of the formula to be proved. This is recorded here. Formulas with this nonempty typically have bc as type. There can be several assumptions for one formula.
5. The target of the formula in backward search. This is a single formula that will result if a contradiction is found with this formula.
6. The time at which the formula was first derived. The time here is the step number.
7. The priority of the formula. This is a number that can be used to set the priority of the inferential tasks.
8. The sets of parents of the formula. This records the set of the sets of formulas used in the derivation of this formula, each set corresponding to a distinct derivation.
9. The set of children of the formula. These are the immediate descendants of the formula.
10. Miscellaneous information, including whether conditionals are if or fif or bif. This is used to determine how the formula is to be used.

Some of the information about the formulas that are stored in that data structure are available for use in Alma formulas through $eval\_bound$.

5.3 Database

The database module provides procedures for storage and retrieval of formulas from the database. The formulas are stored in the data structures described above in the prolog database. The result of this is that prolog indexing is not efficient for the formulas themselves, therefore the database module implements indexing on the database. In addition to storage, retrieval and indexing, the database module implements the inference rules that apply to the secondary database described below. The primary and secondary database use similar procedures for access and indexing.

5.3.1 Primary and secondary databases

In addition to the (primary) database which is visible to the user, there is a secondary database that holds new formulas before they are added to the primary database. These new formulas are either the result of the application of inference rules on the formulas at the previous step or are added to the database from external sources.

At the next step, those formulas in the secondary database that cause contradictions with either other new formulas or old formulas are removed and a $contra$ formula added to the database. Duplicates are also detected and the derivations of the formulas are augmented and the formulas are removed from the set of new formulas.

Then the logical rules of inference (as opposed to the meta-logical operations described above) are applied among these "filtered" new formulas and between these formulas and the old ones. These new formulas are also added to the set of old formulas at this point.

The result of this is that 1. there is a delay of one step between an inference being made and the result appearing in the database 2. at least one of the contradictands does not have any consequences 3. the filtering has to be done to all the new formulas at every set and we can't choose to process just part of the new formulas.

5.3.2 Filtering

At the end of a step a number of new formulas are present that have been produced by applying the inference rules or by being added by the user or by Carne. These are the elements of the secondary database and are indexed by the secondary index. This module gathers these formulas, filters them and sends the resulting formulas to the next stage of processing where the inference rules are applied.

The filtering results in the formulas that are duplicates or contradictions to be removed from the set of formulas to be sent to the next stage.

We search for duplicates among the new formulas and between the new and the old formulas. Formulas ϕ and ψ are considered to be duplicates if they subsume each other. The duplicates represent alternative derivations for the formulas and are recorded in the list of parents for that formula. If the duplication is between a new formula and an old one, the new formula is not present in the filtered output, otherwise one copy of it is.

Contradictions too are searched for both within the new formulas and between the new formulas and the old ones. The formulas that are parts of contradictions are removed from the list, and the contradiction handling module is invoked. Details of this are presented elsewhere.

At the end of this, the new formulas are removed from the secondary database and the secondary index and the resulting filtered formulas are added to the primary database and index. The set of the new filtered formulas is then returned.

5.3.3 History

The formulas that are derived at each step as well as those that are deleted are added to a history. This therefore contains the additions and deletions to the database and can be used to recreate the evolution of the logic.

The history can be asserted in the logic, but this is not currently being done because there has been no need to access history in the applications to date.

The history can however be written out to a file through a command line argument.

5.3.4 Indexing

Since the formulas are not stored directly in the prolog database but in the formula data structures, we need to index the formulas for efficient access.

A hashing function maps the predicate and first argument of literals to be indexed onto integer keys. Special care has to be taken in case the first argument is itself a term or a variable. These keys are used as the first argument for the index terms that are asserted into the prolog database. The index terms have three arguments: the key, the list of formulas where the literal indexed occurs positively; the list where the literal occurs negatively. Different literals can hash to the same key and allowance needs to be made for that.

Therefore, given a new formula to index, for each literal in the formula, is if it is a negative literal, it is negated otherwise not, then the key is computed. This is then used to add the formula in the appropriate list (positive or negative in the index term).

During inference, we need to find the formulas that contain some literal either positively or negatively. The literal in question is hashed and the appropriate set of formulas can be returned. Here too, special attention has to be payed to the case that the first argument is a variable.

Unindexing formulas has to be done prior to removing the formulas from the database and is done by removing the formula from the entries for each of the literals of the formula.

5.3.5 Miscellaneous

Also part of the database module are formula data structure access functions which to retrieve and modify properties of the formula from the data structures. These are simple prolog functions. There are also several ways available to create new formula data structures.

5.4 Inference rules

The inference rule module in addition to defining the inference rules used, also finds candidates for inference rule application and applies the rules.

5.4.1 Candidate applications

Given the set of new filtered (see above) formulas in a step, we need to find the possible successful applications of the rules of inference. This is done by considering each literal in the formulas and finding rules that could apply. The form of the literal determines what rule is considered:

- `not(bs(X))` If this is the only literal in the clause, the backward search is started otherwise, other literals are processed first.
- `bs(X)` This is treated in a similar way.
- `call(X, Y)` If this is the only literal in the clause, the Carne call is made at the end of the step. Otherwise, other literals are processed first. There is no case for `not(call(X, Y))`.
- `gensym(X, Y, Z)` Similar to the above, this is processed if it is the last literal in a clause. In that case, the clause is deleted and the `gensym` done at the end of the step.
- `alma(X)` This is processed in a way similar to the above.
- `reinstate(X)` This too is processed if it is the only literal in a clause, but the processing is done immediately rather than waiting for the end of the step.
- `eval_bound(X, Y)` If all of the variables are bound, `X` is evaluated in prolog, the results are substituted back in the clause and processing continues with the new clauses.
- `not(eval_bound(X, Y))` Similar to above except that this time processing continues if `X` fails.
- *Default* In this case, we lookup the index for the set of formulas that have the same literal but of the opposite sign. These are candidates for resolution with the current clause. The inferential task that results depends on the type of clauses involved, rule used is either a forward chaining or a backward chaining or a user-defined inference rule. If there are no matches in the database, we continue with the next literal in the clause.

The inferential tasks generated in the above are added to the agenda which is a list of outstanding tasks.

5.4.2 Applying rules

Given the agenda, this section determines which of the inferential tasks to execute and executes them. The tasks can be given a priority that can then be used to order the agenda. Once the agenda is ordered, there are two possibilities: we can execute a fixed number of tasks or we can do as many tasks as can be done in a given time (seems not to be implemented). The tasks that are left over remain in the agenda and are reconsidered at the next step.

Each task in the set of tasks that can be generated by the previous section has a corresponding prolog function that implements that inference rule. Therefore, for each task, the corresponding function is called with the appropriate arguments.

5.4.3 Inference rules

Here we describe the prolog functions that are called to implement the inference rules. The tasks possible correspond to if, fif and bif formulas.

- if and bif tasks use resolution in the usual way. bif tasks have to take into account the assumptions under which the formulas are true. If the resolution is successful, the resolvent is made into a new formula that is asserted into the database.
- For the fif tasks, we verify whether each of the conjuncts in the antecedent is in the database, and compute the appropriate relations if necessary. If all the antecedents are in the database, the conclusion is added.

Other tasks

Other inferential tasks include the special cases of literals described above are not inserted into the agenda but are still processed with specific prolog functions.

- If we have a bs literal, a new formula is asserted which is the negation of the argument of the bs, with its assumption being the same formula. This is subsequently used with the usual resolution rule. If the empty clauses is the result, the formula is proven . The search here is breadth first. There are options that allow the user to specify whether the search should terminate after one solution is found.
- The *clock* rule does the following: it deletes the current $now(T)$ and asserts $now(U)$ where $U = T + 1$. This is added directly to the database without the one step delay described earlier.

Secondary database inference rules

There are two secondary inference rules that apply to the secondary database: contradiction detection and duplicate detection.

- Contradiction detection searches for literals that are contradictory. If P is in the database and not(P) is derived at step T, both are left asserted in the database, but they are made unusable for further derivations. contra(NP, NNP, T) is asserted together with distrusted(NP, T) and distrusted(NNP, T), where NP is the name of the formula "P" and similarly for NNP. Other formulas in the database that depend on P or not(P) are also made unusable and "distrusted" is asserted for these too. Leaving the formulas in the database make it possible to reason about them, but we cannot reason with them anymore.

  If while the contradiction has not been resolved P or not(P) reappears through another derivation, a new contradiction is asserted, and that new formula is distrusted.
- Duplication detection simply detects duplicate formulas and ensures that only one token of any formula type is asserted in the primary database. The derivations of the formulas are updated appropriately.

5.5 Interface

5.6 Parser

5.7 Contradiction handler

5.8 Help

5.9 Miscellaneous

5.10 Execution

Each step consists of the following actions by Alma:

1. Applies the secondary inference rules to the contents of the secondary database.
2. Clears out the secondary database and indices.
3. Looks if there are messages from either the Carne socket or the keyboard.
4. Lists the possible primary inference rules that can be applied to the newly added formulas in the primary database.
5. Selects which of these to apply. Currently, all possible inferences are made.
6. Applies these inference rules resulting in a new set of formulas in the secondary database.

Chapter 6

Carne

The specification of the Carne programs consists in writing prolog programs that will implement the procedures desired. Some predicates are available in Carne that enable Carne to affect the operation of Alma. These are described below. **This is as far as I remember. Things have been modified since I last worked on this, so I am not very sure of this description.**

6.1 Interaction

The main interaction Carne does is with Alma and with external processed that read and write KQML messages to the standard input and output of Carne.

6.1.1 Alma-Carne interaction

A `call(X, Y)` when processed in Alma causes the prolog call X to be made in Carne. If the program succeeds, Carne causes `done` to be added to the Alma database. If it fails, `error` is asserted. This happens automatically.

Carne can add arbitrary formulas to Alma using `af` and delete Alma formulas using `df`. These are prolog predicates that are available to be used in specifying Carne programs.

`af(X)` causes X to be added to the Alma database in the same way as one would do at the keyboard.

`df(X)` deletes from Alma all formulas that *unify* with X. Note the difference between this behavior and the behavior of `df` in Alma. The reason for this difference is some people wanting it this way. Don't ask me why.

6.1.2 stdin/stdout interaction

Carne interacts with external processes and with the user at standard input and output. A KQML parser converts the input to a form suitable for further processing in prolog. This causes a formula to be added to the Alma database. Alma can then request further processing of the incoming message based on user-defined message interpretation code.

The KQML parser can also be bypassed for direct commands typed by the user at stdin.

6.1.3 Log files

Carne can be made to write a log file that provides details of the inputs and outputs to Carne and the results of the commands it runs. This can be useful for debugging.

6.2 Other predicates

`cdebug(X)` changes the log file of Carne to be in directory X. This can be used either in Carne programs or at the request of Alma.

Chapter 7

Running Carne

Carne is typically run while connected to Alma. There are three main ways it can be run:

- A Carne executable reading and writing KQML at the standard input and output. This is the usual embedded mode.
- A Carne executable reading commands interactively at the keyboard. This can be used to debug the Alma-Carne system independent of the larger system in which it is embedded.

  If `inkqml false` is not part of the command line arguments, Carne will still expect KQML messages at standard input. Otherwise, to enter commands first do `(short)` then the predicates `af(X), df(X), cdebug(X)` described above are available, as well as any other prolog programs.
- As source code in a prolog process. This allows the Carne code as well as then user specified prolog code to be debugged.

  To do so, load `carne.pl` and initialize it with `qinitialize(L)` where L is a list of the command line arguments one would have run Carne with. The rest proceeds as though Carne were run interactively.

7.1 Command line arguments

Need to verify whether this is correct.

- `sfile X` X is the file into which to write connection information for the alma Default: none
- `load X` X is the file that contains the actions. Default: none
- `init X` X is the name of the file that contains text that is to be spewed onto stdout on startup. Default: none
- `delay X` X is a real number Default: 0.1
- `debug Fname` Fname is the name of the debug dump file
- `inkqml X` X is true or false. True means to parse all input as KQML. Default: true

Chapter 8

Carne implementation

Carne consists of a loop that listens to messages from Alma or from standard input and processes these. Alma messages result in some program being called. Messages on standard input are usually taken to be KQML and are parsed and Alma is informed of the receipt of the message. **Verify this**.

Chapter 9

Applications

In this section we describe some applications of Alma.

9.1 Dialog management

Something to come here.

9.2 Nonmonotonic reasoning

This illustrates an attempt to use Alma to implement a default logic with preferences. The idea is to represent defaults as implications and to apply them whenever they can be applied. The presence of a contradiction signals that a default had been applied when it should not have been. The meta-logical predicates are then used to resolve the problem. This solution makes use of the fact that Alma does not do any inference unless there is a change in the database that would affect the use of some formula.

See the Examples section for an actual execution of this logic.

9.2.1 Representation

We need to represent defaults in the logic and express preferences among them. The defaults are represented as fifs and therefore as inference rules. The reason for using fifs is that with fifs, contraposition is not possible as would be the case with if and bif. We do not want to contrapose defaults in general.

The defaults are named when they are asserted in the system and these names are used to assert preferences between defaults.

For example, we represent that birds fly and that penguins don't as:

```
named(fif(bird(X), conclusion(fly(X))), birdsfly).
named(fif(penguin(X), conclusion(not(fly(X)))), penguinsdontfly).
if(penguin(X), bird(X)).
prefer(penguinsdontfly, birdsfly).
```

9.2.2 Default application

Since defaults are represented as fif inference rules, application of a default is done in the usual way that fifs are applied.

Continuing with the example, if we also have in the database (or add later) that `penguin(joe)`, Alma first concludes that Joe does not fly using `penguinsdontfly`, then derives that Joe is a bird, then concludes that Joe flies using `birdsfly` which causes a direct contradiction.

9.2.3 Resolving the contradiction

As seen above, this results in the contradiction being noted and the formulas resulting from the contradiction being distrusted. A simple resolution of the contradiction using preferences is to inspect the derivations of the contradictands for defaults that may have a preference relation between them. In this case, `fly(joe)` depends on `birdsfly` and `not(fly(joe))` depends on `penguinsdontfly`. Since the second is preferred to the first, we prefer that `not(fly(joe))`, and this can be reinstated in the database.

Finding the derivations is easily done using `name_to_derivation` when we detect a contradiction, by for example `if(and(contra(X, Y, T), eval_bound(name_to_derivation(X, XD), [X])), derivation_of(X, XD))`. Similarly for the derivation of the other contradictand. Once we have the derivations, the easiest way to compare the two is to write a prolog predicate that will do the comparison. This can be invoked using `eval_bound`. Depending on the comparison, we can then reinstate the appropriate formula, using `reinstate`.

Because Alma only applies an inference rule when there is new information in the database that unifies with the antecedent of the rule, `birdsfly` will not be applied to Joe anymore unless we somewhow rederive that Joe is a bird **(is it applied even then? Verify)**. This feature of Alma makes it unnecessary for the application of defaults to verify whether the default has previously been applied and found not to be preferred.

9.2.4 Behavior

The behavior of Alma in this case will be to jump to the first conclusion that can be made and to change its mind later if better information (more preferred defaults, for example) becomes available. This process need not stop at going from `fly(joe)` to `not(fly(joe))` but can involve several iterations of this change, each time the answer being based on better information.

Also the time at which the change occurs is not known. We might have a contradiction immediately, or after a more or less long time depending on how easy it is to derive the new information, or on when new information becomes available.

9.2.5 More work

What happens when there are several preferences at several levels in the derivation is not clear. An approach that looks like it will work is to consider only the "leaf" defaults of the derivations for deciding which contradictand to keep. If the non-leaf defaults are a problem, then they will get into some contradiction without the leaf defaults being used and at that time we will resolve that problem. It may be that by doing this we reverse the decision based on the leaf defaults only. In this case, the initial resolution of the contradiction was not good and the later one is better.

Chapter 10

Bugs

Unfortunately, Alma still has some bugs. Here are those that are known and need to be fixed.

10.1 Number of clauses

The counter for the number of clauses in the database does not look like it works right. See above.

10.2 Resolution

There is a problem with the way resolution is done. It manifests itself when we need to do resolution with formulas that contain or result in tautologies.

10.3 Query

Currently query returns the matches for bs too. Need to filter these out. On the other hand, query(q(X)) does not seem to return all of them. But it is available to inference.

Chapter 11

Future work

Alma is still work in progress and there are several changes that would enhance its usefulness. Some of these are presented here.

11.1 Misc

Need to be able to kill a bs tree individually. I think all that will take is making available one of the internal things.

11.2 Derivations

Need to give a name to derivations that come from the keyboard or the input file etc.

Perhaps also to derivations from eval_bounds somehow.

11.3 Derivations through Carne

Alma does keep track of derivations using 'fif's. This means that we can track the production rule applications which can be useful for metareasoning in the dialog machine.

That is not sufficient though, because the dialog machine makes several Carne calls during reasoning, and Carne adds formulas to Alma whenever it pleases. Alma cannot know how these formulas were derived. This will make it difficult to know why the machine did some action because the derivation will stop dead at the Carne assertion.

11.4 Context

The purpose of contexts is to isolate pieces of computation in the logic. All formulas will be asserted into some context and the inference will be transparently done between formulas in the same or related contexts. You can do inference across different contexts if the contexts are known by name. Also, formulas can cause assertions to be made in any named contexts. If the context does not exist, it is created. This is how new contexts are created.

There is an initial context called 'initial', and all other contexts are derived from that one. These are the child contexts. A child context can use all the formulas in all its ancestor contexts for inference. So, formulas in the initial context are visible in all other contexts. The new formulas produced as a result of inference are asserted in the maximum child context. A parent context cannot use any formula in its child context unless it refers to the contexts by name.

When contradictions are detected, the fact that there is a contradiction will be asserted in a child context of the current one.

Changes to the language:

EVAL_BOUNDable predicates:

form_to_context(P, C): is true if P is in context C, where P is a formula. This is to be used for inference across unrelated contexts.

assert_in(P, C): causes P to be asserted in C. if C does not exist, it is created and is a child of the context in which the formula containing the assert_in is.

this(N): N gets bound to the name of the formula that this(N) is in.

name_to_context(N, C): C is bound to the context in which the formula named N is.

parent(C, D): D is the parent context of C

INTERFACE command:

afa(F, A).

afa means add formula with attributes. A is a list of attributes, for instance [name(Name), context(Context)] and more later if needed. This should supersede "named" for naming formulas.

Other changes:

The interface to Carne will have to be changed so Carne can assert formulas in the right place.

11.5 Focus

This describes a primitive focus/thread mechanism that can provide a framework for us to control which inferences are made by the inference engine. The focus is intended to be set and unset interactively or using eval_bound formulas and once focus is asserted for a formula, this is inherited by its descendants.

This is intended to be a framework in which we can experiment with different policies for focus and priority, and which other facilities ought to be provided in the language.

New operators/predicates.

The new operators/predicates in the language will be:

1. set_focus(F, T) F is a formula, T is true or false.

Asserting that with F instantiated and T true will set focus on formula F.

If T is false, that removes focus from F and its descendants.

If T is not instantiated and F is, if F is in focus, T will be bound to true, else false.

If F is unbound, set_focus will succeed with formulas that are in focus.

2. get_agenda(C, T) C is a list of tasks on the agenda and T is the step number.

This is used to get the agenda at some time. This can be useful when the system is used interactively.

3. set_execute_length(X, T) X is an integer and T is a step.

X is the number of tasks in the agenda that will be executed at each step.

4. set_priority(F, X) F is a formula and X is an integer.

The priority of the thread originating from F is set to X

How this is intended to work.

The user sets the number of tasks to do and the focus. Then, at each step, only that number of tasks will be done, and tasks that involve formulas in focus will be done in priority to other tasks.

A priority can be set for formulas in focus. This is used to decide how many tasks associated with each focussed formula are to be done at each step. This can be used to simulate thread priorities.

The focus and priority could be set by formulas in the language itself, or we could have some sort of procedure to decide how priority should change with time and what things come into and out of focus.

How this is to be implemented.

Each formula in the database will be associated with its focus and its priority. The focus will be a list of formulas from which it descends and that are in focus. The list will be ordered in order of the priority of the formulas in focus. As focus is removed, that list will vary. The priority of the formula will be the maximum priority in the list.

When the list of tasks is obtained, the list will be partitioned into separate lists associated with each focus. If a task is associated with more than one focus formulas, it is assigned to the list with highest priority.

The number of tasks to do for each focus set is computed from the total number of tasks to be done in the step and the priorities of the formulas in focus. The appropriate number of tasks for each focus is removed and added to a list of tasks that will be done. If there is room left, other non-focussed tasks will be added. There will have to be a way to choose which tasks to execute when there are more than can be executed within a given focus. Perhaps recency can be used.

As new formulas are derived, their focus and priority will have to be computed.

As formulas are brought in or out of focus and priorities are changed, all the formulas in the database that are related have to be updated.

If there are always more new tasks to do than are actually done, the agenda will get bigger and bigger. There will have to be a way to throw things away. This could be linked to whatever mechanism is used to keep the size of the database reasonable (which could also be linked to focus and recency).

There are bound to be the usual synchronization problems.

Costs.

There is going to be extra work done in updating the formulas and manipulating the agenda. Computations we currently do are likely to slow down, but as the number of possible inferences gets large, this should be of benefit if we have reasonable policies for focus and priority setting.

11.6 GUI

We need a graphical user interface that would enable the user to see the evolution of the Alma database. This would make debugging and understanding how the Alma theories work much easier. Other desirable features would be:

- Being able to see a derivation tree for any formula in the database.
- Being able to rollback the database to a previous point and restart it.

A first attempt at that has been made by writing a GUI in java and connecting Alma to the GUI through sockets. This does not work very well though.

11.7 History module

Do something about history. Either in the Alma process itself or as an external module. Decide what the queries and responses to the history should be.

11.8 TO WRITE ABOUT

- History file
- Debug file

- cdebug
- history in general

Bibliography

[1] C. Andersen, D. Traum, K. Purang, D. Purushothaman, and D. Perlis. Mixed initiative dialogue and intelligence via active logic,. In *Proceedings of the AAAI'99 Workshop on Mixed-Initiative Intelligence*, 1999.

[2] J. Drapkin and D. Perlis. A preliminary excursion into step-logics. In *Proceedings SIGART International Symposium on Methodologies for Intelligent Systems*, pages 262–269, Knoxville, Tennessee, 1986. ACM.

[3] J. Elgot-Drapkin. A real-time solution to the wise-men problem. In *Proceedings of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, Stanford, CA, 1991.

[4] J. Elgot-Drapkin and D. Perlis. Reasoning situated in time I: Basic concepts. *Journal of Experimental and Theoretical Artificial Intelligence*, 2(1):75–98, 1990.

[5] J. Gurney, D. Perlis, and K. Purang. Interpreting presuppositions using active logic: From contexts to utterances. *Computational Intelligence*, 1997.

[6] M. Nirkhe, S. Kraus, and D. Perlis. How to plan to meet a deadline between now and then. *Journal of Logic and Computation*, 1997.

[7] D. Perlis, J. Gurney, and K. Purang. Active logic applied to cancellation of gricean implicature. AAAI 96 Spring Symposium on Computational Implicature, 1996.

[8] D. Perlis, K. Purang, D. Purushothaman, C. Andersen, and D. Traum. Modeling time and meta-reasoning in dialogue via active logic. In *Working notes of AAAI Fall Symposium on Psychological Models of Communication*, 1999.

[9] K. Purang, D. Purushothaman, D. Traum, C. Andersen, D. Traum, and D. Perlis. Practical reasoning and plan execution with active logic. In *Proceedings of the IJCAI'99 Workshop on Practical Reasoning and Rationality*, 1999.

Appendix A

Keyword reference

In this section we give a bit more detail about the keywords and commands introduced earlier and give examples of their use in Alma formulas and the expected behavior. Extended examples of the use of some of these commands and keywords are presented a later section.

A.1 bs

Synopsis `bs(+X)`

Arguments `X` is a literal that needs to be proved.

Description `bs(X)` starts a proof by contradiction for X as described above. If X is provable using literals and the if and bif implications, it will be asserted in the database when it is proved.

Uses This can be part of a formula or it can be used at the keyboard to start a proof.

Examples `if(p(X), bs(q(X))).`

See also `delete_trees`

A.2 alma

Synopsis `alma(+X)`

Arguments `X` is a callable prolog term, or an Alma command that can be used at the keyboard.

Description `alma(X)` calls X as a prolog program at the end of the step at which it is processed. This is to be contrasted to `eval_bound` which evaluates its argument immediately.

X has to be defined either in prolog, or in Alma or it has to be loaded into Alma using the `load` command line argument.

Uses This can be used in the consequence of implications or at the keyboard.

Examples This is a convenient way to add or to delete formulas through Alma formulas:

```
if(p(X), alma(af(if(q(X, Y), s(X))))).
if(s(X), alma(df(p(X)))).
```

It can also be used to execute tasks that should not be done within a step, like changing log-file directories.

```
if(cd(X), alma(cdebug(X))).
```

See also `load, eval_bound`

A.3 call

Synopsis `call(+X, +A, +Y)`

Arguments
- `X` is a callable prolog predicate.
- `A` is a list of prolog clauses.
- `Y` is a prolog constant. (or term?)

Description `call(X, A, Y)` is used to get Carne to call X in the context of formulas A. Y is used to identify the instance of this call when the status (doing, done, failed) of the call is asserted in the database.

When the request is sent to Carne `doing(X, Y)` is asserted in the database. If the request succeeds, this is replaced by `done(X, Y)` and if it fails, `failed(X, Y)`.

Predicates are available in Carne for the results of the operation to be added to the database.

Uses This can be used in the consequence of implications, or as single literal clauses.

Examples if(new_utterance(X, T), call(parse(X, [], T))).

See also `doing, done, failed`

A.4 cdebug

Synopsis `cdebug(+X)`

Arguments `X` is the name of a directory.

Description `cdebug(X)` causes the log files to be closed and reopened in directory X. The files are named "Alma" for the history file and "ADebug" for the history file.

Uses This is to be used as an argument to `alma`. It can be part of a formula or can be called from the interface. `cdebug` should not be used as an argument to `eval_bound` because this is something that cannot be done in the midst of a step. It has to be done at the end of one step.

Examples This is useful if Alma is used as part of a larger system (like Trains) in which all the debug files are meant to go in one directory.

if(change_directory(X), alma(cdebug(X))).

See also `history, debug`

Synopsis

Arguments

Description

Uses

Examples

See also

Appendix B

Running ALma in prolog

```
Quintus Prolog Release 3.2 (Sun 4, SunOS 5.4)
Copyright (C) 1994, Quintus Corporation.  All rights reserved.
301 East Evelyn Ave, Mountain View, California U.S.A. (415) 254-2800
Licensed to University of Maryland

| ?- [toplevel].
% loading file /fs/disco/kpurang/work/Alma/toplevel.qof
%  loading file /usr/imports/quintus3.2/generic/qplib3.2/library/ask.qof
%

%% many lines deleted

% toplevel.qof loaded in module user, 2.490 sec 297,156 bytes

yes
| ?- initialize.

yes
| ?- handle_args([keyboard, true, run, false]).

yes
| ?- af(if(p(X), q(X))).

X = _14284

| ?- af(p(a)).

yes
| ?- sr.

yes
| ?- sr.

yes
| ?- sdb.
3: q(a)
1: p(a)
0: p(_14392) ---> q(_14392)
```

```
4: now(2)

---Back Searches---


yes
| ?-
```

Appendix C

Demo files

When you run it, if a line is prefixed by ”?’, it is a query, if there is no prefix, it is something that you could type in at the keyboard in a regular alma run. You can hit return or type ’y’ or ’Y’ and so on to answer the questions. Anything else exits.

Appendix D

Script files

Here I have a number of examples to illustrate the running Alma. See the accompanying example files. Included here is an example run with file t1.pl. Comments are prefixed by a

D.1 Example: Forward inference

The file "t1.pl" is:

```
forall(X, if(p(X), q(X))).
forall(X, if(and(q(X), r(X)), s(X))).
p(a).
p(b).
forall(X, r(X)).
if(s(a), alma(format('~n~nGot s(a)~n~n', []))).


> alma run false keyboard true alfile '../alfiles/t1.pl'

% Invocation of Alma. Note that the command line arguments are not
% prefixed with a ''-'' and that single quotes are used to enclose the
% file name because it contains ''.'' and ''/''.

alma run false keyboard true alfile '../alfiles/t1.pl'
Compiled on Saturday December 25 1999 15:14

For help, type alma_help.

alma: sr.

% We allow Alma to step once.
% The strings after the prompt ''alma:'' are what is typed in.
% Note that the commands are terminated by a period.

sr.
alma: sdb.
sdb.
0: p(_7121) ---> q(_7121)
1: r(_7083) & q(_7083) ---> s(_7083)
```

```
2: p(a)
3: p(b)
4: r(_7005)
5: s(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
6: now(1)

---Back Searches---

% These are the contents of the database. The ‘‘_7121’’s are variables.
% The integers at the beginning of each line are the names of the
% formula on that line.

alma: sr.
sr.
alma: sr.
sr.
alma: sr.
sr.
alma: sr.
sr.


Got s(a)


% This is the result of ‘‘alma(format(~n~nGot s(a)~n~n,[]))’’

alma: sr.
sr.


Got s(a)

alma: sr.
sr.
alma: sdb.
sdb.
7: p(_7574) & r(_7574) ---> s(_7574)
1: r(_7534) & q(_7534) ---> s(_7534)
10: q(_7504) ---> s(_7504)
9: q(b)
11: q(a) & r(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
13: r(a) ---> s(a)
14: r(b) ---> s(b)
22: s(b)
3: p(b)
15: p(_7289) ---> s(_7289)
16: p(a) & r(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
0: p(_7214) ---> q(_7214)
20: q(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
8: q(a)
21: s(a)
5: s(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
25: r(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
```

```
4: r(_7031)
32: p(a) ---> alma(format(~n~nGot s(a)~n~n,[]))
2: p(a)
43: now(7)

---Back Searches---

% This is the final state of the knowledge base.
% Notice how the consequences of the database results in multiple
% print-outs of ‘‘Got s(a)’’.

alma: halt.
halt.
>
```

D.2 Example: backward inference

We use the following file, called bstest1.pl:

```
bif(p(X), q(X)). % 1
bif(r(X), q(X)). % 2
if(s(X), r(X)). % 3
p(a). % 4

bs(q(X)). % 5
```

We are interested in finding which objects are qs, so we have a backward search for it in line 5. Notice that we cannot forward chain to any q in this database since the only formulas that result in q are 1 and 2 and both are bifs. With line 4, we can prove $q(a)$ by backward search.

Line 5 starts the backward search for qs immediately. We expect to find $q(a)$ after a few steps. If we set $delete\_trees$ to false (default), we expect that when we later add $p(b)$, $q(b)$ will be derived. Similarly, if we later add $s(c)$, $q(c)$ will be derived. This behavior is illustrated below. Comments are prefixed by "%%".

```
>  ../alma keyboard true run false alfile 'bstest1.pl'
 ../alma keyboard true run false alfile 'bstest1.pl'
Compiled on Friday August 18 2000 16:01

For help, type alma_help.

alma: sr, sdb.
sr, sdb.
0: p(_10300) -b-> q(_10300)
1: r(_10268) -b-> q(_10268)
2: s(_10236) ---> r(_10236)
3: p(a)
4: bs(q(_10184))
5: now(1)

---Back Searches---

%%
```

```
%% No back searches have been started yet.
%%

alma: sr, sdb.
sr, sdb.
0: p(_10541) -b-> q(_10541)
1: r(_10509) -b-> q(_10509)
2: s(_10477) ---> r(_10477)
3: p(a)
4: bs(q(_10425))
7: doing_bs(6,q(_10397))
8: now(2)

---Back Searches---

6: not(q(_10688)) : not(q(_10688))

%%
%% We start a backward search for q(X) and by asserting not(q(X))
%%

alma: sr, sdb.
sr, sdb.
0: p(_10455) -b-> q(_10455)
2: s(_10423) ---> r(_10423)
3: p(a)
4: bs(q(_10371))
7: doing_bs(6,q(_10343))
1: r(_10308) -b-> q(_10308)
11: now(3)

---Back Searches---

9: not(q(_10682)) : not(p(_10674))
10: not(q(_10646)) : not(r(_10638))
6: not(q(_10602)) : not(q(_10602))

%%
%% From the bifs, we see that if not(q(X)) is assumed then not(p(X)) and
%%  not(r(X)) are also true.
%%

alma: sr, sdb.
sr, sdb.
0: p(_10079) -b-> q(_10079)
3: p(a)
4: bs(q(_10027))
7: doing_bs(6,q(_9999))
1: r(_9964) -b-> q(_9964)
12: q(a)
2: s(_9906) ---> r(_9906)
14: now(4)

---Back Searches---
```

```
9: not(q(_10348)) : not(p(_10340))
6: not(q(_10304)) : not(q(_10304))
13: not(q(_10276)) : not(s(_10268))
10: not(q(_10240)) : not(r(_10232))

%%
%% Further, from formula 2, if we assume not(q(X)), not(s(X)) is true.
%% By now we have also concluded q(a) (formula 12) because p(a)
%%   contradicts the supposition that not(p(a)), so q(a) is true.
%%

alma: sr, sdb.
sr, sdb.
0: p(_8855) -b-> q(_8855)
3: p(a)
4: bs(q(_8803))
7: doing_bs(6,q(_8775))
1: r(_8740) -b-> q(_8740)
2: s(_8708) ---> r(_8708)
12: q(a)
16: now(5)

---Back Searches---

9: not(q(_9124)) : not(p(_9116))
13: not(q(_9088)) : not(s(_9080))
10: not(q(_9052)) : not(r(_9044))
6: not(q(_9008)) : not(q(_9008))

alma: af(p(b)).

%%
%% We now add p(b) and expect to see q(b) derived next.
%%

alma: sr, sr, sdb.
sr, sr, sdb.
0: p(_11977) -b-> q(_11977)
3: p(a)
4: bs(q(_11925))
7: doing_bs(6,q(_11897))
1: r(_11862) -b-> q(_11862)
2: s(_11830) ---> r(_11830)
12: q(a)
17: p(b)
19: q(b)
20: now(7)

---Back Searches---

9: not(q(_12258)) : not(p(_12250))
13: not(q(_12222)) : not(s(_12214))
10: not(q(_12186)) : not(r(_12178))
```

```
6: not(q(_12142)) : not(q(_12142))

%%
%% And it is after a step.
%%

alma: af(s(c)), af(s(d)).

%%
%% We add s(c) and s(d) and expect to see th corresponding qs.
%%

alma: sr, sr, sr, sdb.
sr, sr, sr, sdb.
0: p(_22652) -b-> q(_22652)
3: p(a)
4: bs(q(_22600))
7: doing_bs(6,q(_22572))
1: r(_22537) -b-> q(_22537)
2: s(_22505) ---> r(_22505)
12: q(a)
17: p(b)
19: q(b)
22: s(c)
23: s(d)
25: q(c)
27: r(c)
26: q(d)
28: r(d)
34: now(10)

---Back Searches---

9: not(q(_22969)) : not(p(_22961))
13: not(q(_22933)) : not(s(_22925))
6: not(q(_22889)) : not(q(_22889))
10: not(q(_22861)) : not(r(_22853))

%%
%% We do get q(c) and q(d) too.
%%
```

D.3 Example: eval_bound

this should be in the reference section.

D.3.1 Computing factorials

If we want to do some simple fast computations, we can use `eval_bound` to invoke the appropriate prolog predicate to do so. These can be built in predicate or user-defined.

In this example, we compute factorials using a user defined factorial predicate that is invoked in Alma.

D.4 Example 2:

this looks useless

```
> alma2 keyboard true run false
%
% this says that we want alma to listen to the keyboard and that we do
% not want it to run on its own. So each time we want it to step, we have
% to tell it.
%
alma2 keyboard true run false
Compiled on Wednesday March 15 2000 21:18

For help, type alma_help.

alma: af(if(p(X), q(X))).
%
% add $\forall x~ P(x) \rightarrow Q(x)$ in alma/prolog syntax
%
af(if(p(X), q(X))).
alma: sr.
%
% step once
%
sr.
alma: sdb.
%
% display the database.
%
sdb.
p(_6585) ---> q(_6585)
now(1)

---Back Searches---


alma: af(p(a)).
af(p(a)).
alma: af(not(q(s))).
af(not(q(s))).
%
% add two formulas
%
alma: sr.
%
% Step
%
sr.
alma: sdb.
sdb.
p(_6635) ---> q(_6635)
p(a)
not(q(s))
```

```
now(2)

---Back Searches---

%
% the inferences have not been made yet but the formulas are inthe database.
%
alma: sr.
%
% Step once more.
%
sr.
alma: sdb.
sdb.
p(a)
q(a)
not(q(s))
p(_6611) ---> q(_6611)
not(p(s))
now(3)

---Back Searches---

%
% Now we have the inferences we want.
%
alma: halt.
halt.
>
```

Appendix E

Output files

E.1 History

Comment lines added are preceded by “%%” and the comments are preceded and followed by a blank line.

```
step(1)
add(now(1))  2  []

%% The new step number is added at each step. The name is ‘‘2’’ and
%% the derivation is empty.

add(p(a))  1  []
add(p(X1) ---> q(X1))  0  []

%% These are the initial formulas added with names ‘‘1’’ and ‘‘2’’
%% respectively and no derivation.

End of Step----------------------------------

step(2)
add(now(2))  4  []
add(q(a))  3  [[1,0]]

%% Here q(a) has been derived and the derivation is from the formulas
%% ‘‘1’’ and ‘‘2’’ above.

delete(now(1))  2  []

%% The old step number is deleted.

End of Step----------------------------------

step(3)
add(now(3))  5  []
delete(now(2))  4  []
delete(p(a))  1  []

%% This is an explicit deletion from the user.
```

```
End of Step---------------------------------
```

E.2 Debug

The level 3 debug file corresponding to the above is:

```
Args: [keyboard,true,run,false,history,/tmp/hh,debug,3,/tmp/dd]

%% The command line arguments are printed

Adding if(p(_6836),q(_6836))
Asserted 0: [q(_6836),not(p(_6836))]

%% This is the CNF representation of the inpur formula.

Adding p(a)
Asserted 1: [p(a)]

16:37:29:864108

%% This is the time at which the step was started.

Step1
Asserted 0: [q(_7591),not(p(_7591))]
Asserted 1: [p(a)]
Asserted 2: [now(1)]
Agenda
[[fcres,1,0,fc],1]

%% There is just one inference possible which is a forward resolution
%% of formulas 1 and 0

Forward resolve([p(a)],[q(_9687),not(p(_9687))],[],[[[q(a)],[]]])

%% The resolution in the agenda is done and the result is q(a)

16:37:29:957245
Filter time: 10ms
Inference time: 0ms

%% These are the approximate times used for the stages of the computation.

Memory used: 464860

%% The amount of memory used.

Number of clauses: 4

%% The number of clauses in the database at this time.

Length of agenda: 0
```

```
%% The length of the agenda at the end of the step. This need not be 0.

16:37:31:422229
Step2
Deleting 2: [now(1)]
Asserted 3: [q(a)]
Asserted 4: [now(2)]
Agenda

16:37:31:428719
Filter time: 0ms
Inference time: 0ms
Memory used: 464772
Number of clauses: 3
Length of agenda: 0
Deleting 1: [p(a)]

16:37:40:82321
Step3
Deleting 4: [now(2)]
Asserted 5: [now(3)]
Agenda

16:37:40:184360
Filter time: 0ms
Inference time: 0ms
Memory used: 464080
Number of clauses: 0
Length of agenda: 0
```

Note that there are more types of debug nessages printed out. This will be described somewhere else. For exact information on the debug, the best source is the source code.