



The Architecture of the VITAL-KR

Enrico Motta¹ and Arthur Stutt¹.

HCRL Technical Report (VITAL SERIES) TR 80 (V 4)
October 1991

NB See Technical Report 81 for a fuller (and, in some places, revised) account.

This work was conducted as part of the VITAL Project: A methodology - based workbench for KBS
Life cycle support (ESPRIT PROJECT P5365)

Collaborators (VITAL Partners in bold are involved in this task):

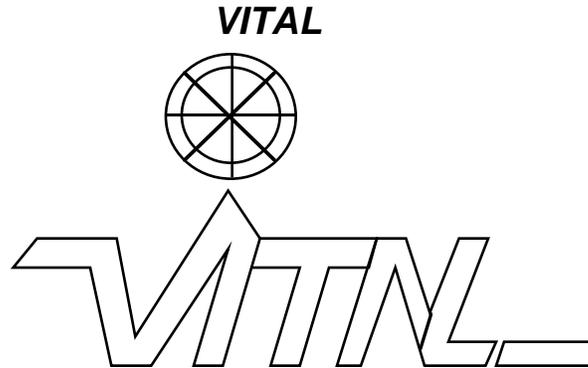
SYSECA - SYSECA TEMPS REEL (Coordinator)	NOTT - UNIVERSITY OF NOTTINGHAM
BULL - BULL CEDIAG	AC - ANDERSEN CONSULTING
ONERA - ONERA	PTT - ROYAL PTT NEDERLAND NV
¹ OU - THE OPEN UNIVERSITY	NOKIA - NOKIA RESEARCH CENTER



**ESPRIT PROJECT P5365
VITAL**

Task 4.1.2, Knowledge Representation Language Specification.

ID412.12 The Architecture of the VITAL-KR



**A methodology - based workbench
for KBS Life cycle support**

Document Reference:	OU/T412.12/W/1	Date: 21/10/91
Activity:	T4.1.2	Status: Working Paper
Distribution:	All T4.1.2 Partners	
Title:	The Architecture of the VITAL-KR	
Abstract:	In this paper the architecture of the VITAL-KR is described.	
Author:	Enrico Motta and Arthur Stutt	

Collaborators (VITAL Partners):

SYSECA - SYSECA TEMPS REEL (Coordinator)	* NOTT - UNIVERSITY OF NOTTINGHAM
* BULL - BULL CEDIAG	AC - ANDERSEN CONSULTING
ONERA - ONERA	PTT - ROYAL PTT NEDERLAND NV
* OU - THE OPEN UNIVERSITY	* NOKIA - NOKIA RESEARCH CENTER

* marked partners are involved in this task

1. INTRODUCTION

As discussed in (Motta, 1991) both the current practice of industrial KBS, and the consensus among researchers (Frisch & Cohn, 1991) suggest that hybrid architectures, embedding a number of specialized representations/reasoners, are required to enable knowledge engineers to build efficient and powerful KBs. This is due to the fact that it has been recognized that no universal knowledge representation language exists, which can efficiently model all types of problem solving handled by AI systems. Each problem may require a different kind of representation, or a different kind of reasoning mechanism. Therefore, it makes sense to think of the next generation of knowledge engineering environments as collections of knowledge representation tools, each of them supporting a particular class of problems. For this reason, the problem of characterizing *hybrid systems*, integrating multiple representations and problem-solving methods, has received particular attention (Frisch & Cohn, 1991), and a number of suggestions have been put forward, as to what integrating method is the most powerful (Myers, 1991).

The VITAL project aims at producing "a workbench for structured KBS development" (Jonker et al., 1991). This will integrate together a number of AI techniques, such as machine learning, knowledge acquisition, model-based reasoning, and knowledge representation, in addition to integrating these AI techniques with software engineering methods. Within this context, the role of the VITAL knowledge representation component (VITAL-KR) is to provide the knowledge engineer with domain-independent representation and reasoning capabilities. Because of the overall emphasis of the VITAL project on integration of existing technology, and because of the need for open, hybrid architectures, comprising a number of inference tools, our work on the VITAL-KR has focussed on producing a practical and open architecture for integrating heterogeneous representation and reasoning systems.

In this paper the architecture of the knowledge representation component of the VITAL workbench (VITAL-KR) is described. The work reported here builds on an earlier document (Motta & Stutt, 1991), which reviewed hybrid architectures for knowledge representation, and formulated a proposal for the architecture of the VITAL-KR. Here, we revise the proposed architecture for the VITAL-KR, and we describe in detail a number of aspects of the VITAL-KR, which were only briefly discussed in the previous paper. These include:

- Functional Interface to the VITAL-KR
- Structure and behaviour of the Inference Scheduler

-
- Role of the GDL and its format
 - Integration Requirements for the Inference Modules (IMs).

Moreover, this document also provides a preliminary description of the IMs to be embedded in the architecture.

Before giving a detailed specification of the various modules, the functional interface to each module, and the communication and control mechanisms in the VITAL-KR, we present a revised description of the coarse-grained architecture described in (Motta and Stutt, 1991).

2 . A REVISED, COARSE-GRAINED DESCRIPTION OF THE VITAL-KR.

The VITAL-KR comprises an *Inference Scheduler* (IS), a number of *Inference Modules* (IMs), a *Global Repository* (GDL), and a *Truth Maintenance System* (TMS) (see figure 1).

Each IM is a separate inference tool, able to represent and reason with its own knowledge. For instance, an IM can be an object-oriented component, a Prolog tool, or a forward-chaining rule interpreter. In fact, any tool can be integrated in this architecture, as long as it meets the requirements listed in section 8.

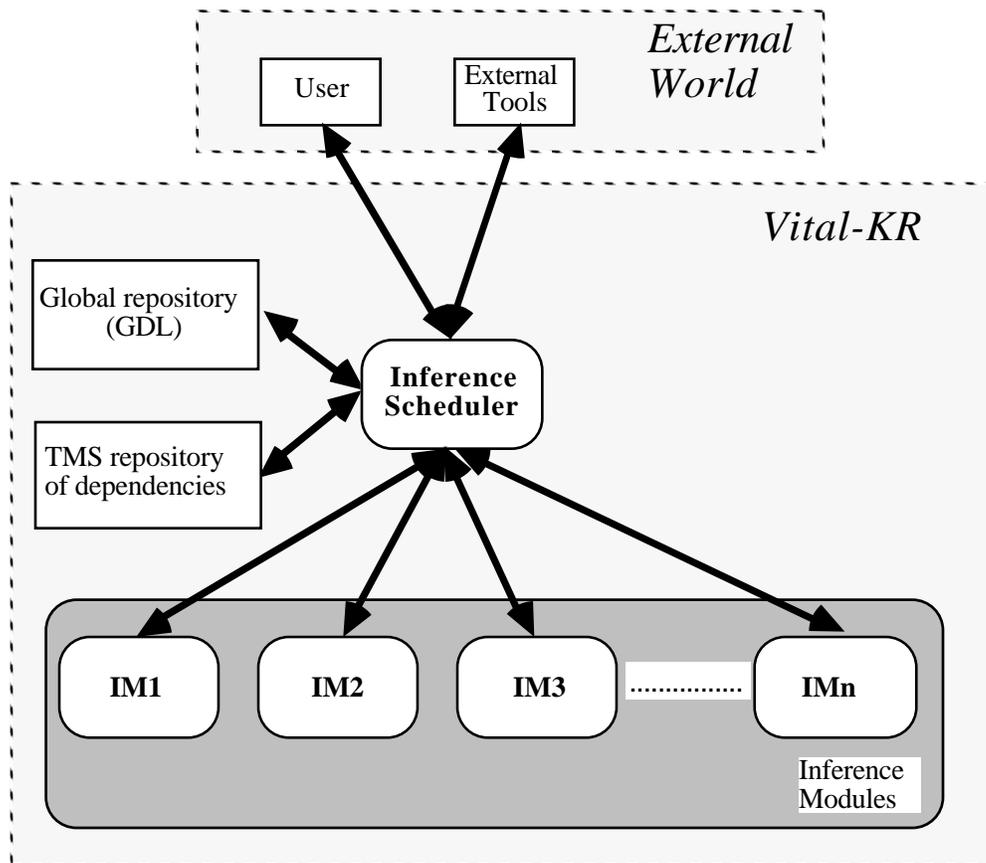


Figure 1. The architecture of the VITAL-KR

The integration of the various IMs in the VITAL-KR is achieved by means of the GDL and the IS. The former fulfills three roles, providing i) a common repository for all global data shared by the IMs, ii) a canonical format for exchanging data between IMs, and iii) a comprehensive, homogeneous description of all *data*, both global and local to a particular IM, which are encoded in the hybrid knowledge base (KB). A KB in the VITAL-KR is the collection of all representations encoded by the IMs, or by the GDL. By 'data' here we mean all ground facts encoded in the KB. The GDL provides a number of advantages both for the internal integration of the VITAL-KR, and for its integration with external tools. Internally, the GDL ensures that integration between all components at data level is achieved, as they all share a common global repository and a canonical format for representing ground facts in terms of *GDL expressions*. Externally, the GDL offers a homogeneous, global view of the data stored in the system. As discussed in (Motta, 1991), this facilitates the integration with external tools such as visualization and debugging tools.

The role of the IS is that of coordinating the working of the various IMs embedded in the VITAL-KR, and interfacing the VITAL-KR with the external world. Thus, all communication within, from, and to the VITAL-KR is centralized. All communication

to the IS is mediated by a *functional interface* which, among other things, supports *generic primitives* for asserting and retracting knowledge, and for posing queries to the system. The role of these generic primitives is similar to the one played by the functional interface to Cyc (Lenat & Guha, 1991), or by *generic functions* in CLOS (Keene, 1989): they specify the functionalities of a system in a way which is independent from its actual implementation. In the case of the VITAL-KR, the generic primitives define an abstract interface not only between the hybrid system and the external world (ie. the user, or other tools), but also between the various VITAL-KR components. The provision of a centralized style of communication together with the generic functional interface makes the architecture more modular. This is because both the communication between the VITAL-KR and the external world, and the communication between two different IMs does not depend on the implementation aspects of the system (i.e. what IMs are included in the hybrid system, and how are they implemented).

Thus, the IS ensures that the various IMs are integrated at the communication and control level. For instance, when queries are posed to the VITAL-KR, these are handled by the IS, which sends them in turn to the relevant IMs, collecting and returning the results. In order to do this efficiently, the IS needs to know what sort of queries each IM can handle, so that only those modules which have a chance of answering a particular query are interrogated. Analogously when new data are asserted, they are first entered into the GDL, and then broadcast to all 'interested' IMs. In turn, each IM can then perform inferences, adding additional new data, which are passed to the IS, and broadcast first to the GDL, and then to all other IMs. The process is repeated until no new inferences are triggered, and no pending data need to be broadcast to the IMs.

Finally, the VITAL-KR also comprises a TMS. As discussed in (Motta & Stutt, 1991), this module performs two roles. First, it makes sure that only facts which have a well-founded support are kept in the knowledge base. Second, when contradictions are encountered, it attempts to restore consistency by adding or retracting *justifications*. These are TMS structures which model data dependencies (see section 7). While we expect some IMs to implement their own belief-revision mechanism, it is useful to embed a general-purpose TMS into the VITAL-KR for two reasons. First, it provides a default facility for performing truth maintenance, to be used by those tools which do not implement their own belief-revision mechanism. Second, it can be used to restore the global consistency of the KB, when contradictions, which are the result of inferences performed by different IMs, are encountered. The details concerning the

integration of the TMS in the VITAL-KR, and the integration of special-purpose and general-purpose TMSs are discussed in section 7.

The architecture described here is truly generic, and, as we said above, any inference system satisfying the requirements listed in section 8 can be integrated with relatively little effort. However, a choice has to be made regarding what inference modules are going to be embedded in the VITAL-KR. Because we intend to provide support for the most established AI paradigms, and because we intend to integrate in the VITAL-KR the results from the KEATS (Motta et al., 1991a) and ACKNOWLEDGE (Akkermans et al, 1991, Vestli & Nordbø, 1991) projects, the VITAL-KR will embed NTP (Elfrink & Reichgelt, 1988), a full 1st order logic theorem prover devised in the context of the ACKNOWLEDGE, as well as the frame-based system and the forward-chaining rule system from the KEATS project. We also intend to integrate a *Task Scheduler* (TS), to provide the user with a facility for representing and reasoning with control knowledge. Finally, in addition to the TMS already mentioned, we also plan to embed a procedural component. A brief description of these systems is given in section 6.

3. THE GDL REPOSITORY

The GDL allows the IMs to share and exchange data, as it provides the VITAL-KR with a global data repository, and a canonical interchange format. While more ambitious scenarios can be envisaged (see Motta, 1991), in which a complete description of all knowledge encoded in the hybrid system is redundantly represented in the GDL, we suggest that in the first instance only ground facts should be explicitly encoded in the GDL. There are various reasons for this choice: i) the overall architecture of the VITAL-KR becomes much simpler and can be implemented using the resources available within the VITAL project, and ii) we expect that the IMs will only need to exchange data. However, a scenario a-la-Cyc (Lenat & Guha, 1990), where all knowledge present in the system is redundantly represented in a logic-based formalism and can be entered into the system in terms of such a formalism, could facilitate the verification and encoding of a hybrid KB. Therefore, we intend to keep open the possibility of extending the specification of the GDL in this direction.

The approach currently taken enormously simplifies the specification of the GDL. Because only ground assertions can be stored, a flat, working memory like format, such as the one specified by the following BNF syntax should be sufficient.

```

<GDL-expression> ::= (pred <arg>*)
<arg> ::= <list> | <lisp atom>
<list> ::= a lisp list
<lisp atom> ::= a lisp atom

```

where * is used to express zero or more occurrences of the symbol.

Because in this scenario the GDL is only meant to be an interchange format, such as the MLT-KRL, no particular reasoning capability is needed. However, if negated assertions can also be added, then the GDL needs to be able to spot inconsistencies, and signal them to the TMS.

Because all data are exchanged in the GDL format, it is up to each IM to make sure that it can input and output data using this canonical representation.

The GDL provides a redundant data storage facility, which contains all ground facts entered into the system. Given this scenario we need to decide whether global data are also stored by individual IMs. There are three possible cases: a) only the GDL stores global data, b) for each assertion an IM is chosen, which 'owns' the assertion and can store it, and c) any IM which is 'interested' in storing the assertion is allowed to store it. The first solution has the advantage of being very efficient from the point of view of memory consumption. However, it is more of an attempted constraint on the behaviour of IMs than a solution. This is because, since data are broadcast to each IM, there is no mechanism that can prevent IMs from caching the data received, or the results of their own inferences. In general, this is done because the way data are stored can facilitate certain types of inferences. For instance, a frame system, where inheritance is the main inference mechanism, will need to cluster its data around the frames to which they are associated. In this way, inheritance of properties can be made very efficient. The same problem applies to solution b, if more than one IM can make use of a particular assertion. Moreover, determining the 'ownership' of a GDL expression can be really tricky. For example, one could determine ownership in terms of predicates. For instance, all assertions with type 'before' could be owned by a module performing temporal reasoning (as with the RHET system (Allen & Miller, 1988)). However, there can be cases where more than one IM can perform inferences on a particular GDL expression, and might need to cache it. Therefore, we have decided not to bother with ownership, and to introduce a liberal policy by which all ground facts are stored in the GDL and, if other IMs want to cache a particular piece of data, they are free to do so. For the sake of efficient retraction of assertions, the GDL should keep a directory of all storage locations for a particular assertion. This could be used by the IS to speed up

the retraction of assertions, as 'Retract' messages could then be sent only to the relevant IMs.

The previous discussion applies to global data, which are shared by all IMs. In the case of data local to a particular IM, these will be transmitted to the GDL (via the IS), with no need for further broadcasting.

In conclusion, we propose that the GDL be a static, redundant repository of global and local ground assertions, stored in a canonical data exchange format. The GDL could be implemented as a hash table, where the key for a particular entry is given by its canonical representation, and its value is a list of pointers to the IMs which store the assertion.

4. FUNCTIONAL INTERFACE

4.1. Roles of the functional interface

The functional interface to the VITAL-KR plays three roles. First, it defines at the black-box level the range of functionalities that the VITAL-KR provides to the user or to external tools. Second, because the VITAL-KR can only afford the functionalities supported by its components, it provides a functional framework for integrating inference modules. That is, an inference module can only be integrated in the VITAL-KR if it supports some subset of the generic primitives specified in the functional interface (FI). Third, because the components of the VITAL-KR also communicate with each other through the FI, it also defines the range of messages that can be exchanged by the various modules embedded in the VITAL-KR.

The details concerning how communication to and within the VITAL-KR is handled by the IS are discussed in section 5. In the next section we specify the operations available in the VITAL-KR.

4.2. The Functional Interface - The Generic Specification

As shown in figure 2, the VITAL-KR supports six classes of generic operations. These provide support for entering new data (TELL), retracting existing data (RETRACT), querying the VITAL-KR (ASK), performing TMS and KB related operations, introducing new IMs, and describing IMs, KBs, or KB units. Moreover, a macro operator (Bundle) is also provided to allow the user (or an IM) to compose a sequence of generic operations.

<i>The VITAL-KR supports</i>
TELL Assert AssertInfer
ASKs Asserted? Prove Prove-all Establish Establish-all
RETRACT Unassert
TMS OPERATIONS NewJustification RemoveJustification Justify Dependencies
KBS OPERATIONS CreateKB DeleteKB LoadKB SaveKB
OTHER OPERATIONS Bundle Describe Register-IM

Figure 2. Generic operations in the VITAL-KR

4.2.1. TELL Operations

Assert

This takes a knowledge base, a GDL expression, and a possibly empty justification, adds the GDL expression to the KB, and sends the justification to the TMS.

Assert: $\sum s \ x \ justification \ x \ KB \ -> \ KB$

AssertInfer

This takes a knowledge base, a GDL expression, and a possibly empty justification, adds the GDL expression to the KB, and sends the justification to the TMS. Moreover, it asks all IMs receiving the assertion to perform all inferences made possible by it.

AssertInfer: $\sum s \ x \ justification \ x \ KB \ -> \ KB$

4.2.2. ASK Operations

The term 'GDL sentence' in this section is used to refer to a GDL expression which is not necessarily ground (ie they may contain variables as arguments).

Asserted?

This takes as input a knowledge base and a GDL sentence. If the sentence is currently part of the GDL repository, then the function returns TRUE and the environment in which the GDL sentence is believed (*bindings*). Otherwise, FALSE is returned.

Asserted?: $\sum x \ KB \ -> \ truth \ value \ x \ bindings$

Prove

This takes as input a knowledge base and a GDL sentence. If the GDL sentence can be proved in the knowledge base, then the function returns TRUE, the environment in which the GDL sentence is believed (*bindings*), and the justification of the proof. Otherwise the function returns FALSE.

Prove: $\sum x \ KB \ -> \ truth \ value \ x \ bindings \ x \ justification$

ProveAll

This takes a knowledge base and a GDL sentence, and returns all possible ways of proving the GDL sentence.

ProveAll: $\sum x \text{ KB} \rightarrow \{\text{bindings } x \text{ justification}\}^*$

Establish

This takes as input a knowledge base and a GDL sentence, and passes them as parameters to Prove. If this succeeds, then the GDL sentence instantiated in the environment returned by Prove is added to the knowledge base, the justification returned by Prove is sent to the TMS, and the result of the call to Prove returned. If Prove fails, then FALSE is returned.

Establish: $\sum x \text{ KB} \rightarrow \text{truth value } x \text{ bindings } x \text{ justification } x \text{ KB}$

EstablishAll

This takes as input a knowledge base and a GDL sentence, and passes them as parameters to a Prove-all. If this succeeds, then, for each pair <bindings, justification> returned by Prove-all, the following routine is executed: i) the GDL sentence is instantiated in the environment specified by the bindings and added to the knowledge base; ii) the justification is sent to the TMS. Finally, the result from Prove-all is returned.

EstablishAll: $\sum x \text{ KB} \rightarrow \{\text{bindings } x \text{ justification}\}^* x \text{ KB}$

4.2.3. RETRACT Operations***Unassert***

This takes a knowledge base and a GDL sentence, finds all GDL expressions matching it, and then removes each of them from the knowledge base, provided that it only has an empty justification. In this case, it also removes the empty justification from the TMS.

Unassert: $\sum x \text{ KB} \rightarrow \text{KB}$

4.2.4. TMS Operations***NewJustification***

This adds a justification to the knowledge base (TMS really).

NewJustification: $\text{justification } x \text{ KB} \rightarrow \text{KB}$

RemoveJustification

This removes a justification from the knowledge base (TMS really).

RemoveJustification: justification x KB -> KB

Justify

This takes a knowledge base and a GDL expression and retrieves the data on which the sentence depends.

Justify: Σ x KB -> Σ s

Dependencies

This takes a knowledge base and a GDL expression and returns all data in the knowledge base which depend on it.

Dependencies: Σ x KB -> Σ s

4.2.5. KB Operations

This operations allow the user to create, delete, load, and save KBs.

CreateKB

This creates a new KB.

CreateKB: -> KB

DeleteKB

This deletes a KB.

DeleteKB: KB->

LoadKB

This loads a new KB into the VITAL system, from a file.

LoadKB: -> KB

SaveKB

This saves a KB onto a file.

SaveKB: KB->

4.2.6. Other Operations**Bundle**

This combines a set of operations together and thus provides a composable interface language.

Describe

This returns either (a) an IM description (b) a KB description or (c) a KB-unit description, depending on its input.

RegisterIM

This registers a new IM with the IS.

RegisterIM: -> IM

4.2.7. Internal Operations

The next set of operations are to be used solely for communication between the various modules embedded in the VITAL-KR. As such they cannot be invoked by a user, or by an external tools. However, because the FI also defines the range of messages that can be exchanged between IMs, they are included in this section.

Answer

This operation is needed for returning results to queries. This is sent either from the IS to IMs, or from IMs to the IS (see section 5.3).

RegisterStorageLoc

This function task as input a GDL expression, an IM name, and a scope (local or global), and informs the GDL that the expression is internally stored by the IM.

This function is needed to allow the GDL to keep track of all data stored in the various IMs.

GetStorageLoc

This takes as input a GDL expression and returns the list of all IMs which internally store the expression.

TellPredDirectory

This is used by IMs, to inform the IS about what types of queries they can deal with, or what types of data they want the IS to broadcast to them.

4.3. The Functional Interface - The Specialized specifications

Each IM needs to support the FI to allow integration into a VITAL system. In practice, however, each module will only support a subset of the FI, while the remaining operations will have null effect. In the following we indicate likely null operations. The discussion is brief and informal, and is only meant to provide the reader with a feeling of what kinds of functionalities the various components of the VITAL-KR are expected to support.

4.3.1. Tell Operations

In addition to the GDL, all IMs are expected to support Tell operations. However, AssertInfer is only going to be fully supported by those IMs which have forward-chaining capabilities.

4.3.2. Retract Operations

All modules are expected to support Retract operations

4.3.3. TMS Operations

The TMS obviously needs to support all TMS operations. However, IMs which perform their own truth maintenance are expected to support both Justify and Dependencies.

4.3.4. KB Operations

These operations have to do with the VITAL-KR as a whole. Therefore, individual IMs are not expected to handle them.

4.3.5. Other Operations

All components in the VITAL-KR need to support Describe. Bundle and Register-IM are only dealt with by the IS.

4.3.6. *Internal Operations*

These are only handled by the IS, with the exception of Answer that needs to be supported by all IMs, and by the TMS.

5. CONTROL AND COMMUNICATION IN THE IS

5.1. Message Structure

The IS provides a mechanism for integrating control and communication both within the VITAL-KR, and between the VITAL-KR and the external world. The functional interface described in section 4 defines the range of messages that can be exchanged between different IMs. Because all communication in our architecture is centralized, generic messages are sent from any IM, from the user, or from external tools to the IS, which then *instantiates* and dispatches them to the relevant IMs. Because generic messages can only be sent to the IS, in the rest of the paper we will not include the receiver of a generic message in the message specification, which will contain the type and arguments of a message, and its sender.

Instantiating a generic message means to produce as many *concrete messages* as there are IMs that can deal with the given generic message. Depending on the type of message, the IS can either dispatch all concrete messages at once to all relevant IMs, or can instead dispatch only one, and wait for its result before deciding whether to dispatch the message to more IMs. For instance, assertions are immediately instantiated and dispatched to all relevant IMs, while 'Prove' messages are dispatched one at the time. The detailed description of how the IS deals with each generic message is given in section 5.4.

The IS associates an *agenda* with each IM, the GDL, and the TMS. This is a control structure used by the IS to manage the flow of messages to and from each of these components. Each agenda stores all messages dispatched to the corresponding component, but not yet processed. Processing of pending messages from an agenda is carried out on a first in, first out policy. In the rest of the paper, we will use the term 'dispatching a message' to mean that the message has been inserted into the relevant agenda. The terms 'send a message', or 'execute a command' will instead be used to mean that a message has been sent to an IM, and control released.

The various details concerning how the IS handles the instantiation, dispatching, and execution of assertions, retractions, and queries are described in the next sections. To simplify the discussion, we'll ignore all aspects of scheduling which have to do with truth maintenance. These will be discussed in section 7.

5.2. Asserting and retracting data in the VITAL-KR

Let's suppose a new ground assertion, say (member_of bill man) is entered by the user. This means that the generic message "Assert (member_of bill man), User" is sent to the IS. The IS first instantiates the message by retrieving the IMs which are interested in receiving it, and then stores the corresponding concrete messages into the relevant agendas. For instance, let's assume that the forward chainer and the frame system are both interested in receiving this generic message. In this case, three concrete messages will be generated, and inserted in the agendas associated with the GDL, the forward chainer, and the frame system. As said before, this is not the same as executing them. This will be either done when the IS has no more processing to do, or when a query is sent to one of these three modules, for instance the frame system. In this case, before the query is executed, its agenda is cleared, and all pending messages executed.

The instantiation of a generic assertion is driven by its predicate which, according to the syntax given in section 3, is the first element of the assertion, in our example: member_of. Thus, given the predicate member_of, the IS retrieves the names of the modules which are interested in receiving this class of data from a directory called *Data Dispatching Directory (DDD)*. Note that the capability of informing the IS about the classes of data an IM wants to have broadcast is one of the requirements that IMs need to fulfill, to be integrated in the VITAL-KR. For instance, a forward-chaining rule interpreter such as OPS-5 will only be interested in receiving data which can match a LHS pattern of a rule belonging to a currently loaded rule base. Therefore, when OPS-5 rules are loaded into the system, the OPS-5 compiler will need to collect the types of all LHS patterns, and send them to the IS (using the message TellPredDirectory).

Retract messages are dealt with in a similar way to Assert ones. When a Retract message is sent to the IS, say "Retract (father bill mary), IM1", the IS first retrieves from the GDL all the storage locations associated with this assertion, and then dispatches the instantiated message to each IM which has internally stored the assertion. These normally include the GDL.

The main advantage of separating dispatching a message from actually sending it is that it avoids the switching of control from one module to another too often, therefore increasing efficiency. One interesting case concerns what to do if a Retract message is added to an agenda, which already contains an assertion matched by the Retract message. For instance, let's suppose we have the following forward chaining rule:

```

(def-rule foo
  if (= n 0) &
    (inc n)
  then
    Retract (= n 0)
    Assert (= n 1))

```

Let's also assume that as well as the forward chainer, also another module, say the theorem prover, is interested in receiving assertions of type '='. Now, the user asserts (inc n), and then (= n 0). Therefore the generic message "Assert (= n 0), User" is sent to the IS, which finds out that both the forward chainer and the theorem prover are interested in receiving the new assertion. Therefore the concrete message "Assert (= n 0)" is inserted in the agendas of these two modules. Because the IS is currently idle, and no other IM is doing any processing, the IS decides to clear the pending assertions in the agendas. Let's suppose the forward-chainer is done first. As a result, all pending assertions in its agenda are executed, rule 'foo' fires, and "Retract (= n 0), forward-chainer", and "Assert (= n 1), forward chainer" are sent to the IS. This then dispatches them to the theorem prover. Figure 3 below shows the contents of the agenda of the theorem prover after the two messages above have been dispatched to it.

Theorem Prover agenda

"Assert (= n 0)"
"Retract (= n 0)"
"Assert (= n 1)"

Figure 3. The theorem prover agenda after rule foo has fired.

When the forward chainer releases control, assuming that no other IM takes control, the backlog of operations pending in the theorem prover agenda will be cleared. Here, the issue arises of whether it is worth to assert (= n 0), given that the very next command retracts it. It makes sense to make retract messages more clever, and use them also to retract preceding assertions still pending in the agenda. However, because there can also be cases where it produces undesirable behaviours, both options should be supported in the implementation. It will be up to the user to choose whether or not a more 'intelligent' retract policy should be enforced.

5.3. Dealing with queries in the VITAL-KR

Queries require a more complicated treatment than assertions, as discussed in this section. Let's suppose we have two IMs, IM1 and IM2, which can perform goal-

driven behaviour, thus handling the generic primitive Prove. We assume that IM1 contains the two following goal-driven rules:

```
(def-rule IM1_a
  (foo x) if
    Prove (boo x))

(def-rule IM1_b
  (bla x) if
    Assert (bar x))
```

while IM2 contains the following two:

```
(def-rule IM2_a
  (boo x) if
    Prove (bla x))

(def-rule IM2_b
  (foo x) if
    (baz x))
```

It is probably already apparent to the reader that these two sets of rules are not intended to have any significant meaning. Their only purpose is to show how goal-driven behaviour is handled in the VITAL-KR.

Let's now assume that at top-level the following three commands are bundled together: "Prove (foo x), User", "Retract (bar x), User", and "Assert (foo x), User". These three generic commands are inserted into the *IS agenda* and the first one, "Prove (foo x), User", is then processed. The *IS agenda* is a structure analogous to the *IM agendas*, except that it stores all pending generic, rather than concrete messages. As in the case of *IM agendas*, the items in the *IS agenda* are cleared using a first in, first out policy. In order to instantiate the generic message, the *IS* first retrieves from the *Query Dispatching Directory (QDD)* the names of the *IMs* which can solve the query, in this case *IM1* and *IM2*. However, the *IS* always tries to solve the query first by interrogating the *GDL*. Therefore, in this case the plan to solve the query "Prove (foo x), User" is as follows (see also section 5.4): "First try the *GDL*. If this fails, then try *IM1*. If *IM1* also fails, then try *IM2*". The result of the query will be the value returned by the first *IM* to solve it, 'false' otherwise.

Assuming that "Prove (foo x), *GDL*" fails, the *IS* tries *IM1*. Before releasing control to *IM1*, the *IS* needs to save its own computational state. Thus, when the answer from *IM1* is received, the *IS* can continue processing the generic message, by either sending the message "Prove (foo x)" to *IM2* (if *IM1* has failed), or by outputting the result returned by *IM1* to the user. To this purpose, that is in order to save its state, the *IS* makes use of a private resource called the *IS stack*. After saving its state, the *IS* sends the message "Prove (foo x)" to *IM1*. Control is then passed to *IM1*, Rule *IM1_a* is invoked, and the message "Prove (boo x), *IM1*" is sent to the *IS*. However, as in the

case of the IS, IM1 also needs to save the state of its computation. Because the details of how each IM deals with the state-saving problem do not concern the VITAL-KR, we should not go into much detail on how this is done. For the sake of explanation, we'll assume that, like the IS, each IM has its own internal stack to save its own state. Figure 4 shows the control information associated with IM1 and IS at this stage of the computation.

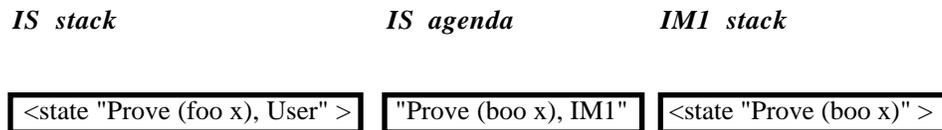


Figure 4. The IS agenda and IM1 stack after the query "Prove (boo x)" has been sent to the IS.

Now, the IS needs to instantiate and execute the generic message "Prove (boo x), IM1". Again, we assume that the GDL cannot solve it, and therefore IM2 is tried. Rule IM2_a is invoked, setting up a new goal "Prove (bla x), IM2", which is then processed, and again assuming that the GDL cannot solve it, is then broadcast to IM1. The situation at this stage is shown in figure 5.

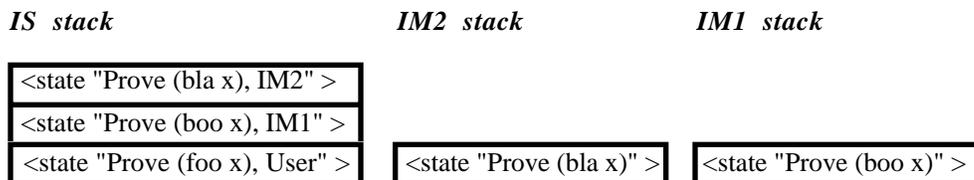


Figure 5. The IS agenda, IM1 stack, and IM2 stack after the query "Prove (bla x)" has been sent to IM1.

Rule IM1_b is then invoked, and the message "Assert (bar x)" sent to the IS, and then broadcast to all interested IMs. Note that this broadcasting doesn't create any problem as there is a separate agenda for each IM. Rule IM1_b succeeds and returns its result to the IS, by using the message Answer, which tells the IS to pop its state from its own stack, and continue the computation. In this case, continuing the computation means to send the result returned by IM1 to IM2. Again, this is done by sending the message Answer to IM2, parametrized by the result returned by IM1. Thus, the original query to IM2, "Prove (boo x)", succeeds, and, using the same protocol, the result is sent to the IS, which then sends it to IM1. Finally, IM1 also succeeds and returns the result of the query "Prove (foo x)" to IS, which then outputs it to the user.

The mechanism described in this section enables both data-driven and goal-driven behaviour to be integrated in a hybrid architecture. In particular, it enables different IMs to cooperate when doing problem solving. The main requirement imposed on a

generic IM, so that it can be integrated in this architecture, is the ability to save its state and to handle multiple, nested queries, as shown in the example.

5.4. Procedural semantics for Functional Interface operations

This section outlines the procedural semantics for the principal FI operations in terms of algorithms which the IS might follow in order to implement them. These serve to clarify the discussion above.

Assert (GDL expression, justification)

Instantiate 1:

 match predicates in GDL expression with DDD

 retrieve list of all interested IMs

Place "Assert GDL expression" on agenda for GDL

Place "Assert GDL expression" on agenda for each interested IM

IF TMS is switched on

THEN send justification to TMS

AssertInfer (GDL expression, justification)

Instantiate 1

Place "AssertInfer GDL expression" on agenda for GDL

IF TMS is switched on

THEN send justification to TMS

Place "AssertInfer GDL expression" on agenda for each interested IM

Clear IM agendas in turn

%% This algorithm ensures that all relevant IMs receive new data and
request for inferences immediately%%

Asserted? (GDL sentence)

Send operation to GDL

IF GDL returns true

THEN succeed

ELSE fail

Prove (GDL sentence)

Instantiate2:

 match predicates in GDL sentence with QDD

 retrieve list of all interested IMs

Order interested IMs

WHILE there are untried IMs

 for each IM_n

 IF IM_n agenda is not empty

 THEN empty IM_n agenda

 send operation to IM_n

 IF IM_n succeeds

 THEN return result to enquirer and stop

 ELSE continue

ENDWHILE

IF no successful IMs

THEN return fail

%% This algorithm ensures that all relevant IMs are tried in turn with all
asserts and retracts made before the proves are sent. The algorithm stops
when an IM is successful %%

ProveAll (GDL sentence)

Instantiate2

Order interested IMs

WHILE there are untried IMs

 for each IM_n

 IF IM_n agenda is not empty

 THEN empty IM_n agenda

 send operation to IM_n

 IF IM_n succeeds

 THEN collect result and continue

ENDWHILE

IF no successful IMs

THEN return fail

ELSE return positive collected results

%% This acts as the above except thta ALL proofs are collected%%

Establish (GDL sentence)

```
Instantiate2
Order interested IMs
WHILE there are untried IMs
  for each  $IM_n$ 
    IF  $IM_n$  agenda is not empty
      THEN empty  $IM_n$  agenda
      send operation to  $IM_n$ 
      IF  $IM_n$  succeeds
        THEN
          ASSERT GDL sentence instantiated in the returned
            environment, and parametrized by the returned justification
          return result to enquirer
          and stop
        ELSE continue
    ENDWHILE
  IF no successful IMs
    THEN return fail

%% This acts as Prove except that the result is asserted. %%
```

EstablishAll (GDL sentence)

```
Instantiate2
Order interested IMs
WHILE there are untried IMs
  for each IMn
    IF IMn agenda is not empty
      THEN empty IMn agenda
      send operation to IMn
      IF IMn succeeds
        THEN
          ASSERT GDL sentence instantiated in the returned
            environment, and parametrized by the returned justification
          collect result
          continue
        ELSE continue
    ENDWHILE
  IF no successful IMs
    THEN return fail
  ELSE return positive collected results

%% This acts as the above except that all relevant IMs are sent the
operation%%
```

Unassert (GDL sentence)

Find all GDL expressions matching the GDL sentence.

For each GDL expression

IF TMS is switched on

THEN

Ask the TMS whether it has a non-empty justification

IF this is not the case

THEN

Send operation to TMS

Retrieve list of storage locations from GDL

FOR each IM in list

Place "Unassert GDL expression" on IM agenda

ELSE

Retrieve list of storage locations from GDL

FOR each IM in list

Place GDL expression on IM agenda

Justify (GDL Expression)

```

Empty all pending agendas
IF TMS is switched on
THEN Send operation to TMS
    and return result
ELSE
    Instantiate1
    Instantiate2
    merge results of instantiations into single list
    WHILE there are untried IMs
        IF IMn agenda is not empty
        THEN empty IMn agenda
            send operation to IMn
            IF IMn succeeds
            THEN
                collect result and continue
            ELSE continue
    ENDWHILE
    return all positive results

%% This algorithm ensures that if the TMS is switched off all relevant
IMs receive the request %%

```

NewJustification (justification)

```

Send operation to TMS

```

RemoveJustification (justification)

```

Send operation to TMS

```

Dependencies (GDL expression)

```
Empty all pending agendas
IF TMS is switched on
THEN Send operation to TMS
    and return result
ELSE
    Instantiate1
    Instantiate2
    merge results of instantiations into single list
    WHILE there are untried IMs
        IF IMn agenda is not empty
        THEN empty IMn agenda
            send operation to IMn
            IF IMn succeeds
            THEN collect result and continue
            ELSE continue
    ENDWHILE
    return all positive results

%% This algorithm ensures that if the TMS is switched off all relevant
IMs receive the request %%
```

Bundle (operations)

```
FOR each Operation in <Bundle>
    do Operation
ENDFOR
```

Answer (result)

Pop IS-stack returning <saved-state>

Determine next operation using the <saved-state> and answer

Continue processing with next operation

%% This algorithm depends on the correct information being stored in the saved state and ensures the IS can continue appropriately%%

5.5. Brief remarks on the implementation of the communication protocol

The mechanism discussed in the previous section requires each IM to save and resume the state of its computation, as well as to exchange messages with the other IMs. From an implementation point of view, there are various options on how the various IMs could actually be integrated and such a behaviour achieved. For instance, if each IM were to reside on a different process, then the message exchanging mechanism and the concept of transfer of control from one to another would actually have to be reflected in the implementation. In such a scenario, when releasing control, an IM would suspend itself, waiting for the result of a query. An alternative scenario, in which each IM is just a program, means that the message passing architecture would be taken metaphorically and would actually be instantiated in a set of procedural calls from a software module to another. In this case, IMs wouldn't really suspend themselves, but they would just call other IMs by means of the predefined set of legal messages (procedure calls). Finally, a note on the state saving mechanism. The use of *closures* in Common Lisp provides a neat mechanism for implementing the state saving mechanism.

6. SPECIFICATION OF THE INFERENCE TOOLS TO BE INTEGRATED IN THE VITAL-KR

In this section we briefly indicate what inference modules we intend to embed in the VITAL-KR. The choice of these modules was mainly dictated by pragmatic considerations, such as availability of software, access to source code, and partners' interests. Given the aims and the approach of the overall VITAL project, and the meagre resources allocated to the knowledge representation task, the alternative approach, devising new languages or integrating external tools, would not be appropriate. Moreover, as discussed in section 1, it was deemed more appropriate for

the work on the knowledge representation task to focus on the integration issues concerning hybrid architectures.

However, although our approach to the choice of the specific IMs for the VITAL-KR has been mainly pragmatic, at a more generic level we were interested in providing inference support for the most established KR paradigms, such as rule-based and frame-based reasoning. Moreover, we also intend to provide support for specifying and reasoning with control knowledge, for truth maintenance, and for integrating arbitrary programs written in a procedural programming language. Therefore, the VITAL-KR will embed the following inference modules:

- A Task Scheduler
- A frame-based language
- An OPS-5 like rule-based language.
- A full 1st order logic theorem prover
- A procedural component.
- A truth maintenance system

In the next sections we briefly describe these modules, mainly by pointing out the relevant references.

6.1. The task scheduler

This module will provide a mechanism for specifying and reasoning with tasks. The issues concerning the specification of this module are described in (Motta et al, 1991b).

6.2. Frame-based representation in the VITAL-KR

This component is going to be based on the Flik language, developed in the context of the KEATS project (Motta et al., 1991a). Flik is an object-oriented formalism that supports multiple inheritance of properties with exceptions. The basic Flik components are *frames* and *slots*. Frames denote entities of the world being modelled and have slots attached which represent their properties. Slots are in turn parametrized by *facets*, which allow the user to express different aspects of a property. For instance, slots typically have facets that describe their value, type, and documentation. The inheritance mechanism in Flik uses both definitional and default properties. These latter can be non-monotonically invalidated when additional information is entered into the system. Flik also provides *blocks*, *methods*, and *daemons*. Blocks provide a mechanism for specifying exceptions to the inheritance of default properties, therefore enabling the user to customize the inheritance behaviour of a particular relation (slot). Thus, they provide a flexible way for solving inheritance conflicts. Methods and daemons enhance the Flik's representational power by specifying two mechanisms for attaching lisp

functions to frames. These functions are then invoked through explicit, program-controlled calls (methods), or as soon as 'important' events, such as creating or deleting a frame, happen (daemons).

One aspect of the Flik language, which is relevant to its integration within the VITAL-KR is that its inheritance-based inferences can be described in terms of non-monotonic justifications. This capability enables it to support the functional interface primitives related to the integration of IMs with the TMS.

6.3. The forward-chainer rule interpreter

We intend to support forward chaining rule inference, according to the OPS-5 model, which is now the most commonly used production systems interpreter. In particular, during the KEATS project we have implemented our own OPS-5 like rule interpreter, called ERI, and we intend to integrate it in the VITAL-KR.

6.4. The theorem prover

Another component that we intend to integrate in the VITAL-KR is NTP (Nottingham Theorem Prover) (Elfrink & Reichgelt, 1988), which is a full 1st order logic theorem proving. This will provide VITAL with logic inference capabilities subsuming those afforded by Prolog interpreters.

6.5. The procedural component

This component will provide the user of the VITAL-KR with support for 'traditional' procedural programming. In this way, it will be possible cleanly to develop KB applications which integrate both traditional as well as inference systems. The issues concerning the integration of a procedural component in the VITAL-KR are discussed in (Toivonen, 1991).

6.6. The truth maintenance system

A TMS ensures that only facts which have a well-founded support are kept in the KB, and restores consistency after a contradiction is generated. Therefore, it provides problem solvers with a utility for representing and maintaining dependencies between data. A number of alternative TMSs have been proposed over the years, including the ones by Doyle (Doyle, 1979), McAllester (McAllerster, 1982), DeKleer (DeKleer, 1986), and Dressler (Dressler, 1988). The particular TMS to be integrated in the VITAL-KR is the one implemented in the KEATS system, which in turn is based on Doyle's TMS. Therefore it supports non-monotonic justifications, although it does not handle reasoning across multiple contexts. When a contradiction is encountered, *contradiction handlers* are in turn invoked, until consistency is restored. The

contradiction handlers are user-definable. The TMS will be discussed further in the next section.

7. ISSUES RELATED TO THE INTEGRATION OF A GENERAL-PURPOSE TMS

While the various IMs provide the VITAL-KR user with different types of inference and problem-solving capabilities, the TMS can instead be regarded as a utility provided for the IMs. These perform inferences, producing dependencies between data, and send them to the TMS. The TMS in turn makes sure that the consequences of these logical dependencies are enforced, by ensuring that only facts which have a well-founded support are believed.

The communication between the TMS and the various IMs is organized as follows. The IMs assert and retract data dependencies (justifications) from the TMS, while the TMS sends back information on whether nodes are *in* or *out*, that is whether particular facts are believed or not. A justification in a Doyle-style TMS has the following format:

$$A \text{ if } B_1, \dots, B_n; \text{ unless } C_1, \dots, C_m$$

with $m, n \geq 0$. A , each B_i , and each C_i are called *TMS nodes*, and represent data in the KB. The syntax for TMS nodes is not surprisingly the same as for GDL expressions. The intuitive reading of the above justification is that A is believed if all B s are believed, and if none of the C s is believed. Each C_i in the justification is called a *default*. In addition to asserting and retracting justifications, it should also be possible for the user or an IM to interrogate the TMS, to find out whether and why a particular assertion is in or out. Therefore, the FI comprises four generic operations which are dealt with by the TMS:

- New-Justification
- Remove-Justification
- Justify
- Dependencies

These are discussed in the next sections.

7.1. Adding justifications

It should be possible for an IM to send justifications either directly by using the message New-justification, or indirectly, by parametrizing Assert messages by means of a justification. The first case is simple: the message is sent to the IS which forwards it to the TMS. The second case is slightly trickier, as it requires that after the new

assertion is dispatched to the relevant IMs, and before releasing control, the IS should inform the TMS about the new justification. In both cases, adding a new justification might cause changes to the belief status of some TMS nodes. If this is the case, the appropriate generic Assert and Retract messages will be sent by the TMS to the IS, to be broadcast to all interested IMs.

Note that if an assertion is made which has no justification, it is treated as a premise (ie supported by an empty justification).

7.2. Retracting justifications

This can only be done by sending the message Remove-Justification. As in the case above, deleting a justification might cause facts to become in from out and vice versa. Empty justifications can also be retracted by sending the Unassert message. In this case, the corresponding empty justification is retracted from the TMS.

7.3. Querying the TMS

The message "Justify" allows the user (or an IM) to find out why a TMS node is currently believed or not. If the state of the node is in, then a list of pairs is returned, where each pair has the format: <INS, OUTS>. The listed pairs provide all reasons which keep the node in. Each reason (each pair) should be interpreted as stating: "the node is in because all the INS are in, and all the OUTS are out". For instance, let's suppose we have the following set of justifications

A if B

B

C if A, B; unless D.

If we now ask the justification for C, the result will be

(<(A), (D)>)

That is, C is in because B is in, and D is out. As shown in the example, Justify only returns the 'ultimate' reasons for believing a fact. These are the leaves of the tree of dependencies whose root is the node in question.

The operation **Dependencies** can be seen as the reverse of the **Justify** operation discussed above.

7.4. A more complicated scenario.

As discussed in (Motta & Stutt, 1991), it is possible that some IMs might decide to take care of handling their own data dependencies, rather than using the general-purpose utility provided by the VITAL-KR. For instance, the frame system is likely to take care itself of inheritance-based dependencies, rather than broadcasting them to the TMS.

The main reason for modules to have special-purpose truth maintenance facilities is efficiency, as a special-purpose module dealing with only a restricted form of data dependency can be much more efficient than the general-purpose TMS. For this reason, systems such as Cyc, which integrate a number of inference mechanisms, do not embed any general purpose TMS but allow each inference mechanism to perform its own truth maintenance.

In the VITAL-KR we want to integrate both general-purpose and special-purpose truth maintenance facilities. In this way, systems which have not got their own truth maintenance facility will be able to use the general-purpose one, while on the other hand, systems which have a way of handling their own data dependencies will still be allowed to do so.

The problem of integrating general-purpose and special-purpose TMSs only concerns dealing with Justify and Dependencies messages. As discussed in the previous sections, only four types of messages can be sent to a TMS, to assert and retract justifications, and to ask the ultimate reasons supporting a fact, and the data supported by a particular GDL expression. The first two messages do not pose any problem. If a module has its own specialised truth maintenance, it will also take care of asserting and retracting its justifications. The situation is slightly more complicated when queries need to be handled. For instance, let's consider again the example discussed in (Motta & Stutt, 1991).

We have the following two frames, which specify that a car has normally four wheels, and that Alfa Romeo cars are produced in Italy. To simplify the problem we assume that these frames contain definitional, rather than default information.

```

Frame Car
  number-of-wheels: 4

Frame Alfa-Romeo-Car
  subclass-of:      Car
  produced-in:     Italy

```

Let's suppose we now create an instance of an Alfa-Romeo car.

```

Frame My-car
  instance-of:      Alfa-Romeo-Car

```

Once this is asserted, the frame system deduces three new theorems which are broadcast to the other IMs, using the GDL format:

```

Car (My-car)
number-of-wheels(My-car, 4)
produced-in(My-car, Italy)

```

Let's now suppose that our hybrid KB also contains the following forward chaining rule:

```
(def-rule test
  if
    Car (x) and produced-in (?x, Italy)
  then
    complies_with_EC_regulations(x))
```

Now we can derive the theorem

`complies_with_EC_regulations(My-car).`

The justification for this theorem is given by the set of dependencies shown in figure 6.

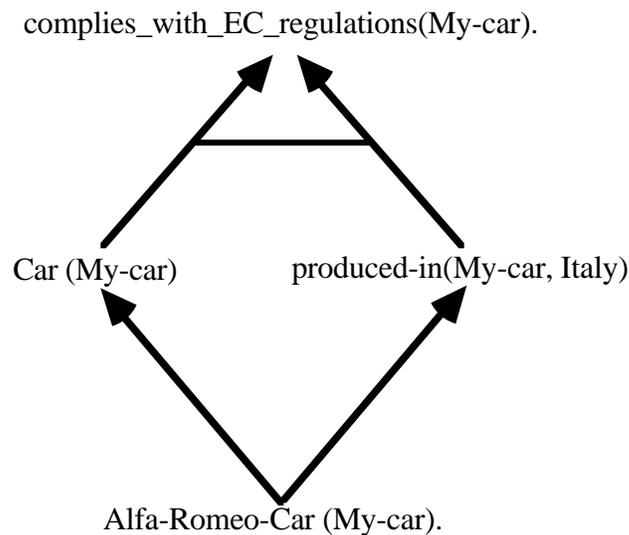


Figure 6. Example of a hybrid proof tree.

The important aspect to note here is that while the two dependencies in the bottom half of the figure are frame-based, and stored internally by the frame system, the one in the top half is rule-based, and handled by the general-purpose TMS. Therefore, if we ask the TMS to justify `complies_with_EC_regulations(My-car)`, it needs to retrieve the relevant information from the frame system. This means that the TMS needs to know that the frame system is the one responsible for asserting `Car (My-car)` and `produced-in(My-car, Italy)`. On efficiency grounds we can discard the solution in which the justifications for these GDL expressions are stored in both the frame system and the TMS. Instead, we require that the name of the IM should be sent to the TMS as justification for the GDL expressions. The idea is that in this way the TMS does not handle these data dependencies itself, while it can still find out about them, by querying the module in question. This information is needed in a number of cases, such as when a contradiction is encountered, or a justification is removed. Note that in order for the

integration between general-purpose and special-purpose TMSs to be possible, all IMs performing their own truth maintenance will have to support the generic messages Justify, and Dependencies.

8. REQUIREMENTS IMPOSED ON IMS BY THE ARCHITECTURE OF THE VITAL-KR

In this section we summarize the requirements which will be imposed on the designer of any IM which is to be fully integrated into the VITAL knowledge representation architecture. Most of these are obvious. Some however result from the particular mechanisms we have defined for the IS. In what follows we will present the requirement and in addition some rationale for its inclusion.

1. *An IM must support the Functional Interface.* This requirement is imposed for two reasons. First, because it is worth to integrate an IM only if it provides a subset of the generic capabilities specified by the FI. Second, because the FI also specifies the range of messages that can be sent to and from an IM. Therefore, supporting the FI is essential to guarantee that an IM is integrated in the hybrid architecture. There are also two further related requirements, which have to do with the TMS:

The IM must be able to supply justifications. This is because basic operations in the FI, such as Assert and Prove, take justifications as input or return them as output.

IMs which perform their own Truth Maintenance must support the TM primitives defined in the FI.

2. *The IM must be able to provide a directory of the predicates included in its part of the KB.* These are needed so that the IS can make sensible decisions about which IMs to send operations to. The operation **TellPredDirectory** has been defined for this purpose.

3. *Multiple instantiations of IMs are needed* in order to deal with possible sub-queries to the same IM. The mechanism for state storage is left to the implementer of the IM. This will probably be similar to that given above for the IS in that each IM will need a stack for handling queries.

4. *The IM must be able to keep track of all changes and pass these to the IS.* These will usually occur as a result of **AssertInfers**. In order to ensure that the IS broadcasts the changes in the most logical fashion the IM must transmit the changes ordered so that sentences which depend on other sentences are sent later.

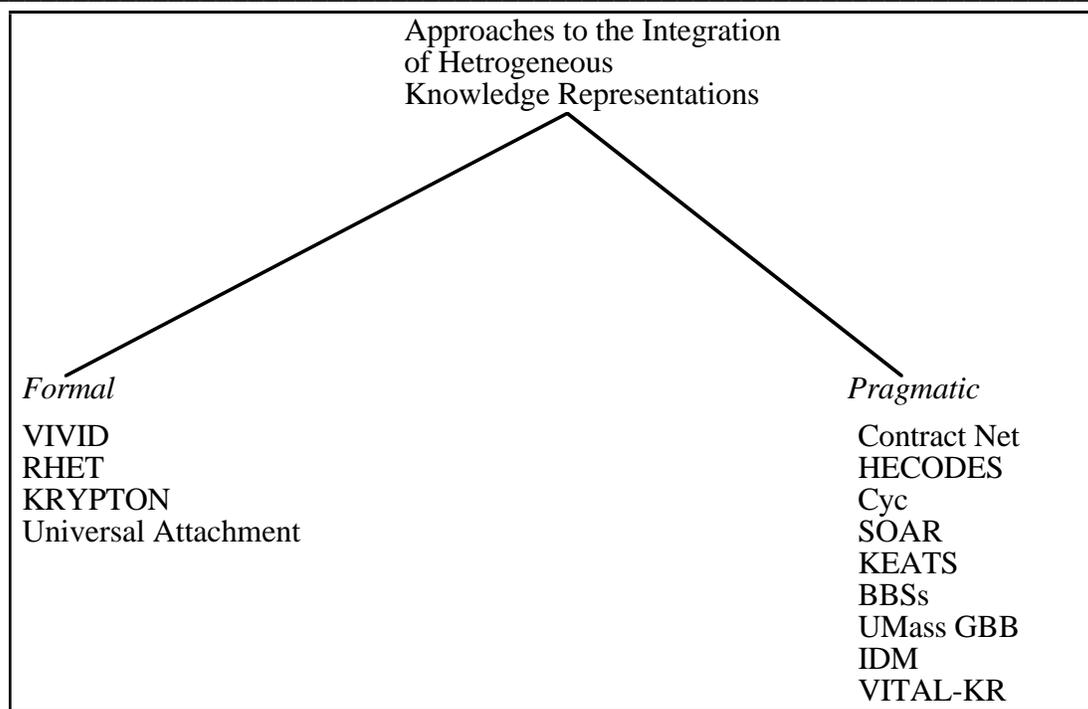
5. *The IM must be able to communicate in the GDL format.* That is to say that it must either use the GDL format internally or have built in translators.

9. CONCLUSION - COMPARISON WITH OTHER HYBRID SYSTEMS

Concluding, we will briefly try to situate the above architecture in relation to other hybrid systems. In Motta and Stutt (1991) we discussed the various approaches to integrating diverse knowledge representations and we went on to outline an architecture for the VITAL-KR. We did not, however, discuss the differences between our architecture and that of other approaches and the advantages which accrue from this. We will remedy this in the present section.

For the sake of clarity we will present the main differentiating features of VITAL-KR rather than discuss all of the ways in which the system differs. The interested reader is referred back to Tables 1 and 2 of Motta and Stutt (1991) for full details.

In general the VITAL-KR represents a non-formal solution to the problem of integrating diverse representations which at the same time avoids the ad hoc feel of some systems. We suggest that there are two main categories of systems: (a) The formally integrated systems such as KRYPTON, VIVID, RHET (plus Myers' technique of Universal Attachment); (b) The pragmatically oriented solutions which enforce integration either by incorporating research from the formal approaches or by having a tight control regime or both. In the latter we include Cyc, HECODES and VITAL among others.



There is much in common between our approach and that of BBSs. We have for instance included a common data area which can be used (indirectly) for communication between IMs. However, this represents an overall global picture of the state of the system's knowledge (as a VIVID KB) while the blackboard only carries partial results and hypotheses. We also retain many of the scheduling ideas found in BBSs (such as the explicit representation of control knowledge). However, we go beyond BBSs, as we integrate both data-driven and query-driven behaviour.

With regard to other pragmatic approaches we have most in common with Cyc and HECODES. SOAR is predominantly a forward chaining rule system with a chunking mechanism and therefore is less general. The Contract Net system is principally intended to overcome problems with distributed processing. In common with Cyc we have a functional interface and a scheduling mechanism. However, the VITAL-KR also provides an explicit framework for integrating additional inference mechanisms, both data-driven and query-driven. In common with HECODES we have a communication protocol, a central shared data area and a means of incorporating multiple KBs. Unlike HECODES we have a VIVID KB, control integrated as an IM and an extended functional interface.

Concluding, we summarize what we feel are the main strengths of our architecture.

1. The VITAL-KR architecture enables the integration of data and query driven behaviour.

2. The VITAL-KR is entirely open: any inference tool satisfying the requirements listed in section 8 can be integrated.
3. While each IM is free to store and reason with its own data, an integrated architecture is achieved by means of the IS, the GDL, and the TMS.
4. The functionalities of the VITAL-KR as a whole, and the range of messages that can be sent from one component to another are clearly specified by the functional interface.

REFERENCES

- Aho, A.V., Hopcroft, J.E. and Ullman, J.D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
- Brachman, R.J., Fikes, R.E. and Levesque, H.J. (1983). *KRYPTON: A Functional Approach to Knowledge Representation*. FLAIR Technical Report No. 16, Fairchild Lab. for AI Research, Pal Alto, CA.
- Corkhill, D.D. (1991). Embedable Problem-Solving Architectures: A Study of Integrating OPS5 with UMass GBB. *IEEE Transactions on Knowledge and Data Engineering*, 3(1), March 1991.
- Elfrink, B. and H. Reichgelt (1989). 'Assertion-Time Inference in Logic-Based Systems' in Peter Jackson, Han Reichgelt and Frank van Harmelen (eds.) *Logic-Based Knowledge Representation* (Cambridge, Mass. 1989)
- Frisch, A. and Cohn, T. (1991). 1988 Workshop on Principles of Hybrid Reasoning, *AI Magazine*, 11(5).
- Guttag, J. (1977). Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6), 396-404.
- Jonker, W., Kontio, J., and Motta, E. (1991). Definition and positioning of the VITAL project. Internal Project Memo PTT/T732/I/1.
- Keene, S. (1989). *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, Massachusetts.
- Lenat, D.B. and Guha, R.V. (1990). Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project. Addison-Wesley.
- Lenat, D.B., Guha, R.V., Pittman, K., Pratt, D. and Shepherd, M. (1990). CYC: Towards programs with common sense. *Communications of the ACM*, 33(8), 30-49, August 1990.
- Levesque, H. (1984). Foundations of a Functional Approach to Knowledge Representation. *Artificial Intelligence*, 23, 155-212.
- Motta, E. (1991). Initial Formulation of the VITAL-KR Framework. Internal Project Memo VITAL/ID412.1/W/2.
- Motta, E., Rajan, T., Domingue, J., and Eisenstadt, M. (1991a). Methodological Foundations of KEATS, The Knowledge Engineers' Assistant. *Knowledge Acquisition*, 3(1), March 1991.
- Motta, E., Stutt, A., O'Hara, K., Kuusela, J., Toivonen, H., Reichgelt, H., Watt, S., Aitken, S. and Verbeck, F. (1991b). Knowledge Representation Language Specification: Final Deliverable. VITAL internal document OU/DD412/W/1. (HCRL Tech. Report 81V).
- Motta, E. and Stutt, A. (1991). Integration of Heterogeneous Languages (Architectural Issues). Internal Project Memo VITAL/ID412.2/W/2.
- Myers, K.L. (1991). Universal Attachment: An Integration Method for Logic Hybrids. *Proceedings of the 2nd International Conference on Knowledge Representation*. (Eds. Allen, J., Fikes, R. and Sandewall, E. *Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann.)

Zhang, C. and Bell, D.A (1991). HECODES: a framework for HEterogeneous COoperative Distributed Expert Systems. *Data & Knowledge Engineering* 6, 251-273.