International Journal of Cooperative Information Systems
Vol. 10, Nos. 1 & 2 (2001) 171–195

KRAFT: AN AGENT ARCHITECTURE FOR KNOWLEDGE FUSION

ALUN PREECE and KIT HUI
Department of Computing Science, University of Aberdeen
Aberdeen, Scotland, AB24 3UE, UK

ALEX GRAY and PHILIPPE MARTI
Department of Computer Science, Cardiff University
Cardiff, Wales, CF24 3XF, UK

TREVOR BENCH-CAPON
Department of Computer Science, University of Liverpool
Liverpool, L69 7ZF, UK

ZHAN CUI and DEAN JONES
Intelligent Business Systems Research, Advanced Communications Research
BT Labs, Ipswich, IP5 3RE, UK

Knowledge fusion refers to the process of locating and extracting knowledge from multiple, heterogeneous on-line sources, and transforming it so that the union of the knowledge can be applied in problem-solving. The KRAFT project has defined a generic agent-based architecture to support fusion of knowledge in the form of constraints expressed against an object data model. KRAFT employs three kinds of agent: *facilitators* locate appropriate on-line sources of knowledge; *wrappers* transform heterogeneous knowledge to a homogeneous constraint interchange format; *mediators* fuse the constraints together with associated data to form a dynamically-composed constraint satisfaction problem, which is then passed to an existing constraint solver engine to compute solutions. The paper presents the KRAFT architecture and the three kinds of agent, and includes a description of a demonstration KRAFT application in the domain of telecommunications service provision.

Keywords: Agent architectures, multi-agent systems, information agents, constraints, knowledge fusion, ontologies.

1. Introduction and Motivation

The KRAFT project (Knowledge Reuse And Fusion/Transformation) began in 1996 with the objective of defining a generic architecture for knowledge fusion. Knowledge fusion refers to the process of locating and extracting knowledge from multiple, heterogeneous on-line sources, and transforming it so that the union of the knowledge can be applied in problem-solving. KRAFT builds upon work done

in the early 1990s on knowledge sharing and reuse, most notably the results of the Knowledge Sharing Effort (KSE) project.[1]

Although it did result in a number of practical applications,[2,3] the early work on knowledge sharing and reuse has not had the expected impact in the construction of large-scale, open, distributed knowledge systems. One possible explanation for this is that, until recently, the demand for the technology in terms of appropriate application domains has been limited. However, this is changing with the rapidly-growing demand for systems that support the exchange and processing of rich information in areas such as electronic commerce[4] and knowledge management.[5] Another explanation for the limited take-up of the results of the KSE work is the complexity of the KSE's Knowledge Interchange Format (KIF), resulting in a significant effort to publish and process knowledge in KIF.

The KRAFT project aims to ease the development of knowledge fusion systems by restricting the form of knowledge to *constraints* expressed against an object data model.[6,7] The KRAFT architecture supports the following:

- locating appropriate on-line sources of knowledge;
- transforming heterogeneous knowledge to a homogeneous constraint interchange format;
- fusing the constraints with associated data to form a dynamically-composed constraint satisfaction problem (CSP);
- harnessing existing constraint solver engines to compute CSP solutions.

These features — heterogeneous on-line sources, an open, dynamic environment, legacy processing engines — led to the choice of an agent architecture as the basic model for KRAFT:

- *facilitator* agents support the description and location of on-line sources;
- sources are *wrapped* by agent software to transform local knowledge to and from the interchange format;
- *mediator* agents support the querying of sources, and fusion of knowledge from the sources;
- legacy solver engines are provided with agent *wrappers* as front-ends to their services.

This paper describes the KRAFT architecture with emphasis on its agent-based model. Before presenting an overview of the architecture in Sec. 3, motivating applications will be discussed in Sec. 2. Sections 4–6 examine the main types of agent in KRAFT. Section 7 describes a KRAFT application in the domain of telecommunications service provision. Section 8 surveys related work, and Sec. 9 concludes.

2. Motivating Applications

The KRAFT architecture was conceived to support *configuration design applications* involving multiple component vendors with heterogeneous knowledge and

data models. This kind of application is very general, covering not only the obvious manufacturing-type applications (for example, configuration of personal computers or telecommunications network equipment) but also service-type applications such as travel planning (for example, composing package holidays or business trips involving flights, ground travel connections, and hotels).

Configuration design problems are commonly tackled as constraint satisfaction problems.[8] Where components in the design will come from a number of different vendors, the domains of many of the variables in the CSP are entities stored in each vendor's local product catalogue database. Many of the constraints in the CSP will be on these entity types, defining how the components can be used in configured designs. Some constraints will refer to related instances of other entity types, whose values must be extracted from some other vendor's database and checked for compatibility. KRAFT provides mechanisms by which local database contents can be advertised on the network, so that constraints can be selected and fused together by specialized mediator agents, and passed to a constraint solver. The solver then has to find variable instantiations to satisfy the constraints.

Component suppliers make their catalogue databases available on the network. Locally to each supplier, the databases will have different semantics and assumptions. In printed catalogues, these assumptions often appear as asterisked footnotes or "small print". For example, the product catalogue for (fictitious) disk drive vendor, Storage Inc, may have the following "small print" associated with each of its range of Zip disk drives: *this Zip disk drive requires a PC with a USB-type port.* This kind of small print can readily be expressed as constraints in a database catalogue. For example:

```
forall z in zip_drives:
    port_types(connected_pc(z)) must include "USB";
```

This constraint would be stored in Storage Inc's catalogue database, but note that it refers to a property of the PC (`port_types`) to which the drive would be connected in a configured PC system (stored as the `connected_pc` attribute of the Zip disk drive `z`).

The locally-stored constraints will typically be expressed against a local data model for the product catalogue. In order to fuse together constraints from multiple heterogeneous product catalogues, it is necessary to translate the constraints to a common *constraint interchange format*, expressed in the terms of a *shared ontology* as is commonly done in knowledge-sharing systems.[1] The following constraint shows the above small print Zip disk constraint translated from Storage Inc's local constraint language to the KRAFT Constraint Interchange Format (CIF) language, expressed in the terms of a shared ontology for PC system configuration:

```
constrain
    each d in disk_drive
        such that name(vendor(d)) = "Storage Inc"
        and type(d) = "Zip"
    at least 1 p in ports(host_pc(d))
to have type(p) = "USB";
```

KRAFT CIF is based on the CoLan language used to express semantics in the object database P/FDM.[9] The terminology used in this transformed constraint (for example, the concept `disk_drive`, meaning all disk drive components, the attributes of this concept, `vendor` and `type`) must be defined in the shared ontology for the PC design domain. Some of the transformations needed here were:

- *Addition of contextual information.* In the local Storage Inc database, all constraints implicitly refer to this vendor's products; in the shared ontology, the name of the vendor of the component must be stated explicitly.
- *Mapping classes to attribute values.* In the local Storage Inc database, the type *Zip disk* is represented by the class `zip_drives`; in the shared ontology, "Zip" is the value of the attribute type of elements in the class `disk_drive`.
- *Coping with varying granularities of description.* In the local Storage Inc database, the connected PC is modelled in less detail, with the types of ports being stored in the attribute `port_types` of the `connected_pc` entity; in the shared ontology, the connected PC of the disk drive (attribute renamed to `host_pc`) is modelled in more detail, with individual ports as entities in their own right, and `type` as an attribute of the `port` entity.

Note that, to allow data instances to be represented, the shared ontology must be formalized at a level of detail which allows a schema to be extracted. For example, the ontology defines the concept `disk_drive` as a sub-concept of `storage_device`, which is in turn a sub-concept of `hardware_device`. It also defines `disk_drive` as having a `type` attribute, the value of which is represented as a string data structure. This information can be used as a schema to permit instances of `disk_drive` to be represented, stored, manipulated, and transmitted across the network.

Constraint transformations are implemented within the wrapper agent for each individual vendor, as part of the setting-up needed for the vendor to join the

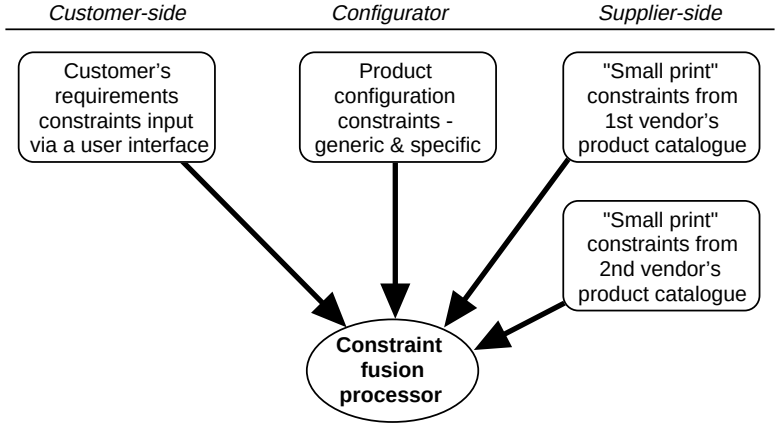


Fig. 1. Fusion of constraints from multiple sources.

KRAFT network. Once transformed, the small print constraints can be fused together with other constraints from various sources, as shown in Fig. 1.

In a typical configuration design application, some constraints will be provided by the customer; others will come from the vendors as discussed above; there will also be constraints coming from the service-provider who will act as the configurator of the product or service provided by the application. Typically, the configurator service-provider will be a value-adding reseller from the point-of-view of the component vendors. Note that there may be multiple configurators, each providing a different product or service; also, the design process may have additional stages, where one reseller sells to another reseller, each adding their own constraints to the final product or service. Details of how the constraint fusion process operates within the KRAFT architecture are given in Sec. 6.

3. The KRAFT Agent Architecture

All knowledge processing components in the KRAFT architecture are realized as software agents. External services (including legacy databases, constrant solving engines, and user interfaces) are "wrapped" with agent software to make them appear as agents to a KRAFT network. The perceived benefits of agent architectures as applicable to KRAFT are:

- *Openness.* Agent architectures are inherently open: agents can freely join a network, advertise their capabilities to one another, and dynamically form alliances for knowledge exchange and problem-solving.
- *Knowledge-level communication.* Agents communicate using knowledge-level protocols: these protocols accommodate knowledge representation languages as their content, and high-level conversational transactions define their sequencing.

The KRAFT architecture is designed to be consistent with emerging agent standards, notably the de facto KQML standard[10] and the de jure FIPA standard.[11]

An overview of the generic KRAFT architecture is shown in Fig. 2. KRAFT agents are shown as ovals. There are three kinds of these: wrappers, mediators, and facilitators. All of these are in some way knowledge-processing entities. External services are shown as boxes. There are three kinds of these: user agents, resources (typically databases or knowledge bases), and solvers. All of these external services are producers and consumers of knowledge: users supply their requirements to the network in the form of constraints via a user agent service, and receive results in the same way. Resources store, and can be queried for, knowledge and data. Solvers accept CSPs and return the results of the solving process.

3.1. *KRAFT agent types*

Wrappers. Wrappers are agents that act as proxies for external resources. These are often legacy systems, so one task of a wrapper is to provide a bridge between

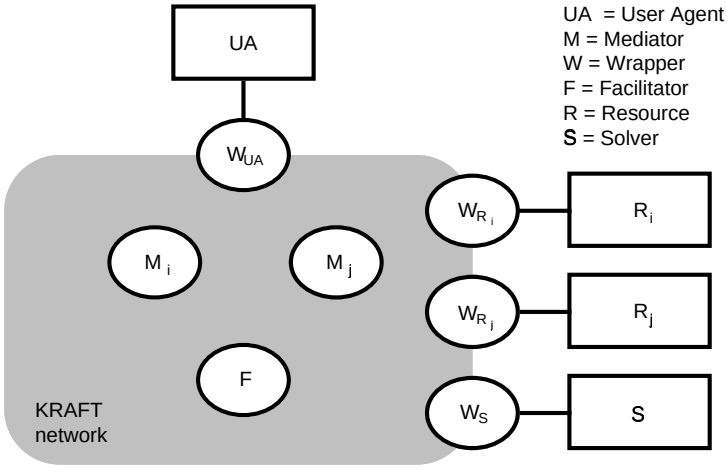


Fig. 2. Overview of the generic KRAFT architecture.

the legacy system interface and the KRAFT agent interface. For example, the legacy interface of a relational database will typically be SQL/ODBC; the KRAFT wrapper will accept incoming request messages from other agents in the KRAFT agent communication language, transform these into SQL queries, run them on the database, and transform the returned results to an outgoing message in the KRAFT agent communication language.

Wrappers also provide entry-points into the KRAFT system for user agents. User agents allow end-users access to a KRAFT knowledge processing system. A user agent will offer some kind of user interface, with which the user will present queries to the KRAFT network. The user agent will transform the users' requirements into the internal knowledge representation language of the KRAFT system, and interact with other KRAFT agents to answer the queries. A user agent will typically also do some local processing on knowledge, at least to transform it for presentation. The knowledge-transforming capabilities of wrappers are addressed in Sec. 5.

Mediators. Mediators are the internal knowledge-processing agents of the KRAFT system: every mediator adds value in some way to knowledge obtained from other agents. The most important mediator task in a KRAFT system is to fuse constraints obtained from other agents to generate a CSP at run-time. As part of this task, mediators will typically pre-process the constraints in various ways. They will also need to plan and perform selective database queries as explained in Sec. 6.

There will typically be several mediators in a KRAFT network: one to perform each distinct value-adding service. For example, in a KRAFT network performing configuration of PC products, there may be a single configurator mediator, or there may be several configurators, perhaps one for each of several different kinds of PC (laptops, generic desktops, special-purpose workstations, etc.) or one for

each of several distinct subsystems (CPU, peripheral systems, application software bundles, etc.).

Facilitators. Facilitators are the "matchmaker" agents that allow agents to discover one another. Agents coming online register their identities, network locations, and advertisements of their knowledge-processing capabilities with a known facilitator. When an agent needs to request a service from another agent, it asks a facilitator to recommend an agent that appears to provide that service. Facilitators are knowledge-processing entities: establishing that a service request "matches" a service advertisement requires reasoning with the declarative representations of request and advertisement. Details of the facilitation operations in KRAFT are given in Sec. 4.

Each KRAFT network requires at least one facilitator. In a large network, there may be multiple facilitators, either for reasons of specialization or efficiency.

3.2. *KRAFT communication protocols*

KRAFT agents communicate via messages using a nested protocol suite. KRAFT messages are implemented as character strings transported by a suitable carrier protocol. A simple message protocol encapsulates each message with low-level header information, including a timestamp and network information. The body of the message consists of two nested protocols: the outer protocol is the agent communication language CCQL (*Constraint Command and Query Language*) which is a subset of the Knowledge Query and Manipulation Language (KQML).[10] Nested within the CCQL message is its content, expressed in the CIF shown in Sec. 2.

In the current implementation, KRAFT messages are syntactically Prolog term structures. An example message is shown below. The outermost `kraft_msg` structure contains a `context` clause (header information) and a `ccql` clause. The message is from an agent called `storage_inc` to an agent called `pc_configurator`. The `ccql` structure contains, within its content field, a CIF expression (in the implementation, CIF expressions are actually transmitted in a compiled internal format).

```
kraft_msg(
    context(1,id(19), pc_configurator, storage_inc,
        time_stamp(date(29,9,1999), time(14,45,34))),
    ccql(tell, [
        sender : storage_inc,
        receiver : pc_configurator,
        reply_with : id(18),
        ontology : shared,
        language : cif,
        content : [
            constrain
                each d in disk_drive
```

```
                    such that name(vendor(d)) = "Storage Inc"
                    and type(d) = "Zip"
                at least 1 p in ports(host_pc(d))
            to have type(p) = "USB"
    ])
)
```

Use of Prolog term structures is chiefly for convenience, as most of the current knowledge-processing components in the KRAFT implementation are written in Prolog. However, the Prolog term structures are easily parsed by non-Prolog KRAFT components; currently there are several components implemented in Java, for example.

As explained in Sec. 2, the terms used in the CIF part of the message are defined in the shared ontology. To support the representation, storage, and transmission of data instances, the ontology has schema-level information in addition to conceptual-level definitions.

3.3. *Operational view of the KRAFT architecture*

This section presents an operational "walk-through" of the generic KRAFT network shown in Fig. 2. The generic network features a user agent UA, its wrapper W_{UA}, a facilitator F, two sample mediators M_i, M_j, two sample resources R_i, R_j and their wrappers W_{R_i}, W_{R_j}, and a solver S and it's wrapper W_S. In general, of course, there may be multiple user agents, solvers, and any number of mediators and wrapped resources. There may also be multiple facilitators.

The walk-through traces the steps involved in solving a single request, issued by a user to the user agent, UA. Each numbered step is from the point-of-view of a particular component, named at the start of the step. Messages between components are shown in the form:

`CCQL-performative`(*Message content*) $\rightarrow$ *Receiver*

(1) UA submits a request Q_{UA} in a format local to the user agent. Q_{UA} will typically be some kind of query, and may include constraints (expressed in the local constraint language).
(2) W_{UA} transforms Q_{UA} into a KRAFT request Q_K, in CIF expressed against the shared ontology. Again, Q_K may include constraints (now expressed in CIF).
(3) If W_{UA} already holds an advertisement `advertise`(A, C_A), where:
    — A is a named KRAFT agent
    — C_A is a capability of A
    — C_A matches Q_K
    Then goto step 5.
    Else, send message to facilitator F:
    `recommend_one`$(Q_K) \rightarrow F$

(4) F searches it's directory for an advertisement $\texttt{advertise}(A, C_A)$, where C_A matches Q_K, and sends:[a]
$\texttt{forward}(\texttt{advertise}(A, C_A)) \rightarrow W_{UA}$

(5) W_{UA} sends Q_K to the agent identified in the advertisement:
$\texttt{ask}(Q_K) \rightarrow A$

(6) A processes Q_K according to the kind of agent, it is:

- If A is a *wrapped resource*, W_{R_i}:
  W_{R_i} transforms Q_K into a local query Q_{R_i}, in the local ontology, which it submits to the Resource R_i; when W_{R_i} receives the response data D_{R_i}, it transforms D_{R_i} to a KRAFT result data object D_K, in CIF/shared ontology:
  $\texttt{tell}(D_K) \rightarrow W_{UA}$
- If A is a *mediator*, M_i:
  M_i decomposes Q_K into subtasks $Q_{K_1} \cdots Q_{K_n}$; then, in parallel, serially, or in some combination thereof, A recursively performs steps 3–6 with:
  — each Q_{K_i} substituting for Q_K
  — M_i substituting for W_{UA}
  M_i receives responses, and fuses them into a unified KRAFT result data object, D_K, in CIF/shared ontology:
  $\texttt{tell}(D_K) \rightarrow W_{UA}$
- If A is a *wrapped solver*, W_S:
  W_S transforms Q_K into statements in the solver's local language, which it submits to S.
  (*) If the Solver's response is a request for more data, D_S, then W_S:
  Transforms D_S to a KRAFT request, Q_{K_S}
  Recursively performs steps 3–6 with:
  — Q_{K_S} substituting for Q_K
  — W_S substituting for W_{UA}
  Receives response(s), transforms them, submits them to the solver, and goes to (*).
  Else W_S:
  Transforms response to a KRAFT result data object, D_K, in CIF/shared ontology:
  $\texttt{tell}(D_K) \rightarrow W_{UA}$
  If W_S needs to recursively perform steps 3–6 as noted above, then in performing step 3, it is possible that the solver's wrapper will consult the facilitator to find an agent that can handle its data request; however, it is likely that the solver's wrapper will direct it's requests for more data back to the originator of Q_K (probably a mediator). This is because the mediator will likely be constructing variable domains on behalf of the solver (see Sec. 6).

[a]This walk-through assumes that the agents are using "recommend-style" facilitation, as presented in Sec. 4. KRAFT facilitators also support "broker-style" facilitation, where the facilitator will relay the request Q_K directly to the advertising agent A on W_{UA}'s behalf. See Sec. 4 for further information on facilitator operations.

(7) W_{UA} receives the KRAFT result object, transforms it into the local format, and passes it to UA for display.

3.4. *Implementation of the KRAFT architecture*

Inter-agent communication in KRAFT is implemented by message passing using the Linda tuple-space communication model.[16] A Linda server manages the tuple space; clients connect to the space to write or read tuples (messages). KRAFT uses a Prolog implementation of Linda, where tuples are Prolog term structures: instances of the `kraft_msg` term structure shown in Sec. 3.2. To send a message, an agent writes it to a Linda server with the name of the recipient; to receive a message, an agent reads any tuples with its own name as the value of the `receiver` field. An advantage of using this model is that the individual agents do not need to be multithreaded; they choose when to receive any waiting messages synchronously. KRAFT agents can be written in any language provided that they have a Linda client module. Currently, these are available for Prolog and Java agents.

The Linda model is most effective for local-area communication, so to support wide-area KRAFT networks a federated Linda space has been implemented. Each local-area (called a *hub*) has its own Linda server with which local agents interact. The agent namespace has a URL-like *hubname/agentname* syntax. Each Linda server is coupled to a *gateway* agent that relays messages between hubs in a manner similar to an internet router: if a message is posted on the local Linda server with a non-local *hubname* for the recipient, the gateway relays the message to the correct hub gateway agent, which in turn writes it to the hub's local Linda server. This architecture is shown in Fig. 3. In the current implementation, the protocol used to carry messages between hubs is TCP via the socket interface; preliminary work has also been done on inter-hub communication using CORBA IIOP.[12]

To support debugging, a *Monitor* user agent has been implemented to trace and display the passage of messages across a KRAFT network, shown in Fig. 4. Monitor

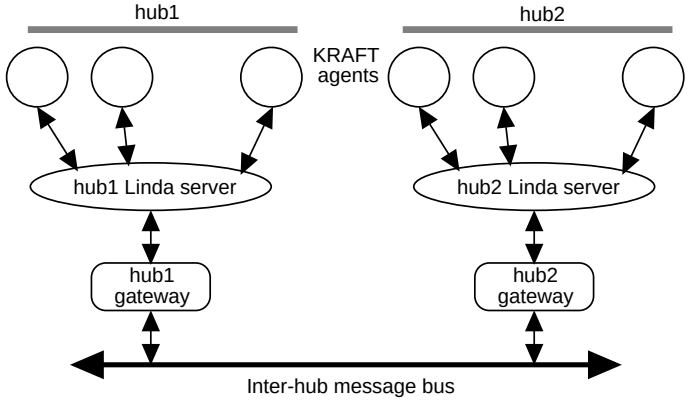


Fig. 3. Implementation of the KRAFT architecture.

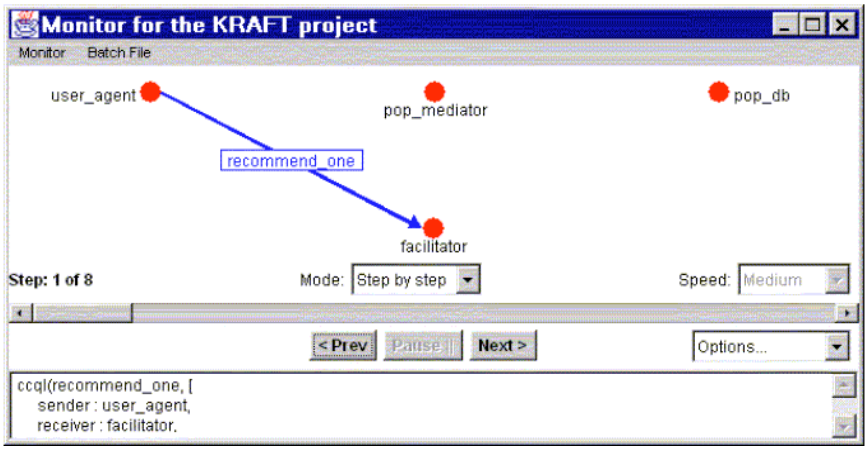


Fig. 4. Screenshot of the KRAFT Monitor user agent.

agents are able to register with the gateway agents in order to display activity at non-local hubs, allowing a user to see interactions across the entire KRAFT network.

The following sections examine the operations of each of the three kinds of KRAFT agent in more detail.

4. Facilitator Agents

Facilitators provide matchmaking services within a KRAFT network:

- *Advertisement handling*. When a facilitator receives an advertisement from a resource, it is able to process and store this information for further use. An advertisement is a (set of) capabilities that a resource commits to provide. Every resource willing to advertise its capabilities does so first by registering with a facilitator (by sending a CCQL `register` message containing the references and location of the resource), then by sending a CCQL `advertise` message containing the formal description of its capabilities (shown below).
- *Request facilitation*. For a given request, this is the action of finding a (set of) resources that comply with a satisfiability (or suitability) criterion. In other words, it is the action of finding a resource whose advertised capability matches the requirements derived from the query. This satisfiability criterion depends on the search strategy adopted for the resolution of the query but it is always somewhere in the spectrum bounded by "the most exact match" on one hand and by "the set of all approximate matches" on the other hand. In other words, the satisfiability criterion is a tradeoff between correctness and completeness in the set of solutions.

  Following the KQML model,[10] CCQL offers two ways for agents to interact with a facilitator:

Recommend-style facilitation: On receiving a query enclosed in a `recommend_one` (or `recommend_all`) message, the facilitator will reply to the agent at the origin of the query, a singleton (or a list) of matching advertisements (including the references to the corresponding resource). The replied advertisement(s) is contained in a `forward` message.
Broker-style facilitation: On receiving a query enclosed in a `broker_one` (or `broker_all`) message, the facilitator will send the enclosed query to the most (or all) relevant resource. The answer to the query is then brought back to the facilitator, which in turn forwards the reply to the originator of the query.

4.1. *Expressing capabilities*

A resource capability must be represented intentionally for compactness, but in a way that minimizes imprecision. The abstract characteristics of a resource are:

- network information to locate and query the wrapper of the resource;
- the CCQL performatives that the wrapper can handle;
- the intentional content of the resource (or a subset of it);
- an abstract representation of the functionality of the resource.

Advertisements are the basic data structure used to communicate these characteristics. The terms used in the body of advertisments are defined in the shared ontology. The possible components of an advertisement are:

- list of allowed CCQL *performatives*;
- list of available *services*, where each service is defined in the shared ontology;
- list of *domains* that the database can deal with, where each domain is defined in the shared ontology;
- specification of a subset of the CIF language, delimiting the expressiveness of the resource query language within CIF as a whole.

The facilitator encapsulates a database of received advertisements with the above components; the CCQL facilitation operations (forwarding and brokerage) are implemented as queries on this advertisement database.

An example CCQL advertisement message follows (the KRAFT message header is not shown). This advertisement is from the wrapper of a PC software vendor called `storage_inc`, sent to a facilitator called `yellow_pages`. The content says that the advertiser can handle CCQL `ask_one` and `ask_all` messages, expressed in a subset of CIF corresponding to SQL queries (from the service description, defined in the shared ontology), about ontology concepts `storage_device` and `pc_peripheral`.

```
ccql(advertise, [
    sender : storage_inc,
    receiver : yellow_pages,
    reply_with : id(48),
    ontology : advertise_ontology,
```

```
    language : cif,
    content : [
        advertisement.performativeList = [ ask_one, ask_all ],
        advertisement.serviceList = [ <database, sql> ],
        advertisement.domainList = [ storage_device, pc_peripheral ] ]
])
```

5. Wrapper Agents and the Shared Ontology

There are three levels of heterogeneity that inhibit the reuse of knowledge stored in resources that are to be connected to a KRAFT network:

- *Interaction*: Different knowledge sources can be interacted with in different ways; for example, some systems only allow the user to pose queries whereas other systems will ask the user for information.
- *Syntactic*: Knowledge sources use different representation formats.
- *Semantic*: Data models and terminology vary across different knowledge sources.

As outlined in Sec. 3, interaction and syntactic heterogeneity are addressed by the use of the CCQL and CIF protocols within the KRAFT network and by providing wrapper agents that translate all messages into and out of these protocols. Heterogeneity of data models is handled in KRAFT by the use of a common object data model against which the CIF constraints are expressed, and against which data instances can be stored and transmitted. This section focusses on the problem of semantic heterogeneity arising from the use of different terminology across the various knowledge sources.

To overcome this problem, a shared ontology is specified, which formally defines the terminology of the problem domain. The content of messages within a KRAFT network must be expressed using terms that are defined in the shared ontology. For each knowledge source, a local ontology is specified. For example, where the knowledge source is a database, the local ontology defines the terms that are used in the database schema. Between a local ontology and the shared ontology, there will be a number of ontology mismatches, which are instances of semantic heterogeneity.[13] These include the use of different terms to refer to the same concept (synonyms) and the use of the same term to refer to different concepts (homonyms). When a constraint is translated from one vocabulary to another, we would like to ensure that the knowledge expressed by the constraint is kept constant. If this were not the case, constraints passed to a mediator using terms defined in the shared ontology could express very different knowledge than the original constraints expressed in terms defined in the local ontology. In order to achieve this, an *ontology translation* is defined that specifies how an expression using terms defined in a source ontology is translated to an expression using terms defined in the target ontology.

The first step in defining an ontology translation is to specify a set of ordered pairs or *ontological correspondences*. An ontological correspondence specifies the

term or expression in the target ontology that represents as closely as possible the meaning of the source ontology term or expression. For each term in the source ontology, an attempt is made to identify a corresponding term in the target ontology. It may not be possible to directly map all of the source ontology terms to a corresponding target ontology term. For some of the terms in the source ontology that cannot be mapped in this way, it may be possible to include them in the ontology translation by defining correspondences between compound expressions. This leads to the following classification of ontological correspondences:

- *Class correspondences*: Map from source ontology terms that denote classes to target ontology terms that denote classes.
- *Instance correspondences*: Map from source ontology terms that denote individuals to target ontology terms that denote individuals.
- *Slot correspondences*: Map from source ontology terms that denote binary relations to target ontology terms that denote binary relations.
- *Compound-expression correspondences*: Map from a source ontology compound expression to a target ontology compound expression.

A pair of terms and/or expressions in an ontological correspondence are not necessarily semantically equivalent. As mentioned above, when a wrapper translates a CIF expression, we want to ensure that the target CIF expression is semantically equivalent to the source CIF expression. We ensure that the meaning of expressions is not changed by defining sets of conditions that must be satisfied by the expressions involved in the translation. Conditions that must hold true of the source expression are called *pre-conditions* and those that must hold true of the target expressions are called *post-conditions*. Given two ontologies O_1 and O_2 which define the terminology of two languages L_1 and L_2 with vocabularies V_1 and V_2 respectively, a translation T from terms defined in O_1 to terms defined in O_2 is a structure $\langle F_{12}, \mathbf{Pre}, \mathbf{Post} \rangle$ where:

- F_{12} is a partial function $F_{12} : V_1 \rightarrow V_2$,
- the set **Pre**, which is a set of pre-conditions,
- the set **Post**, which is a set of post-conditions.

For example, consider the conversion of the constraint from the terminology of the Storage Inc. ontology to the terminology of the shared ontology given in Sec. 2. The relevant ontology translation for this conversion would include the following class correspondence:

```
<zip_drives, disk_drive>
```

Since the source term denotes a subclass of the class denote by the target term, we include the following post-condition for this correspondence (expressed using a Prolog-style list of two elements, the first of which is a list of types for variables used in the second element, which represents the condition):

```
[[d:disk_drive], [name(vendor(d)) = "Storage Inc" and type(d) ="Zip"]]
```

To enable bidirectional translation between a KRAFT network and a knowledge source, two such ontology translations must be defined.

As the local and shared ontologies are not represented in the same format that is used for the CIF, the semantic transformation of CIF expressions by wrappers is not done by interpreting ontology translations directly. Rather, the relevant ontology translations are used as part of the specification of a wrapper. Consequently, developers have complete autonomy in the implementation of wrappers. In the KRAFT application described in Sec. 7, the transformation of CIF expressions is implemented in wrappers using rewrite rules.[14] A wrapper that implements an ontology translation must ensure that the pre- and post-conditions are satisfied when translating CIF expressions from the source to the target ontology.

6. Mediator Agents and Constraint Fusion

As explained in Sec. 2, an application problem in KRAFT is specified by constraints extracted from different sources on the network. Each constraint becomes part of a conjunctive statement describing the application problem as a constraint satisfaction problem (CSP). In the PC hardware configuration domain, problem solving knowledge comes from the user requirements, restrictions attached to hardware components from different vendors, and generic design knowledge governing a workable configuration. The following example constraint from the user agent specifies that the PC must use a "`pentium3`" processor but not the "`win98`" OS:

```
constrain each p in pc
    to have cpu(p) = "pentium3"
    and name(has_os(p)) <> "win98"
```

For the components to fit together, they must satisfy certain basic configuration constraints. For example, the size of the OS must be smaller or equal to the hard disk space for a proper installation:

```
constrain each p in pc
    to have size(has_os(p)) =< size(has_disk(p))
```

This kind of constraint comes from the configurator's knowledge base.

Now the candidate components from different vendors may have instructions attached to them as "small print" kinds of constraint. In the vendor database of operating systems, "`winNT`" requires a memory of at least 32 megabytes:

```
constrain each p in pc such that name(has_os(p)) = "winNT"
    to have memory(p) >= 32
```

The KRAFT approach to this task employs a constraint-fusing mediator which extracts and combines constraints from distributed sources for problem solving purposes. Constraints as mobile pieces of knowledge are transported and transformed to compose a constraint satisfaction problem (CSP), which is then analyzed and solved by a combination of distributed database queries and constraint logic programs. With the help of a facilitator, this approach allows tailoring of an execution

plan in a dynamic environment, depending on the capability and availability of active online resources.

The sample constraints above give the following fused constraint, which describes the overall requirement on the variables involved:

```
constrain each p in pc
    to have cpu(p) = "pentium3"
    and name(has_os(p)) <> "win98"
    and size(has_os(p)) =< size(has_disk(p))
    and if name(has_os(p)) = "winNT"
        then memory(p)) >= 32
        else true
```

The reason for fusing the constraint fragments is to provide the basis for exploring how the CSP can best be divided into sub-problems of distributed database queries and sub-CSPs. When a single piece of constraint is insufficient to solve a CSP effectively, we hope to combine information from multiple constraint fragments to arrive at a more tractable solution. It is from the fusion process that useful information can be inferred and captured for problem solving purposes.

6.1. *CSP solving*

The constraint fusion process composes a concrete description of the overall CSP in a declarative form. To solve a composed CSP efficiently, a mediator feeds it into a problem decomposer which extracts selection information from the CSP description to generate distributed database queries, with the remaining constraints forming a smaller sub-CSP. The mediator then sends these database queries in multiple messages to different database wrappers to retrieve candidate data values.

Database query generation constitutes an important phase of pre-processing. It shifts part of the problem solving process into the distributed databases by composing data filters as database queries. This prevents unnecessary transportation of irrelevant data into the KRAFT domain and relieves network traffic in a distributed system. Data filtering by database query generation, however, is not sufficient to resolve all constraints. The amount of selection information which can be represented as database queries depends on the expressiveness of the database query language. The remaining sub-CSP has to be resolved by a more powerful constraint solver in the next stage. The final stage of the problem solving process is to feed data and constraints into a constraint solver so that solutions to the CSP can be obtained. In the application system, described in Sec. 7, we use the finite domain constraint solver in the ECLiPSe constraint logic programming (CLP) system.[b]

To form the initial value domains of variables in a CLP program, candidate data retrieved in the previous stage are compiled into CLP data structures. The sub-CSP which is formed by the problem decomposer is then compiled into CLP program

[b] `http://www.ecrc.de/eclipse/`

codes to impose constraints on these variables. Finally, the mediator sends the CLP program and data to the constraint solver and waits for the result to be returned.

7. Applying the KRAFT Architecture

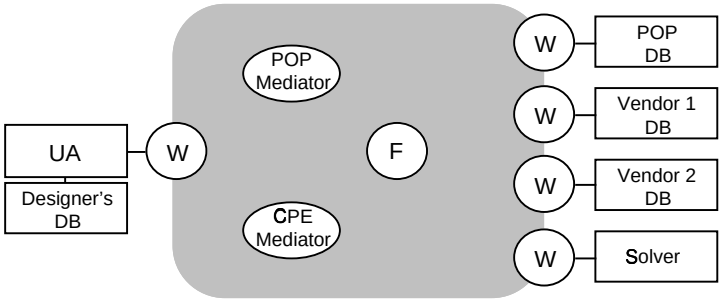
To establish proof-of-concept, the KRAFT architecture has been tested with a realistic application in the domain of telecommunications network data services design; this application was specified by the KRAFT project's industrial partner, BT. The network data services design problem considered by KRAFT is in the phase of network configuration from the viewpoint of a customer at a single site, allowing a BT network designer to select services and equipment to meet the customers' requirements:

- A suitable Point of Presence (POP) at which to connect to the BT network.
- Suitable Customer Premises Equipment (CPE) with which to service the connection; types of CPE include routers, bridges, and FRADs, though it was decided to focus initially solely on *router* products.

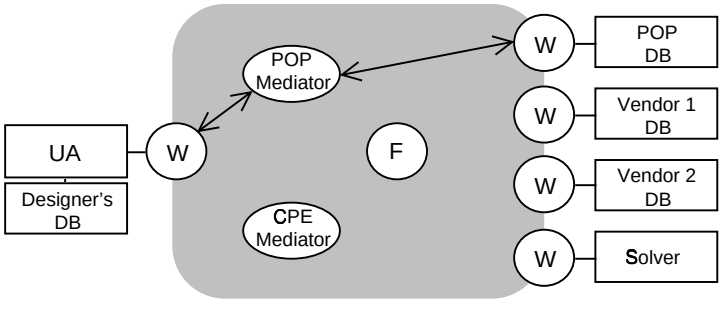
A conceptual view of the test application as implemented is shown in Fig. 5(a). This application maps onto the generic architecture shown in Fig. 2 as follows:

- A single wrapped *User Agent*, designed by BT, provides a user interface for the two kinds of request listed above. Coupled to this user agent is a database of designer knowledge, which will be accessed during the network data services configuration design process. The *User Agent Wrapper* provides network access to and from both the user agent and the *Designer's DB*.
- As the two kinds of request are independent (it is possible to select a CPE on the basis of a customer's LAN and WAN requirements, without knowing which POP will be used, and vice versa), it was decided to provide a separate mediator for each task: the POP request is handled by the *POP Mediator*, and the CPE request is handled by the *CPE Mediator*.
- There is a single *Facilitator* which is not specific to the application domain, except that it has access to the shared ontology.
- There are four wrapped resources:
  POP Database, a database of POPs (based on BT's own POP database);
  Vendor 1 *DB*, a product catalogue database for a CPE vendor (based on the actual product catalogue of 3Com);
  Vendor 2 *DB*, a product catalogue database for a second CPE vendor (based on the actual product catalogue of Cisco).
  Designer's DB, a source of network data services design constraints (based on knowledge acquired from BT network data services designers).
- There is a single wrapped legacy constraint *Solver* engine.

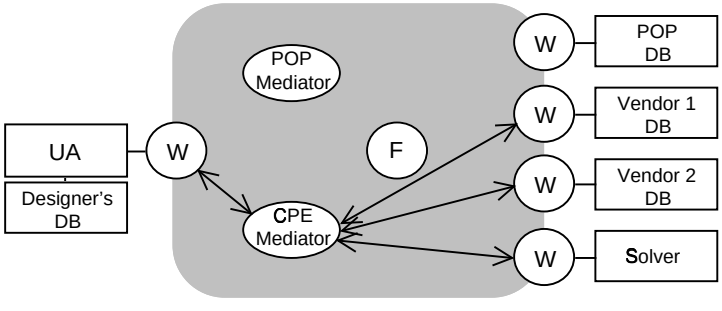
All the agents here (mediators, facilitators, and wrappers) are implemented in Prolog. The user interfaces (BT's user agent, and the Monitor shown in Sec. 3.4) are



(a) KRAFT Network DataServices application architecture.



(b) KRAFT Network DataServices application interaction 1: locate a POP.



(c) KRAFT Network DataServices application interaction 2: choose CPE.

Fig. 5. The architecture of the KRAFT test application. NOTE In (b) and (c), interactions with the facilitator are not shown.

Java applications. The database resources are managed by independent instances of the P/FDM DBMS[c], each with its own local schema. The constraint solver is ECLiPSe.

The four wrapped resources are considered to be pre-existing legacy databases. For the purposes of the prototype, simplified versions of these databases were built; however, care was taken to ensure that the databases of CPE information

[c] `http://www.csd.abdn.ac.uk/~pfdm`

were created independently, so as to ensure realistic heterogeneity. Each of the databases was populated with data and constraints; for example, a vendor database was populated with data on the vendor's CPE products, and constraints defining the valid usage of each product. The main aim of creating the four resources was to demonstrate the feasibility of creating wrapper agents to transform between the internal knowledge representation (data and constraints) and the KRAFT CIF language.

When the various service-providing agents come on-line, each sends an appropriate `advertise` message to the faciliator:

- *POP DB Wrapper* advertises that it can supply *POP* data objects;
- *Vendor* 1 *Wrapper* advertises that it can supply *router* data objects where the manufacturer is "Vendor 1";
- *Vendor* 2 *Wrapper* advertises that it can supply *router* data objects where the manufacturer is "Vendor 2";
- *Solver Wrapper* advertises that it can process finite domain CSPs;
- *POP Mediator* advertises that it can supply information on POPs that are closest to a given location;
- *CPE Mediator* advertises that it can supply CPE data objects from multiple vendors that meet given customer requirements.
- *User Agent Wrapper* advertises that it can supply network data services design constraints.

The following subsections provide a flavour of how the application operates, by tracing a single walk-through of each of the two main operations: selecting a POP, and selecting CPE.

7.1. *Handling POP requests*

A POP request issued by the user agent results in the following sequence of actions, summarized in Fig. 5(b):

(1) Via the *User Agent*, the user specifies the location of the customer's site, and the customer's required wide-area network (WAN) services (for example, Frame Relay and ISDN).
(2) The *User Agent Wrapper* formulates the POP query as a KRAFT message, and attempts to locate an agent that can answer the query by contacting the *Facilitator* through a `recommend` CCQL message, indicating that it needs to find a POP closest to a given location.
(3) The *Facilitator* matches the *User Agent Wrapper*'s request to the advertisement from the *POP Mediator*, and `forward`s the matching advertisement back to the *User Agent Wrapper*.
(4) The *User Agent Wrapper* sends an `ask-one` message to the *POP Mediator*, requesting a POP that meets the user's requirement constraints (location and services).

(5) The *POP Mediator* contacts the *Facilitator* to find a source of POP data, and is `forwarded` the advertisment from the *POP DB Wrapper*. It then sends an `ask-all` message to the *POP DB Wrapper*, requesting all POP data objects with the required services.
(6) Assuming that the *POP DB Wrapper*'s reply was non-empty, the *POP Mediator* computes which POPs are nearest the customer's site, and sends these data objects in a `tell` message to the *User Agent Wrapper*. This computation is simple enough that the *POP Mediator* performs it itself, and does not need to invoke the *Solver*.
(7) Upon receipt of the data from the *POP Mediator*, the *User Agent Wrapper* transforms it to the local format for presentation to the user via the *User Agent* itself.

7.2. *Handling CPE Requests*

A CPE request issued by the user agent results in the following sequence of actions, summarized in Fig. 5(c):

(1) Via the *User Agent*, the user specifies additional constraints on the type of equipment needed, including support for various LAN protocols used within the customer's site (TCP/IP, AppleTalk, 10 base T Ethernet, etc.) and support for the required WAN services that determined the choice of POP (Frame Relay, ISDN, etc.).
(2) The *User Agent Wrapper* interacts with the *Facilitator* as above, this time looking for vendor-independent CPE data objects. It is `forward`ed the *CPE Mediator*'s advertisement.
(3) The *CPE Mediator* receives an `ask-all` request from the *User Agent Wrapper*, specifying all the customer's requirement constraints. It sends a `recommend-all` message to the *Facilitator* to discover all CPE vendors currently on-line.
(4) The *Facilitator* finds no CPE vendors have advertised but, knowing from the shared ontology that *router is a kind of CPE*, it is able to `forward` *CPE Mediator* the advertisements from *Vendor* 1 *Wrapper* and *Vendor* 2 *Wrapper*.
(5) The *CPE Mediator* uses some of the customer's requirement constraints to formulate `ask-all` requests to each vendor's wrapper. Each wrapper responds, `tell`ing the *CPE Mediator* the router data objects that meet the given requirements, and any attached "small print" constraints on these router data objects.
(6) The *CPE Mediator* formulates a CSP by fusing the constraints it now has:
   - all the customer requirement constraints;
   - all the "small print" constraints on router data objects from both vendors;
   - network data services design constraints which it obtains by sending an `ask-all` message to the *User Agent Wrapper*, having discovered its location from the *Facilitator*.

(7) The CSP is formulated as a finite domains CSP, so the *CPE Mediator* interacts with the *Facilitator* to discover a finite domain solver. It then sends the *Solver Wrapper* the CSP.
(8) Assuming there is at least one solution to the CSP, the *Solver Wrapper* `tell`s the solution set to the *CPE Mediator*, which then returns these results to the *User Agent Wrapper*.
(9) The user can examine the solutions (if any) via the *User Agent* and, if necessary, refine the constraints and invoke further requests to the KRAFT network.

The implemented application was tested to demonstrate the essential functionality of all the components (facilitation, constraint transformation, and constraint fusion), and also to test the performance of a wide-area KRAFT network (with agents running at all four of the project sites across the UK). While the functionality was proven satisfactorily, the performance was sluggish due largely to the choices of platform (Prolog and Java). Further conclusions are drawn in Sec. 9. Further details of the testbed application are available in Ref. 15.

8. Related Work

Agent-based architectures are proving to be an effective approach to developing distributed information systems,[17] as they support rich knowledge representations, meta-level reasoning about the content of on-line resources, and open environments in which resources join or leave a network dynamically.[18] KRAFT employs such an agent-based architecture to provide the required extensibility and adaptability in a dynamic distributed environment. Unlike most agent-based distributed information systems, however, KRAFT focuses on the exchange of data and constraints among agents in the system.

Recent research in the area of software agent technology offers promising ways of supporting new kinds of distributed information system applications, but the area is still far from mature. Early projects such as PACT[2] and SHADE[3] showed that agent technology could support exchange of rich business information — using the Knowledge Interchange Format (KIF) — between organizations using heterogeneous technologies. While demonstrating the promise of the agent-based approach, these projects revealed problems: chiefly, the complexity of the KIF representation has prevented it from gaining widespread use.

The ADEPT project offers a flexible environment for distributed information system applications, with an emphasis on the dynamic management of workflow between partner organizations.[19] Service agreements are negotiated, formed, and re-formed over time, supporting both competitive and collaborative interactions, albeit with rather limited forms of information exchange.

The design of the KRAFT architecture builds upon recent work in agent-based distributed information systems. In particular, the roles identified for KRAFT agents are similar to those in the InfoSleuth system;[17] however,

while InfoSleuth is primarily concerned with the retrieval of data objects, the focus of KRAFT is on the combination of data and constraints. KRAFT also builds upon the work of the Knowledge Sharing Effort,[1] in that some of the facilitation and brokerage methods are employed, along with a subset of the 1997 KQML specification.[10] Unlike the KSE work, however, which attempted to support agents communicating in a diverse range of knowledge representation languages (with attendant translational problems), KRAFT takes the view that constraints are a good compromise between expressivity and tractability.

The specification and design for KRAFT's facilitators is informed by the many different definitions of facilitators in the literature.[3,20–22] Few of these approaches make use of constraints to support the matching of customer requirements and supplier capabilities. One constraint-based approach to the facilitation problem is the Matchmaker project, which is currently being applied to electronic commerce applications.[23] Like the Xerox work, the Matchmaker project does not deal with the extraction of constraints from distributed sources, and their use in problem-solving.

The exploration of ontologies, how to construct them and how to use them, is the focus of a great deal of activity. For a good selection of current and recent work see The Proceedings of the Eleventh[24] and Twelfth[25] Workshops on Knowledge Acquisition, Modeling and Management. For a description of issues relating to mappings in particular, see Ref. 26.

In its emphasis on constraints, KRAFT is similar to the Xerox Constraint Based Knowledge Brokers project;[27] the difference is that the Xerox work focusses upon the use of constraints to support querying of distributed data sources, rather than the extraction of constraints from distributed sources, and the use of these constraints in configuration design problem-solving. Also, KRAFT recognizes the need to transform constraints when they are extracted from local resources, typically for reasons of ontological or schema mismatch.[6,13]

The Smart Clients project[28] is related to KRAFT in the way they conduct problem-solving on a CSP dynamically specified by the customer, using data extracted from remote databases. Their approach differs from KRAFT in that only data is extracted from the remote databases, no small print constraints come attached to the data; also, all the problem-solving is done on the client, rather than by mediator agents. No constraints are therefore transmitted across the network; conversely, it is the constraint solver that is transmitted to the client's computer, to work with the constraints specified locally by the customer.

Finally, ongoing work at IBM is similar in concept to KRAFT's use of "small print" constraints.[29] The difference is that this work uses a rule-based formalism to specify contractual "fine print" in the form of business rules. Logic programming techniques are then used to reason with the rules.

9. Conclusion

This paper has described the KRAFT agent-based architecture for knowledge fusion. The testbed application specified by BT has established the feasibility of the KRAFT approach to supporting *virtual organizations* where vendors are able to advertise their product catalogues to resellers, who in turn offer value-adding services to customers via customized user agents.[30] Specific software components of the KRAFT system are highly reusable, including the CCQL messaging system, facilitators, wrapper shells, and several mediator and solving-related components.

Clearly, there is a cost associated with setting up a KRAFT network, in that members must wrap their knowledge sources to conform to the shared protocols and knowledge exchange languages. However, KRAFT aims to demonstrate that the use of constraints offers an effective "middle way" between the off-putting complexity of KIF at one extreme, and the limited expressivity of basic data exchange approaches at the other extreme.

The prototype network data services application has proven the concept of supporting configuration design problems by constraint fusion. In doing so, it has raised a number of issues that will be the subject of future work:

- the small scale of the prototype does not provide convincing evidence of the scalability of the KRAFT approach needed to cope with Internet-scale applications — further testing for scalability on larger KRAFT networks is needed;
- control is decentralized within the KRAFT network, and transactions are overly loose at present: more robustness and control is needed to support real-world applications.

Acknowledgments

KRAFT is a collaborative research project between the Universities of Aberdeen, Cardiff and Liverpool, and BT. The project is funded by EPSRC and BT. We would like to thank Nick Fiddian, Mike Shave, and Jean-Christophe Pazzaglia for their comments on this and earlier versions of this paper.

References

1. R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator and W. Swartout, Enabling technology for knowledge sharing, *AI Magazine* **12** (1991) 36–56.
2. M. Cutkosky, R. Engelmore, R. Fikes, M. Genesereth, T. Gruber, W. Mark, J. Tenenbaum and J. Weber, PACT: An experiment in integrating concurrent engineering systems, *IEEE Comput.* **26** (1993) 8–27.
3. D. R. Kuokka, J. G. McGuire, J. C. Weber, J. M. Tenenbaum, T. R. Gruber and G. R. Olsen, SHADE: Technology for knowledge-based collaborative engineering, *J. Concurrent Eng.*: *Appl. Res.* **1**, 2, 1993.
4. R. Kalakota and A. Whinston, *Electronic Commerce*: *A Manager's Guide* (Addison-Wesley, 1997).

5. J. Liebowitz and L. Wilcox, *Knowledge Management and Its Integrative Elements* (CRC Press, New York, 1997).
6. P. Gray, A. Preece, N. Fiddian, W. Gray, T. Bench-Capon, M. Shave, N. Azarmi and M. Wiegand, KRAFT: Knowledge fusion from distributed databases and knowledge bases, *Proc. 8th Int. Workshop Database Expert Syst. Appl.* (*DEXA-97*) (IEEE Press, 1997) 682–691.
7. P. Gray, K. Hui and A. Preece, Finding and moving constraints in cyberspace, *AAAI Spring Symp. Intell. Agents Cyberspace* (AAAI Press, Menlo Park, CA, 1999) 121–127.
8. E. Freuder and B. Faltings, *Configuration: Papers from the AAAI-99 Workshop* (AAAI Press, 1999).
9. N. Bassiliades and P. M. D. Gray, CoLan: A functional constraint language and its implementation, *Data and Knowledge Engineering* **14** (1994) 203–249.
10. Y. Labrou, Semantics for an agent communication languagex, PhD Thesis, University of Maryland, Baltimore MD, USA, 1996.
11. L. Chiariglione, FIPA — Agent technologies achieve maturity, *AgentLink Newsletter* **1** (1998) 2–4.
12. A. Preece, A. Borrowman and T. Francis, Reusable components for KB and DB integration, *Proc. ECAI'98 Workshop Intell. Inf. Integration*, 1998, 157–168.
13. P. R. S. Visser, D. M. Jones, T. J. M. Bench-Capon and M. J. R. Shave, Assessing heterogeneity by classifying ontology mismatches, *Proc. Int. Conf. Formal Ontology Inf. Syst.* (*FOIS98*) (IOS Press, 1998) 148–162.
14. P. M. D. Gray, S. M. Embury, K. Y. Hui and G. Kemp, The evolving role of constraints in the functional data model, *J. Intell. Inf. Syst.*, 1999, 1–27.
15. N. J. Fiddian, P. Marti, J-C. Pazzaglia, K. Hui, A. Preece, D. M. Jones and Z. Cui, Application of KRAFT in data service network design, *BT Tech. J.* **14** (1999) 117–130.
16. N. Carriero and D. Gelernter, Linda in Context, *Commun. ACM* **32** (1989) 444–458.
17. R. Bayardo *et al.*, InfoSleuth: Agent-based semantic integration of information in open and dynamic environments, *Proc. SIGMOD'97*, 1997.
18. G. Wiederhold and M. Genesereth, The basis for mediation, *Proc. 3rd Int. Conf. Coop. Inf. Syst.* (*COOPIS95*), 1995.
19. N. Jennings, P. Faratin, M. Johnson, T. Norman, P. O'Brien and M. Wiegand, Agent-based business process management, *Int. J. Coop. Inf. Syst.* **5** (1996) 105–130.
20. W. P. Birmingham, An agent-based architecture for digital libraries, *D-Lib Magazine*, July 1995.
21. U. M. Borghoff, R. Pareschi, H. Karch, M. Nöhmeier and J. H. Schlichter, Constraint-based information gathering for a network publication system, *Proc. 1st Int. Conf. Practical Appl. Intell. Agents Multi-Agent Tech.*, April 1996, 45–59.
22. N. Singh, M. Genesereth and M. A. Syed, A distributed and anonymous knowledge sharing approach to software interoperation, *Int. J. Coop. Inf. Syst.* **4**, 4 (1995) 339–367.
23. E. Freuder and R. Wallace, Matchmaker agents for electronic commerce, *Artificial Intelligence for Electronic Commerce*: *Papers from the AAAI–99 Workshop* (AAAI Press, 1999).
24. B. Gaines and M. Musen (eds.), *Proc 11th Int. Workshop Knowledge Acquisition, Model. Management* (*KAW'98*), April 1998.
25. B. Gaines, M. Musen and R. Kremer (eds.), *Proc 12th Int. Workshop Knowledge Acquisition, Model. Management* (*KAW'99*), October 1999.

26. J. Park, J. Gennari and M. Musen, Mappings for reuse in knowledge-based systems, *Proc 11th Int. Workshop Knowledge Acquisition, Model. Management* (*KAW'98*), April 1998.
27. J. Andreoli, U. Borghoff and R. Pareschi, Constraint agents for the information age, *J. Universal Comput. Sci.* **1** (1995) 762–789.
28. M. Torrens and B. Faltings, Smart clients: Constraint satisfaction as a paradigm for scaleable intelligent information systems, *Artificial Intelligence for Electronic Commerce: Papers from the AAAI-99 Workshop* (AAAI Press, 1999).
29. D. Reeves, B. Grosof, M. Wellman and H. Chan, Toward a declarative language for negotiating executable contracts, *Artificial Intelligence for Electronic Commerce: Papers from the AAAI-99 Workshop* (AAAI Press, 1999).
30. A. Preece, K. Hui and P. Gray, KRAFT: Supporting virtual organizations through knowledge fusion, *Artificial Intelligence for Electronic Commerce: Papers from the AAAI-99 Workshop* (AAAI Press, 1999) 33–38.