Chapter 1

CONSTRAINTS AS MOBILE SPECIFICATIONS IN E-COMMERCE APPLICATIONS

Kit-ying Hui,
Peter M. D. Gray,
Graham J. L. Kemp,
and Alun D. Preece
Department of Computing Science
King's College, University of Aberdeen
Aberdeen AB24 3UE
Scotland, United Kingdom
{khui|pgray|gjlk|apreece}@csd.abdn.ac.uk

Abstract We show how quantified constraints expressed in a sub-language of first-order logic, against a shared data model that is free to evolve, provide an excellent way of transporting domain-specific semantics along with the data. In this form it can be processed automatically by various intelligent components, instead of requiring human intervention, as it would do if expressed in natural language. It can also be combined with other constraints, by algebraic transformation against a common data model, and then passed to an appropriate solver. These techniques have been tested in a classic e-business application scenario: configuring a product from parts selected from e-vendors' catalogues, whilst conforming to requirements specific to the parts, expressed as mobile constraints.

1. Introduction

Providing technological support to the formation and operation of dynamic and open virtual organisations is a central concern in business-to-business e-commerce (Preece et al., 1999; Schein, 1994). In a virtual organisation, member companies integrate their resources to create a more competitive whole. To support these organisations, the communication mechanisms must cope with both the cooperative and the competitive nature of the enterprise. Further, business processes in a virtual organisation interact like agents by exchanging information to achieve certain tasks. Thus the communication mechanism must be powerful enough to support the exchange of data, information and knowledge among members.

Currently, the main technologies offered to support virtual organisations are Electronic Data Interchange (EDI) and Extranets. Unfortunately, current EDI systems are largely proprietary

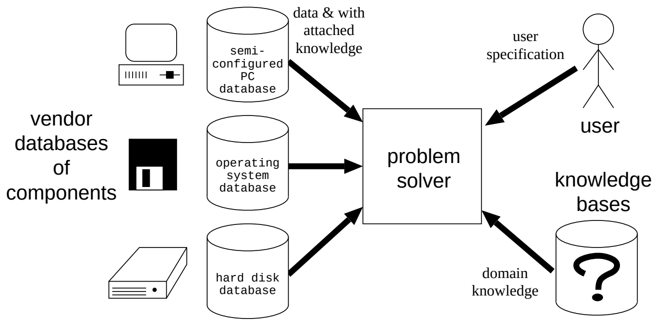

data & with attached knowledge
semi-configured PC database
operating system database
hard disk database
vendor databases of components
problem solver
user specification
user
knowledge bases
domain knowledge

Figure 1.1. This figure shows an application where components are put together to configure a PC. Both data and knowledge have to be combined before a solution can be found.

and limited to the exchange of relatively simple relational data. The new XML standard is non-proprietary and it will be good for exchanging semantics according to an agreed document type definition (DTD), but it does not rule out using natural language comments to convey semantics. Business data needs to be much more "self-describing" and to have attached meta-knowledge on how the information can be used and combined with other information (Jeffery, 1998). We present ideas on how this can be done using constraints, so that the semantics of the data are made explicit to remote programs.

The KRAFT project[1] (Gray et al., 1997) has an architecture that is suitable to support virtual organisations in which members exchange information in the form of constraints expressed against an object data model (Preece et al., 1999). The constraints allow member companies to design new products from components in their individual catalogues, and also to advertise the content of their catalogues in a way that is meaningful to remote programs and not just to humans. Constraints are exchanged via messages expressed in an agent communication language, supporting flexible transactions.

1.1. Motivation

KRAFT was conceived primarily to support configuration design applications among multiple partner organisations with heterogeneous knowledge and data models. This makes it suitable for the support of virtual organisations.

Consider the problem of configuring a computer from the set of product catalogues provided by different vendors as databases (figure 1.1). We call these product data *candidate data* as they are potential values of solutions in valid configurations.

Configuration specifications come in the form of constraints from various sources. For example, a particular requirement can be:

> *"The PC must use a Pentium II processor."*

There are also constraints that govern a usable configuration. Here is one of them:

> *"The size of a hard disk must be big enough to accommodate the chosen operating system."*

[1]KRAFT = Knowledge Reuse And Fusion/Transformation.

To arrive at a usable configuration, we may issue a distributed database query that performs a join across multiple database tables to get the candidate components and then check the retrieved data for compatibility and requirement. However, as problem domains become more sophisticated, it is insufficient to store only data but also knowledge in order to capture the *semantics* of the application domain.

Thus databases may have different semantics and hidden assumptions stored together with data, as components stored in vendor databases may have specific instructions attached to them, describing how the products have to be used. For example, a particular operating system may have the following requirement attached:

> "*Windows NT requires a minimum memory of 64M bytes in your PC.*"

Therefore, it is usually inadequate to use a distributed database query for finding a list of compatible parts. We must also ensure that the hidden semantic knowledge is properly utilised.

This problem originates from the fact that knowledge no longer statically resides in a resource but becomes mobile. Like footnotes in a product catalogue, some mobile knowledge is attached to data objects and thus they must be involved (or satisfied) whenever those data objects are used.

1.2. A Distributed Configuration Design as a Constraint Satisfaction Problem

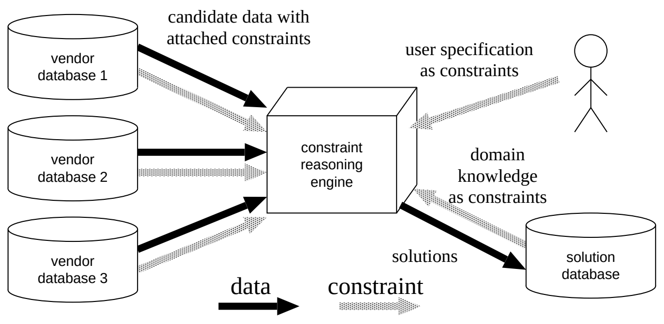
A configuration problem is a design activity in which an artifact is assembled from a set of components by connecting them in certain ways. Many early configurators, like R1/XCON (McDermott, 1982), are rule-based systems in which domain and strategic knowledge are tightly coupled. This makes them relatively expensive to maintain, especially for large and complex problem domains with a high rate of change of knowledge.

Our approach is to represent the configuration problem as a constraint satisfaction problem (CSP) but to bring the constraints together into one place for solving. Constraint solving provides a domain-independent framework for the representation of configuration problems by declarative knowledge which is relatively cheaper to maintain (Sabin and Freuder, 1996; Mailharro, 1998). The principle is to find values for problem variables subject to constraints that restrict which combinations of values are allowed (Sabin and Freuder, 1996).

The configuration problem in the KRAFT environment has the following characteristics:

- Candidate data, which form the initial solution space, are distributed in different resources, according to different schemas.
- Domain knowledge in the form of constraints come from different sources, expressed in different but related ontologies.
- Constraint knowledge is mobile. It can be attached to data objects or move freely within the network.
- Constraint knowledge can be reused by transforming it to fit another ontology. When a piece of constraint is attached to a data object, it must be utilised whenever that data object is used.

These characteristics have some important implications. When a resource joins the network, it must be incorporated automatically, which includes using both the stored data and constraints. This dynamic environment, together with mobile constraints which can attach to data objects, make the problem specification dynamic, since it may change as different candidate data objects become involved. Therefore, it is difficult for us to compose it into a static database query. The problem is also data-intensive. Thus feeding all candidate data into a single problem solver may create the problem of memory overflow, and should be avoided.



candidate data with attached constraints
vendor database 1
vendor database 2
vendor database 3
user specification as constraints
constraint reasoning engine
domain knowledge as constraints
solutions
solution database
data
constraint

Figure 1.2. KRAFT utilises both data and constraints which are mobile in the KRAFT domain. Constraint knowledge flow is shown as grey arrows while data flow is in black.

2. Modelling the Configuration Task in KRAFT

KRAFT uses the constraint formalism to model both user specifications and domain knowledge on component compatibility, thus providing a declarative and uniform representation of both restrictions and specifications of the problem specification. A declarative constraint (Gray et al., 1999a) is a self-contained mobile knowledge object which can be extracted and transported in a distributed environment. Internally, selection information in a declarative specification can be moved within a computation. These promising features liberate KRAFT from a fixed execution plan and enable the system to explore different problem-solving strategies as problem tasks can be transformed, delivered and processed in capable problem-solving components.

Figure 1.2 shows the KRAFT system from the perspective of constraint and data flow.

We store components as data instances in different *vendor databases*, together with their attached knowledge in the form of constraints. These component instances define the domains of variables in the CSP. Other constraints come from an otherwise empty *solution database*, which we will discuss in section 2.1, and also the *user*. As different vendors may have their local domain model, constraints from different resources may be expressed in different vocabularies and against different schemas. As we will see in section 3, the KRAFT architecture is flexible enough to cope with heterogeneous resources but to simplify our problem, we assume the use of a uniform *integration schema* within the KRAFT domain. Constraints and data expressed against local schemas will be transformed and mapped into the *integration schema*.

Mittal and Frayman (Frayman and Mittal, 1987; Mittal and Frayman, 1989) presented a generic domain-independent model of configuration tasks where each component is described by a set of *attributes* and *connection ports*. Sabin and Freuder (Sabin and Freuder, 1996) further proposed the constraint-based framework of *composite CSP*. In a composite CSP, variables are not restricted to take atomic values but also an entire subproblem. Thus instantiating variables may change the CSP dynamically. Instead, we model a restricted configuration task where the set of variables and their domains are fixed at the time of problem composition. However, we still allow constraints to be dynamically added as the solving process proceeds.

2.1. Database Integrity Constraints as CSP Specifications

To specify a CSP by database integrity constraints, we visualise a *solution database* which is empty and yet to be populated by the solutions of a CSP, after it is solved. As all integrity constraints are satisfied in a semantically consistent database, we can restrict the combination of

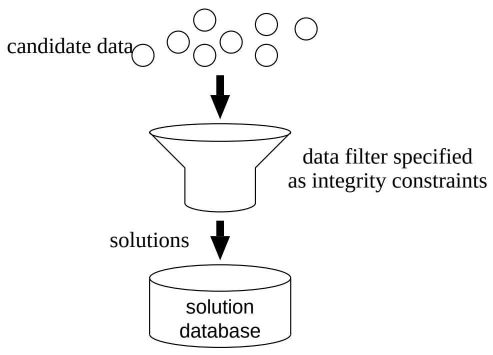


Figure 1.3. Integrity constraints on the solution database act as a data filter that controls which candidate data will become a solution.

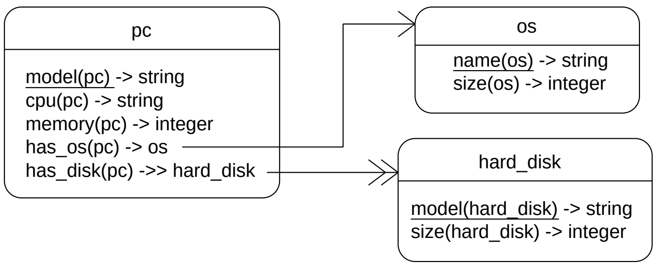


Figure 1.4. Our example solution database schema of configured PCs.

values which can be stored and qualified as solutions to the CSP by imposing integrity constraints against the *solution database* schema (figure 1.3).

Figure 1.4 shows an example solution database schema that stores all properly configured PCs. The requirement of having only *"pentium2"* CPU is expressed as the following integrity constraint:

$$(\forall p, c) \quad pc(p) \wedge cpu(p, c) \longrightarrow c = \text{"}pentium2\text{"}$$

In our prototype system, we use the P/FDM database system (Embury, 1995) that is based on Shipman's functional data model (Shipman, 1981). P/FDM uses the constraint language CoLan (Bassiliades and Gray, 1994) and the same constraint can be expressed as:

```
constrain all p in pc
    to have cpu(p)="pentium2"
```

Compatibility between components can also be expressed as integrity constraints. The following constraint specifies that an operating system (OS) must be able to fit into one of the installed hard disks in a properly configured PC:

$$(\forall p, o, so)\left(\begin{array}{c} pc(p)\wedge \\ has\_os(p,o)\wedge \\ size(o,so) \end{array}\right) \longrightarrow$$

$$(\exists d, sd)\left(\begin{array}{c} has\_disk(p,d)\wedge \\ size(d,sd)\wedge \\ sd \geq so \end{array}\right)$$

Or in CoLan:

```
constrain all p in pc
    all o in has_os(p)
so that some d in has_disk(p)
    has size(d) >= size(o)
```

Thus the *solution database* provides a framework for CSP specification. However, in most cases, only the schema of the *solution database* exists and no value is actually being stored. Instead, solution values are returned to the user through the user-agent.

2.2. Database Integrity Constraints as Mobile Knowledge

Database integrity constraints in P/FDM are quantified constraints that apply to a set of data objects. When expressed against a “KRAFT domain-wide” *integration schema*, these constraints are self-contained abstract objects which can be used to represent domain-specific knowledge, partially solved solutions and intermediate results. Effectively, they carry otherwise hidden operational semantics along with the data. This is vital for its proper use in e-commerce. When a data object is retrieved from a database and migrated into a network, its attached constraints must also be extracted and mobilised to retain its annotated instructions. In other words, data objects are annotated with declarative instructions on how these objects should be used.

Suppose a manufacturer produces tailor-made OS for the *“Pentium III”* platform only. So he puts a universally quantified constraint on all OS in his product database so that any PC using this OS must use a *“Pentium III”*. This constraint is quantified over all OSes:

$$(\forall o, p, c) \quad \left(\begin{array}{c} os(o)\wedge \\ has\_os(p,o)\wedge \\ cpu(p,c) \end{array}\right) \longrightarrow c = \text{"}pentium3\text{"}$$

The same constraint can be expressed in Colan:

```
constrain each o in os
  to have cpu(has_os_inv(o))="pentium3"
```

The function `has_os_inv` is the inverse function of `has_os`, which traverses the relationship in the reverse direction. Inverse functions in P/FDM are discussed in (Embury, 1995).

By using an optional filter, we can selectively apply the constraint to a reduced set of data instances instead of all objects of that class. This allows constraint knowledge to be attached as if to an individual data object. For example, we can apply a constraint to the set of OSes with the name of `"winNT"` only:

$$(\forall o, n, p, m) \quad \left(\begin{array}{c} os(o) \wedge \\ name(o, n) \wedge \\ n = \text{"}winNT\text{"} \wedge \\ has\_os(p, o) \wedge \\ memory(p, m) \end{array} \right) \longrightarrow m \geq 64$$

The same constraint can be expressed in Colan:

```
constrain all o in os
    such that name(o)="winNT"
to have memory(has_os_inv(o))>=64
```

This is an example of a *conditional constraint* which only applies when a certain *guarding condition* is true. In this example, the *guarding condition* is the boolean test *"$name(o) = \text{"}winNT\text{"}$"*. Once again, we use the inverse function `has_os_inv` to navigate from the class `os` to `pc`.

Traditionally, database integrity constraints are used for validation checks on populated data and define the acceptable states of the database which all stored data have to satisfy. In using database integrity constraints as CSP specifications, we extend the use of integrity constraints to include un-populated entity classes. Thus the manufacturers and designers are putting constraints on objects which will form relationships with the components but are not yet connected! In this way, integrity constraints specified against a *solution database* work on data objects which are yet to be created, as well as existing data in the database. We call these unpopulated entity classes *empty-slots*, as they represent objects which will be plugged into the configuration to form a workable design. These *empty-slots* cannot be filled by just any value, instead, we restrict the allowed values by the attached constraints. Thus an integrity constraint does not only restrict the stored value in a database record, but also any potential value that will be found for an *empty-slot*.

2.3. Categorising Constraints

The easiest way to categorise distributed constraints from various sources is to classify them according to their origin.

- **Small-Print Constraints**

  These constraints are stored in a database in association with class descriptors for data objects, and they can be viewed as an attachment of instructions on how a data object should be used. We call these *small-print constraints* (Gray et al., 1999b), representing *small-print* conditions in a contract or footnotes in a catalogue. Note that constraints are actually stored with a class descriptor but selection conditions in the constraint can be used to make it specific to a particular object, as discussed in section 2.2.

- **Design Constraints**

  When constraints are used to represent specification knowledge in the form of design constraints, they capture expert knowledge about feasible designs. We store such constraints in the *solution database*. Although initially empty of data, the *solution database* provides a framework for specifying and integrating the problem-solving knowledge, through its attached constraint metadata.

- **User Requirement Constraints**

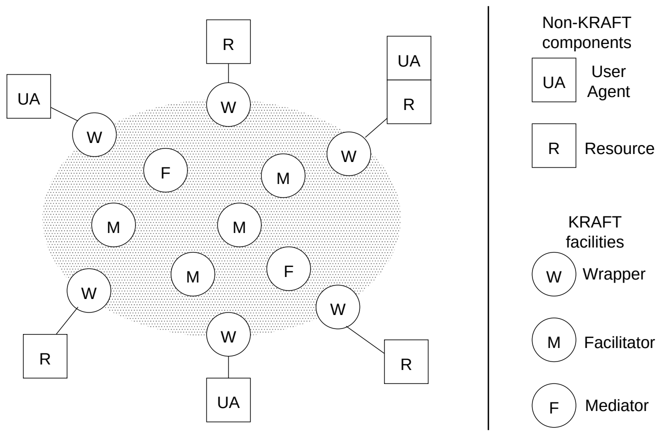


Figure 1.5. This figure shows a conceptual view of the KRAFT architecture. KRAFT components are round in shape while non-KRAFT ones are marked as squares. The grey area represents the KRAFT domain where a uniform language and communication protocol is respected.

> Constraints can also be used to represent user specifications for the required solutions. Like other resources in the system, the user serves as another information source, feeding knowledge into the system in the form of constraints.

Categorising constraints according to their different sources and origins is a natural classification but it does not explain why some constraints behave differently from others. A closer examination reveals that the difference in behaviour comes from their different scopes of application, as they are attached to objects on different abstraction levels.

A *small-print constraint* forms part of the data object to which it is attached. Therefore, it applies to all application problems and problem instances that utilise such data. In other words, it is specific to the attached data object but generic to all application problems and problem instances that use the data.

Design constraints capture specifications of an application problem. They can be viewed as attached to a particular problem, and thus apply to different instances of the same problem. When a class of problems shares some common *design constraints*, these constraints may be parametrised to capture the generality, and so have to be specialised before being applied to a specific design problem.

User requirement constraints are attached to a problem instance. As a result, they are specific to a particular problem instance and may differ between different sessions.

This alternative classification focuses on “where a constraint applies” instead of “where a constraint comes from”, as knowing when to satisfy a constraint is more important than knowing its origin. As a result, we can have a constraint that comes from the user but is attached to particular data objects, thus behaving as a *small-print constraint*.

3. The KRAFT Architecture

KRAFT has an agent-based architecture where knowledge processing components are realised as software agents. The basic philosophy of the architecture design is to define a KRAFT domain where certain communication protocols and languages must be respected, as shown in figure 1.5. *KRAFT facilities* forming part of the KRAFT architecture must conform to these protocols and languages, and must also provide sharable and asynchronous access to the services they provide.

Three important KRAFT facilities of distinctive roles have been identified:

- **Wrapper**
  Wrappers interface non-KRAFT components to the KRAFT network by providing translation services between the internal data formats of user agents and resources and the data format used within the KRAFT domain. They also provide the high-level communication mechanisms needed to link the user agents and resources to the internal facilities of the KRAFT domain.
- **Facilitator**
  Facilitators maintain directories of KRAFT facilities, their locations and what services they provide, and also details of their availability, load and reliability. Their principal function is to accept messages from other KRAFT facilities and route them appropriately.
- **Mediator**
  Mediators are KRAFT components that can utilise domain knowledge to transform data in order to increase their information content. They also operate to implement particular tasks, although a task can be highly specific to a particular application domain or it can be domain-independent.

The architecture in figure 1.5 also shows other non-KRAFT components which are linked to the KRAFT network via *wrappers*:

- **User Agent**
  Users access the services of the KRAFT domain via *user agents*. *User agents* are considered external to the KRAFT domain and are connected to the network via *wrappers*.
- **Resource**
  Resources include information sources such as databases, knowledge bases and also processing engines like constraint solvers. Similar to *user agents*, *resources* are independent of the KRAFT domain and are connected to the network via *wrappers*.

The design of KRAFT is consistent with several emerging agent standards, notably KQML (Finin and et al., 1993; Finin et al., 1994) and FIPA (Chiariglione, 1998). Agents are peers; any agent can communicate with any other agent with which it is acquainted. Agents become acquainted by registering their identity, network location, and an advertisement of their knowledge-processing capabilities with a facilitator.

KRAFT agents communicate by sending messages to other peer agents using a nested protocol stack. The outermost shell of a KRAFT message is a message header that encapsulates the body of the message with low-level time-stamp and network information. Wrapped by this header is the body of the message which in turn consists of two nested protocols. The outer protocol is the agent communication language CCQL (Constraint Command and Query Language) and nested within the CCQL message is its content, expressed in the CIF language (Constraint Interchange Format).

CCQL describes the “speech act” of the message and is defined by a set of performatives based upon a subset of KQML (Knowledge Query and Manipulation Language) (Finin and et al., 1993; Finin et al., 1994). A more detailed discussion of KQML performatives can be found in (Labrou, 1996).

The CIF language is based on CoLan (Bassiliades and Gray, 1994) which is a high-level declarative constraint description language for use with the object-oriented database P/FDM (Embury, 1995). CoLan has features of both first-order logic and functional programming and is based on Daplex (Shipman, 1981) query language.

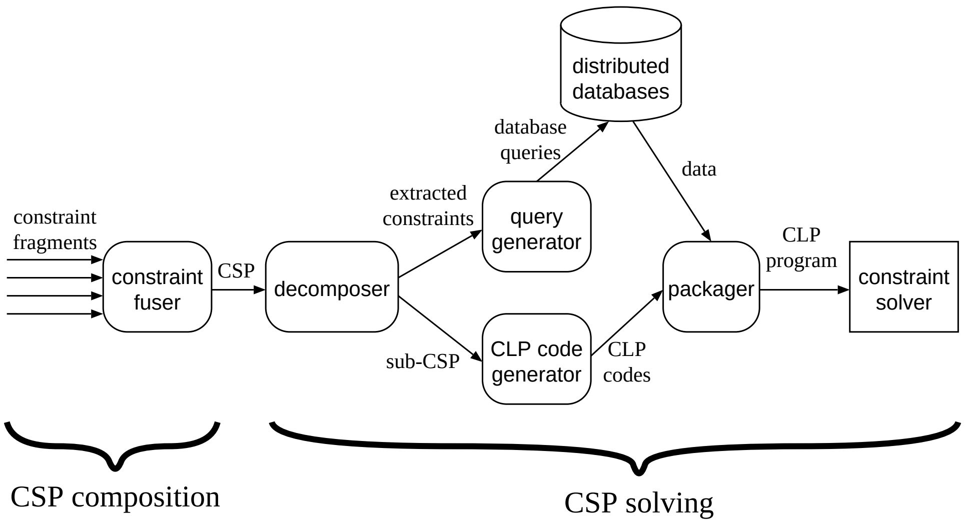


Figure 1.6. The KRAFT problem-solving process is divided into two phases: CSP composition and CSP solving.

4. CSP Composition

Problem solving in KRAFT is divided into two stages (figure 1.6). In the first stage, distributed constraints are fused to compose a concrete description of the overall CSP. In the second stage, the composed CSP is analysed and decomposed into sub-queries which are solved by multiple problem solving components in the system.

The CSP composition process can be further divided into three stages:

- **Constraint Extraction and Transformation**

  Constraint knowledge in KRAFT comes from different sources. Despite their different origins, these constraints must be exported from their respective sources before they can be utilised. They must also be transformed to resolve any semantic mismatch between heterogeneous resources.

- **Constraint Fusion**

  Constraint fusion is a necessary step in solving distributed CSPs as each constraint fragment is only a partial description of the whole problem. It is only by fusing the constraint pieces together that the overall problem description is revealed. While a single piece of constraint may not contain enough information to solve a CSP, we hope that by combining constraints together, their total value is enhanced, and thus making the problem solvable.

- **CSP Formation from Integrity Constraints**

  KRAFT constraints are expressed in the form of database integrity constraints which are meant to restrict the combination of existing data stored in a database rather than creating value combinations that satisfy certain requirements. Therefore, these integrity constraints cannot be used directly as a CSP specification but have to be transformed before they can be compiled into an executable program that searches for solutions.

```
with common p in pc
  rewrite simm(has_mother_board(p)) +
          sdram(has_mother_board(p))
  into memory(p);

with common p in pc
  rewrite os_name(p)
  into name(has_os(p));
```

Figure 1.7. Example rewrite rules.

4.1. Constraint Extraction and Transformation

From the viewpoint of constraint extraction, there are two main categories of constraint knowledge in KRAFT. The first type of constraints, like user specification constraints, are actively fed into the system and do not require any extraction. The second type of constraints are stored in resources and have to be extracted before they become mobile and move into the network. Examples are *designer constraints* stored in the solution database[2] and *small-print constraints* in vendor databases.

To achieve the required mobility of constraint knowledge, it is necessary for resources to support meta-level queries that retrieve stored constraint information instead of data. A resource which does not support constraint extraction confines constraint knowledge within its domain and forces localised constraint utilisation, thus restraining the system from composing a global execution plan.

Our prototype uses the P/FDM database system (Embury, 1995) which allows the retrieval of meta data through queries on the meta-schema. P/FDM provides a uniform access interface where meta-data in the database are retrieved like normal data by the Daplex language. To retrieve meta-data, we simply express a query against the meta-schema instead of the data schema.

Before fusion can take place it is also necessary to ensure that the constraints to be combined all have the same terms of reference. This can be achieved by rewriting each constraint to refer to an *integration schema*. In KRAFT, this rewriting is done automatically by *wrappers*. Each local database in the KRAFT network has a wrapper which can apply declarative *rewrite rules* to constraints expressed against the local schema to give a transformed constraint expressed against the integration schema.

Rewrite rules are represented internally in P/FDM as functional expressions implemented as Prolog term structures. Rewriting is done by pattern matching to find whether the left hand side of any rewrite rule is present as a sub-expression within the constraint. If a match is found, the sub-expression is replaced by the right hand side of the rewrite rule with appropriate variable substitutions. Figure 1.7 shows two examples of rewrite rules. The effect of applying these rewrite rules to a constraint is shown in figure 1.8.

The rewrite rule is a powerful mechanism that maps constraints from one schema into another. However, moving a constraint from the local schema into the integration schema may not be just a simple operation of replacing sub-expressions in a constraint. When a constraint is moved

[2]As we saw in section 2.1, the solution database may not physically exist. In this case, designer constraints may be readily stored as application-specific knowledge in the user-agent.

```
constrain each p in pc
    such that os_name(p) = "winNT"
to have simm(has_mother_board(p)) +
        sdram(has_mother_board(p))>=32;

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

Figure 1.8. Example constraints. The first constraint is expressed against a local schema. The second constraint shows the result of transforming this constraint to refer to the integration schema.

from a local resource into the network, the domain of quantification changes. Thus a constraint which is true in a local resource may not remain true when it migrates out of that resource. For example, a specific vendor HAL may have “Linux” installed in all its PCs. So the following constraint is universally true in its local database:

```
constrain each p in pc
  to have os_name(p)="Linux"
```

When this constraint is extracted from the local resource, transformed and moved into the network, it fails to remain universally true as other vendors may not have the same requirement on their PCs. To utilise this constraint, we have to add an extra restriction to limit the set of quantified values:

```
constrain each p in pc
    such that manufacturer(p)="HAL"
to have name(has_os(p))="Linux"
```

In general, when a universally quantified constraint is moved from a local resource into the unified space, we must add an extra condition to restrict the domain of the quantified variable so that its set of values remains the same as it was in the local resource. In the current implementation, wrappers provide the required knowledge and mechanism to automate this tagging process. This is not easily scalable and will be the subject of future work.

4.2. Constraint Fusion

Declarative constraints stored as self-contained knowledge objects in a distributed system form a shared library of building blocks which can be retrieved, transformed and combined. The key to reusing and sharing this knowledge is the process of *constraint fusion*, which dynamically combines their semantic content to compose problem specification instances. The software mediator performing this task of constraint fusion provides a commonly used service to higher-level applications by adding value to individual knowledge fragments. This is also a crucial process which provides the required scalability and flexibility where new resources can join a distributed system by bringing in new knowledge dynamically.

Semantically, constraint fusion is the logical conjunction of constraints. When quantified constraints are conjoined together, they undergo a kind of information exchange which adds

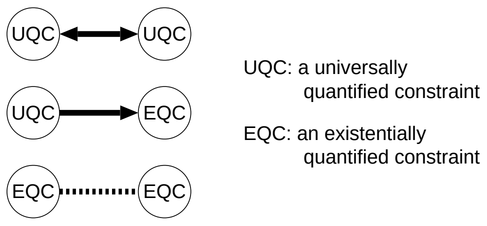

UQC
UQC
UQC
EQC
EQC
EQC
UQC: a universally quantified constraint
EQC: an existentially quantified constraint

Figure 1.9. This diagram summarises the behaviour when different quantifiers are fused together. The solid arrow shows the potential constraint information flow when a universally quantified constraints is conjoined with another quantified constraint. An existentially quantified constraint does not have this tendency. Thus fusing a UQC with an EQC results in a single-direction arrow while fusing two EQCs have no potential information exchange, as shown by the dotted line.

value to individual constraint fragments by enhancing the semantics of each other. The behaviour of conjoining two quantified constraints depends on their quantifiers. The consequence of constraint conjunction originates from the *universal quantifier* which has the tendency of imposing constraints to all potential variables, when the condition allows. The *existential quantifier*, however, does not have this tendency. When a *universally quantified constraint* takes part in a constraint fusion process, there is a potential exchange of constraint information where the combined constraints can be enriched. Thus the presence of a *universally quantifier* is a necessary condition for constraint fusion to take place. Figure 1.9 summarises the behaviour of different quantifier combinations when two quantified constraints are conjoined together.

Operationally, constraint fusion is the identification of correspondences between variables in different constraint fragments, which allow potential constraint information flow between them. We call these correspondences *variable-links*. *Variables-links* are identified by examining how variables are 'generated', which can be easily illustrated by the following two constraints:

$$\begin{array}{l}(\forall p_1, m_1) \quad pc(p_1) \wedge memory(p_1, m_1) \longrightarrow m_1 \geq 32 \\ (\forall p_2, m_2) \quad pc(p_2) \wedge memory(p_2, m_2) \longrightarrow m_2 \leq 1024\end{array}$$

By comparing the predicates on the left-hand-side of the implication, we can identify the correspondences between variables p_1-p_2 and m_1-m_2. The two constraints can then be combined into one:

$$(\forall p, m) \quad \begin{pmatrix} pc(p)\wedge \\ memory(p, m) \end{pmatrix} \longrightarrow \begin{pmatrix} m \geq 32\wedge \\ m \leq 1024 \end{pmatrix}$$

More complicated situations may arise when constraints are fused. A possible result is a *conditional constraint* that applies only when a *guarding condition* is satisfied, as illustrated by the following example:

$$\begin{array}{ll}(\forall p_1, m_1) & \begin{pmatrix} pc(p_1)\wedge \\ memory(p_1, m_1) \end{pmatrix} \longrightarrow m_1 \geq 32 \\ (\forall p_2, m_2, c) & \begin{pmatrix} pc(p_2)\wedge \\ cpu(p_2, c)\wedge \\ c = "pentium2" \wedge \\ memory(p_2, m_2) \end{pmatrix} \longrightarrow m_2 \geq 64\end{array}$$

Fusing them results in a *conditional constraint* where an extra restriction is imposed when the *cpu* of a *PC* is a *"pentium2"*:

```
constrain each p in pc
  to have cpu(p)="pentium2"
  and name(has_os(p)) <> "win95"

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

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

Figure 1.10. Three example constraints representing a user requirement, a designer constraint and a small-print constraint.

$$(\forall p, m, c) \quad \left(\begin{array}{c} pc(p) \wedge \\ memory(p,m) \end{array} \right) \longrightarrow$$
$$\left(\begin{array}{c} m \geq 32 \wedge \\ \left(\left(\begin{array}{c} cpu(p,c) \wedge \\ c = \text{"}pentium2\text{"} \end{array} \right) \longrightarrow m \geq 64 \right) \end{array} \right)$$

Once all *variable-links* are identified between two constraints, there are two approaches to fuse them: *implicit* and *explicit*. The *implicit* constraint fusing approach just combines constraints by maintaining the identified *variable-links*, which can be easily implemented by the unification mechanism in a constraint logic programming (CLP) system. When two variables are unified together, constraints imposed on one variable will propagate to the other variable through the CLP system and restrict its value domain accordingly, and vice versa. The *implicit fusion approach* does not transform any constraint, and so it does not suffer from any limitation of the constraint representation language.

In an *explicit* constraint fusing approach, by contrast, variable-link information is used to transform and manipulate constraint expressions to give a concrete representation of the fused constraint. In general, this includes transforming natural language sentences, rewriting first-order predicate calculus expressions, combining CIF expressions represented as Prolog term structures and conjoining goals in a constraint logic program. The fused constraint is then compiled into CLP program code and sent to the constraint solver. A major drawback in the *explicit* approach is that the fused constraint is expressed in a target language and thus the power of the approach depends on the expressiveness of the language in representing different fusing situations.

The two approaches of *implicit* and *explicit* constraint fusion are not mutually exclusive. Instead, a *hybrid* approach is a more appealing solution for fusing constraints. A more detailed discussion of constraint fusion is given in (Hui, 2000).

Figure 1.10 shows three constraints representing a user requirement, a designer constraint and a small-print constraint. When we fuse these constraints together, we get the constraint in figure 1.11 describing the overall restriction on the *solution database*.

```
constrain each p in pc
  to have cpu(p)="pentium2"
  and name(has_os(p))<>"win95"
  and size(has_os(p))=<size(has_disk(p))
  and if name(has_os(p))="winNT"
      then memory(p)>=32
      else true
```

Figure 1.11. The result of fusing the three constraints in figure 1.10.

4.3. CSP Formation from Database Integrity Constraints

Fusing distributed constraints reveals a complete picture of the desired states of the *solution database*. However, we cannot directly compile them into an executable program to find the solution values, as they are meant to restrict the value combination of existing data stored in a database, instead of creating new value combinations that satisfy certain requirements. In particular, they contain references to unpopulated values and relationships, called *empty-slots*, as discussed in section 2.1 and 2.2.

Empty-slots represent relationship instances which are to be created from the solutions after the CSP is solved. The *empty-slot* problem arises because we are moving a constraint expressed like an integrity constraint from a database where some slots are unpopulated, into the context of the *solution database*, where the slots are assumed to be populated. A database integrity constraint that references such an *empty-slot* always trivially succeeds or fails[3] because there are no stored instances that can satisfy the slot predicate. Similarly, a database query that tries to retrieve any value from an *empty-slot* always gets nothing. Instead of compiling database integrity constraints into an executable program, we have to transform them into a CSP, which when solved, gives value combinations that satisfy the original integrity constraints on the solution database.

The transformation from an integrity constraint into a CSP, however, is surprisingly simple. We will use the following integrity constraint in first-order logic form for explanatory purpose:

$$(\forall p, o, n) \quad \left(\begin{array}{c} pc(p)\wedge \\ os(o)\wedge \\ has\_os(p,o)\wedge \\ name(o,n) \end{array} \right) \longrightarrow n \neq \text{"}win95\text{"}$$

Once the *solution database* is populated, the relation `has_os(p,o)` will relate a `PC` to its installed `OS`. Under this situation, the stored instances of `has_os(p,o)` in the database define and restrict the valid combination of `p` and `o`.

Now if we go back to the problem of constructing a CSP to find the valid combination of `p` and `o`, `has_os(p,o)` puts no restriction on `p` and `o` as there is no stored value. Instead, restrictions on the `PC-OS` combination come from constraints on other attributes. Thus `has_os(p,o)` is actually redundant in the context of the *solution database* as it is subsumed by the other selection conditions. An easy way of transforming a set of database integrity constraints into a CSP,

[3]A weak translation of the implication in a universally quantified constraint makes it trivially succeed or fail, depending on whether the reference to an empty-slot is on the 'left-hand-side' or 'right-hand-side' of the implication. An existentially quantified constraint referencing an empty-slot always fails.

therefore, is to take out all the references to *empty-slots*, meaning that the *empty-slots* no longer put any restriction on the involved variables. In this way, we are effectively representing the value domain of `has_os(p,o)` by the Cartesian product of the domains of `p` and `o` which provides the initial finite domains for the variables in the constraint solver. Any value combination that satisfies these constraints with empty-slot references removed is a solution. In our example, we get the following CSP by taking out the reference to `has_os(p,o)`:

$$(\forall p, o, n) \quad \left(\begin{array}{c} pc(p) \wedge \\ os(o) \wedge \\ name(o, n) \end{array} \right) \longrightarrow n \neq \text{"}win95\text{"}$$

Any `p`, `o` and `n` in the solution database will have to satisfy this constraint. From a constraint-solving point of view, it means: *"any* `PC` *and* `OS` *combination is valid if the name of the* `OS` *is not* `"win95"`".

The identification of *empty-slots* (i.e. unpopulated relationships) plays an important role in composing a CSP from database integrity constraints. This piece of meta-knowledge is best supplied by the KRAFT programmer who also provides the application specific *design constraints*.

It is also important to emphasize that the *empty-slots* meta-data is not discarded after the CSP is composed but saved for later use, as we have to keep the association between variables in an *empty-slot*.

5. CSP Solving

Once a CSP is composed from distributed constraints, it is analysed and decomposed into sub-problems. The decomposition step is not a simple reverse process of constraint fusion. Depending on the current status of the system and availability of different resources, different execution plans are derived. Constraints are fused in the first place because we want to find the best way to split the problem and divide labour.

In our prototype system, we chose to decompose a CSP into distributed database queries and a reduced sub-CSP. Database queries are sent to databases to retrieve data values for the formation of variable domains in the CSP, while the reduced sub-CSP is compiled into CLP code. We use the ECLiPSe CLP system (ecl, b; ecl, a) as it supports flexible code generation as in LP systems but being more efficient in execution. The generated CLP code and variable domain information are then sent together to the constraint solver for execution, which either finds the solution(s) to the CSP or detects a conflict.

CSP solving in KRAFT involves four stages:

- **Database query formation**

  Constraint information is extracted from the CSP specification to compose database queries. This early filtering technique helps to reduce the amount of data that need to be transported to form the initial solution space.

- **Variable domain formation**

  Database queries composed are used to retrieve candidate data values and form the initial variable domains.

- **Constraint posting**

  After extracting constraint information to form database queries, the remaining CSP specification is compiled into CLP program code, which reasons about the domain variables.

- **Variable labelling**

The final stage of variable labelling instantiates domain variables such that all required constraints are satisfied.

5.1. Database Query Formation from the CSP

Databases in the KRAFT environment have a limited CSP solving capability in the form of database query answering. By extracting constraint information from the CSP specification to compose database queries, we delegate part of the CSP solving process to the involved databases. In other words, CSP solving is distributed. Database query formation from a CSP will promote early data filtering, thus reducing the amount of candidate data transported from databases into the constraint solving components.

Simply speaking, we can compose database queries by extracting constraint information that unconditionally applies. Although their constraint reasoning capability may vary, most databases support the use of a filter to be applied uniformly. The conversion of such filters into a database query is straightforward, for example, by generating a `WHERE` clause in SQL. In the case of a constraint that applies conditionally, like `memory(p)>=32` in figure 1.11, we can use a technique that transforms a conditional constraint into several separate database queries with their own data filters. However, transforming a constraint with a complex *guarding condition* into multiple queries will be complicated and difficult, especially when the condition may involve nested quantifications. To solve this problem, we only compose database queries by extracting constraint information that always applies to the solutions. Conditional constraints that remain in the CSP will then be compiled into CLP program code and handled by the constraint solver.

To complete our discussion, an *existentially quantified constraint* does not give us enough information to construct any database query, as it does not require all data to satisfy the constraint. Now our strategy in forming database queries from a CSP specification is simple:

Every universally quantified constraint that un-conditionally applies to a set of entity classes is composed into a database query. Other constraints remain in the CSP specification and will be solved by constraint solvers.

5.2. Variable Domain Population

Database queries composed from the CSP are used to retrieve candidate data and form the initial solution space.

As a CLP program reasons over CLP data structures, we have to compile the retrieved data into CLP data structures before they can be used to populate the domains of variables in the CSP.

In the functional data model, attributes of an entity are modelled as functions on a data object, which is identified by a unique *object identifier*. Scalar attributes are modelled as scalar functions while relationships between objects are modelled as non-scalar functions that return the *object identifiers* of the related data objects. We represent the relationship between the object and each attribute by a separate constraint. The following example shows how the `object/2` and `fnval/5` (meaning '*function value'*) term structures are used to represent data objects, the single-valued attributes `model`, `cpu`, `memory` and multi-valued attribute `has_disk` of the object `pc1`:

```
object(pc,pc1).
fnval(model,[pc],[pc1],string,
      'P5-120').
fnval(cpu,[pc],[pc1],string,
      pentium).
```

```
fnval(memory,[pc],[pc1],integer,32).
fnval(has_disk,[pc],[pc1],hard_disk,
      disk1).
fnval(has_disk,[pc],[pc1],hard_disk,
      disk2).
```

This approach offers a uniform representation across different entity classes and attributes by modelling the relationship between the input arguments and output value of a function. Type information is included to make it self-describing and to discriminate between overloaded functions. In many logic programming systems, say Prolog, these facts can only be used as passive tests against instantiating values. That means the constraint information they contain are only utilised by instantiated variables. Variable instantiation, however, is a strong commitment which costs a lot to be undone when proved to be wrong. Instead of selecting a value prematurely, ECLiPSe supports the use of such a user-defined predicate (e.g. `fnval/5`) as an active constraint by *generalised constraint propagation* (Provost and Wallace, 1991), which incorporate the given constraint information by reducing the variable domain but without instantiating it. Given the above representation of PC objects, the following ECLiPSe goals set up the constraints between the domain variables `Pc`, `Model`, `Cpu`, `Memory` and `Disk`:

```
object(pc,Pc) infers most,
fnval(model,[pc],[Pc],string,Model)
      infers most,
fnval(cpu,[pc],[Pc],string,Cpu)
      infers most,
fnval(memory,[pc],[Pc],integer,Memory)
      infers most,
fnval(has_disk,[pc],[Pc],hard_disk,Disk)
      infers most,
```

Now the reason of applying data filters in composing the initial variable domains becomes obvious. As data objects are mapped and stored as Prolog facts in the CLP system, a big data set will put a high demand on the tuple space of the CLP system, whose ability lies in constraint reasoning instead of data management.

5.3. Constraint Solving

With the support of the runtime library, our CLP code generator systematically compiles CIF constraints into ECLiPSe code. The generated program has a top level predicate `solve/1` calling three other subgoals, which resemble the three stages of *variable declaration*, *constraint posting* and *variable labelling* in CLP (Frühwirth et al., 1993; Wallace, 1998). Information is communicated by a shared variable:

```
solve(Shared) :-
    declare_vars(Shared),
    post_constraints(Shared),
    label_vars(Shared).
```

Figure 1.12 shows the structure of the generated ECLiPSe program. The `post_constraints/1` clause is the entrance to constraint posting, which calls a conjunction of `post_constraint/2`

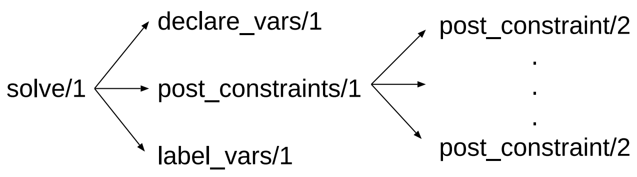


Figure 1.12. The structure of our generated ECLiPSe program.

goals, where each of them is a result of a single quantified constraint in the original CIF expressions.

Posting constraint alone usually cannot obtain a solution to the CSP but removes invalid values from the variable domains. *Labelling* is the final stage of CSP solving where variables are instantiated to values in their respective domains to reach a consistent constraint network.

We use the ECLiPSe `indomain/1` predicate to instantiate a list of finite domain variables. When variables are gradually instantiated, delayed constraints are awakened and backtracking may occur. Once enough information is available for the evaluation of their guarding conditions, Their *guarding conditions* can now be evaluated to determine whether the constraints are applicable.

6. Related Work

KRAFT employs an agent-based architecture which is proving to be an effective approach to developing distributed information systems, as it provides the required extensibility and adaptability by supporting rich knowledge representations, meta-level reasoning about the content of online resources, and an open environment in which resources can join or leave a network dynamically. Early projects like PACT (Cutkosky et al., 1993) and SHADE (Kuokka et al., 1994) have already shown that agent technology can the support exchange of rich business information using the Knowledge Interchange Format (KIF) (Genesereth and Fikes, 1992). The ADEPT project further shows the flexibility of an agent-based system in supporting agile organisations, with an emphasis on the dynamic management of workflow between partner organisations (Jennings et al., 1996). 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 KRAFT architecture shares similarities with other agent-based distributed information systems, in particular, the InfoSleuth project (Bayardo et al., 1997; Nodine et al., 1998). Architecturally, both systems comprise a network of cooperating agents. Scalability is provided by match-making agents, like broker-agents or facilitator, which associates agents with resources at runtime. The roles identified for KRAFT agents are also similar to those in InfoSleuth. However, the major difference lies in KRAFT's emphasis on the use of both constraints and data, while InfoSleuth is primarily concerned with data retrieval. In its emphasis on constraints, KRAFT is similar to the Xerox Constraint Based Brokers project (Andreoli et al., 1995). However, KRAFT recognises the need to transform constraints when they are extracted from local resources.

KRAFT also builds upon the work of the Knowledge Sharing Effort (KSE) (Fikes et al., 1991; Neches et al., 1991; Patil et al., 1992), in that some of the facilitation and brokerage methods are employed, along with a subset of the 1997 KQML specification (Labrou, 1996). However, unlike the KSE work which attempted to support agents communicating many diverse forms of knowledge, KRAFT takes the view that constraints are a good compromise between expressivity and tractability.

The Smart Clients project (Arnal and Faltings, 1999) 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 (Reeves et al., 1999) is similar in concept to KRAFT's use of small-print constraints. The difference is that this work uses a rule-based formalism to specify contractual *fine print* in the form of business rules. Logic program techniques are then used to reason with the rules.

7. Conclusions

A crucial insight in KRAFT is that quantified constraints, expressed in a sub-language of first-order logic against a shared data model that is free to evolve, provide an excellent way of transporting semantics along with data. Thus we recognise the fact that constraints have evolved from database states restrictors to a kind of portable knowledge that can be exported and processed. We use constraints to capture domain knowledge, which is distributed among different resources. These distributed knowledge fragments are combined to give *added value* by a process called *knowledge fusion*.

Once we have the semantic knowledge in this form, remote programs can reuse it very flexibly. We have developed an extensible problem solving approach that dynamically composes a problem specification by fusing reusable blocks of constraint knowledge. Our constraint fusion algorithm puts no restriction on the constraints, except that they must be expressible in the CIF language.

Our idea of decomposing the fused CSP into sub-problems is an open and flexible approach that allows different problem solving methods to be used. We fuse constraints in order to determine a better way to solve them by combining different problem solving paradigms. The decomposition process is based on a simple heuristic of minimising retrieved data sets and it adapts to problem instances by analysing the CSP at runtime. The current database query formation algorithm is a simple one but more sophisticated strategies can be used. Similarly, database queries and CLP code are generated at runtime for greater flexibility.

KRAFT employs an agent architecture which makes it very suitable to support virtual organisations. The use of this open architecture is an important feature that allows problem solving knowledge, strategies, heuristics, partial results and problem solutions to be communicated within the KRAFT domain for the purpose of distributed problem solving.

The KRAFT architecture has been applied to the design of data service networks for telecommunication (Fiddian et al., 1999). Future work will focus upon testing and evaluating the KRAFT architecture in a broader range of business-to-business e-commerce scenarios.

Acknowledgments

This work was initially funded by EPSRC and BT. The research is continuing with funding from EPSRC to explore applications to Knowledge Reuse under the Advanced Knowledge Technologies (AKT) project [4] at Aberdeen; this funds the work of Kit Hui.

[4]URL: `http://www.csd.abdn.ac.uk/ research/akt/`

References

ECLiPSe Library Manual. ECRC and IC-Parc.

ECLiPSe User Manual. ECRC and IC-Parc.

Andreoli, J.-M., Borghoff, U. M., and Pareschi, R. (1995). Constraint agents for the information age. *Journal of Universal Computer Science*, 1:762–789.

Arnal, M. T. i. and Faltings, B. (1999). Smart clients: Constraint satisfaction as a paradigm for scaleable intelligent information systems. In Finin, T. and Grosof, B., editors, *Artificial Intelligence for Electronic Commerce*, pages 10–15. AAAI Press.

Bassiliades, N. and Gray, P. (1994). CoLan: a Functional Constraint Language and Its Implementation. *Data and Knowledge Engineering*, 14:203–249.

Bayardo, Jr., R. J., Bohrer, B., Brice, R. S., Cichocki, A., Fowler, J., Helal, A., Kashyap, V., Ksiezyk, T., Martin, G., Nodine, M. H., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., and Woelk, D. (1997). Infosleuth: Semantic integration of information in open and dynamic environments (experience paper). In Peckham, J., editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 195–206. ACM Press.

Chiariglione, L. (1998). FIPA – agent technologies achieve maturity. *Agent Link Newsletter*, pages 2–4.

Cutkosky, M., Engelmore, R., Fikes, R., Genesereth, M., Gruber, T., Mark, W., Tenenbaum, J., and Weber, J. (1993). PACT: an experiment in integrating concurrent engineering systems. *IEEE Computer*, 26(1):8–27.

Embury, S. (1995). User Manual for P/FDM V.9.1. Technical report, Dept. of Computing Sc., University of Aberdeen. URL: `http://www.csd.abdn.ac.uk/∼pfdm`.

Fiddian, N. J., Marti, P., Pazzaglia, J.-C., Hui, K., Preece, A., Jones, D. M., and Cui, Z. (1999). A knowledge processing system for data service network design. *BT Technical Journal*, 17(4):117–130.

Fikes, R., Cutkosky, M., Gruber, T., and Van Baalen, J. (1991). Knowledge Sharing Technology: Project Overview. Technical Report KSL 91-71, Knowledge Systems Laboratory, University of Stanford.

Finin, T. and et al., J. W. (1993). Draft Specification of the KQML Agent Communication Language. The ARPA Knowledge Sharing Initiative, External Interfaces Working Group.

Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an Agent Communication Language. In *Proceedings of Third International Conference on Information and Knowledge Management (CIKM'94)*. ACM Press.

Frayman, F. and Mittal, S. (1987). COSSACK: A constraints-based expert system for configuration tasks. In Sriram, D. and Adey, R. A., editors, *Knowledge Based Expert Systems in Engineering: Planning and Design*, pages 143–165. Computational Mechanics Publications.

Frühwirth, T., Herold, A., Küchenhoff, V., Provost, T. L., Lim, P., Monfroy, E., and Wallace, M. (1993). Constraint logic programming – an informal introduction. Technical Report ECRC-93-5, ECRC.

Genesereth, M. and Fikes, R. (1992). Knowledge Interchange Format, Version 3.0, Reference Manual. Technical Report Report Logic-92-1, Logic Group, Computer Science Department, Stanford University.

Gray, P., Preece, A., Fiddian, N., Gray, W., Bench-Capon, T., Shave, M., Azarmi, N., Wiegand, M., Ashwell, M., Beer, M., Cui, Z., Diaz, B., S.M.Embury, K.Hui, A.C.Jones, D.M.Jones, G.J.L.Kemp, E.W.Lawson, K.Lunn, P.Marti, J.Shao, and P.R.S.Visser (1997). KRAFT: Knowledge Fusion from Distributed Databases and Knowledge Bases. In Wagner, R., editor, *Proceedings of the Eighth International Workshop on Database and Expert Systems Applications*, pages 682–691, Toulouse, France. IEEE Computer Society Press.

Gray, P. M. D., Embury, S. M., Hui, K., and Kemp, G. J. L. (1999a). The evolving role of constraints in the functional data model. *Journal of Intelligent Information Systems*, 12:113–137.

Gray, P. M. D., Hui, K., and Preece, A. D. (1999b). Finding and moving constraints in cyberspace. In *Intelligent Agents in Cyberspace*, pages 121–127. AAAI Press. Papers from the 1999 AAAI Pring Symposium Technical Report SS-99-03.

Hui, K. (2000). *Knowledge Fusion and Constraint Solving in a Distributed Environment*. PhD thesis, University of Aberdeen.

Jeffery, K. (1998). Metadata: an overview and some issues. *ERCIM News*, (35).

Jennings, N., Faratin, P., Johnson, M., Norman, T., O'Brien, P., and Wiegand, M. (1996). Agent-based business process management. *International Journal of Cooperative Information Systems*, (5):105–130.

Kuokka, D., McGuire, J., Weber, J., Tenenbaum, J., Gruber, T., and Olson, G. (1994). SHADE: Knowledge based technology for the re-engineering problem.

Labrou, Y. (1996). *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland, Baltimore MD, USA.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:383–397.

McDermott, J. (1982). A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88.

Mittal, S. and Frayman, F. (1989). Towards a generic model of configuration tasks. In *Proceedings of The 11th International Joint Conference on Artificial Intelligence*, pages 1395–1401. AAAI Press.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senatir, T., and Swartout, W. (1991). Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56.

Nodine, M., Perry, B., and Unruh, A. (1998). Experience with the infosleuth agent architecture. In *Proceedings of AAAI 98 Workshop on Software Tools for Developing Agents*.

Patil, R., Fikes, R., Patel-Schneider, P., McKay, D., Finin, T., Gruber, T., and Neches, R. (1992). The DARPA Knowledge Sharing Effort: Progress Report. In Nebel, B. and Swartout, W., editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 777–788, Cambridge, MA, USA. Morgan Kauffman Publishers.

Preece, A. D., Hui, K., and Gray, P. M. D. (1999). KRAFT: Supporting virtual organisations through knowledge fusion. In Finin, T. and Grosof, B., editors, *Artificial Intelligence for Electronic Commerce*, pages 33–38. AAAI Press.

Provost, T. L. and Wallace, M. (1991). Generalised constraint propagation over the CLP scheme. Technical Report ECRC-91-1, ECRC. Also appears in *Journal of Logic Programming*, 16(3):319–359, 1993.

Reeves, D. M., Grosof, B. N., Wellman, M. P., and Chan, H. Y. (1999). Toward a declarative language for negotiating executable contracts. In Finin, T. and Grosof, B., editors, *Artificial Intelligence for Electronic Commerce*, pages 39–45. AAAI Press.

Sabin, D. and Freuder, E. C. (1996). Configuration as composite constraint satisfaction. In *Workshop Notes of AAAI Fall Symposium on Configuration*, pages 28–36, Menlo Park, California. AAAI Press.

Schein, E. (1994). Innovative cultures and organisations. pages 125–146.

Shipman, D. (1981). The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173.

Wallace, M. (1998). Constraint programming. In Jay, L., editor, *The Handbook of Applied Expert Systems*. CRC Press.