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. 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., 1999a; 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 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

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35658-7_21

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., 1999a). 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

Consider the problem of configuring a computer from the set of product catalogues provided by different vendors as databases. User requirements and design restrictions can be represented as constraints. Examples are:

"The PC must use a Pentium II processor."

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

To arrive at a usable configuration, we may issue a distributed database query that performs a join across multiple database tables and then check the retrieved components 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, describing how the data have to be used. For example, a particular operating system may have a 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.

1.2. A Distributed Configuration Design as a Constraint Satisfaction Problem

A configuration problem is a design activity in which an artifact is assembled by connecting a set of components in certain ways. The configuration problem in KRAFT has some interesting characteristics which make it difficult to be handled by traditional rule-based configuration systems, like R1/XCON (McDermott, 1982).

When a resource joins the network, both stored data and semantic knowledge must be incorporated automatically. This dynamic environment, together with mobile knowledge which can attach to data, make the problem specification dynamic, since

[1]URL: http://www.csd.abdn.ac.uk/~apreece/Research/KRAFT.html/

it may change as different data objects become involved. 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.

Our approach is to represent the configuration problem as a constraint satisfaction problem (CSP) and 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).

2. Modelling the Configuration Task in KRAFT

KRAFT uses constraints as a uniform formalism to represent user specifications and domain knowledge on component compatibility. A declarative constraint is a self-contained mobile knowledge object, in which selection information can be moved within a computation (Gray et al., 1999a). These features allow different problem-solving strategies to be explored.

Component instances, which define the domains of variables in the CSP, are stored in different *vendor databases* with attached constraints. Other constraints come from an otherwise empty *solution database* (section 2.1) and also the *user*. Constraints from different resources may be expressed in different vocabularies and against different schemas. The KRAFT architecture is flexible enough to cope with heterogeneous resources (section 3) but to simplify our problem, we assume the use of a single *integration schema* within the KRAFT domain. Constraints and data expressed against local schemas will be transformed and mapped into this *integration schema*.

Mittal and Frayman (Frayman and Mittal, 1987; Mittal and Frayman, 1989) presented a generic domain-independent model of configuration based on constraint solving. Sabin and Freuder (Sabin and Freuder, 1996) further proposed the framework of *composite CSP*, in which instantiating variables may change a CSP dynamically. In KRAFT, 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 initially empty and yet to be populated by the solutions of a CSP, after it is solved. We restrict the combination of values which can be stored and qualified as solutions to the CSP by imposing integrity constraints against the *solution database* schema. Although initially empty of data, the *solution database* provides a framework for specifying and integrating the problem-solving knowledge, through its attached constraint metadata. Figure 1 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 on the *solution database*:

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

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:

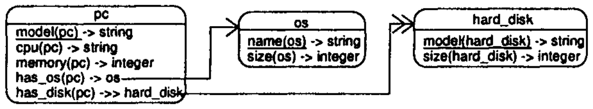

Figure 1. Our example solution database schema of configured PCs.

$$(\forall p, o, so) \begin{pmatrix} pc(p) \wedge \\ has\_os(p,o) \wedge size(o,so) \end{pmatrix} \longrightarrow (\exists d, sd) \begin{pmatrix} has\_disk(p,d) \wedge \\ size(d,sd) \wedge sd \geq so \end{pmatrix}$$

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 (Embury, 1995) 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.

A manufacturer producing tailor-made OS for the *"Pentium III"* platform may put the following universally quantified constraint on all OS in his product database:

$$(\forall o, p, c) \quad \big(\; os(o) \wedge pc(p) \wedge has\_os(p,o) \wedge cpu(p,c) \; \big) \longrightarrow c = \text{"}pentium3\text{"}$$

With an optional filter, we can selectively apply a constraint to a reduced set of data instances instead of all objects of a class. This allows constraint knowledge to be attached as if to an individual data object. The following is an example of a *conditional constraint* which only applies when the name of the OS is "winNT":

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

Database integrity constraints are traditionally used for validation checks on populated data. In using database integrity constraints as CSP specifications, we extend the use of integrity constraints to include unpopulated entity classes. Thus the manufacturers and designers are putting constraints on objects which will form relationships with the components but are not yet connected! 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.

2.3. Categorising Constraints

Constraints can be categorised according to their origin. *Small-print constraints* (Gray et al., 1999b), resemble *small-prints* and footnotes in a product catalogue.

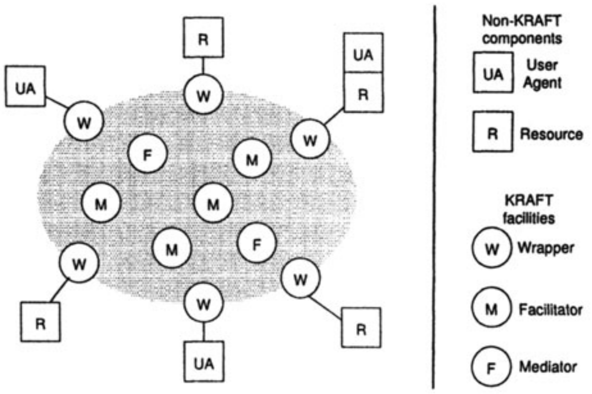

Figure 2. This figure shows a conceptual view of the KRAFT architecture. The grey area represents the KRAFT domain where a uniform language and communication protocol is respected.

They are stored in databases in association with class descriptors for data objects, and can be viewed as an attachment of instructions on how a data object should be used. *Design constraints* capture expert knowledge about feasible designs and are stored in the *solution database*. *User requirement constraints* come from the user and represent user specifications on the desired configurations.

This categorisation, however, 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. *Design constraints* capture domain knowledge of an application problem. They can be viewed as attached to a particular problem, and thus apply to different instances of the same 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, a constraint from the user may be attached to a particular data object, thus behaving as a *small-print constraint.*

3. The KRAFT Architecture

Knowledge processing components in KRAFT 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 (figure 2).

Three important KRAFT facilities of distinctive roles have been identified. *Wrappers* interface non-KRAFT components to the KRAFT network by providing translation services and high-level communication mechanisms. *Facilitators* maintain directories of KRAFT facilities. Their principal function is to accept messages from other KRAFT facilities and route them appropriately. *Mediators* are KRAFT components

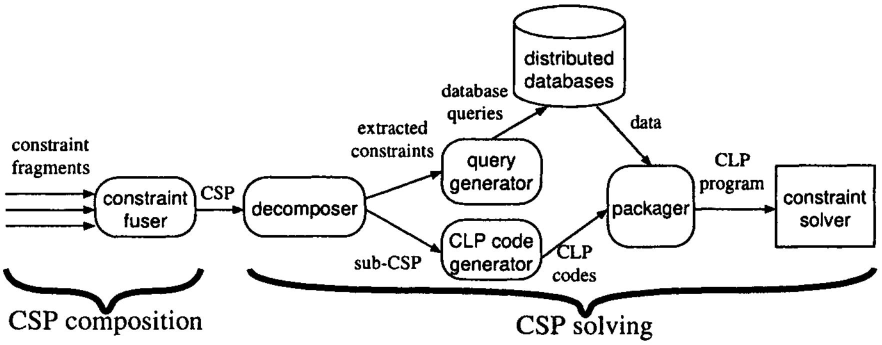

Figure 3. The KRAFT problem-solving process is divided into two phases: CSP composition and CSP solving.

that can utilise domain knowledge to transform data in order to increase their information content. Non-KRAFT components which are linked to the KRAFT network via *wrappers* are *user agents* and *resources*. Users access the services of the KRAFT domain via *user agents*. *Resources* include information sources such as databases, knowledge bases and also processing engines like constraint solvers.

The design of KRAFT is consistent with several emerging agent standards, notably KQML (Finin et al., 1993; Finin et al., 1994) and FIPA (Chiariglione, 1998). A detailed discussion of the KRAFT architecture can be found in (Gray et al., 1997; Preece et al., 1999a; Preece et al., 1999b; Preece et al., 2001).

4. CSP Composition

Problem solving in KRAFT is divided into two stages (figure 3). 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-problems which are solved by multiple problem solving components. The CSP composition process can be further divided into three stages: *constraint extraction and transformation*, *constraint fusion* and *CSP formation from integrity constraints.*

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 extract constraints, it is necessary for resources to support meta-level queries that retrieve stored constraint information instead of data. A resource which does

[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.

```
with common p in pc
  rewrite simm(has_motherboard(p))+sdram(has_motherboard(p))
  into memory(p);
with common p in pc
  rewrite os_name(p) into name(has_os(p));
```

Figure 4. Example rewrite rules.

```
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 5. Example constraints. The first constraint is expressed against a local schema. The second constraint shows the result of transforming the first one to refer to the integration schema.

not support constraint extraction forces a localised constraint utilisation, thus restraining the system from composing a global execution plan. Our prototype uses the P/FDM database system (Embury, 1995) which provides a uniform access to meta data through queries on the meta-schema by the Daplex language (Embury, 1991).

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 is achieved by rewriting each constraint to refer to an *integration schema.* Each local resource in KRAFT 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. Figure 4 shows two examples of rewrite rules. The effect of applying these rewrite rules to a constraint is shown in figure 5.

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. A constraint which is true in a local resource may not remain true when it migrates out of that resource. 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. 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 constraints are conjoined together, they exchange information and enhance the semantics of each other. The result 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. Thus the presence of a *universally quantifier* is a necessary condition for constraint fusion to take place.

Operationally, constraint fusion is the identification of correspondences between variables in different constraint fragments, which allow potential constraint information flow between them. These correspondences, called *variable-links*, are identified by examining how variables are 'generated'. Consider the following example:

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

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:

$$(\forall p, m) \quad \big(\ pc(p) \wedge memory(p, m)\ \big) \longrightarrow \big(\ m \geq 32 \wedge m \leq 1024\ \big)$$

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:

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

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

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

Once all *variable-links* are identified between two constraints, there are two approaches to fuse them: *implicit* and *explicit*. These two approaches of 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 6 and 7 shows an example of fusing three constraints.

4.3. CSP Formation from Database Integrity Constraints

The fused constraints precisely describe the desired states of the *solution database* but they cannot be directly compiled into an executable program to find the solution values. In particular, they contain references to unpopulated values and relationships, called *empty-slots* (section 2.2). The *empty-slot* problem arises because we are moving a constraint expressed like an integrity constraint from a database where some slots

```
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 6. Three example constraints representing a user requirement, a designer constraint and a small-print constraint.

```
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 7. The result of fusing the three constraints in figure 6.

are unpopulated, into the context of the *solution database*, where the slots are assumed to be populated. A database integrity constraint that references an *empty-slot* always trivially succeeds or fails[3] because there are no stored instances that can satisfy the slot predicate. Similarly, a query that tries to retrieve from an *empty-slot* always gets nothing. The transformation from an integrity constraint into a CSP, however, is surprisingly simple. Consider the following integrity constraint:

$$(\forall p, o, n) \quad \big(\ pc(p) \wedge os(o) \wedge has\_os(p,o) \wedge name(o,n)\ \big) \longrightarrow n \neq \text{"}win95\text{"}$$

In a populated *solution database*, 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 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 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, therefore, is to take out all the references to *empty-slots*, meaning that the *empty-slots* put no restriction on any variable. 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):

$$(\forall p, o, n) \quad \big(\ pc(p) \wedge os(o) \wedge name(o,n)\ \big) \longrightarrow n \neq \text{"}win95\text{"}$$

[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.

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* is a important piece of meta-knowledge which 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, 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 *constraint logic programming* (CLP) code. We use the ECLiPSe CLP system (Aggoun et al., 1999; Brisset et al., 1999) as it supports flexible code generation as in logic programming (LP) systems but being more efficient in execution. The generated CLP code and variable domain information are 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*, *variable domain formation*, *constraint posting* and *variable labelling*.

5.1. Database Query Formation from the CSP

By extracting constraint information from a CSP to compose database queries, we delegate part of the CSP solving process to the involved databases. This promotes early data filtering, thus reducing the amount of candidate data transported from databases into the constraint solving components.

Most databases support the use of a uniform data filter, for example, by generating a WHERE clause in SQL. In the case of a conditional constraint (figure 7), we can use a technique that transforms it 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. As a result, we compose database queries by extracting constraint information from universally quantified constraints that always apply to the solutions. Conditional constraints in the CSP will be compiled into CLP program code and handled by the constraint solver. *Existentially quantified constraints* are usually ignored in database query formation as the constraint information they contain are not strong enough. More detailed discussions can be found in (Hui, 2000).

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. A detailed discussion on *variable domain population* can be found in (Hui and Gray, 2000).

5.3. Constraint Posting and Variable Labelling

Our CLP code generator systematically compiles CIF constraints into ECLiPSe code (Hui and Gray, 2000). The generated program has a top level predicate calling three subgoals, resembling the three stages of *variable declaration*, *constraint posting* and *variable labelling* in CLP (Frühwirth et al., 1993; Wallace, 1998). Information is communicated through a shared variable:

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

Variable labelling is the final stage of CSP solving where variables are instantiated to values in their respective domains. When variables are gradually instantiated, delayed constraints are awakened and backtracking may occur, until a consistent constraint network is reached or a conflict is detected.

6. Related Work

KRAFT employs an agent-based architecture which is proving to be an effective approach to developing distributed information systems. Early projects like PACT (Cutkosky et al., 1993) and SHADE (Kuokka et al., 1994) have already shown that agent technology can support the 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).

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. We recognise the fact that constraints have evolved from database states restrictors to a kind of portable knowledge that can be exported and processed (Gray et al., 1999a). 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.

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 (EPSRC grant no. GR/N15764) [4] at Aberdeen; this funds the work of Kit Hui.

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

References

Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Harvey, W., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Novello, S., Perez, B., van Rossum, E., Schimpf, J., Shen, K., Tsahageas, P., and de Villeneuve, D. (1999). *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.

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.

Brisset, P., El Sakkout, H., Frühwirth, T., Gervet, C., Harvey, W., Meier, M., Novello, S., Le Provost, T., Schimpf, J., Shen, K., and Wallace, M. (1999). *ECLiPSe Library Manual.* ECRC and IC-Parc.

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. (1991). The Implementation of an Interface to Metadata in P/FDM. Technical Report AUCS/TR9114, Dept. of Computing Science, University of Aberdeen, Aberdeen, Scotland, AB2 9UE.

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., 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.

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

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.

Hui, K. and Gray, P. M. D. (2000). Developing finite domain constraints – a data model approach. In *Proceedings of the 1st Internatinal Conference on Computational Logic (CL2000)*, pages 448–462. Springer-Verlag.

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., Hui, K., Gray, A., Marti, P., Bench-Capon, T., Cui, Z., and Jones, D. (2001). KRAFT: An agent architecture for knowledge fusion. *Inernational Journal of Cooperative Information Systems*, 10(1 & 2):171–195.

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

Preece, A. D., Hui, K., Gray, W. A., Marti, P., Bench-Capon, T. J. M., Jones, D. M., and Cui, Z. (1999b). The KRAFT architecture for knowledge fusion and transformation. In *19th SGES International Conference on Knowledge-based System and Applied Artificial Intelligence (ES99).* Springer.

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.

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