ELSEVIER Science of Computer Programming 30 (1998) 201–225

Science of Computer Programming

Constraint-based protocols for distributed problem solving [1]

Uwe M. Borghoff[a,*], Remo Pareschi[a], Francesca Arcelli[b], Ferrante Formato[c]

[a] *Rank Xerox Research Centre, Grenoble Laboratory, 6, chemin de Maupertuis, F-38240 Meylan, France*
[b] *Dipartimento di Ingegneria dell' Informazione e di Ingegneria Elettrica, Università di Salerno, I-84084 Fisciano (SA), Italy*
[c] *Centro di Ricerca in Matematica Pura e Applicata (CRMPA), Salerno, I-84084 Fisciano (SA), Italy*

Abstract

Distributed Problem Solving (DPS) approaches decompose problems into subproblems to be solved by interacting, cooperative software agents. Thus, DPS is suitable for solving problems characterized by many interdependencies among subproblems in the context of parallel and distributed architectures. Concurrent Constraint Programming (CCP) provides a powerful execution framework for DPS where constraints define local problem solving and the exchange of information among agents declaratively. To optimize DPS, the protocol for constraint communication must be tuned to the specific kind of DPS problem and the characteristics of the underlying system architecture. In this paper, we provide a formal framework for modeling different problems and we show how the framework applies to simple yet generalizable examples. © 1998 Elsevier Science B.V.

Keywords: Constraint propagation; Distributed artificial intelligence; Distributed problem solving; Constraint-based knowledge brokers; Cooperative agents; Protocols

1. Introduction

From a problem solving point of view, distribution requires the decomposition of a problem into a set of subproblems, where the solution of the problem amounts to concurrently solving all of the subproblems and then composing their solutions. Subproblems are solved by parallel processes, or by dynamic software agents in networked environments. As characterized in [15, 29], if there are many interdependencies among the subproblems (and thus the agents assigned to solve them must interact a great deal) then we pass from simple distributed processing into the realm of true DPS. For instance, gathering information from distributed knowledge repositories in the context

* Corresponding author. E-mail: borghoff@grenoble.rxrc.xerox.com.
[1] Extended version of a paper [5] presented at the 1st International Workshop on Concurrent Constraint Programming CCP '95 (Venice, Italy, 29–31 May 1995).

of executing a complex plan belongs to DPS: information gathered as part of a certain task in the plan may influence other tasks.

Concurrent Constraint Programming (CCP) is a powerful framework to support DPS, since it provides a direct implementation of the notion of shared, reusable, incrementally refined information. To optimally solve DPS problems with CCP, we need, however, to be able to tune the scope of inter-agent communication according to such factors as the type of the DPS problem and of the characteristics of the underlying system architecture. Specifically, we need to determine how many agents can make use of, and thus should be exposed to, given information in the form of constraints. In this aim, we define a *design space* of inter-agent communication protocols by formally defining two opposite extremes of information sharing. One extreme captures the hypothesis of *minimal* information reuse: all generated information is potentially made available, but can be delivered only in response to specific requests. We shall then consider the opposite case, based on the idea of *maximal* information reuse: all generated information is immediately broadcast to all agents. If the generation of a result is costly and if the result is frequently reused at many agent sites, the broadcast of the result can decrease the amount of network traffic in the entire system. Delivery of results is done once and forever. On the other hand, there is a risk of cluttering agents with useless information.

In the course of the paper, we shall formally introduce the notion of *quantity of reuse* of information to obtain a heuristic to decide between the two protocols, as well as a basic metric over the space of protocols that lie in between the two extremes. To provide a clear illustration of how this heuristic works, we shall apply it to a very simple, abstract case, namely, the distributed parsing of a context-free grammar generating boolean expressions. We shall also show, however, that this simple example can be easily generalized to real-life problems. Conversely, we shall show that any kind of DPS problem can be mapped into a grammatical problem, where experimented algebraic techniques can be exploited to derive the heuristic.

Our study of protocols for constraint-based communication is done within the framework of the Constraint-Based Knowledge Broker (CBKB) model [1, 2], a recently introduced approach to CCP where a sharp separation is drawn between the use of constraints for (1) local problem solving; and (2) communication. Compared to other formalizations of CCP, the CBKB model focuses more directly on communication issues and, hence, on problem solving done via the interaction of cooperative agents. An illustration of the use of CBKBs for information gathering in network-wide environments is given in [3]. Ref. [9] (see also [10, 43]) provides a detailed description of a CBKB software implementing information gathering facilities on the World-Wide Web.

This paper indicates several directions for future work. As we pointed out, the two protocols above are at the very opposite ends of a spectrum of possible protocols. Intermediate cases correspond to group deliveries for subsets of agents. For many practical applications these intermediate cases seem to be the most useful. Thus, our heuristic provides an elementary compass to help navigate this space of protocols. Further developments are needed to refine it into a set of techniques for automatically assigning agents to appropriate "interest groups" and for allowing flexible tuning of group-based communication. Moreover, there are situations where the optimal strategy for DPS may involve splitting the problem into subproblems which are optimally solved according to different protocols; these may in turn be themselves composite. Again, we will need flexible ways for identifying and for expressing such protocols.

The paper is organized as follows. Section 2 outlines the main characteristics of the concurrent language LO that is used to specify the DPS protocols. Section 3 formally introduces the notion of Constraint Based Knowledge Broker, which is then used to formalize and compare the two opposite DPS protocols, exploiting, respectively, the *minimal* and the *maximal* reuse of generated results. In Section 4, an analysis follows where the notion of *quantity of reuse* is introduced to provide a choice criterion for the appropriate protocol. A simple example from context-free parsing illustrates the heuristic thus defined. This example is then generalized to examples in the DPS domain of information gathering. Conversely, it is shown how algebraic techniques from formal languages can be applied to derive the heuristic. In Section 5, related work is discussed, and finally, Section 6 concludes by addressing future work.

2. A declarative, executable specification language: LO

We use LO [4] as a specification language for our DPS protocols. LO is a declarative concurrent language which amalgamates aspects of several concurrent languages based on generative communication, including CHAM [7], Gamma [6], Linda [11], Maude [27], Swarm [31] and borrows also from concurrent logic programming languages [12, 20, 24, 38, 40]. LO's logical primitives for agent coordination and communication make it particularly suitable for our purposes; however, our approach is also compatible with other choices of specification language.

The LO programming constructs are rules that define interaction among concurrent agents. Agents are seen as pools of tokens where a pool is represented as a multiset of these tokens. There are three basic constructs in LO:

- The @-construct is used to manage intra-pool concurrency. It is used in rules of the form

  $a\,@\,b$ **<>-** $c\,@\,d$

  meaning that a and b must both be present in the pool before application of the rule and that both are withdrawn from the pool after application. Moreover, c and d are present in the pool after application.
- The &-construct is used to manage inter-pool concurrency. It is used in rules of the form

  $a\,@\,b$ **<>-** $c\,\&\,d$

  meaning that a and b must both be present in the pool before application of the rule and that both are withdrawn from the pool after application. Moreover, two new pools are created by cloning the remaining tokens, adding c to one of the created pools, and d to the other.

- The ^-construct is used to broadcast tokens between agents. It is used in rules of the form

  $a@b@\hat{}c$ <>- d

  meaning that a and b must both be present in the pool before application of the rule and are both withdrawn from the pool after application. Moreover, after application, d is present in the pool and c is present in any pool different from the one in which the rule has been applied. [2]

Tokens can be *indexed* so as to restrict broadcasting to multicasting up to the degenerate case of one-to-one communication. Indices allow linking of senders and receivers for specific tokens: indexed tokens are sent only to agents that are "registered" for that index. Indexing and de-indexing of tokens will be the crucial means for differentiating the DPS protocols to be specified in the course of the paper.

Rules are triggered via pattern matching. The state of a pool is matched against the left-hand sides of rules. When a match is found, the selected rule is applied, i.e. the tokens in its left-hand side are removed from the pool, broadcast tokens are transmitted to other pools, and new multisets are defined according to the right-hand side of the rule.

For a formal operational semantics of LO, based on the proof theory of Linear Logic [19], see [4].

3. A formal framework for DPS

We can formalize the problem–subproblem relationship, which is at the basis of DPS, via the notion of *generator*. Intuitively, a generator defines the subproblems which need to be solved in order to solve a given problem. A generator works by formalizing problem solving in terms of stable models and fixed-point operators. *Constraints* provide a declarative way to prune the search space.

In order to make the paper self-contained, we restate in the following some of the formal definitions given in [2]. In the remainder, moreover, we illustrate many of these formal aspects through concise examples.

3.1. Generator

Given an abstract domain of values $\mathscr{D}$, representing tokens of knowledge, a *generator* is a mapping $\mathscr{D}^n \mapsto \wp(\mathscr{D})$, which produces new tokens from existing ones. That is, a generator (having arity n) takes some n tokens of existing knowledge, applies the generator's function, and provides some new tokens of knowledge (if any).

A set of generators identifies a class of subsets of the abstract domain which are stable by these generators. Let E be a subset of the domain and Γ a set of generators.

[2] A library of higher-level communication facilities on top of the basic mechanism of broadcasting in LO is described in [8].

Definition 1. E is called Γ-stable if and only if

$$\forall g \in \Gamma,\ \forall x_1,\dots,x_n \in E: \quad g(x_1,\dots,x_n) \subset E.$$

The class of stable sets is closed under intersection, so that it has a smallest element in the sense of inclusion, given by the intersection of all the stable sets. This minimal stable set, also called *minimal model*, represents the intended semantics of the set of generators. The class of stable sets is also closed under non-decreasing union, so that a standard fixed-point procedure can be applied to compute the minimal model. The core of the procedure is given by the mapping:

$$T : \wp(\mathscr{D}) \mapsto \wp(\mathscr{D}) \qquad \text{where } \forall E \in \wp(\mathscr{D}): \quad T(E) = \bigcup_{\substack{g \in \Gamma \\ x_1,\dots,x_n \in E}} g(x_1,\dots,x_n).$$

The minimal model M is then given by the least fixed-point of T, expressed as

$$M = \bigcup_{n \in \mathbb{N}} T^n(\emptyset).$$

Thus, T provides a way of incrementally computing the minimal stable set, by computing the sequence $(T^n(\emptyset))_{n\in\mathbb{N}}$, given by $E_0 = \emptyset$ and $E_{n+1} = T(E_n)$.

Note, however, that a minimal model can be infinite. Take for instance as abstract domain the set of words built out of the letters 'a' and 'b'. Assume, furthermore, a generator (having arity 2) that takes two existing words, concatenates them, appends an 'a' or a 'b' randomly, and provides this (new) word as additional knowledge. This new word can then serve as new input to generate more knowledge. Obviously, this procedure will generate infinitely many words of the domain. A well-known mechanism to prune this generation procedure uses constraints. In our example, constraining the maximal length of the words would immediately result in a finite minimal model.

In addition to simply constrain an individual generator's function, we shall define some mappings to express dependencies between the input and the output tokens of different generators.

Generators can be associated with a class of LO-agents which we call *broker* agents. A broker implements a generator function taking several input tokens (according to its arity) and producing corresponding output tokens.

The fixed-point procedure is initialized by putting the following special input tokens inside each broker agent:

- the token **arity**-n, where n is a non-negative integer, holds the arity of the generator g
- the token **free**-k (for each positive integer k, less than or equal to the arity) which represents a place holder, one for each argument of the generator g. These tokens are consumed as arguments get bound.

This basic protocol, which relies on a simple forward-chaining scheme, can be refined into protocols appropriate for DPS by taking into account dependencies between the input and output tokens of different broker agents' generators. Both refinements to be illustrated in this paper rely on the idea that a broker agent has some knowledge of the dependencies between the input and output tokens of its associated generator. Thus the broker is able, given a *constraint* on the outputs (the constraint given by a *request*), to infer constraints on the inputs. In this way, constraints provide top-down filtering over the set of values that can be produced by a generator as candidate solutions for a given problem, and permit the exploitation of interdependencies between subproblems.

More formally, let us consider the mapping $\bar{g} : \wp(\mathscr{D}) \mapsto \wp(\mathscr{D}^n)$ defined by

$$\forall w \in \wp(\mathscr{D}), \quad \bar{g}(w) = \prod_{k=1}^{n} \bar{g}_k(w).$$

Note that, in order to infer constraints on the inputs, $\bar{g}_k(w)$, the so-called *backdependency* function for argument position k, must be known.

By definition, we have: $\forall \boldsymbol{x} \in \mathscr{D}^n$, $\forall w \in \wp(\mathscr{D})$, if $g(\boldsymbol{x}) \cap w \neq \emptyset$ then $\boldsymbol{x} \in \bar{g}(w)$. If $\mathscr{G}$ denotes the graph of the function of generator g (hence $\mathscr{G} \in \wp(\mathscr{D}^{n+1})$), then we have equivalently, $\forall w \in \wp(\mathscr{D})$, $\mathscr{G} \cap (\mathscr{D}^n \times w) \subset (\bar{g}(w) \times w)$.

In other words, $\bar{g}(w) \times w$ provides an upper approximation (in the sense of inclusion) of the part of the set $(\mathscr{D}^n \times w)$ which is contained in the graph of the generator's function. To capture inter-argument dependencies, we generalize this notion of approximation, and we assume that each broker is equipped not only with a generator g but also with an *interdependency* function $\tilde{g}$. The interdependency function computes an upper approximation of the part of any subset of $\mathscr{D}^{n+1}$ which is included in the graph of the generator's function.

Intuitively,

- the *backdependency* functions describe how a broker will decompose a given problem description into subproblems. These subproblems are then propagated to other brokers as their problem descriptions.

  The set of all brokers act on request. The request may come from an end user, or, as part of a subproblem propagation, from other brokers. All involved brokers cooperate to solve their part of the overall problem.
- the *interdependency* function describes how the subproblem solutions relate to each other.

We illustrate this behavior in Fig. 1. The body of this figure shows graphically how interdependencies in the activities of broker agents are exploited to feed the generator's function with the needed input tokens.

3.2. Scope of a broker

We define the *scope* of a broker as the subset of the domain which does not intersect any of the constraints of the requests it has already processed. In other words, the scope of a broker denotes the complement of the set of elements of the minimal model the broker (or one of its clones) has already explicitly generated (or is in the process of generating). A broker can be viewed as an agent which explores a domain and explicitly generates the elements it encounters which are in the minimal model. The scope of the broker denotes the un-explored area of the domain.

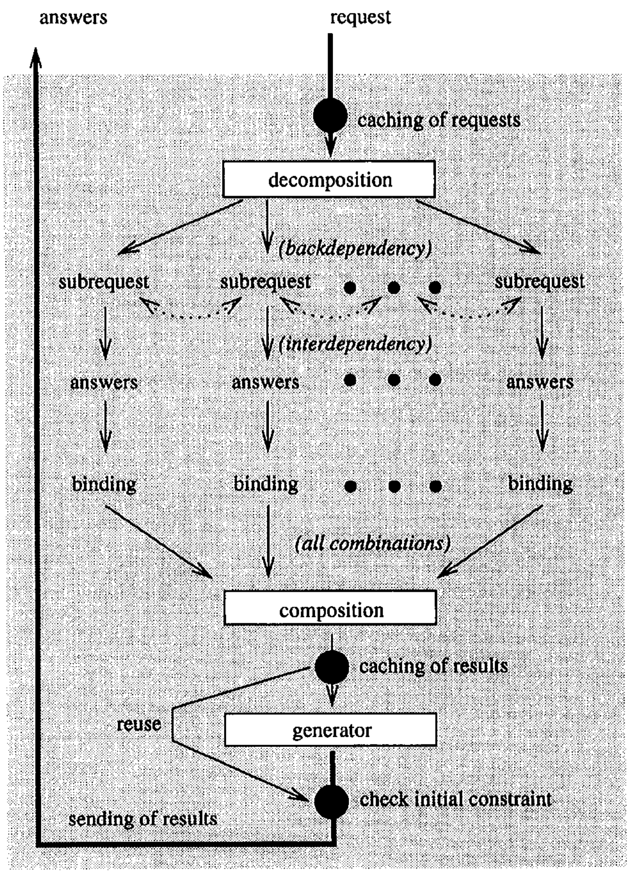

Fig. 1. Exploiting interdependencies in a broker of arity n.

Let w_0 be the scope of a broker at some point, and let w be the constraint of a request it receives. The broker spawns an agent in charge of exploring the subset $w_0 \cap w$ (and answering the request), and then continues with a reduced scope $w_0 \cap \neg w$ (where $\neg w$ is the complement of w). The interested reader is referred to [3, 42] for a detailed discussion on how to implement, work with, and split the scope of a broker.

3.3. The request–subrequest protocol

The request–subrequest protocol refines constraint-based communication as illustrated in the section above by exploiting yet another dependency: the request carries an index that is added to all output tokens that are sent out. In this way, requester and requestee are directly linked. Information is provided only if requested, and is sent only to the original requester.

The initial request carries an index which acts as an address for the requesting agent, as well as a description of the problem to be solved (instantiated as a constraint on the problem domain). A broker agent takes the problem description and simplifies it into subproblems. These descriptions of the subproblems are then submitted as subrequests in the same way as the initial request. The subrequests are individually indexed so that they can be collected into a solution by the requestee agents which is eventually returned to the original requester.

We encode the request–subrequest protocol with the following LO rules.

(RS1): **broker**(w_0) @ **requ**(I, w) @ **split**(w_0, w, w_1, w_2) @ **init**(w_1, s)
<>- **broker**(w_2) & **process** @ **requ**(I, w) @ **const**(s).

The token **split**(w_0, w, w_1, w_2) defines, given two subsets w_0, w of $\mathcal{D}$, the subsets w_1, w_2 of $\mathcal{D}$ defined by $w_1 = w_0 \cap w$ and $w_2 = w_0 \cap \neg w$. As mentioned before, each broker is equipped not only with a generator g, but also with an interdependency function $\tilde{g}$ which computes an upper approximation of the part of any subset of $\mathcal{D}^{n+1}$ which is included in the graph of the function of generator g. Now, the token **init**(w, s) is used to compute the *initial* approximation: It builds, given a subset w of $\mathcal{D}$, the subset s of $\mathcal{D}^{n+1}$ – called *broker-local constraint store* – defined as $s = \tilde{g}(\mathcal{D}^n \times w)$. It is assumed that if the broker-local constraint store s is empty, the token is not present. The index I of a request acts as an address for the requester.

(RS2): **const**(s) @ **free**-k @ **seek**-k(s, w) @ ^**requ**(I, w)
<>- **const**(s) @ **wait**-k(I) @ **requ**(I, w).

The token **seek**-k(s, w) is used to extract information from an approximation (by simple projection in the broker-local constraint store). It builds, given a subset s of $\mathcal{D}^{n+1}$, the subset w of $\mathcal{D}$ defined as $w = \pi_k\langle s\rangle$. The agent sends an indexed subrequest for the kth subproblem specified by w.

(RS3): **const**(s) @ **wait**-k(I) @ **answer**(I, x) @ **ins**-k(x, s, s')
<>- **const**(s) @ **wait**-k(I) & **const**(s') @ **bound**-k(x).

The token **ins**-k(x, s, s') provides further refinements of the approximation upon receiving answers to the kth subrequest. It builds, given a subset s of $\mathcal{D}^{n+1}$ (the broker-local constraint store) and an element x of $\mathcal{D}$ (a particular answer), the subset s' of $\mathcal{D}^{n+1}$ defined as $s' = \tilde{g}(s \cap \pi_k^{-1} < x >)$. In other words, s' is the approximation of the subset of s consisting of the tuples which are in the graph of the generator's function and whose kth component is precisely x. Thus, the binding of the kth argument may reduce the scope of the other arguments. It is assumed here that if the broker-local constraint store s' is empty, the token is not present.

(RS4): **arity**-n @ **bound**-1(x_1) @ $\cdots$ @ **bound**-n(x_n)
<>- **tuple**-n($x_1, \ldots, x_n$).

The broker agent feeds the generator's function with input tokens of the corresponding arity. Using these input tokens provided in **tuple**-n($x_1, \ldots, x_n$), the generator produces a result tuple **res**(x) $=$ $g(x_1, \ldots, x_n)$.

(RS5): **process** @ **res**(x) **<>-** **process** & **process**(x).

A result tuple **res**(x) is used to create a process **process**(x) that is in charge of a specific result x.

(RS6): **process**(x) @ **requ**(I, w) @ **sat**(x, w) @ ^**answer**(I, x)
< >- **process**(x).

Send out the answer using the index I. The token **sat**(x, w) is needed for consistency reasons: it simply checks whether a partial result x complies with the given constraint w.

The request–subrequest protocol as given above avoids redundancies in subproblem generation *within a broker*. Once obtained, results are cached and reused.

Example 2. Suppose a broker has a generator g with $g(t) = \{t\}$ and receives a request with a constraint w such that $t \in w$. As illustrated in rule (RS1), a specialized agent ***process*** is created to process this request. Suppose furthermore that the initial problem (given as a constraint) $\tilde{g}(\mathscr{S} \times w)$, returned by the token ***init***(w, s), is $s = w \times w$. Now, since ***seek***-1(s, w) $= \pi_1 \langle s \rangle = w$, the subrequest for the (single) argument of the generator also has w as constraint. Sent in rule (RS2), the same broker receives this subrequest. However, due to the scope reduction performed by ***split***(w_0, w, w_1, w_2), the broker is no longer in charge of a problem constrained by w. Instead, the specialized agent ***process***(x) will reuse, as illustrated in rule (RS6), the already calculated result.

3.4. *The local caching protocol*

In contrast with the request–subrequest protocol, the local caching protocol does not link requesters with requestees. Instead, as soon as they are available, solutions to subproblems are broadcast to all existing broker agents. The local caching protocol aims to avoid redundancies in subproblem generation *among brokers*.

The initial request carries only a description of the problem to be solved; no index is associated with it. As before, a broker agent takes the problem description and simplifies it into subproblems. However, as a consequence of this protocol, some of the subproblems solutions may already be known to the broker agents. The description of still unsolved subproblems are submitted as subrequests in the same way as the initial request, i.e., again without index. In this way, we obtain a situation of local *caching* of information for all existing broker instantiations, thus decreasing the overall amount of result generations and, possibly, network traffic, as we avoid the re-generation of the same requests from different requesters. On the other hand, we may end up storing information which never gets used.

In general, this approach is appropriate when there may be several requests of the same item.

We encode the local caching protocol with the following LO rules.

(LC1): **broker**(w_0) @ **requ**(w) @ **split**(w_0, w, w_1, w_2) @ **init**(w_1, s)
< >-**broker**(w_2) & **process** @ **requ**(w) @ **const**(s).

The request carries no index.

(LC2): **const**(s) @ **free**-k @ **nosolution**-k($\mathscr{X}, s$) @ **seek**-k(s, w) @ ^**requ**(w)
< >-**const**(s) @ **requ**(w).

If a solution to the kth subproblem specified through w has not yet been obtained, the agent sends a de-indexed subrequest. The token **nosolution**-$k(\mathscr{X},s)$ checks the multiset $\mathscr{X}$ of already obtained results; if and only if none of the elements of $\mathscr{X}$ complies with the constraint for the kth argument position, the token holds. Even if an answer arrives which complies with the constraint for the kth argument position, but is not checked in time, the only drawback is that a redundant subrequest will be generated. Effective handling of negative information of this kind is a well-known problem in concurrent programming, for which first results can be found in [33].

(LC3): **const**(s) @ **answer**(x) @ **seek**-$k(s,w)$ @ **sat**(x,w) @ **ins**-$k(x,s,s')$
< >-const(s) & **const**(s') @ **bound**-$k(x)$.

If a partial result x is (already) found that complies with the constraint w for the kth argument position, the answer x is (re-)used.

(LC4): **arity**-n @ **bound**-$1(x_1)$ @ ... @ **bound**-$n(x_n)$
< >-tuple-$n(x_1,\dots,x_n)$.

As in rule (RS4), using **tuple**-$n(x_1,\dots,x_n)$, the generator produces a result tuple **res**$(x)=g(x_1,\dots,x_n)$.

(LC5): **process** @ **res**(x) **< >-process** & **process**(x).

Again, this rule generates a new token in the pool representing the "memory" of the broker. It can be reused by triggering the next rule, whose task is to check whether the answer produced by the generator contained in the token **res**(x) satisfies the constraint of a request. Eventually, the token is broadcast to all existing broker agents, by means of the unindexed token **answer**(x).

(LC6): **process**(x) @ **requ**(w) @ **sat**(x,w) @ ^**answer**(x)
< >-process(x).

The generated token is cached in **process**(x) and is broadcast to all other agents.

4. Analysis of reuse of information in the two protocols

In this section, we compare the behavior of request–subrequest and local caching from the point of view of the reuse of information. In particular, we shall substantiate the hypothesis that local caching performs better than request–subrequest when (and only when) reuse of partial results is high.

To define the terms of comparison, we introduce the notion of a *measure of reuse* R in terms of the number of the partial results that can be reused. Thus, R will provide a heuristic for the choice of the appropriate protocol. High values of R suggest choosing the local caching protocol, while low values of R indicate using the request–subrequest protocol.

To be as explicit as possible, we define R for two domains given: one, the set of propositional formulas, and the second, the algebra of closed first-order terms. Indeed, the latter domain fully generalizes the former. Throughout the paper, propositional formulas will provide a suitably simple yet non-trivial specific case where to exemplify our concepts and techniques for the analysis of the reuse of information in DPS.

4.1. *Specific case: propositional formulas*

The value R gives a measure of the reusability of tokens in the generation of a well-formed propositional formula, by counting the number of occurrences of the same subformula in a given propositional formula.

Definition 3. Let α and β represent well-formed propositional formulas, and let Λ stand for a possibly empty multiset of formulas. Let $\{A_1,A_2,\dots,A_n\}$ represent a set of propositional atomic formulas. Furthermore, let $m(A_i)$ express the multiplicity of occurrences of A_i in $\{A_1,A_2,\dots,A_n\}$.

$$\begin{aligned}
R(\{\alpha \vee \beta\} \cup \Lambda) &= R(\{\alpha,\beta\} \cup \Lambda)\\
R(\{\alpha \vee \alpha\} \cup \Lambda) &= R(\{\alpha,\alpha\} \cup \Lambda) + 2\\
R(\{\alpha \wedge \beta\} \cup \Lambda) &= R(\{\alpha,\beta\} \cup \Lambda)\\
R(\{\alpha \wedge \alpha\} \cup \Lambda) &= R(\{\alpha,\alpha\} \cup \Lambda) + 2\\
R(\{\neg\alpha\} \cup \Lambda) &= R(\{\alpha\} \cup \Lambda)\\
R(\{A_1,\dots,A_n\}) &= \sum_{\substack{A_i \in \{A_1,\dots,A_n\}\\ \text{where } m(A_i) > 1}} m(A_i)
\end{aligned}$$

Example 4. The following are two formulas with the same number of propositional atomic formulas where the measure of reuse R is low in the former and high in the latter:

$$\begin{aligned}
\alpha_m &= A_1 \vee (A_2 \vee (A_3 \vee A_4))\\
\alpha_M &= (A_1 \vee A_1) \vee (A_1 \vee A_1)
\end{aligned}$$

In the first case, the value of reuse $R(\{\alpha_m\})$ is 0 (no reuse at all), while in the second the value of reuse $R(\{\alpha_M\})$ is 10 (the maximum value of reuse for a formula of length 4).

At first glance, the definition of the measure of reuse looks rather arbitrary. However, it can be easily generalized, as we show just below.

4.2. *Algebra of closed first-order terms*

We give now a definition of R for the case where the minimal model is any algebra of closed first-order terms. As can be seen,[3] this definition generalizes Definition 3.

[3] I.e., by interpreting '$\vee$' and '$\wedge$' as functions of arity 2, and seeing propositional atomic formulas as constants.

Definition 5. Let A indicate a possibly empty multiset of terms in a signature and let $\{a_1,\dots,a_n\}$ represent a set of constants. Furthermore, let $m(t_i)$ express the multiplicity of occurrences of term t_i in the multiset $\{t_1,\dots,t_n\}$, and $m(a_i)$ express the multiplicity of occurrences of constant a_i in the multiset $\{a_1,\dots,a_n\}$.

$$R(\{f(t_1,\dots,t_n)\} \cup A) = R(\{t_1,\dots,t_n\} \cup A) + \sum_{\substack{t_i \in \{t_1,\dots,t_n\} \\ \text{where } m(t_i) > 1}} m(t_i)$$

$$R(\{a_1,\dots,a_n\}) = \sum_{\substack{a_i \in \{a_1,\dots,a_n\} \\ \text{where } m(a_i) > 1}} m(a_i)$$

Example 6. In analogy to the previous example, we give two terms with the same number of constants for which there are two different values for the measure of reuse:

$$t_m = f(a_1,a_2,a_3,a_4)$$
$$t_M = f(f(a_1,a_1),f(a_1,a_1))$$

In the former case, the value of reuse $R(\{t_m\})$ is 0 (no reuse at all), while in the second the value of reuse $R(\{t_M\})$ is 10 (the maximum value of reuse for a term of length 4; note that, $R(\{f(t_1,\dots,t_n)\}) = R(\{f(f(t_1,\dots,t_n))\})$.

4.3. Example for the reuse in the local caching protocol

Below we present an example of reuse of information in the local caching protocol, in the case of the generation of the minimal model of an algebra of closed first-order terms under the constraint w represented by the set of terms $t \in \mathcal{T}$ whose length $L(t)$ is not more than 3.

Let g be a generator of arity 2. According to the computational model that takes into account the interdependencies of the tokens of the generators, we take as $\tilde{g}$ the following function mapping $\wp(\mathcal{T}^3)$ into itself:

$$\tilde{g}((t_1,t_2,t_3)) = \{((t_1,t_2,t_3) \in \mathcal{T}^3 \mid (t_1,t_2) \in \mathcal{T}^2 \text{ and } t_3 = g(t_1,t_2)\}.$$

It can be immediately verified that $\tilde{g}$ satisfies the properties of the approximating function described in [1] as: $\tilde{g} : \wp(\mathscr{D}^{n+1}) \mapsto \wp(\mathscr{D}^{n+1})$ where

$$\forall s \in \wp(\mathscr{D}^{n+1}): \quad \mathscr{G} \cap s \subset \tilde{g}(s).$$

Thus, the interdependency function computes an upper approximation of the part of any subset of $\mathscr{D}^{n+1}$ which is included in the graph of the generator's function. Being an approximation function, we assume that $\tilde{g}$ has the usual (anti-)closure properties, i.e., it is reductive, monotonous and idempotent, i.e.

$$\forall s, s_1, s_2, \quad \begin{cases} \tilde{g}(s) \subset s \\ \text{if } s_1 \subset s_2 \text{ then } \tilde{g}(s_1) \subset \tilde{g}(s_2) \\ \tilde{g}(\tilde{g}(s)) = \tilde{g}(s) \end{cases}$$

When $\tilde{g}$ is thus defined, the semantics of the token **ins**-k of rule (LC3) and **init** of rule (LC1) are well defined. Let us now analyze the flow of messages in the local caching protocol.

A broker is initially given the scope $w_0 = \{t \in \mathscr{T} \mid L(t) \leqslant 3\}$. Suppose it receives the request **requ**(w) where $w = \{t \in \mathscr{T} | L(t) \leqslant 2\}$. As explained before, the token **split**(w_0, w, w_1, w_2) produces two new sets w_1 and w_2. In our example, it immediately follows that $w_1 = w_0 \cap w = w$ and $w_2 = w_0 \cap \neg w = \{t \in \mathscr{T} | L(t) = 3\}$. Besides, the token **init**(w_1, s) will set $s = \{t \in \mathscr{T} \mid L(t) \leqslant 2\}$. By triggering the rule (LC1), a new agent will be spawned, whose scope will be held in s.

Let us focus our attention on this new agent. Suppose the tokens **seek**-1 and **free**-1 are already present in the agent's multiset of tokens; then, token **seek**-1 will activate the backdependency function on the broker-local constraint store s, establishing a constraint over the arguments. Notice that since $L(t) \leqslant 2$, the backdependency function will deliver a subrequest to a zero-ary or unary broker. This will produce a subrequest to another agent, encoded as the argument of the token **requ**. If the answer to this request is already present in the broker's local storage, encoded as the argument of the token **answer**, then rule (LC3) will be triggered, instantiating the first argument and generating the answer to the initial request w. The argument of the answer could instead have been generated by the broker itself during a past computation, and then broadcast to all existing brokers, as shown in rule (LC6).

4.4. *Comparing reuse in the two protocols*

Both the request–subrequest and the local caching protocol provide support for re-using already generated results. The main difference between the two protocols is that local caching allows a *structured* possibility of reuse, i.e., it automatically binds the new results of a generator with the results that are already known. By contrast, as illustrated in rule (RS6), the request–subrequest protocol offers only unstructured memory, namely it merely stores the results via the persistence of the token **process**(x) in the state of the requestee broker.

This difference in behavior is illustrated through the example of Fig. 2, where a broker associated with a generator g of arity 2 asks two other brokers a request of the same kind; these, in turn, ask two other brokers a subrequest to generate tokens a and b. The request–subrequest protocol allows a simple form of reuse, amounting to storing the partial results a and b in the requestee broker which can then re-deliver such results upon new requests without newly generating them. By contrast, the local caching protocol allows complete re-use of results, shifting all the computational load to the broker associated with the generator g. Generators for a and b send their answer to g to yield $g(a,b)$; this token is returned in a de-indexed fashion and reaches all existing brokers, that can then use it to build the new token $g(g(a,b), g(a,b))$. The local caching protocol's structured reuse dramatically reduces the number of brokers involved, even though the de-indexed message flow characteristic of local caching produces in some cases bundles of wasted messages.

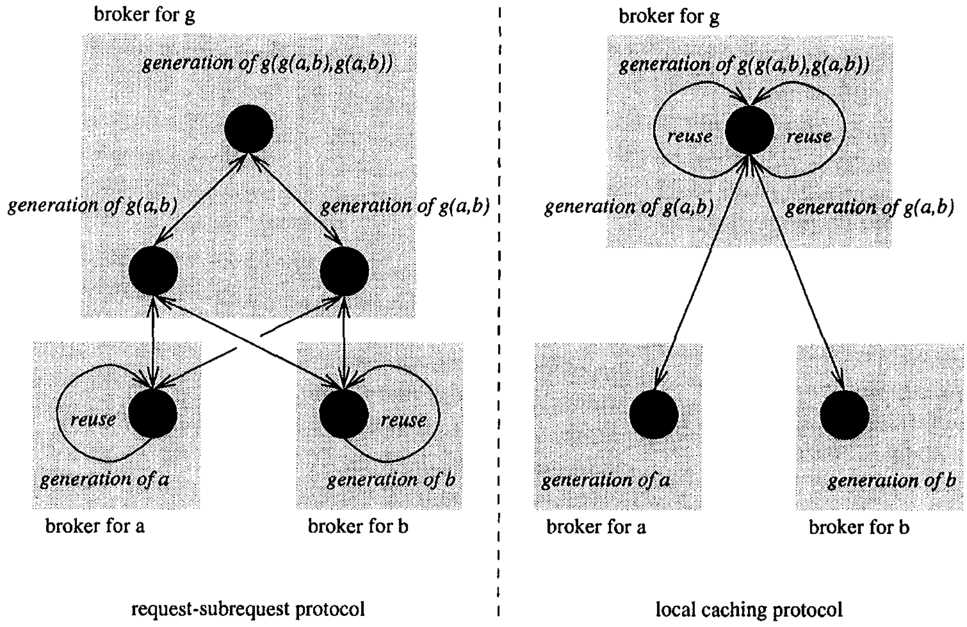

Fig. 2. Comparison of the message flow in the both protocols.

4.5. The average quantity of reuse

In the previous section, we have shown how a numerical value can be assigned to a given DPS problem expressed as a first-order term. This numerical value corresponds to the measure of reuse of information for that specific problem. To make practical sense of this measure of reuse, think of the following simple procedure:

- Define a criterion for the size of DPS problems in a given domain. For instance, we can choose to treat problems up to an arbitrary length n, where length is defined w.r.t. the encoding of a problem as a first-order term.
- Calculate the mean value of reuse μ_n for all problems of length less than or equal to n. This can be done by exhaustively computing the individual value of reuse for each term of length less than or equal to n, and then dividing the sum of all values by the corresponding number of terms.
- Use further criteria to measure additional costs in the two protocols. For instance, determine the average number BS_n of brokers spawned for all problems of length less than or equal to n in the given domain according the request–subrequest protocol. See [2] for techniques for determining these parameters.
- Now take the value $\mu_n * BS_n$ and, when faced with the question of which protocol to use for the treatment of a specific DPS problem of length less than or equal to n in the given domain, apply the following rule of thumb: if the individual value of reuse R for the specific problem is greater than $\mu_n * BS_n$ then opt for local caching, otherwise stick to request–subrequest.

An interesting problem arises when the average value of reuse cannot be computed because the criterion for fixing the size of a DPS problem does not determine a finite set. Take for instance the following recursive ambiguous context-free grammar G for generating boolean expressions, where with the symbol F we denote a generic propositional formula, with A an atomic formula, and with a, b, c and d we denote four particular atomic formulas:

$$\begin{aligned}
&S \Rightarrow F\\
&F \Rightarrow A \mid F \vee F \mid F \wedge F \mid \neg F \mid (F)\\
&A \Rightarrow a \mid b \mid c \mid d
\end{aligned}$$

Now take the criterion for defining the size of a problem to be given by the number n of occurrences of atomic formulas in a given formula; it is easy to see that for any $n \geqslant 1$ the number of such formulas is infinite. Yet the criterion is intuitively finite, under the assumption that the equivalence from classical logic $\neg\neg\alpha \leftrightarrow \alpha$ (law of double negation) holds. In fact, in this way any even number of occurrences of '$\neg$' can simply be ignored, and any odd number of occurrences of '$\neg$' can be reduced to a single occurrence of '$\neg$'. An easy variation of this case can be taken from text processing: in the LaTex text processing system, any sequence of occurrences of the operator {\mbox} can be reduced to a single occurrence of the same operator.

In the next section we formalize these arguments by showing how the criterion above for the language of boolean formulas can still lead to a computable average of reuse if the language is simplified by taking into account the semantic equivalences generated by the law of double negation. Section 4.8 provides a general set of simplifying techniques that can be applied to all DPS problems as formalized in the CBKB framework. Interestingly enough, this is done by reducing DPS problems to parsing problems; in a way, this generalizes to DPS the relationship between parsing and database querying as characterized in [30, 41].

On the other hand, the case of a boolean formula with multiple occurrences of the same subformula can be thought as an abstract version of many real-life examples, where the same piece of information is used over and over again in different places. Consider for example a hyper-document where e.g. all Xerox copiers are described in terms of specification, costs, suppliers, references, table of comparisons, other company's products, etc. Obviously, the same pieces of information (e.g., price list, vendor addresses, warranty text, Xerox logo and Xerox common information,...) are reused in most of the documents of this sort. In a hyper-document, typically, there are many links that point to these pieces of information. Likely reuse of this information during a browsing session suggests local caching as the appropriate protocol.

4.6. Language quotienting

Let $\tilde{\alpha}$ be an ambiguous propositional formula and M the set of parsed trees generated by the grammar G, we denote with $M_{\tilde{\alpha}}$ the set of parsed trees having $\tilde{\alpha}$ as leaf. The cardinality of $M_{\tilde{\alpha}}$ is given by the $(n-1)$th number of Catalan C_{n-1},[4] where n is the number of atomic formulas in $\tilde{\alpha}$, i.e.

$$|M_{\tilde{\alpha}}| = C_{n-1}.$$

This holds, since we can set a bijection between the binary trees with n leaves and the ways of putting parentheses to a word of length n. See [25] for details.

Example 7. Let $\tilde{\alpha}$ be $a \vee b \wedge c \vee d$. Then, we get $M_{\tilde{\alpha}} = \{((a \vee b) \wedge (c \vee d)),\ (a \vee (b \wedge (c \vee d))),\ (a \vee ((b \wedge c) \vee d)),\ (((a \vee b) \wedge c) \vee d),\ ((a \vee (b \wedge c)) \vee d)\}$, and hence $|M_{\tilde{\alpha}}| = C_3 = 5$.

Let M_n be the (infinite) set of well-formed propositional formulas with n atomic formulas yielded by the grammar G. Thus, the quantity

$$\mu_n = \frac{\sum_{\alpha \in M_n} R(\{\alpha\})}{|M_n|}$$

gives the average value for the measure of reuse R in the domain of well-formed propositional formulas of length n. According to its value, the quantity μ_n supplies us with a reliable criterion for deciding which of the two protocols is more suitable for implementing the CBKB model in the domain of the well-formed formulas without variables.

We now define the least equivalence relation $\simeq$ on the language $L(G)$ generated by the grammar G (which contains the following relation $\sim$) where α stands for a well-formed propositional formula:

$$((\neg)^n \alpha) \sim \alpha \quad \text{if } n \text{ is even}$$

$$((\neg)^n \alpha) \sim \neg\alpha \quad \text{if } n \text{ is odd}$$

Now consider

$$L(G)^{\neg} = \frac{L(G)}{\simeq}.$$

We can now simplify $L(G)$ to the quotient language $L(G)^{\neg}$ without loss of generality (by assuming the equivalence $\neg\neg\alpha \leftrightarrow \alpha$). Clearly in $L(G)^{\neg}$, for any n the set M_n of well-formed propositional formulas with n atomic formulas is finite, and this makes possible to compute the average value of reuse for formulas with n or less occurrences of atomic formulas. Furthermore, we can prove a proposition which shows an interesting relation between the individual measure of reuse R and the number of Catalan.

Proposition 8. *Let $L(G)^{\neg}_n$ be the set of well-formed formulas in $L(G)^{\neg}$ having n atomic occurrences, and let ϕ be the number of different atomic formulas in the grammar G, then $|L(G)^{\neg}_n| = C_{n-1}\phi^n 2^{3n-2}$.*

[4] $C_n = \frac{(2n)!}{n!(n+1)!}$.

Proof. Let T be a binary tree representing the parse of a formula with n atomic occurrences. Obviously, the number of nodes of T is $2n-1$, and the total number of inner nodes (i.e., nodes that are not leaves) is $(2n-1)-n=n-1$. Now there will be only two ways of generating a well-formed propositional formula from the tree T. Briefly,

(i) since every inner node is binary, and since a propositional formula has only two binary connectives ('$\vee$' and '$\wedge$'), there are 2^{n-1} ways of placing these connectives to the formula.

(ii) since the double negation is absorbed by a blank, there are $\sum_{i=0}^{2n-1}\binom{2n-1}{i}$ $(=2^{2n-1})$ ways of adding the negation symbol ($i=0$ represents the case where no negation symbol is set).

It is easy to see that there is a bijection between the binary trees generated in this way and the well-formed propositional formulas of $L(G)^{\neg}$. Since there are C_{n-1} different binary trees and only ϕ^n different dispositions of ϕ atomic formulas into n atomic occurrences, the claim follows. $\square$

By Proposition 8, we can now capture the quantity μ_n defining the average value for the measure of reuse, with the following closed formula

$$\mu_n=\frac{\sum_{\alpha\in M_n}R(\{\alpha\})}{C_{n-1}\phi^n 2^{3n-2}}.$$

4.7. Discussion

While local caching would seem in general appropriate for parsing boolean expressions with the grammar above, there are cases where request–subrequest is instead preferable.

As shown in Fig. 3, in the generation of the parse tree for the formula $\alpha=((a\wedge b)\wedge(c\vee d))$ where only little reuse of previous partial results is possible, the request–subrequest protocol is the right choice. As stated in the grey box of the '*and*'-broker's generator for the partial result $(c\vee d)$, the possible reuse of information is low, and as a consequence the use of the local caching protocol is not recommended.

The local caching protocol is particularly appropriate when parsing formulas with multiple occurrences of the same subformulas, as in the case of the formula $\alpha=((a\wedge a)\wedge(a\wedge a))$. This aspect is put in evidence in Fig. 4 by showing the traffic among brokers agents during the generation of α. Solid lines represent messages which are effectively used in constructing the token, while dashed lines stand for broadcast messages which could possibly get wasted.

The broker agent in charge of a sends a to the '*and*'-broker's generator which binds (and reuses) this argument to generate $(a\wedge a)$; finally, this new token is again reused and bound twice for the generation of the token $((a\wedge a)\wedge(a\wedge a))$. This is a significant case of reuse, obtained at the price of a bundle of possibly wasted messages.

Fig. 5 focuses on how the number of broadcast messages grow with increasing the number of brokers. It displays a rich bundle of messages, very few of which (printed

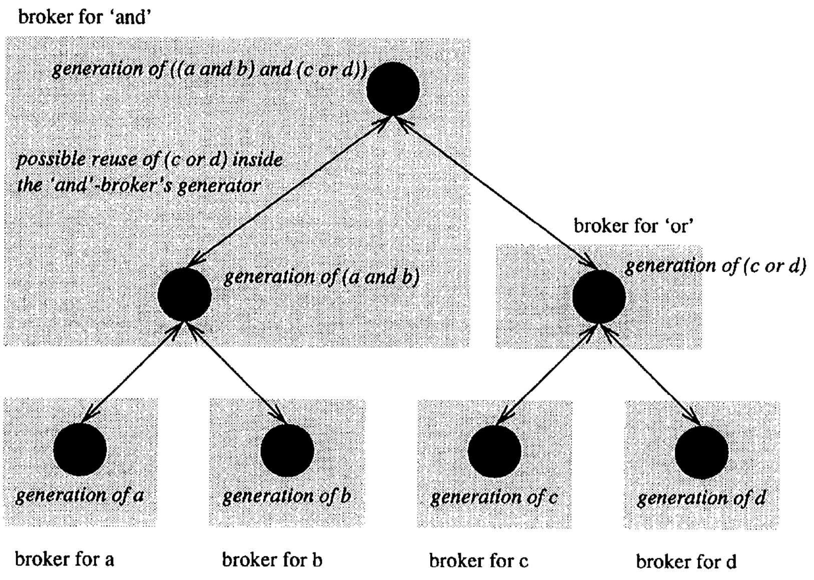

Fig. 3. Message flow during parsing of a formula with possible reuse.

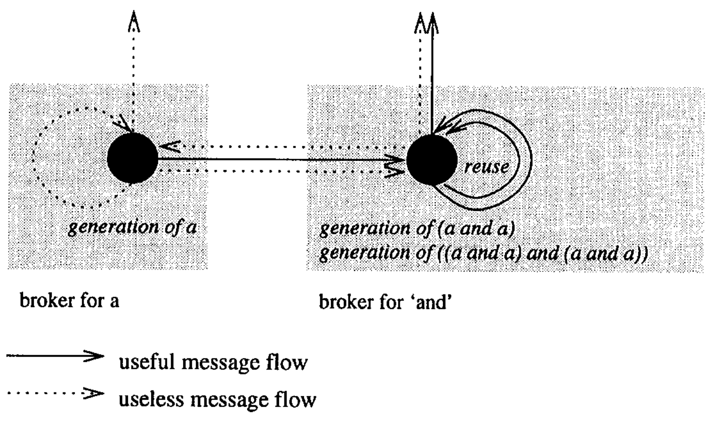

Fig. 4. Message flow in the local caching protocol with two broker agents.

in solid lines) are actually used for the generation of the token $((a \vee b) \wedge (a \vee b))$. Here the '*and*'-broker's generator can benefit of the deindexation of the answer because both pools containing the tokens **free**-1 and **free**-2 receive the partial result $(a \vee b)$, causing the generator to yield the answer.

4.8. Algebraic analysis

We now generalize the parsing example of Section 4.6 by showing how DPS problems can be mapped into grammars. The computation of the average value of reuse

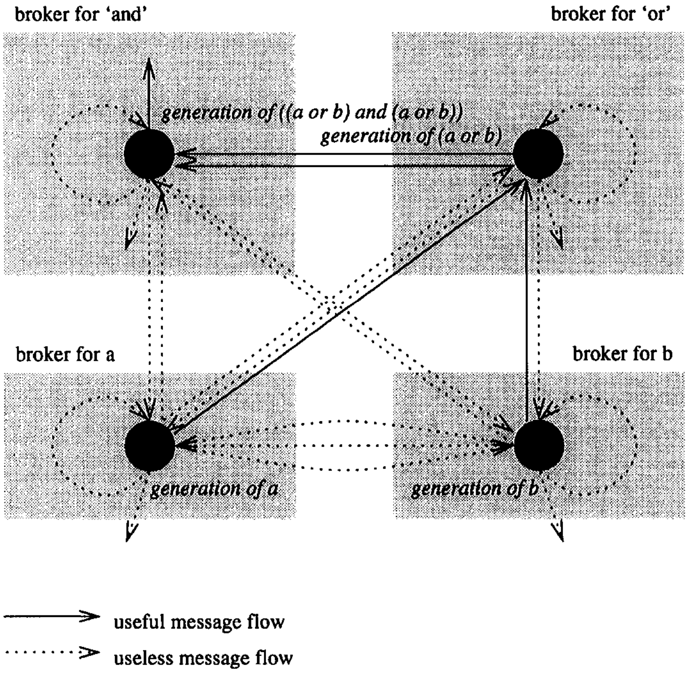

Fig. 5. Message flow in the local caching protocol with four broker agents.

can then take advantage of Schützenberger's techniques [36, 37] based on algebraic and transcendental generating functions. These techniques allow the analytical computation of combinatorial quantities relative to the language generated by a grammar G, allowing a full (although possibly asymptotic) analysis of the reuse. More specifically, these techniques support the calculation of the mean value of reuse μ_n: again, this is done by exhaustively computing the individual value of reuse for each word in the language of length less than or equal to n, and then dividing the sum of all values by the number of words up to length n. The number of words up to length n is provided as part of the generating function for G.

In two steps, we elaborate on generalizing the heuristic presented before:
- first, we convert a Herbrand domain into a grammar G, and
- then we apply some well-known algebraic techniques to evaluate the measure of reuse for the terms in the Herbrand domain, i.e., we calculate the mean value of reuse μ_n for all words in $L(G)$.

Let us focus on the first step. Given a signature Σ, the algebra of closed terms T_Σ can be generated by a suitable algebraic operator T by means of a minimum fixed-point procedure. It can be seen that $T_\Sigma = \bigcup_{n=1}^{\infty} T^n(\emptyset)$. This is a worthy, although informal, argument for the recursive enumerability of T_Σ, so we are able to state the following proposition.

Proposition 9. *Let T_Σ be an algebra of closed terms, then there exists a grammar G such that the language generated by G is $L(G) = T_\Sigma$.*

The grammar generating the closed terms algebra T_Σ could be given by the quadruple

$$G = \langle \{term\}, \Sigma_0, \{term \mapsto f(term, \ldots, term)\}, \{term\} \rangle,$$

where $\{term\}$ is the set of non-terminal symbols, Σ_0 is the set of terminal symbols, and where $f \in \Sigma$. Since this is an unambiguous grammar, we can apply to G all the methods and tools derived from the theory of algebraic languages. In order to simplify the algebraic analysis from the point of view of reuse, we will quotient the Herbrand domain according to the symmetric transitive closure of a rewrite-rules system with roots in the semantic properties of the domain. Then we derive once again the generating grammar for the quotient of $L(G)$ and reapply the algebraic methods to the simplified domain, increasing computational complexity. The main problem is to keep the quotient of a generated language as *computable* as possible. From the point of view of computability theory, this amounts to keeping the quotient set at least *recursively enumerable*. For this property to be meaningful, one must identify an equivalence class with a particular element, namely a normal form with respect to an equational theory E. Thus we can set an injective map I between T_Σ/E and T_Σ such that $I([t]) = nf_E(t)$. Then we can embed T_Σ/E into the free monoid $(\Sigma)^*$ which is codifiable in $\mathbb{N}$.

This creates an interesting connection between the results of our approach and the decidability problem of equational first-order theories. Indeed the following proposition is an immediate consequence of Birckhoff's basic result on equational logic.

Proposition 10. *Let T_Σ be an algebra of closed terms and let E be an equational first-order theory over Σ. If E is decidable then there exists a grammar such that $L(G) = T_\Sigma/E$.*

Example 11. Consider the algebra of well-formed propositional formulas of the preceding section and let $E = \{\neg\neg\alpha = \alpha\}$. This set is trivially decidable, so the quotient algebra modulo E can still be generated by a grammar (it will be the set $\{\alpha, \neg\alpha$ where α is a or $b\}$).

After the discussion of the conversion of a Herbrand domain into a grammar G, we now use Schützenberger's method to simplify the combinatorial complexity of the language and to calculate the mean value of reuse μ_n for all words in $L(G)$.

Algebraic languages can be associated with an algebraic function f_L, such that, if $f_L(x) = \sum_{i=0}^{\infty} w_i x^i$ for some subset of the real numbers, then w_i gives the number of words of the language L whose length is i. Consequently, we can use the w_i in our heuristic to give a more general measure for μ_n. Having a measure of reuse R for an individual word and letting M_n represent the set of words of L up to length n, the mean value of reuse μ_n for all words in M_n is given by

$$\mu_n = \frac{\sum_{m \in M_n} R(\{m\})}{\sum_{i=0}^{n} w_i}.$$

Example 12. Let $L(G)$ be the language generated by the following grammar

$$F \rightarrow \neg F \mid a \mid b \tag{1}$$

Applying Schützenberger's method to such a grammar, we get the following generating function $f_{L(G)}(x) = 2x/(1-x)$ which, in the interval $(-1, 1)$ is the functional limit of the power series $\sum_{i=1}^{\infty} 2x^i$. Therefore, for each integer i there will be two words of length i.

Now let us quotient the language $L(G)$ with respect to the equivalence relation generated by equational system $E = \{\neg\neg x = x\}$. This is a trivial decidable equational system where the normal form of a term is easily derivable; thus, according to Proposition 10, we expect $L(G)/E$ to be a language generated by a grammar. This holds since the following grammar G', defined by

$$F \rightarrow \neg G \mid G \tag{2}$$

$$G \rightarrow a \mid b \tag{3}$$

generates the language $L(G)/E$.

If we now apply the Schützenberger method to $L(G')$ we find a corresponding generating function $f_{L(G')}(x) = 2x + 2x^2$ whose power-series expansion is immediate. Again, the central idea of this method is that the "quotienting" of a grammar-generated language actually trims the recursive structure of the production rules. This is reflected in the Schützenberger method by a *lower degree* or, more generally, by a simpler algebraic expression of the generating function.

5. Related work

This paper relates directly to previous results published in [1] where the notion of Constraint Based Knowledge Broker was first introduced. More recent work on the CBKB model [2] has provided a number of complexity results concerning the number of agents needed in the request-subrequest protocol, and the number of messages sent, both in the form of requests and answers. These results have directly influenced the idea of investigating different protocols, so as to handle a variety of problem domains and system architectures.

For a conceptual characterization of Distributed Problem Solving see [17, 18]. A number of different cooperation strategies between agents have been proposed, ranging from strongly hierarchical master–slave relationship (as in the CBKB framework using the request–subrequest protocol), to the less hierarchical Contract-Net [39], to the sharing of common goals. Both the Contract-Net protocol and the CBKB framework apply structured messages to model agent interaction. The Contract-Net protocol supports an application protocol for communication between problem solving agents and facilitates distributed control during the problem solving effort. Special emphasis is put on finding those agents which are eligible for solving the created subproblems, and on the negotiation between agents for information exchange with respect to subproblem descriptions, required agent capabilities, and subproblem solutions [15]. The Contract-Net protocol can be seen as a particular instance of a CBKB-system with two broker roles, namely a manager and a set of bidders. The manager tries to locate a contractor among the bidders to solve a particular problem. As a result, the manager is equipped with a very specific generator for the bid selection. The protocol is negotiation-based along five different phases. In the first phase, the manager announces the problem by sending a request for bids to the set of bidders, e.g. by sending a request constraining the problem domain and constraining the capabilities a potential contractor must have. In the next phase, the bidders check the problem constraints and propagate their bids, e.g. by tailoring the problem domain to their scope. The next phase comprises the selection of a contractor by the manager. Upon receipt of answers the manager invokes its generator. The result generation is quite simple. If the bid satisfies the initial constraint, a potential contractor is found. Among all potential contractors, a "best" (according to some problem-dependent criteria) potential contractor is selected as contractor. In the fourth phase, the problem solving task is transmitted to the contractor. The final phase concludes with the problem solving itself.

More recently, a cooperative information gathering approach using a multi-agent system based on DPS has been illustrated in [29]. Additional relevant literature can be found in [26]. Cooperative Solutions for Constraint Satisfaction Problems have been addressed in [13] where quantitative measures (e.g. time to solution distributions) based on experiments are given. However, so far, DPS has been missing a real computational model, and our work can be seen as a first contribution to fill this gap.

Concurrent Constraint Programming – the background for our CBKB model – introduced in [32] (see also [35]), applies the insights gained through Constraint Logic Programming [21, 23] to concurrent programming. A process transition is controlled by the presence of a constraint in the constraint store, or, more precisely, by its entailment from the constraint store. This enforces a strictly monotonous view of constraints, which has been partially relaxed in [14, 16, 34] (for a similar kind of relaxation, see also [22]). This approach does not make the distinction between two different uses of constraints: for local problem solving and for communication (and hence for Distributed Problem Solving). By contrast, this distinction appears clearly in the CBKB model, where communication of constraints is achieved via the global flow of messages while each individual broker agent encapsulates its own local problem solver. Thus, our approach can also be seen as a contribution to merging the two paradigms of Concurrent Constraint Programming and of Coordination Languages [11], where a coordination language is seen as the layer for gluing together independent software components to perform a global activity.

An early contribution in this direction (dating well before the programming paradigms of both Concurrent Constraint Programming and Coordination Languages) can be found in [28], where a framework is proposed in which a set of constraints is solved by a system of cooperative, distributed, specialized constraint solvers which exchange their relevant results. This framework is based on a simple broadcast mechanism for constraint propagation, without further investigations on refinements, variations and complexity of this basic protocol.

6. Conclusions and future developments

In this paper, we have defined a design space of constraint-based communication protocols for Distributed Problem Solving (DPS) by formally defining two opposite extremes of information sharing. One extreme captures the hypothesis of minimal information reuse: all generated information is potentially made available, but can be delivered only in response to specific requests. The opposite case is based on the idea of maximal information reuse: all generated information is immediately broadcast to all agents.

We have formally introduced the notion of *quantity of reuse* of information to obtain a heuristic to decide between the two protocols, as well as a basic metric over the space of protocols that lie in between the two extremes. These other protocols define intermediate attitudes towards information, in between the hypotheses of minimal and maximal reuse, and in general will be the more useful for most practical applications. We need flexible ways for expressing such protocols, and for mixing them freely in the overall solution of a problem. In addition, we need ways to guess the right protocol, or the right *melange* of protocols, for specific problems. This calls for contributions from such diverse fields as programming linguistics, learning and simulation.

Another important area of investigation is extending the choice criterion for protocols to take into account not only the type of DPS problem but also the characteristics of the underlying system architecture. For instance, transputers seem particularly well-suited for the request–subrequest protocol, as the cost of communication is low and there are strong limitations on storage. By contrast, for distributed architecture based, say, on networks of workstations, local caching would appear as the most appropriate solution.

Acknowledgements

We would like to thank Natalie Glance and the anonymous referees for their helpful comments on earlier versions of this paper.

References

[1] J.-M. Andreoli, U.M. Borghoff, R. Pareschi, Constraint-based knowledge brokers, in: H. Hong (Ed.), Proc. 1st Internat. Symp. on Parallel Symbolic Computation (PASCO'94), Hagenberg/Linz, Austria, 1994, Lecture Notes Series in Computing, Vol. 5, World Scientific, Singapore, pp. 1–11.

[2] J.-M. Andreoli, U.M. Borghoff, R. Pareschi, The constraint-based knowledge broker model: Semantics, implementation and analysis, J. Symbol. Comput. 21 (4) (1996) 635–667.

[3] J.-M. Andreoli, U.M. Borghoff, R. Pareschi, J.H. Schlichter, Constraint agents for the information age, J. Universal Comput. Sci. 1 (12) (1995) 762–789. Electronic version available at http://www.iicm.edu/jucs.

[4] J.-M. Andreoli, R. Pareschi, Communication as fair distribution of knowledge, in: Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91), Phoenix, AZ, November 1991, ACM SIGPLAN Notices 26 (11) 212–229.

[5] F. Arcelli, U.M. Borghoff, F. Formato, R. Pareschi, Tuning constraint-based communication in distributed problem solving, in: Proc. 1st Internat. Workshop on Concurrent Constraint Programming (CCP'95), Venice, Italy, May 1995.

[6] J.-P. Banâtre, D. Le Métayer, The GAMMA m and its discipline of programming, Science of Computer Programming 15 (1990) 55–77.

[7] G. Berry, G. Boudol, The chemical abstract machine, in: Proc. 17th ACM SIGACT/SIGPLAN Annual Symp. on Principles of Programming Languages, San Francisco, CA, 1990, pp. 81–94.

[8] U.M. Borghoff, LOKIT: A toolkit for building distributed collaborative applications, Rank Xerox Research Centre, Grenoble Lab., France, Technical Report CT-002, May 1994.

[9] U.M. Borghoff, R. Pareschi, H. Karch, M. Nöhmeier, J.H. Schlichter, Constraint-based information gathering for a network publication system, in: B. Crabtree, N. Jennings (Eds.), Proc. 1st Internat. Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK, 1996, The Practical Application Company Ltd., Blackpool, UK, pp. 45–59.

[10] U.M. Borghoff, J.H. Schlichter, On combining the knowledge of heterogeneous information repositories, J. Universal Comput. Sci. 2 (7) (1996) 515–532. Electronic version available at http://www.iicm.edu/jucs.

[11] N. Carriero, D. Gelernter, How To Write Parallel Programs. A First Course, MIT Press, Cambridge, MA, 1990.

[12] K.L. Clark, PARLOG and its applications, IEEE Trans. Software Engrg. 14 (12) (1988) 1792–1804.

[13] S.H. Clearwater, B.A. Huberman, T. Hogg, Cooperative solution of constraint satisfaction problems, Science 254 (1991) 1181–1183.

[14] P. Codognet, F. Rossi, NMCC: Constraint enforcement and retraction in CC programming, in: Proc. 12th Internat. Conf. on Logic Programming (ICLP'95), Kanagawa, Japan, 1995, MIT Press, Cambridge, MA.

[15] R. Davis, R.G. Smith, Negotiation as a metaphor for distributed problem solving, Artif. Intell. 20 (1) (1983) 63–109.

[16] F.S. de Boer, J.N. Kok, C. Palamidessi, J. Rutten, Non-monotonic concurrent constraint programming, in: D. Miller (Ed.), Proc. Internat. Logic Programming Symp. (ILPS'93), 1993, MIT Press, Cambridge, MA, pp. 315–334.

[17] K.S. Decker, E.H. Durfee, V.R. Lesser, The evaluation of research in cooperative distributed problem solving, in: M.N. Huhns, L. Gasser (Eds.), Distributed Artificial Intelligence, 1989, Research Notes in Artificial Intelligence, Vol. 2. Pitman/Morgan Kaufmann, San Mateo.

[18] E.H. Durfee, V.R. Lesser, D.D. Corkill, Trends in cooperative distributed problem solving, IEEE Trans. on Knowledge and Data Engrg. 1 (1) (1989) 63–83.

[19] J.-Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1987) 1–102.

[20] S. Haridi et al., Concurrent constraint programming at SICS with the Andorra Kernel Language, in: P. Kanellakis, J.-L. Lassez, V. Saraswat (Eds.), Proc. 1st Workshop on Principles and Practice of Constraint Programming, Providence, RI, 1993.

[21] P. v. Hentenryck, Constraint satisfaction in logic programming, Logic Programming Series, MIT Press, Cambridge, MA, 1989.

[22] M. Henz, G. Smolka, J. Würtz, Object-oriented concurrent constraint programming in Oz, in: P. v. Hentenryck, V. Saraswat (Eds.), Principles and Practice of Constraint Programming, MIT Press, Cambridge, MA, 1995, pp. 27–48.

[23] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: Proc. 14th ACM SIGACT/SIGPLAN Annual Symp. on Principles of Programming Languages, Munich, Germany, January 1987, ACM, New York, pp. 111–119.

[24] S. Janson, S. Haridi, Programming paradigms of the Andorra kernel language, in: V. Saraswat, K. Ueda (Eds.), Proc. Internat. Logic Programming Symp. (ILPS'91), San Diega, CA, 1991, MIT Press, Cambridge, MA.

[25] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[26] S. Lander, V.R. Lesser, Customizing distributed search among agents with heterogeneous knowledge, in: T.W. Finin, C.K. Nicholas, Y. Yesha (Eds.), Proc. 1st Internat. Conf. on Information and Knowledge Management, Baltimore, MD, November 1992, Springer, Berlin.

[27] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G.A. Agha, A. Yonezawa, P. Wegner (Eds.), Research Directions in Concurrent Object Oriented Programming, MIT Press, Cambridge, MA, 1992, pp. 314–390.

[28] G. Nelson, D.C. Oppen, Simplification by cooperating decision procedures, ACM Trans. Programming Languages Systems 1 (2) (1979) 245–257.

[29] T. Oates, M.V.N. Prasad, V.R. Lesser, Cooperative information gathering: A distributed problem solving approach, Dept. Computer Science, Univ. Massachusetts, Amherst, MA, Technical Report TR-94-66, 1994.

[30] F.C.N. Pereira, D.H.D. Warren, Parsing as deduction, Proc. 21st Annual Meeting of the Association for Computational Linguistics, MIT, Cambridge, MA, 1983.

[31] G.-C. Roman, H.C. Cunningham, Mixed programming metaphors in a shared dataspace model of concurrency, IEEE Trans. Software Engrg. 16 (12) (1990) 1361–1373.

[32] V.A. Saraswat, Concurrent constraint programming languages, PhD thesis, Carnegie-Mellon University, Pittsburg, PA, 1989.

[33] V.A. Saraswat, R. Jagadeesan, V. Gupta, Foundations of timed concurrent constraint programming, Proc. 9th IEEE Symp. on Logic in Computer Science, Paris, France, 1994, IEEE Comp. Soc. Press, Los Alamitos, CA, pp. 71–80.

[34] V.A. Saraswat, P. Lincoln, Higher order linear concurrent constraint programming, Technical Report, Xerox Palo Alto Research Center, Palo Alto, CA, 1992.

[35] V.A. Saraswat, M. Rinard, P. Panangaden, Semantic foundations of concurrent constraint programming, in: Proc. 18th ACM SIGACT/SIGPLAN Annual Symp. on Principles of Programming Languages, Orlando, FL, January 1991, ACM, New York, pp. 333–352.

[36] M.P. Schützenberger, Sur une variante des fonctions sequentielles, Theoret. Comput. Sci. 4 (1977).

[37] M.P. Schützenberger, The critical factorization theorem, in: M. Lothaire (Ed.), Combinatorics on Words, Encyclopedia of Mathematics and its Applications, Vol. 17, Addison-Wesley, Reading, MA, 1983.

[38] E. Shapiro, The family of concurrent logic programming languages, ACM Comput. Surveys 21 (3) (1989) 413–510.

[39] R.G. Smith, The contract net protocol: High-level communication and control in a distributed problem solver, IEEE Trans. Comput. C-29 (12) (1980) 1104–1113.

[40] K. Ueda, Guarded horn clauses, in: E. Wada (Ed.), Logic Programming, 1986, Lecture Notes in Computer Science, Vol. 221, Springer, Berlin, pp. 168–179.

[41] L. Vielle, Recursive axioms in deductive databases: The query–subquery approach, in: L. Kerschberg (Ed.), Proc. 1st Conf. on Expert Database Systems, Benjamin/Cummings, Menlo Park, CA, April 1986.

[42] J.-M. Andreoli, U.M. Borghoff, R. Pareschi, Signed feature constraint solving, Proc. 3rd Internat. Conf. on the Practical Application of Constraint Technology (PACT'97), London, UK, April 1997, The Practical Application Company Ltd., Blackpool, UK.

[43] U.M. Borghoff, P.-Y. Chevalier, J. Willamowski, Adaptive refinement of search patterns for distributed information gathering, in: A. Verbraeck (Ed.), Proc. Internat. Conf. EuroMedia/WEBTEC '96, London, UK, December 1996, The Society for Computer Simulation, San Diego, CA, pp. 5–12.