In *the Journal of Symbolic Computation, 21:4, pp 635–676 (1996)*

The Constraint-Based Knowledge Broker Model: Semantics, Implementation and Analysis

J.M. Andreoli, U.M. Borghoff and R. Pareschi
Rank Xerox Research Center, Grenoble, France

Abstract

Imagine distributed knowledge processing with autonomous activities and decentralized control where the handling of partial knowledge does not result in unclear semantics or failure-prone behavior. In this paper, a modular approach is taken where concurrent agents, called *constraint-based knowledge brokers* (CBKBs), process and generate new knowledge in the presence of partial information. CBKBs apply constraint solving techniques to the domain of information gathering in distributed environments. Constraints are exploited to allow partial specification of the requested information, and to relate information requests from multiple sources.

We present a mathematical model where the semantics of the knowledge system is described using a standard fixed-point procedure. A basic execution model is then provided. This is incrementally refined to tackle problems of inter-argument dependencies (that arise with constraints relating information requests from different sources), of knowledge reuse inside the knowledge generators, and of recursion control. The model refinements are illustrated by a detailed complexity analysis in terms of the number of agents needed and of the number of messages sent, distinguished by requests and answers of the involved broker agents. A detailed example shows a broker-based chart-parser for unification grammars with feature terms implemented using CBKBs. As we shall point out, this apparently abstract example can be easily generalized to full-fledged information gathering.
Keywords: knowledge brokers, concurrent constraint programming, recursion control, knowledge reuse, distributed processing, information gathering, feature-unification grammar.

1 Introduction

This paper[1] combines basic research in two important areas of constraint-based programming of autonomous activities: firstly, it provides a mathematical model based on a fixed-point semantics for a multiagent characterization of distributed search and, secondly, it introduces a corresponding execution model that is refined step-wise to meet different areas of interest and to solve problems such as inter-argument dependencies, reuse of previously computed results, and recursion control. The mathematical model provides an algebra of information tokens to formalize a world of distributed knowledge. Correspondingly, proven techniques from symbolic computation, suitable for solving traditional algebraic problems, are here adapted to the task of symbolic manipulation of information tokens. The main contribution of this combination of results is in bringing constraint solving techniques to the domain of information gathering in distributed environments. Constraints are exploited to allow partial specification of the requested information, and to relate information requests from multiple sources.

In distributed systems with decentralized control, autonomous activities for knowledge processing face the problem of how to react efficiently when only partial knowledge is provided. Assume, for example, a query to some service providers where the request has not been fully specified. Partial knowledge on the side of the service providers should not result in the abortion of the entire query. Instead, some knowledge based on the available information should be derived. When some threshold of information is reached, some action should be triggered to acquire more knowledge from some other source of information and, eventually, answer the request.

Another example concerns the merging of information, e.g., during the preparation of a multi-author document. Although the final document should consist of all the authors' submissions, intermediate documents (where only some of the authors have actually provided their material) can be used for further processing, such as reviewing. In the reviewing process, new knowledge (the reviewed document) is again derived using partial information.

[1] Extended version of a paper [2] presented at the 1st Int. Symp. on Parallel Symbolic Computation (PASCO'94), Hagenberg/Linz, Austria, 26–28 Sep. 1994.

There are many examples where distributed processing can effectively exploit knowledge that is not fully available. This kind of processing, however, may sometimes result in unclear semantics of the overall model, concerning the exact meaning of a piece of information extracted from partial knowledge. In our examples, we can ask, for instance, what a reply to a partially specified query or the status of a "partial paper" review. Can we increase the ability to avoid unnecessary/redundant processing when previously neglected information is finally provided?

Constraint-based knowledge brokers (CBKBs) are potential candidates to tackle these problems. The purpose of the paper is to provide concise information on how to model these brokers and how to implement them. Our work merges several lines of research: the request/subrequest model of knowledge processing and its refinements (in particular the lemma caching techniques), dynamic programming, multi-agent systems and constraint-based programming, in particular Constraint Logic Programming (CLP) and Concurrent Constraint Programming (CCP).

Organization of the Paper

The paper is organized as follows. Sect. 2 introduces our mathematical model to describe knowledge brokers with a well-defined semantics. The traditional fixed-point procedure is implemented by means of the rule-based coordination language LO. Any language could obviously have been used for this purpose, however, LO offers in a single environment based on very few primitives, a number of powerful constructs which are well-adapted to our problem (e.g. broadcast and cloning). LO could be termed a "reduced primitive set language" by analogy with RISC. A brief introduction to LO is given. In Sect. 3, we discuss a first enhancement of the basic model. We extend the current knowledge of each broker by adding a back-dependency function to its generator. Sect. 4 deals with constraint-based knowledge brokers to solve the problems of inter-argument dependencies, reuse of previous (i.e. already calculated) results and, most importantly, recursion control. From there, we further extend the knowledge of the brokers by adding the notion of *threshold* of information. The main model refinements are illustrated by a detailed complexity analysis w.r.t. the number of agents needed in the massively parallel schemes and the number of messages sent, distinguished by requests and answers of the involved broker agents. Sect. 5 shows an example where a broker-based chart-parser for unification grammars with feature terms is introduced using the adapted constraint-based knowledge brokers model. In Sect. 6, we briefly describe previous work related to our approach, e.g. CLP and CCP. Finally, Sect. 7 concludes by discussing future work.

2 Knowledge Brokers

2.1 The Theoretical Framework

We consider here a class of systems based on the notion of *generators*. Given an abstract domain of values $\mathcal{D}$, representing tokens of knowledge, a generator is a mapping $\mathcal{D}^n \mapsto \wp(\mathcal{D})$, which produces new tokens from existing ones. A set of generators identifies the class of subsets of the domain which are stable by these generators:

Definition 1 *Let E be a subset of the domain and Γ a set of generators. E is Γ-stable if and only if*

$$\forall g \in \Gamma,\ \forall a_1, \ldots, a_n \in E,\ g(a_1, \ldots, a_n) \subset E.$$

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 traditional fixed-point procedure is available to compute the minimal model. The core of the procedure is given by the mapping:

$$\begin{array}{l} T : \wp(\mathcal{D}) \mapsto \wp(\mathcal{D}) \\ \forall E \in \wp(\mathcal{D}), \\ T(E) = \bigcup_{\substack{g \in \Gamma \\ a_1, \ldots, a_n \in E}} g(a_1, \ldots, a_n). \end{array}$$

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

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

T thus provides a way of incrementally computing the minimal stable set, by computing the sequence $(T^n(\emptyset))_{n \in \mathcal{N}}$, given by

$$E_0 = \emptyset \qquad E_{n+1} = T(E_n).$$

<*rule*>	=	<*trigger*> <*bcast*> '`<>-`' <*body*>
<*trigger*>	=	<*token*>
	|	<*token*> '`@`' <*trigger*>
<*bcast*>	=	<>
	|	<*bcast*> '`@ ^`' <*token*>
<*body*>	=	<*token*>
	|	<*body*> '`@`' <*body*>
	|	<*body*> '`&`' <*body*>
	|	'`#t`'
	|	'`#b`'

Table 1: Syntax of LO rules.

2.2 Succinct Introduction to LO

LO is a rule-based coordination language. An LO system consists of a set of agents which evolve concurrently. The state of each agent is represented by a pool of tokens. An LO program is a set of rules which operate on the state of the agents. Rules have the syntax as given in Table 1. Tokens in agents states are tuples of values, prefixed with a predicate name. In a rule, a <*token*> consists of a tuple of variable names prefixed with a predicate name. Variables in rules start with an upper-case letter.

A <*rule*> of the form

```
p(X) @ q(X,Y) @ ^r(Y) <>- BODY
```

enables a state transition on agents which contain the <*trigger*> of the rule (in this case, two tokens of the form p(a) and q(a,b)), in their state. When the transition is performed on such an agent, these two tokens are *removed* from the state of the agent; then the token r(b), specified in the <*bcast*> part of the rule, is *broadcast* to (i.e. added to the state of) all the *other* agents; finally, the body of the rule (`BODY` in which `X` is replaced by a and `Y` by b) is *executed* in the agent.

Thus, the left-hand side of a rule combines: a precondition to its application (trigger), a local, destructive, state modification (atomic removal of the trigger), and a global communication (by broadcast).

A <*body*> of the form

```
p(a) @ q(b,c) & r(d)
```

is executed as follows: the agent is cloned (this is indicated by the cloning operator `&`); the tokens `p(a)` and `q(b,c)` (on one side of the cloning operator) are added to the state of one of the clones, while the token `r(d)` (on the other side) is added to the state of the other clone. There might be no cloning, in which case the tokens of the body are simply added to the state of the agent. The body of a rule can also be reduced to the operator `#t`, meaning the termination of the agent, or the operator `#b`, meaning no operation is to be performed.

The right-hand side of a rule therefore enables dynamic creation and termination of agents, as well as local, incremental state modification (adjunction of tokens).

Instantiation of the variables of the trigger of an applicable rule, is obtained by pattern matching with the corresponding tokens in the state of the agent it applies to. Other variables (which do not occur in the trigger) are instantiated with new *unique* constants, thus providing a basic name service. These constants can then be used as addresses to implement address-based communication using broadcast (see [5, 8] for more details).

Finally, each agent can be equipped with specialized pool handlers, declared in programs through class declarations. The class declarations give a lucid overview of the input/output behavior of the pool handlers. They have the form given in Table 2. A <*classdec*> of the form

class `C` **holds**
    p(x,y,z) $\{x,y \mapsto z\}$
    q(x) $\{\mapsto x\}$.

represents a pool handler, which handles tokens of the form p(a,b,c) and q(d). The meaning of the signatures can informally be described as follows.

- The first signature $\{x,y \mapsto z\}$ indicates that, when a rules tries to extract a token of the form `p(X,Y,Z)` (the first pattern above), `X` and `Y` must already have been instantiated with values, while `Z` gets instantiated by the extraction.

<*classdec*>	=	<*name*> 'holds' <*patterns*> '.'
<*patterns*>	=	<*pattern*>
	|	<*pattern*><*patterns*>
<*pattern*>	=	<*token*> <*signature*>

Table 2: Syntax of class declarations.

- Similarly, the second signature $\{\mapsto x\}$ indicates that, in `q(X)`, `X` gets instantiated by the extraction.

Such signatures enforce a partial sequentialization in the evaluation of the trigger of the rules.

This brief overview of the LO language will be sufficient to understand the algorithms given in the remainder of this paper. The interested reader will find more details in [1, 3, 4].

2.3 An Agent-Based Implementation

We provide a modular implementation of the computation of the minimal model as described in Sect. 2.1. More precisely, modularity is obtained by considering each generator as an autonomous activity, called a *knowledge broker*. The activity of the overall system is obtained by simple coordination of the activities of the individual brokers. In particular, we assume that each broker knows nothing about the other brokers, and we only wish to provide a coordination layer which synchronizes them. In this section, we provide a straightforward implementation of the basic fixed-point procedure, using the coordination language LO. Extensions and refinements of the basic model are proposed in the next sections.

The constraint-based knowledge brokers could be implemented in any languages in particular, constraint-based languages. However, by using LO, we can exploit specific strong features such as broadcast, cloning, persistence of tokens until they are used, and atomic removal of tokens after use.

2.3.1 Implementing Knowledge Brokers with LO

Consider a broker, associated with a generator g of arity n. It is represented in LO as a separate agent. We assume that the broker has an internal activity capable of transforming, for each n-tuple of values $a_1, \ldots, a_n$ of the domain, the token `tuple-`$n(a_1, \ldots, a_n)$ into zero, one or several tokens of the form `res`(b) – one for each element b in $g(a_1, \ldots, a_n)$. This activity is performed by a specialized pool handler described by the following class declaration:

class `Generator` **holds**
  `tuple-`$n(x_1, \ldots, x_n)$ $\{x_1, \ldots, x_n \mapsto\}$
  `res`(x) $\{\mapsto x\}$.

To compute the minimal model, the coordinator makes heavy use of the broadcast mechanism of LO. In fact, the intended effect of the execution of the system of broker agents, is to broadcast each element t of the minimal model as a token of the form `in-model`(t). These elements are thus available to any agent which has access to the broadcast tokens.

2.3.2 Preliminaries to the Fixed-point Procedure

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

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

The activity of the brokers consists of three tasks:

1. Collect the elements broadcast by all existing brokers (including itself) and then form all possible tuples of them (the arity of the tuples being that of the attached generator);
2. For each tuple, thus obtained, produce all the results of the application of the generator;
3. Broadcast each result.

These three tasks are clearly interdependent and must be run concurrently. Each of these three tasks is handled by a separate set of LO rules.

The following rules may look quite informal: they must be understood as being "formal" skeletons which encode all the actual rules obtained, by replacing `k` by a positive integer and `n` by a non-negative integer. These rules can thus be read as templates.

1. The production of tuples is achieved by

   ```
   in-model(X) @ arity-n
        <>- arg-1(X) @ ... @ arg-n(X) @ arity-n.

   arg-k(X) @ free-k
        <>- bound-k(X) & free-k.

   bound-1(X_1) @ ... @ bound-n(X_n) @ arity-n
        <>- tuple-n(X_1,...,X_n).
   ```

   The first rule picks an element x previously generated and proposes it as a candidate for each argument position k in the generator, by producing the token `arg-`$k(x)$.

   For each such candidate at argument positions which are still free, the second rule creates two clones, one in which the candidate is actually bound to the argument position and one in which the position is left free (so as to consider other candidates). The set of brokers therefore evolves dynamically.

   Finally, when a broker has bound all its arguments, the third rule composes the corresponding tuple of arguments `tuple-`n for the generator.

2. The transformation of the tuple token `tuple-`n into result tokens `res` is performed by the generator attached to a broker. This transformation is specific to each generator.

3. The broadcast of the results is simply achieved by:

   ```
   res(X) @ ^in-model(X) <>- #b.
   ```

Fig. 1 illustrates the behavior of a broker with arity n for this naïve model.

2.3.3 Complexity

For our complexity analysis we use the following notations. n_g is the arity of a generator $g \in \Gamma$. $\overline{n}$ denotes the maximal arity of the generators in Γ whereas $\underline{n}$ denotes the minimal non-zero arity of the generators in Γ. $|M|$ denotes the size of the minimal model. ξ denotes the number of result tokens that are built by a generator from a given tuple token. w is a constraint on the elements in the minimal model. $|M_w|$ denotes the size of the minimal model when the elements are constrained by w. Furthermore, let $\mathcal{E}(g,w)$ be defined such that

$$|M_w| = \sum_{g \in \Gamma} \mathcal{E}(g,w)$$

i.e., $\mathcal{E}(g,w)$ is the number of elements in the minimal model provided by a generator g when the elements are constrained by w. Trivially, the overall number of broadcast messages in this model is

$$\mathcal{M}(\Gamma,w) = |M_w|.$$

The overall number of agents that will be spawned by a broker with generator g, during the calculation of the minimal model is

$$\mathcal{B}(g,w) = \left(1 + \sum_{g\prime \in \Gamma} \mathcal{E}(g\prime,w)\right)^{n_g}$$

Obviously, the overall number of agents in this model is

$$\mathcal{B}(\Gamma,w) = \sum_{g \in \Gamma} \mathcal{B}(g,w) = \sum_{g \in \Gamma} (1 + |M_w|)^{n_g}$$

where

$$\mathcal{B}(\Gamma,w) \sim |M_w|^{\overline{n}}.$$

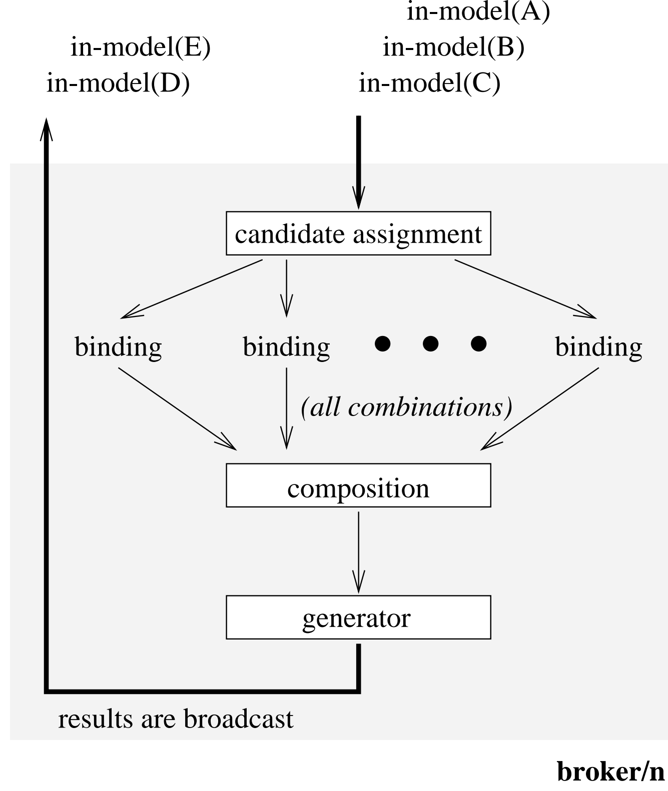

Figure 1: Scheme for the naïve model.

A very important value for the complexity of a given model is the number of transformations of tuple tokens into result tokens. Here, the real work is performed by the generators.

$$\mathcal{T}(g,w) = |M_w|^{n_g}$$

transformations are performed by a broker with generator g. Therefore,

$$\mathcal{T}(\Gamma,w) \sim |M_w|^{\overline{n}}.$$

Example 1 Let f_0^μ be constants, and let f_n^μ be functions of arity n, where $n \geq 1$ and $\mu \in \{1,\ldots,\xi\}$.

A term t is recursively defined by: f_0^μ is a term. If $t_1,\ldots,t_n$ are terms then $f_n^\mu(t_1,\ldots,t_n)$ are terms.

The length $L(t)$ of a term t is recursively defined by: $L(f_0^\mu)$ is 1, $L(f_n^\mu(t_1,\ldots,t_n))$ is $1+\sum_{i=1}^n L(t_i)$.

Given a tuple token, e.g. $\texttt{tuple-}2(t_1,t_2)$, the attached generator of arity 2 provides ξ results: $\texttt{res}\big(f_2^1(t_1,t_2)\big)$, $\texttt{res}\big(f_2^2(t_1,t_2)\big)$,..., $\texttt{res}\Big(f_2^\xi(t_1,t_2)\Big)$. There is always a broker of arity 0 that provides ξ constants.

We define the constraint w as the upper bound for the length of the terms in the minimal model. Now, the values for $\mathcal{E}(g,w)$ and, therefore, the size of the minimal model $|M_w|$, depend not only on w but also on ξ. In this example,

$|M_1| = \xi$,

$|M_2| = \xi + \xi^2$,

$|M_3| = \xi + \xi^2 + 2\xi^3$,

$|M_4| = \xi + \xi^2 + 2\xi^3 + 5\xi^4$,

$|M_5| = \xi + \xi^2 + 2\xi^3 + 5\xi^4 + 13\xi^5$, and so forth. In the remainder of this analysis, it is worth noting that there are minimal models M_w with a model size that increases exponentially in w.

If we install four brokers with arities 0, 1, 2, and 3, respectively, and if we choose $\xi = 2$ for each broker's generator, then to compute the minimal model M_3, consisting of all terms with length $L \leq 3$, i.e. consisting of the 22 terms f_0^1, f_0^2, $f_1^1(f_0^1)$, $f_1^1(f_0^2)$, $f_1^1(f_1^1(f_0^1))$, $f_1^1(f_1^1(f_0^2))$, $f_1^1(f_1^2(f_0^1))$, $f_1^1(f_1^2(f_0^2))$, $f_1^2(f_0^1)$, $f_1^2(f_0^2)$, $f_1^2(f_1^1(f_0^1))$, $f_1^2(f_1^1(f_0^2))$, $f_1^2(f_1^2(f_0^1))$, $f_1^2(f_1^2(f_0^2))$, $f_2^1(f_0^1,f_0^1)$, $f_2^1(f_0^1,f_0^2)$, $f_2^1(f_0^2,f_0^1)$, $f_2^1(f_0^2,f_0^2)$, $f_2^2(f_0^1,f_0^1)$, $f_2^2(f_0^1,f_0^2)$, $f_2^2(f_0^2,f_0^1)$, $f_2^2(f_0^2,f_0^2)$, our implementation needs exactly 12 720 agents. 11 155 transformations are performed by the generators.

Analogously, to compute the minimal model M_4 consisting of all terms with length $L \leq 4$, our implementation needs exactly 1 103 440 agents for the resulting 102 terms. 1 071 714 transformations are performed by the generators.

The rules of our agent-based implementation can be improved for efficiency. For example, the "argument-binding" rule could maintain the set of free argument positions, so that the "candidate-generating" rule generates candidates only for arguments which are not yet bound. This would avoid the production of useless candidates, but would enforce some sequentialization in the access to the set of free argument positions. Also, the "result-broadcast" rule could maintain a count of the broadcast results so that brokers may terminate once all their results have been broadcast (assuming that the generator also provides the total number of results it produces for each tuple it is given).

3 Discussion

3.1 Controlling the Fixed-point Procedure

The previous implementation of the brokers is based on a plain forward-chaining mechanism. This has several well-known drawbacks in particular, the complete lack of control over the generation of the elements of the minimal model. For example, recursion is a classical source of trouble: assume a unary generator g such that $g(t) = \{t\}$ and suppose that the term t is produced by some other generator. The broker for g will receive and consume the token `in-model`(t), propose t as a candidate argument, and produce two clones:

1. `arity-1`, `free-1`
2. `arity-1`, `bound-1`(t)

The second clone above produces `tuple-1`(t) and, by application of the generator to that tuple, the result `res`(t). So, a new token `in-model`(t) is broadcast and feeds the first clone above which, in turn, clones itself and becomes two new brokers identical to those above – hence creating a loop.

This kind of loop is not too dramatic if we assume a fair execution strategy: it leads to the production of multiple copies of the same elements of the minimal model and, possibly, to a combinatorial explosion. Since $\mathcal{E}(g, w) \to \infty$ (that is, the number of tokens `in-model`(t) broadcast by g is infinite), theoretically, an infinite number of agents will be in charge of calculating the elements of the minimal model. But, assuming unbounded execution time and space, all the elements will ultimately be produced. Instead, the real question is: do we really want all the elements of the minimal model to be produced?

In fact, the fixed-point procedure can only be used in one run, to produce all the explicit knowledge implicitly contained in the set of generators. It is obviously not desirable to be fully explicit in most applications simply because the implicit knowledge is too huge to be entirely addressed (even without considering the risk of a combinatorial explosion mentioned above). For example, consider the case of a deductive database, where the implicit knowledge consists of *all* the tuples, i.e. the base ones as well as all the deducted ones.

Therefore, rather than trying to render all the knowledge explicit, i.e. generate all the elements of the minimal model, the implementation presented in the next section constrains the knowledge brokers to actually generate tokens only upon receipt of a *request*. A request specifies a subset of the domain, called its *criterion*. The *answer* to a request w.r.t. a set of generators, is the intersection set of the criterion of the request with the minimal model defined by the generators. This is similar to the case of traditional logic programming where requests are uninstantiated terms, specifying the subset of the minimal model the user wants to retrieve.

3.2 A Request-Based Protocol

In order to process requests, each broker is equipped with some knowledge about the behavior of its generator: the idea here is to view the criterion of a request acting as a *constraint* on the result of the generator attached to a broker. We then assume that the broker is equipped with some constraint propagation capabilities, and can infer, given a constraint on the result of its generator, some constraints on each of its arguments individually.

3.2.1 Knowledge Extension – first step

Let w be the criterion of a request, i.e. a subset of the domain. The constraint it expresses on the expected result of a generator g is of the form

$$w \cap g(a_1, \ldots, a_n) \neq \emptyset,$$

meaning that at least one of the results of the generator belongs to the criterion of the request. Inferred constraints on the arguments of the generator are of the form $a_k \in w_k$ for each $k = 1, \ldots, n$. The subsets w_k depend on the

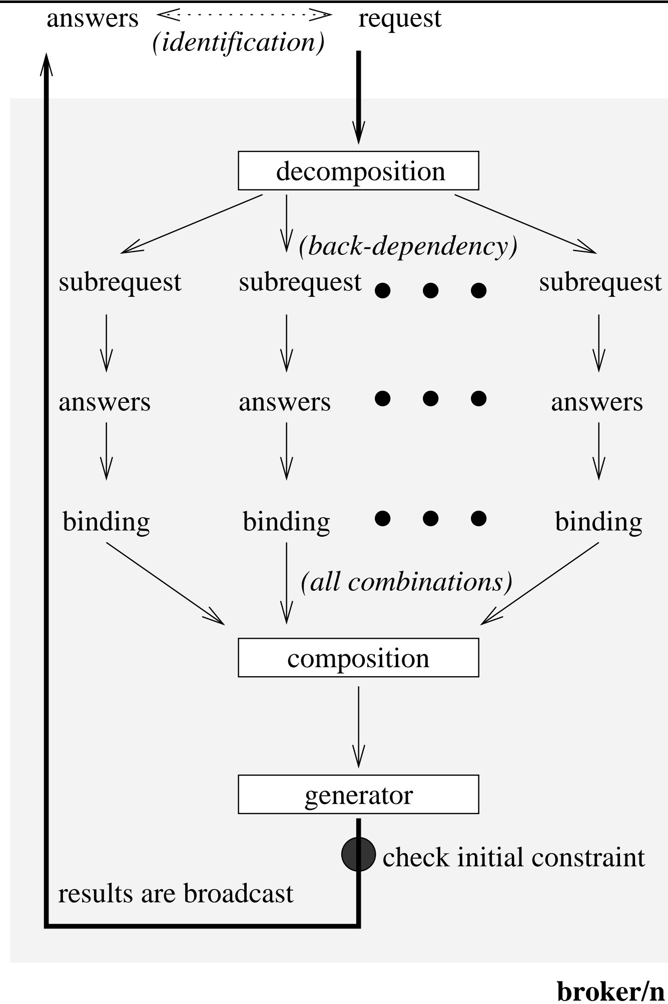

Figure 2: Scheme for the request/subrequest model.

criterion of the request w. Therefore, we assume that each broker is equipped not only with a generator g (of arity n) but also with a set of *back-dependency* functions (one per argument of the generator) $(\bar{g}_k)_{k=1,\ldots,n}$ which are mappings $\bar{g}_k : \wp(\mathcal{D}) \mapsto \wp(\mathcal{D})$ verifying:

$$\forall a_1, \ldots, a_n \in \mathcal{D}, \quad \forall w \in \wp(\mathcal{D}),$$
$$\texttt{if } \; g(a_1, \ldots, a_n) \cap w \neq \emptyset \; \texttt{ then } \; \forall k = 1, \ldots, n, \;\; a_k \in \bar{g}_k(w).$$

The extended knowledge provided by the back-dependency functions, considerably modifies the communication protocol between the brokers. In the plain forward-chaining case, this protocol is rather elementary: each broker receives tokens produced by the other brokers and sends them, in turn, new tokens. A request-based implementation involves a more complex protocol: each broker receives requests; for each request, then it sends subrequests (for the arguments); it receives answers to these subrequests; it subsequently sends answers to the initial request, built from the answers to the subrequests. Fig. 2 illustrates this behavior for a broker with arity n. The interface of the `Generator` class is now extended to account for the back-dependency functions:

class `Generator` **holds**
  `backdep-`$k(w, w')$ $\{w \mapsto w'\}$.

The pattern `backdep-`k is used to handle the k-th back-dependency function: the (permanent) token `backdep-`$k(w, w')$ builds, given a subset w of the domain, the subset w' of the domain defined by

$$w' = \bar{g}_k(w).$$

Obviously, the back-dependency functions reduce the scope of the search for the arguments of the generator, given a constraint on its result. This does not however mean that any combination of arguments which satisfy

the constraints, defined by the back-dependency functions, necessarily produces results which satisfy the initial constraint. The initial constraint must still be checked for each result produced. Hence, we define the class `Constraint` as

class `Constraint` **holds**
`sat`(x, w) $\{x, w \mapsto\}$.

The (permanent) token `sat`(x, w) tests, given an element x of the domain and a subset w of the domain, whether x belongs to w.

Each request is encoded as a single token `requ`(q, w), where w is the criterion of the request and q is its identifier used to retrieve the answer: elements of the answer to a request are expected as broadcast tokens of the form `answer`(q, a). Brokers are characterized by the token `broker`. Each time that a broker receives a token of the form `requ`(q, w), it spawns a specialized agent, characterized by the token `process`(q, w), in charge of this request only. The obvious attempt of using the token `requ`(q, w) instead of `process`(q, w) would lead to uncontrolled clashes with other broadcast requests.

The criterion of the request is also separately held in a token `crit`(w).

```
broker @ requ(Q,W)
     <>- broker & process(Q,W) @ crit(W).
```

The specialized request processor first applies the back-dependency functions to the criterion of the request it processes. A subrequest is broadcast for each argument of the generator. The criterion of each subrequest is simply obtained by application of the corresponding back-dependency function. The subrequests are identified with new constants which are held in tokens of the form `wait-`$k(q)$.

```
crit(W) @ free-k @ backdep-k(W,W') @ ^requ(Q,W')
     <>- crit(W) @ wait-k(Q).
```

When an answer to a subrequest is received, it is proposed as a candidate value for the corresponding argument of the generator. When all the arguments have thus been filled-in, a tuple is formed which is submitted to the generator. Moreover, the results are filtered by the criterion of the initial request and are broadcast as answers to that request:

```
wait-k(Q) @ answer(Q,X)
     <>- bound-k(X) & wait-k(Q).

bound-1(X_1) @ ... @ bound-n(X_n) @ arity-n
     <>- tuple-n(X_1,...,X_n).

process(Q,W) @ res(X) @ sat(X,W) @ ^answer(Q,X)
     <>- process(Q,W).
```

Example 2 Let w be an integer value that specifies a subset of the term domain by giving an upper bound for the length of the terms in the minimal model. Using our example of Sect. 2.3 and letting `requ`(q, w) be a request for all terms in the minimal model with $L \leq w$, we define the back-dependency functions of each generator $g \in \Gamma$ of arity n_g as $(k = 1, \ldots, n_g)$: $w' = \bar{g}_k(w) = w - n_g$. Since

$$L\left(f_{n_g}\left(t_1, \ldots, t_{n_g}\right)\right) \leq w,$$

i.e.,

$$\sum_{i=1}^{n_g} t_i \leq w - 1,$$

the greatest possible length of one of the arguments for this generator t_i is $w - n_g$. I.e., the chosen back-dependency functions are quite efficient.

Fig. 3 shows the dynamic creation of processes and subrequests for an initial request `requ`$(0, 4)$. The flow of requests and answers is only illustrated for *broker/1* in full detail. In the example, a broker of arity 0 (*broker/0*) provides the constant *f0* on request.

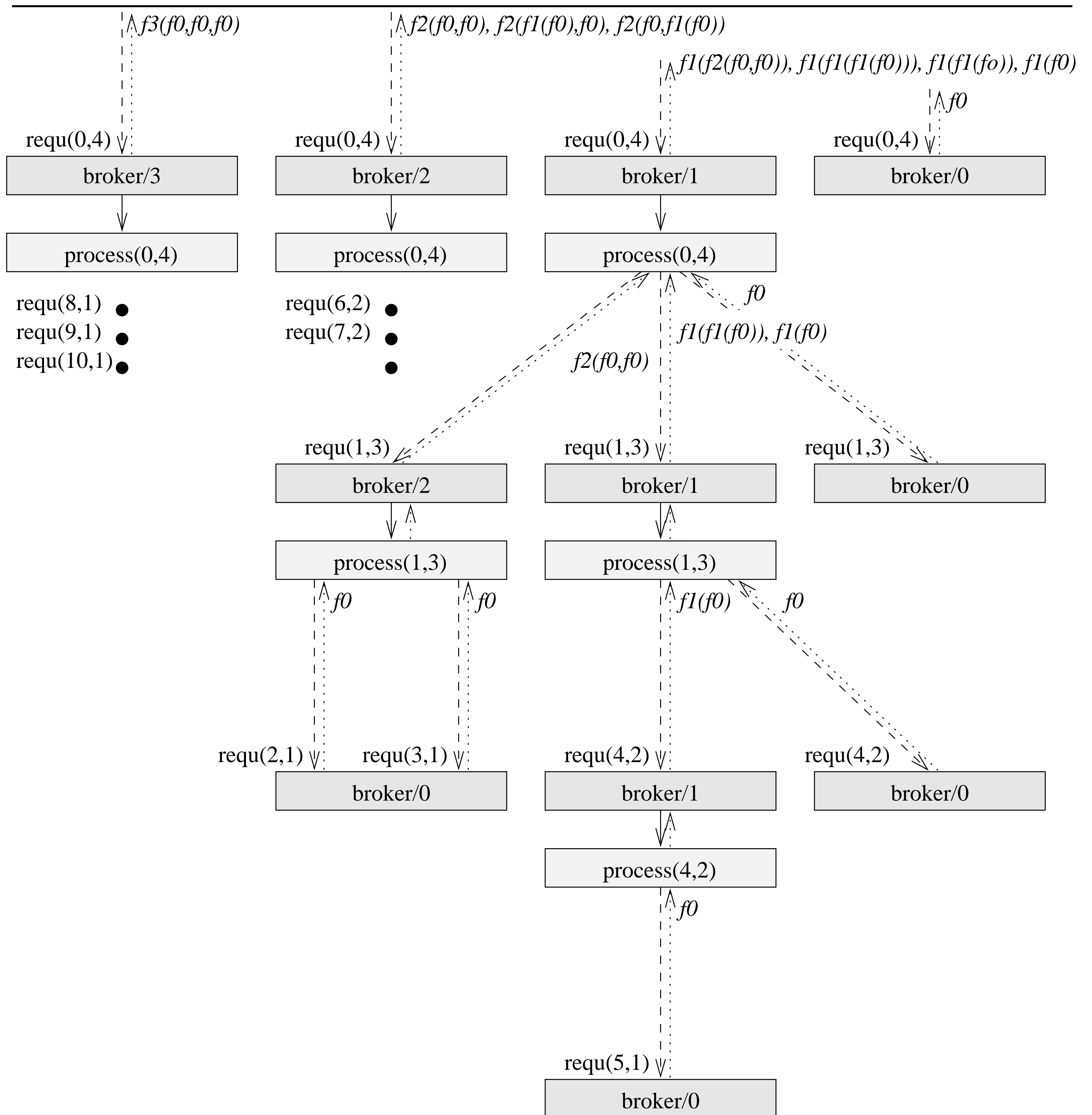

Figure 3: Example for the request/subrequest model.

3.2.2 Complexity

For the general case of our implementation, we compute an upper bound for the number of agents needed. First, let us consider a broker with a generator g of arity n_g. If this broker receives a request $\texttt{requ}(q, w)$, $w > n_g$, it spawns a specialized agent, characterized by the token $\texttt{process}(q, w)$, which in turn, using the back-dependency functions, produces n_g subrequests of the form $\texttt{requ}(q_1, w - n_g)$, $\texttt{requ}(q_2, w - n_g)$,. . ., $\texttt{requ}(q_{n_g}, w - n_g)$. For each of these subrequests, the agent receives $|M_{w-n_g}|$ answers.

When an answer is received, it is proposed as a candidate value for the corresponding argument of the brokers' generator. If we focus on the worst-case assumption that all `wait`-clones are installed before the first answer is received, it immediately follows that

$$1 + \sum_{i=1}^{n_g} \binom{n_g}{i} |M_{w-n_g}|^i = \left(1 + |M_{w-n_g}|\right)^{n_g}$$

broker agents are spawned when all $n_g \times |M_{w-n_g}|$ answers have finally been consumed. This represents the number of broker agents that are spawned when a request of the form $\texttt{requ}(q, w)$ is received by a broker g. During the computation of the minimal model, new requests are produced.

Let $\mathcal{R}_{w'}(g, w)$ be the overall number of requests of the form $\texttt{requ}(q, w')$ that are produced by broker g due to the initial request $\texttt{requ}(q, w)$. For every $g \in \Gamma$

- $w - n_g < w'$: $\mathcal{R}_{w'}(g, w) = 0$
- $w - n_g = w'$: $\mathcal{R}_{w'}(g, w) = n_g$
- $w - n_g > w'$: $\mathcal{R}_{w'}(g, w) = n_g \sum_{g' \in \Gamma} \mathcal{R}_{w'}(g', w - n_g)$

The number $\mathcal{B}(g, w)$ is the overall number of agents that will be spawned by a broker with generator g due to the initial request $\texttt{requ}(q, w)$.

$$\mathcal{B}(g, w) = \left(1 + |M_{w-n_g}|\right)^{n_g} + \sum_{w'=n_g+1}^{w-\underline{n}} \left(1 + |M_{w'-n_g}|\right)^{n_g} \times \sum_{g' \in \Gamma} \mathcal{R}_{w'}(g', w).$$

Analogously, the number $\mathcal{A}(g, w)$ is the overall number of answers that will be sent by a broker with generator g due to this initial request.

$$\mathcal{A}(g, w) = n_g \times |M_{w-n_g}| + \sum_{w'=n_g+1}^{w-\underline{n}} n_g \times |M_{w'-n_g}| \times \sum_{g' \in \Gamma} \mathcal{R}_{w'}(g', w).$$

Obviously, the number of transformations performed by a broker with generator g is

$$\mathcal{T}(g, w) = |M_{w-n_g}|^{n_g} + \sum_{w'=n_g+1}^{w-\underline{n}} |M_{w'-n_g}|^{n_g} \times \sum_{g' \in \Gamma} \mathcal{R}_{w'}(g', w).$$

In our example, using the given back-dependency functions, to compute the minimal model M_4 consisting of all terms with length $L \leq 4$ ($\xi = 2$), our implementation needs only 128 agents for the resulting 102 terms. Thereby, 13 requests/subrequests are submitted, and 158 answers are broadcast.

For the asymptotical behavior, we give an upper bound for $\mathcal{R}_{w'}(g, w)$. As mentioned before, the back-dependency functions reduce the scope of the search for the arguments of the generator. The chosen definitions of the back-dependency functions obviously have some impact on the numer of requests: Weaker back-dependency functions result in a higher number of requests:

Let $N_\Gamma = \sum_{g \in \Gamma} n_g$, $N_\Gamma \geq 1$, be the sum of the arities of all generators, and let the back-dependency functions be defined as ($k = 1, \ldots, n_g$): $w^\star = \bar{g}_k(w) = w - 1$. This yields to a total number of

- $N_\Gamma = 1$: $\mathcal{R}^\star(\Gamma, w) = w$
- $N_\Gamma > 1$: $\mathcal{R}^\star(\Gamma, w) = \frac{N_\Gamma^w - 1}{N_\Gamma - 1}$

request/subrequests where, for $N_\Gamma > 1$, a broker with generator $g \in \Gamma$ of arity n_g is in charge of

$$\mathcal{R}^\star(g, w) = n_g \frac{N_\Gamma^{w-1} - 1}{N_\Gamma - 1}$$

subrequests. For $N_\Gamma = 1$, the broker with the generator $g \in \Gamma$ of arity 1 is in charge of $w - 1$ subrequests. Now, for every $g \in \Gamma$,

- $w-1 < w\prime$: $\mathcal{R}^{\star}_{w\prime}(g,w) = 0$
- $w-1 = w\prime$: $\mathcal{R}^{\star}_{w\prime}(g,w) = n_g$
- $w-1 > w\prime$: $\mathcal{R}^{\star}_{w\prime}(g,w) = n_g \sum_{g\prime\in\Gamma} \mathcal{R}^{\star}_{w\prime}(g\prime, w-1)$

i.e., $\mathcal{R}^{\star}_{w\prime}(g,w) = n_g N_{\Gamma}^{w-w\prime-1}$. Using $\mathcal{R}^{\star}_{w\prime}(g,w)$ as upper bound in our analysis, we get:

$$\mathcal{B}(g,w) \leq \sum_{w\prime=n_g+1}^{w} \left(1+|M_{w\prime-n_g}|\right)^{n_g} \times N_{\Gamma}^{w-w\prime},$$

and

$$\mathcal{A}(g,w) \leq \sum_{w\prime=n_g+1}^{w} n_g \times |M_{w\prime-n_g}| \times N_{\Gamma}^{w-w\prime},$$

and

$$\mathcal{T}(g,w) \leq \sum_{w\prime=n_g+1}^{w} |M_{w\prime-n_g}|^{n_g} \times N_{\Gamma}^{w-w\prime}.$$

In models where $|M_w|$ increases itself exponentially in w – as in our example of Sect. 2.3 –, $\left(1+|M_{w\prime-n_g}|\right)^{n_g} \times N_{\Gamma}^{w-w\prime}$ can become intractable even for small values of w. By any means, it is needed to reduce the number of subrequests dramatically. This is the main goal of the next section.

4 Constraint-Based Knowledge Brokers

In this section, we enhance the basic implementation to tackle several kinds of problems. We focus in particular on

- inter-argument dependencies,
- reuse of previously computed results and,
- recursion control.

4.1 Inter-Argument Dependencies

The request-based model relies on the idea that each broker has some knowledge on the dependency between the result of its generator and its arguments, so that it is capable, given a constraint on the result of the generator (the criterion of a request), to infer constraints on its arguments (turned into subrequests). Compared with the plain forward-chaining scheme, the request/subrequest model obviously reduces the extent of implicit knowledge, contained in the generator, which is explicitly generated. Greater refinement can be obtained with more knowledge on the generators. In particular, it is often realistic to assume some knowledge not only about the dependency between the result and the arguments of a generator, but also about the dependencies between its arguments.

To capture this notion of inter-argument dependencies, we revise and extend the notion of back-dependency functions.

4.1.1 Knowledge Extension – second step

Let's consider the mapping $\bar{g} : \wp(\mathcal{D}) \mapsto \wp(\mathcal{D}^n)$ defined by

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

By definition, we have

$$\forall \vec{a} \in \mathcal{D}^n,\ \ \forall w \in \wp(\mathcal{D}),\ \ \text{if}\ \ g(\vec{a}) \cap w \neq \emptyset\ \ \text{then}\ \ \vec{a} \in \bar{g}(w).$$

If $\mathcal{G}$ denotes the graph of g (hence $\mathcal{G} \in \wp(\mathcal{D}^{n+1})$), then we have equivalently,

$$\forall w \in \wp(\mathcal{D}),\ \ \mathcal{G} \cap (\mathcal{D}^n \times w) \subset (\bar{g}(w) \times w).$$

In other words, $\bar{g}(w) \times w$ provides an upper approximation (in the sense of inclusion) of the part of the set $(\mathcal{D}^n \times w)$ which is contained in the graph of the generator. To capture inter-argument dependencies, we generalize this notion

of approximation and we now assume that each broker is equipped not only with a generator g, but also with an *interdependency* function $\tilde{g} : \wp(\mathcal{D}^{n+1}) \mapsto \wp(\mathcal{D}^{n+1})$ verifying:

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

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

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

The back-dependency functions are now redundant, since they are defined in terms of the interdependency function as

$$\bar{g}_k(w) = \pi_k < \tilde{g}(\mathcal{D}^n \times w) >$$

where $\pi_k : \mathcal{D}^{n+1} \mapsto \mathcal{D}$ is the k-th projection function:

$$\pi_k(\vec{a}) = \vec{a}[k].$$

The interdependency function is accounted for at the interface level for the `Generator` class:

class `Generator` **holds**
 `tuple-`$n(x_1, \ldots, x_n)$ $\{x_1, \ldots, x_n \mapsto\}$
 `res`(x) $\{\mapsto x\}$
 `init-`$n(w, s)$ $\{w \mapsto s\}$
 `ins-`$k(x, s, s')$ $\{x, s \mapsto s'\}$.

The patterns `tuple-`n and `res` are the same as in the previous sections. The pattern `init-`n is used to compute the initial approximation (in fact, the back-dependency functions): the (permanent) token `init-`$n(w, s)$ builds, given a subset w of $\mathcal{D}$, the subset s of $\mathcal{D}^{n+1}$ defined as

$$s = \tilde{g}\left(\mathcal{D}^n \times w\right).$$

It is assumed that if s is empty, the token is not present. Similarly, the patterns `ins-`k provide further refinements of the approximation upon receipt of answers to the subrequests. The (permanent) token `ins-`$k(x, s, s')$ builds, given a subset s of $\mathcal{D}^{n+1}$ and an element x of $\mathcal{D}$, the subset s' of $\mathcal{D}^{n+1}$ defined as

$$s' = \tilde{g}\left(s \cap \pi_k^{-1} < x >\right).$$

In other words, s' is the approximation of the subset of s consisting of the tuples which are in the graph of the generator and whose k-th component is precisely x. Thus, the binding of the k-th argument may reduce the scope of the other arguments. It is assumed here that if s' is empty, the token is not present.

Elementary constraint manipulations are given by the following interface:

class `Constraint` **holds**
 `sat`(x, w) $\{x, w \mapsto\}$
 `seek-`$k(s, w)$ $\{s \mapsto w\}$.

The pattern `sat` is the same as in the previous section. The pattern `seek-`k is used to extract information from an approximation (by simple projection). The (permanent) token `seek-`$k(s, w)$ builds, given a subset s of $\mathcal{D}^{n+1}$, the subset w of $\mathcal{D}$ defined as

$$w = \pi_k < s > .$$

In Fig. 4, the program for the brokers, given in the previous section, is augmented to account for interdependency constraints, held in a token of the form `const`(s) where s is the subset of $\mathcal{D}^{n+1}$.

```
(1) broker @ requ(Q,W) @ init(W,S)
        <>- broker & process(Q,W) @ const(S).

(2) const(S) @ free-k @ seek-k(S,W) @ ^requ(Q,W)
        <>- const(S) @ wait-k(Q).

(3) const(S) @ wait-k(Q) @ answer(Q,X) @ ins-k(X,S,S')
        <>- const(S) @ wait-k(Q) & const(S') @ bound-k(X).

(4) arity-n @ bound-1(X_1) @ ... @ bound-n(X_n)
        <>- tuple-n(X_1,...,X_n).

(5) process(Q,W) @ res(X) @ sat(X,W) @ ^answer(Q,X)
        <>- process(Q,W).
```

Figure 4: The request/subrequest protocol with interdependency constraints.

4.2 Control of the Request Generation

The request/subrequest protocol does not avoid recursion *per se*. Suppose a broker has a generator g with $g(t) = \{t\}$ and receives a request with a criterion w such that $t \in w$. A specialized agent A is created to process this request. Suppose furthermore, that the initial constraint $\tilde{g}(\mathcal{D} \times w)$, returned by the token **init**, is $s = w \times w$. Now, since $\pi_1 < s > = w$, the subrequest for the (single) argument of the generator also has w as a criterion. The same broker receives this subrequest (broadcast by A) and spawns again a new agent in charge of the same request – hence a loop.

Another, related issue, is that of reuse. Suppose that two brokers generate subrequests with the same criterion (or overlapping criteria). With the basic request/subrequest protocol, these two requests are processed separately, i.e., all the work of the generation of explicit elements of the minimal model, satisfying the criterion of the requests, is duplicated (at least partially). This redundant processing can also be detected in our example of Fig. 3. Here, e.g. *broker/1* processes the element *f1(f0)* three times; once due to **requ**(4,2) of *broker/1*, and twice due to the subrequests **requ**(6,2) and **requ**(7,2) (processing not shown in full detail) that are produced by *broker/2*.

4.2.1 Recursion Elimination and Reuse of Results

We now show that both recursion elimination and reuse can easily be achieved by a lemma caching mechanism (i.e. caching of requests and results) which requires only a few modifications to the program of Fig. 4. We define the "scope" of a broker as the subset of the domain which does not intersect any of the criteria 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 the domain and explicitly generates the elements it encounters which are in the minimal model. The scope of the broker denotes the unexplored area of the domain.

4.2.2 Knowledge Extension – third step

Let w_o be the scope of a broker at some point, and let w be the criterion of a request it receives. The broker spawns an agent in charge of exploring the subset $w_o \cap w$ (and answering the request), and gets on with a reduced scope $w_o \cap \neg w$ (where $\neg w$ is the complement of w). To perform the intersection and complement operations on subsets of the domain, we augment the `Constraint` interface as follows:

class `Constraint` **holds**
    `split`(w_o, w, w_1, w_2) $\{w_o, w \mapsto w_1, w_2\}$.

Given two subsets w_o, w of $\mathcal{D}$, the (permanent) token `split`(w_o, w, w_1, w_2) builds the subsets w_1, w_2 of $\mathcal{D}$ defined by

$$w_1 = w_o \cap w \qquad w_2 = w_o \cap \neg w.$$

We can now modify the first rule of Fig. 4 which handles incoming requests, assuming that the scope w_o of a broker is held in the token `broker`(w_o), and that initially $w_o = \mathcal{D}$.

```
(1') broker(W0) @ requ(Q,W) @ split(W0,W,W1,W2) @ init(W1,S)
          <>- broker(W2) & process @ requ(Q,W) @ const(S).
```

Notice that the agent which is spawned by the request still contains the request: indeed, this agent is now in charge not only of the request which has spawned it, but also of all the requests whose criteria intersect its area of exploration.

The generation of results is the same as before (rules 2, 3, and 4 of Fig. 4, respectively), except that, when an agent broadcasts a subrequest, this subrequest should remain visible to the agent (since it may itself generate answers to it). Therefore, the second rule of Fig. 4, which is in charge of broadcasting the subrequests, should be modified as follows, so that each subrequest once broadcast is reasserted inside the agent.

```
(2') const(S) @ free-k @ seek-k(S,W) @ ^requ(Q,W)
          <>- const(S) @ wait-k(Q) @ requ(Q,W).
```

The transformation of results into answers (last rule of Fig. 4) must be more deeply modified. Indeed, it is no longer the case that each agent spawned by a broker has only a single request to process. Here, an agent is spawned for each generated result and this agent will in turn process all the requests it can answer. This is achieved by the following rules (replacing the last rule in Fig. 4).

```
(5') process @ res(X)
          <>- process & process(X).

(5") process(X) @ requ(Q,W) @ sat(X,W) @ ^answer(Q,X)
          <>- process(X).
```

These refinements are schematically illustrated in Fig. 5. Again, we focus on a broker with arity n.

4.2.3 Complexity

$[lb, \ldots, ub]$ represents an integer interval that specifies a subset of the term domain by giving both a lower and upper bound for the length of the terms in the minimal model. In conformance with our example, initially, $[1, \ldots, w]$ represents all terms in the minimal model with length $L \leq w$.

Let us consider a broker with a generator g of arity n_g. If this broker receives a request $\mathtt{requ}(q, [1, \ldots, w])$, $w > n_g$, it spawns a specialized agent, characterized by the tokens $\mathtt{process}$, $\mathtt{requ}(q, [1, \ldots, w])$, and $\mathtt{const}(s)$, which in turn, using the back-dependency functions being defined as ($k = 1, \ldots, n_g$)

$$[lb', \ldots, ub'] = \bar{g}_k([lb, \ldots, ub]) = [max(1, lb - n_g), \ldots, ub - n_g],$$

produces n_g subrequests of the form $\mathtt{requ}(q_1, [1, \ldots, w-n_g])$, $\mathtt{requ}(q_2, [1, \ldots, w-n_g])$,. . ., $\mathtt{requ}(q_{n_g}, [1, \ldots, w-n_g])$. As in the simple request/subrequest model, for each of these subrequests, the agent receives $|M_{w-n_g}|$ answers. However, in contrast with the previous model, the broker here refines the scope of the domain it is in charge of (see rule (1')). This refinement reduces the number of subrequests dramatically. If we assume the weak back-dependency functions ($k = 1, \ldots, n_g$)

$$[lb^\star, \ldots, ub^\star] = \bar{g}_k([lb, \ldots, ub]) = [max(1, lb - 1), \ldots, ub - 1],$$

and if we assume the worst-case split of the initial scope of the domain from the interval $[n_g, \ldots, \infty]$ into the intervals $[n_g, n_g]$ and $[n_g + 1, \ldots, \infty]$ then the upper bound for the overall number of subrequests of the form $\mathtt{requ}(q, [1, \ldots, w\prime])$ produced by this broker due to the initial request $\mathtt{requ}(q, [1, \ldots, w])$ and the corresponding subrequests of all the other brokers (including himself), is $\forall w\prime = 1, \ldots, w - 1$:

$$\mathcal{R}^\star_{w\prime}(g, w) = n_g.$$

Hence,

$$\mathcal{B}(g, w) \leq \sum_{w\prime = n_g + 1}^{w} \left(1 + |M_{w\prime - n_g}|\right)^{n_g} \times N_\Gamma$$

and

$$\mathcal{A}(g, w) \leq \sum_{w\prime = n_g + 1}^{w} n_g \times |M_{w\prime - n_g}| \times N_\Gamma.$$

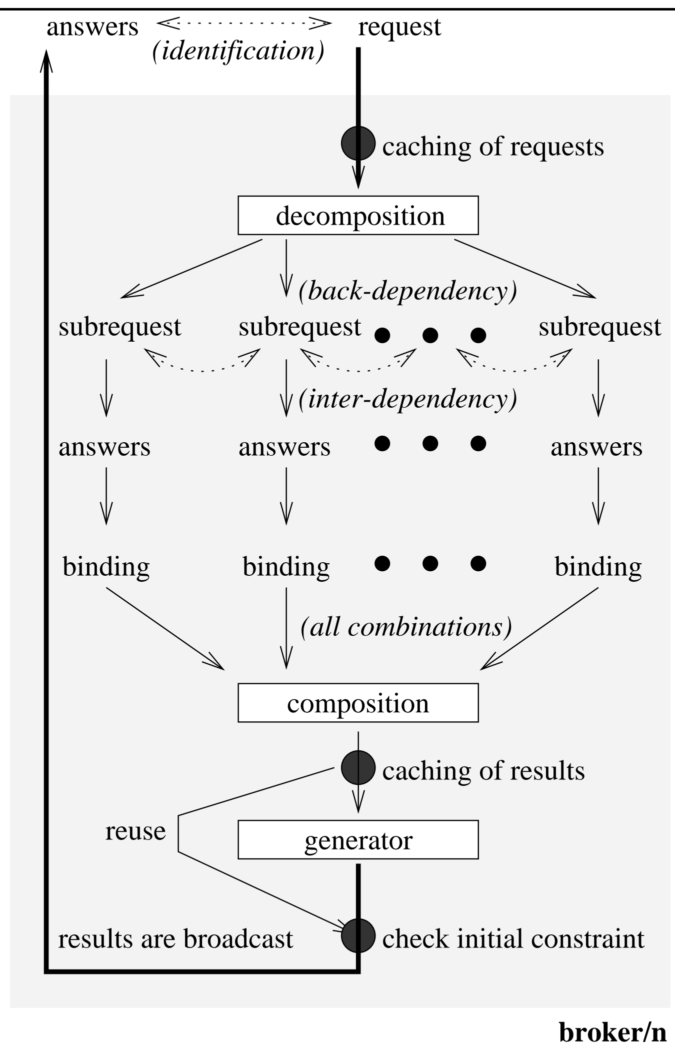

Figure 5: Scheme for the constraint-based knowledge broker model.

Due to the reuse of results, another considerable advantage is gained over the transformations. It immediately follows, that

$$\mathcal{T}(g,w) \leq \sum_{w'=n_g+1}^{w} |M_{w'-n_g}|^{n_g}.$$

Therefore,

$$\mathcal{B}(g,w) \sim \mathcal{T}(g,w) \sim |M_{w-n_g}|^{n_g},$$

and

$$\mathcal{A}(g,w) \sim |M_{w-n_g}|,$$

respectively, as well as

$$\mathcal{B}(\Gamma,w) \sim \mathcal{T}(\Gamma,w) \sim |M_{w-\overline{n}}|^{\overline{n}},$$

and

$$\mathcal{A}(\Gamma,w) \sim |M_{w-\underline{n}}|,$$

respectively.

4.3 Knowledge Thresholds

The program of Fig. 4, with or without the lemma caching facility, has one important drawback. Although interdependency constraints reduce the number of tuples submitted to the generator, they do not influence the generation of subrequests. Indeed, let us consider the second rule in the program of Fig. 4, the *subrequests generation* rule. It generates a subrequest for each free argument position k, based on the k-th projection of the current interdependency constraint.

Unfortunately, nothing prevents this rule from being applied simultaneously to all the free argument positions, immediately after creation of the specialized agent in charge of a request. In this case, the answers to one subrequest will not influence the generation of the other subrequests. This is unfortunate since, in some cases, the binding of one argument might considerably refine the constraint on another and, hence, the criterion of its corresponding subrequest.

On the other hand, if the subrequests generation rule is not applied immediately, but according to some strategy, then, eventually, the binding of one argument may influence the subrequest corresponding with another through the interdependency constraint. In other words, a strategy for the generation of the subrequests can easily be implemented by gaining control over the subrequests generation rule.

4.3.1 Knowledge Extension – final step

To achieve this control, we let the application of the subrequest generation rule depend on some condition on the current interdependency constraint. Such conditions, which we call *thresholds*, are expressed as unary predicates over interdependency constraints (which are elements of $\wp(\mathcal{D}^{n+1})$).

Consequently, we now assume that each broker is equipped not only with a generator and an interdependency function, but also with a set of thresholds $(t_k)_{k=1,\ldots,n}$, one per argument of the generator, where each t_k is a subset of $\wp(\mathcal{D}^{n+1})$. Thresholds are accounted for in the interface for the `Generator` class as follows:

class `Generator` **holds**
  `threshold-`k(s) $\{s \mapsto\}$.

The subrequests generation rule is now simply modified as follows:

```
(2") const(S) @ free-k @ threshold-k(S) @ seek-k(S,W) @ ^requ(Q,W)
          <>- const(S) @ wait-k(Q) @ requ(Q,W).
```

It is now only applied if the interdependency constraint held in `const`(s) satisfies the threshold condition. Different classes of strategies are obtained by considering different kinds of thresholds:

4.3.2 Strategies Based on Argument Ordering

The most simple strategy class relies on argument ordering. Suppose we want to express that subrequests for argument k should not be issued before argument h has been bound. We simply write

$$t_k = \left\{ s \in \mathcal{D}^{n+1} \text{ such that } \exists a \in \mathcal{D}, \pi_h < s >= \{a\} \right\}.$$

Several control-level constraints of this kind can of course be combined. If there is no loop in the (possibly partial) ordering of the arguments, then the strategy is complete, in the sense that all the answers to any request can be produced. When two arguments are not actually ordered, the corresponding subrequests are, potentially, issued concurrently. For a broker with arity n, this strategy is schematically illustrated in Fig. 6.

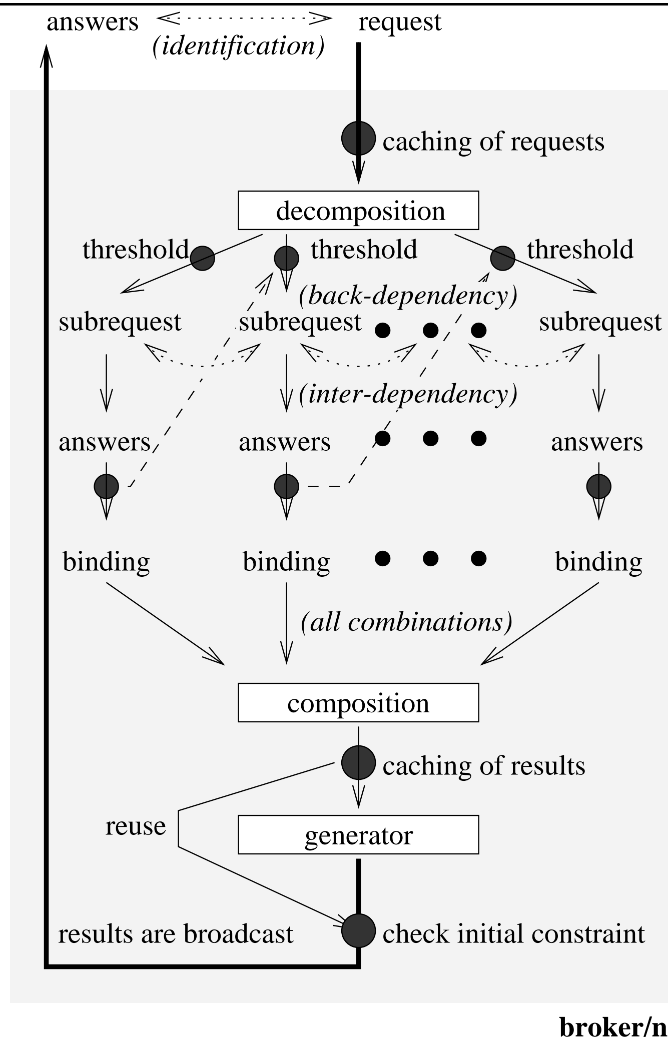

Figure 6: Scheme for the constraint-based knowledge broker model with thresholds and argument ordering.

4.3.3 Strategies Based on Argument Content

A more refined class of strategies is obtained by imposing control-level constraints on the content of the arguments. For example, one may wish to express that subrequests for argument k should be issued only if enough is known about this argument (this local threshold being characterized by a subset w_o of $\mathcal{D}$). This can be obtained by

$$t_k = \left\{ s \in \mathcal{D}^{n+1} \text{ such that } \pi_k < s > \subset w_o \right\}.$$

Here completeness is lost in the sense that requests which are too general, may deadlock and produce no answer (simply because no argument has enough knowledge to come to the decision to send its subrequest).

Example 3 Let us consider a broker with a generator g of arity n_g. If this broker receives a request $\texttt{requ}(q, [1, \ldots, w])$, $w > n_g$, it spawns a specialized agent, characterized by the tokens `process`, $\texttt{requ}(q, [1, \ldots, w])$, and $\texttt{const}(s)$. In our example we can use the following thresholds:

$$L\left(f_{n_g}\left(t_1, \ldots, t_{n_g}\right)\right) \leq w,$$

i.e.,

$$\sum_{i=1}^{n_g} t_i \leq w - 1,$$

i.e., the greatest possible length of one of the arguments t_i is $w - n_g$. Therefore, the broker produces a first subrequest of the form $\texttt{requ}(q_1, [1, \ldots, w - n_g])$. If it receives an answer for t_1 of length $L(t_1)$, $1 \leq L(t_1) \leq w - n_g$, the constraint for the other arguments can be refined:

$$\sum_{j=2}^{n_g} t_j \leq w - 1 - L(t_1),$$

i.e., the greatest possible length of one of the other arguments t_j is $w - n_g - L(t_1) + 1$. A second, more refined, subrequest is then of the form $\texttt{requ}(q_2, [1, \ldots, w - n_g - L(t_1) + 1])$. If it now receives an answer for t_2 of length $L(t_2)$, $1 \leq L(t_2) \leq w - n_g - L(t_1) + 1$, the constraint for the remaining arguments can be refined analogously.

As a consequence, the n_g-th subrequest can then be of the simple form $\texttt{requ}(q_{n_g}, [1, 1])$.

4.4 Summary of Complexity Analysis

Table 3 summarizes our results of the complexity analysis and illustrates the asymptotical behavior of the different models.

The analysis of the model refinements shows the complexity in terms of the number of agents needed and of the number of messages sent, distinguished by requests and answers of the involved broker agents. Furthermore, the number of transformations (of tuple tokens into result tokens) is given.

5 Broker-Based Chart-Parsing

This section presents an application of the constraint-based knowledge brokers to parsing. The problem is to parse sentences of a language described by a context-free feature-unification grammar [29, 32].

The technique we use here is a particularly interesting case of distributed problem solving [12, 14] which illustrates well the use of local resource consumption in generic communication: each broker receives a separate copy of each broadcast request and can consume it only from its own local environment.

The problem we address specifically is concurrent parsing, a topic which has attracted the interest of several researchers in the object-oriented programming community [25, 35]. On the other hand the problem-solving technique we employ here can be fruitfully generalized to more complex examples, like distributed expert systems operating on complex domains, where different experts are required to work independently on shared data, feeding back different outputs which all need to be taken into consideration for the final solution of a given problem.

5.1 Chart-Parsing

The application we describe amounts to a concurrent implementation of the Earley algorithm for context-free parsing [15] and is drawn much from active chart-parsing methodology [20], where incomplete phrasal subtrees are viewed as agents consuming already completed elements to produce other (complete or incomplete) subtrees.

	naïve model (exact)	request/subrequest model (upper bounds)	constraint-based knowledge brokers (upper bounds)
$\mathcal{R}(\Gamma, w)$ (requests)	$\lvert M_w \rvert$	$\frac{N_\Gamma^w - 1}{N_\Gamma - 1}$ asymptotically: exponential in w	N_Γ asymptotically: constant
$\mathcal{A}(\Gamma, w)$ (answers)	asymptotically: equivalent to the order of the model size	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} n_g \lvert M_{w\prime-n_g} \rvert N_\Gamma^{w-w\prime}$ asymptotically:[a] exponential in the minimally reduced[c] w	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} n_g \lvert M_{w\prime-n_g} \rvert N_\Gamma$ asymptotically: equivalent to the order of the minimally reduced[b] model size
$\mathcal{B}(\Gamma, w)$ (broker agents)	$\sum_{g\in\Gamma} (1 + \lvert M_w \rvert)^{n_g}$ asymptotically: equivalent to the order of the model size to the power of the maximal arity	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} \left(1 + \lvert M_{w\prime-n_g} \rvert\right)^{n_g} N_\Gamma^{w-w\prime}$ asymptotically:[a] exponential in the minimally reduced[c] w	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} \left(1 + \lvert M_{w\prime-n_g} \rvert\right)^{n_g} N_\Gamma$ asymptotically: equivalent to the order of the maximally reduced[d] model size to the power of the maximal arity
$\mathcal{T}(\Gamma, w)$ (transformations)	$\sum_{g\in\Gamma} \lvert M_w \rvert^{n_g}$ asymptotically: equivalent to the order of the model size to the power of the maximal arity	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} \lvert M_{w\prime-n_g} \rvert^{n_g} N_\Gamma^{w-w\prime}$ asymptotically:[a] exponential in the minimally reduced[c] w	$\sum_{g\in\Gamma} \sum_{w\prime=n_g+1}^{w} \lvert M_{w\prime-n_g} \rvert^{n_g}$ asymptotically: equivalent to the order of the maximally reduced[d] model size to the power of the maximal arity

[a] assumption, that $\overline{n} << w$
[b] i.e., $\lvert M_{w-\underline{n}} \rvert$
[c] i.e., $w - \underline{n} + 1$
[d] i.e., $\lvert M_{w-\overline{n}} \rvert$

Table 3: Some results of the complexity analysis.

<*feature-constraint*>	=	<*path*> ':=' <*atomic-value*>
	$\vert$	<*path*> '==' <*path*>
<*path*>	=	<*label*>
	$\vert$	<*label*> '.' <*path*>

Table 4: Syntax of feature constraints.

However, in our case even the rules of the grammar and the entries of the lexicon act as independent units directly partaking in the computation. Moreover, as distinct from the usual sequential formulations of chart-parsing, here no superimposed scheduler is in charge of the task of feeding incomplete subtrees with complete ones; instead, incomplete elements as well as grammar rules, are handled by brokers, and behave as truly active decentralized computational units which progress by exchanging requests and answers (complete trees). In that sense, it is a typical example of cooperative problem solving.

It has been noted that the chart-parsing methodology is not limited to parsing but provides a general inference mechanism for rule-based engines [28]. For example, it has been applied to deductive databases in Datalog [34], and the broker-based parser presented below could straightforwardly be adapted to this case.

5.2 Knowledge Representation – Feature Structures

The domain of knowledge $\mathcal{D}$ used in the following is that of "feature terms". For simplicity's sake, we here take a feature term to be a simple record, i.e. a list of pairs label/value; the labels are assumed to be distinct pairwise and the values can either be atomic values (integer, strings, etc.), or feature terms themselves. A label can be interpreted as a partial function which maps each feature term to the value of this label in that feature term, when it exists. This functional interpretation can be straightforwardly extended to paths, i.e. sequences of labels.

A set of feature terms (element of $\wp(\mathcal{D})$) can be represented intentionally by a set of feature constraints which relate the values obtained through different paths. The syntax of feature constraints is given in Table 4. The denotation of a set of feature constraints is the intersection of the denotations of its elements. For example, if `a`, `b` and `c` are labels, a feature term t belongs to the denotation of {`a.b:=3, a==c`} if and only if

- t contains the label `a` and its value is a feature term t';
- t' contains the label `b` and its value is 3;
- t contains the label `c` and its value is also t'.

All the feature constraint manipulations required here are encoded in the following class:

class `Constraint` **holds**
  `sat`(x, w) $\{x : \mathcal{D},\ w : \wp(\mathcal{D}) \ \mapsto \}$
  `seek-`$k(s, w)$ $\{s : \wp(\mathcal{D}^{n+1}) \ \mapsto \ w : \wp(\mathcal{D})\}$
  `split`(w_o, w, w_1, w_2) $\{w_o, w : \wp(\mathcal{D}) \ \mapsto \ w_1, w_2 : \wp(\mathcal{D})\}$**.**

Notice that a $n+1$-tuple of feature terms can itself be represented by a feature term obtained by concatenating to its $n+1$-th component distinguished labels `arg-`k, for $k = 1, \ldots, n$, holding the n first components of the tuple. Thus, a subset of tuples of feature terms, such as the first argument of the `seek` member above, can be directly specified as a subset of feature terms, using regular feature constraints.

- The member `sat`(x, w) simply tests whether a feature term x satisfies a set of feature constraints w.
- The member `seek-`$k(s, w)$ builds the set of feature constraints w obtained by extracting all the information concerning the distinguished label `arg-`k, from a set of feature constraints s. Thus, constraints of the form `arg-`$k.P$ `:=` V entailed by s become P `:=` V in w; constraints of the form `arg-`$k.P$ `==` `arg-`$k.P'$ entailed by s become P `==` P' in w; all other constraints, if any, in s have no counterpart in w.
- The member `split` is more involved: given sets of feature constraints w_o and w, the set of feature constraints w_1 is simply obtained as the union of w_o and w (corresponding to the intersection of their denotation), assuming this union is not inconsistent; the set w_2 must encode the intersection of the denotation of w_o with the *complement* of the denotation of w. This cannot be expressed within the simple framework adopted in this paper, but can easily be expressed in the full feature language (see for example [32]).

5.3 Knowledge Representation – Grammar Rules

A context free grammar rule can be seen as a generator operating on parse tokens expressing statements of the form "Phrase ... is an instance of syntactic category ...". Such tokens are represented as feature terms using the following labels:

- `text` holds a piece of text to be parsed;
- `beg` and `end` hold pointers in the text which delimits the portion of the text on which the parsing statement is done;
- `cat` holds the syntactic category of the specified portion of the text.

Syntactic categories are feature structures specifying a basic type (e.g. `s` for correct sentences, `np` for noun phrases, `vp` for verb phrases, etc.) plus a number of syntactic or semantic attributes (gender, number, etc.). For simplicity's sake, in the sequel, we consider that categories are reduced to their basic type and ignore the other attributes.

Initially, each grammar rule is handled by a broker whose generator is fully defined by a sub-class of the generic class

class `Generator` **holds**
  `tuple-`$n(x_1, \ldots, x_n)$ $\{x_1, \ldots, x_n \mapsto\}$
  `res`(x) $\{\mapsto x\}$
  `init-`$n(w, s)$ $\{w \mapsto s\}$
  `ins-`$k(x, s, s')$ $\{x, s \mapsto s'\}$
  `threshold-`$k(s)$ $\{s \mapsto\}$.

Let's consider the simple rule `s ::= np vp`. Its corresponding generator has arity 2 and defines the class members as follows.

- The member `tuple-2`(x_1, x_2) builds the feature term corresponding to the head `s` given the two feature terms x_1, x_2 corresponding to the tail categories `np` and `vp`.
- The member `res`(x) makes this new term x available as a result.
- The member `init-2`(w, s) takes a set of feature constraints w and builds the set of feature constraints s obtained by adding to w the set s_o of feature constraints corresponding to the rule, assuming no inconsistencies occur. s_o denotes a subset of $\mathcal{D}^3$, according to the convention described above for tuples of feature terms, and expresses the dependencies between the head and the tail of the grammar rule. Here, it is represented by the following set of feature constraints:

  ```
  cat := s,
  arg1.cat := np, arg2.cat := vp,
  arg1.text == text, arg2.text == text,
  beg == arg1.beg, end == arg2.end, arg1.end == arg2.beg
  ```

  In other words, a phrase of category `s` is obtained from a phrase of category `np` (noun phrase, held in `arg1`) and a phrase of category `vp` (verb phrase, held in `arg2`). The first three constraints encode the category of each phrase; the other constraints ensure that the phrases belong to the same text and are appropriately juxtaposed.

- The member `ins-`k takes a feature term x and a set of feature constraints s and builds the set of feature constraints s', obtained by inserting in s, the feature constraint `arg-`k `:=` x.
- Finally, the member `threshold-`k can be defined to implement various strategies; the most naïve strategy forces the categories of the tail of the rule to be parsed from left to right. This is obtained by an argument ordering strategy, saying that the predicate `threshold-`$k(s)$ holds whenever s entails {`text :=` T, `arg-`k`.beg :=` P, `arg-`k`.cat :=` C} for some text T, some pointer P in that text and some category C. Notice that alternative strategies can be defined, such as, for example, simultaneous left-to-right and right-to-left strategies, where the threshold predicate holds when either the begin or the end pointer is assigned.

The whole computation is initiated by an interface which builds and broadcasts requests on demand, waits for possible answers to these initial requests, and propagates the answers back to the user. At the other end, a dictionary agent is capable of processing some requests corresponding with terminal syntactic categories by fetching information from a database of entries which associates words with terminal categories.

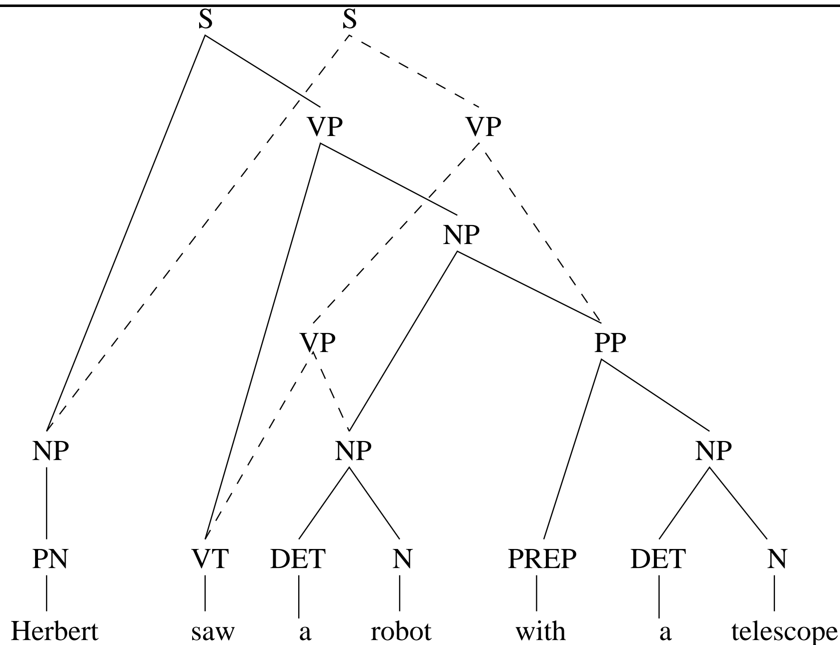

Figure 7: A sample run of the broker-based parser.

5.4 A Sample Run

Consider the (trivial) sample grammar

```
s  ::= np vp
np ::= det n
np ::= pn
np ::= np pp
vp ::= tv np
vp ::= vp pp
pp ::= prep np.
```

The sentence "Herbert saw a robot with a telescope" can be parsed according to this grammar (and an appropriate dictionary), yielding the two parse-trees of Fig. 7. These two answers originate from the fact that the same complete trees can be consumed by several agents encoding different, incomplete trees. As a consequence, we end up with two different parses for the substring 'saw a robot with a telescope'. Our generic constraint-based knowledge broker model allows maximal reuse of results in the generation of such alternate parse-trees, and also avoids infinite loops deriving from left-recursive rules such as the fourth and sixth rules of the grammar above. The approach to enforce redundancy checking is quite simple and elegant and comes naturally in a decentralized, object-oriented style of programming; it can be contrasted with the more usual way of enforcing it, which is obtained by explicitly comparing newly created trees with previously existing ones.

6 Related Work

Our work is related to two main areas: on the one hand, Constraint Logic Programming (CLP), which, in general, has been heavily investigated (see [33, 19] for seminal papers), and on the other hand, multi-agent (or multi-process) support for coordination [23], in particular applied to knowledge retrieval.

In [22], the concept of generalized propagation is introduced, in which disjunctive constraints are propagated by taking their most specific common generalization. The propagation mechanism itself has some similarities with ours, in the sense that the rules the authors consider (strict Horn clauses) are also viewed as agents which perform the propagation.

In a more traditional setting, [24] proposes a framework in which a set of constraints is solved by a system of cooperative, distributed, specialized constraint solvers which exchange their relevant results. Propagation is also achieved through broadcasting but is not controlled by a request/subrequest mechanism.

Concurrent constraints were introduced in a process-oriented setting, in [30]. There, 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 [31]. This approach does not make the distinction between two separate uses of constraints: for communication and for solving. This distinction clearly appears in our proposal, where communication is achieved by the – global – broadcasting mechanism and solving is performed – locally – inside each individual broker agent.

The Oz system [18] also proposes a framework where multiple agents ("elaborators" in Oz terminology) interact by exchanging constraints through a monotonic constraint store. Non monotonicity is super-imposed by allowing logical variables to be unified with "names" which can themselves be *destructively* bound to other variables.

Concerning knowledge retrieval, a cooperative information gathering approach using a multi-agent system for distributed problem solving was recently published in [26]. More relevant literature can be found in [21]. Logic-based models are also used in [16] to capture the domain of expertise of information brokers. Rather than using constraints, their modeling language is based on a predicate logic with contexts. The TSIMMIS project [11] takes a different approach using a self-describing object model for the internal representation of information and requests.

Similarly the Teamwork approach proposed in [13] applies knowledge combination to distributed search problems whose descriptions offer no natural way of dividing them a priori into subproblems.

We have chosen the word "broker" within our framework to mirror the fact that all knowledge (even only partially available or heterogeneous) that is encapsulated in objects, may be manipulated and retrieved by the simple coordination of the activities of individual agents (the brokers), which may then be distributed. An illustration of the use of constraint-based knowledge brokers (CBKBs) in network-wide environments was given in [9]. For a concrete application domain, [10] provides a detailed description of a CBKB software implementing information gathering facilities to help knowledge workers access and leverage the World-Wide Web.

In our framework, brokers are fully autonomous entities which can channel and transform requests to objects which encapsulate possibly heterogeneous data and are potentially distributed. In that sense, they have a strong similarity with the Object Request Brokers (ORBs), widely used to model and implement distributed applications. The Object Management Group's Common Object Request Broker Architecture (OMG's CORBA [17]), for example, is a quasi-standard in defining objects and accessing them through their interfaces. However, CORBA brokers assume essentially synchronous communication between objects and do not address issues of request filtering and combination.

7 Conclusion

Our approach used a mathematical model to describe constraint-based knowledge brokers where the traditional fixed-point procedure was exploited. This procedure was implemented in the rule-based coordination language LO. Several enhancements of the basic model were motivated and illustrated, such as the additional knowledge of inter-argument dependency functions in the context of the broker's generators. It was shown how this additional knowledge helps the brokers to solve the problems of reuse of previously calculated results and of recursion control. Furthermore, we showed how additional sets of thresholds can increase the broker's ability to avoid unnecessary/redundant knowledge flow. The entire methodology presented here is related to the introduction of goal-directed backward-chaining in forward-chaining rule-based systems. An important outcome of the paper is that all the different extensions have been incorporated in a single framework simply through the tuning of basic rule templates. In general, our mathematical model provided an algebra of information tokens to formalize a world of distributed knowledge. Correspondingly, proven techniques from symbolic computation, suitable for solving traditional algebraic problems, were adapted to the task of symbolic manipulation of information tokens.

A complexity analysis was given. Table 3 summarized some results of this analysis. It became clear how the different refinements of the model have a favorable effect on strategic parameters such as the number of agents needed, the number of knowledge transformations, or the messages broadcast.

It was illustrated how the constraint-based knowledge broker model can be used to implement a broker-based chart-parser for unification grammars with feature terms [32]. The same approach can be adapted to Datalog [6, 7, 34] (using the magic set method), following the classical analogy between parsing and deduction (see the Earley deduction [27]).

We are adapting this model to the case of a partial request processor in the presence of distributed service providers. We further plan to enhance our model with the capability of handling not only incomplete requests but also incomplete answers. In this case, an incomplete answer to a subrequest could bring enough information for a threshold to be reached, and thus for a new subrequest to be launched. A suitable application domain for such a refinement is document merging where, for example, a first draft of a synthesis document can be provided as an (incomplete) answer based on components which are themselves incomplete drafts.

Finally, we investigate the problem of identifying appropriate application classes for our methodology, for which no obvious criterion is currently available.

References

[1] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[2] J-M. Andreoli, U. Borghoff, and R. Pareschi. Constraint based knowledge brokers. In *Proc. of PASCO'94*, Linz, Austria, 1994.

[3] J-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction abstract machines. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 257–280. MIT Press, Cambridge, Ma, U.S.A., 1993.

[4] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA'91*, Phoenix, Az, U.S.A., 1991.

[5] F. Arcelli, U. Borghoff, F. Formato, and R. Pareschi. Tuning constraint-based communication in distributed problem solving. In *Proc. of the 1st International Workshop on Concurrent Constraint Programming*, Venice, Italy, 1995.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th Symposium on Principles of Database Systems*, 1986.

[7] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3&4):255–299, 1991.

[8] U. Borghoff, R. Pareschi, F. Arcelli, and F. Formatto. Constraint based protocols for distributed problem solving. *Science of Computer Programming*, 29, 1995.

[9] U. Borghoff, R. Pareschi, H. Karch, M. Nöhmeier, and J.H. Schlichter. Constraint-based information gathering for a network publication system. In *Proc. of 1st Int'l Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, London, U.K., 1996.

[10] U. Borghoff and J. Schlichter. On combining knowledge from heterogeneous information repositories. Technical report, Rank Xerox Research Centre, Grenoble, France, 1995.

[11] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. of IPSJ'94*, Tokyo, Japan, 1994.

[12] K. Decker, Durfee E., and Lesser V. The evaluation of research in cooperative distributed problem solving. In Huhns M. and L. Gasser, editors, *Distributed Artificial Intelligence*. Pitman/Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1989.

[13] J. Denzinger. Knowledge-based distributed search using teamwork. In *Proc. of 1st Int. Conf. on Multi-Agent Systems (ICMAS'95)*, San Francisco, Ca, U.S.A., 1995.

[14] E. Durfee, V. Lesser, and D. Corkill. Trends in cooperative distributed problem solving. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):63–83, 1989.

[15] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 1970.

[16] R. Fikes, R. Engelmore, A. Farquhar, and W. Pratt. Network-based information brokers. In *Proc. of AAAI Spring Symposium on Information Gathering from Distributed Heterogeneous Environments*, Stanford, Ca, U.S.A., 1995.

[17] Object Management Group and X/Open. OMG, CORBA, the common object request broker: architecture and specification, 1991. OMG Document Number 91.12.1, Rev. 1.1.

[18] M. Henz, Smolka G., and J. Würtz. Object-oriented concurrent constraint programming in oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 27–48. MIT Press, Cambridge, Ma, U.S.A., 1995.

[19] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proc. of the 14th ACM POPL Symposium*, Munich, Germany, 1987.

[20] M. Kay. Algorithm schemata and data structure in syntactic processing. Technical report, Xerox Parc, Palo Alto, Ca, U.S.A., 1980.

[21] S. Lander and V. Lesser. Customizing distributed search among agents with heterogeneous knowledge. In *Proc. of 1st International Conference on Information and Knowledge Management*, Baltimore, Md, U.S.A., 1992.

[22] T. Le Provost and M. Wallace. Generalized constraint propagation over the clp scheme. *Journal of Logic Programming*, 16(3&4):319–359, 1993.

[23] T.W. Malone and K. Crowstone. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[24] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[25] C. Numaoka and M. Tokoro. A decentralized parsing method using communicating multiple concurrent objects. In *Proc. of 2nd International Conference of Technology of Object Oriented Languages and Systems*, Paris, France, 1990.

[26] T. Oates, M. Prasad, and V. Lesser. Cooperative information gathering: A distributed problem solving approach. Technical Report TR-94-66, Dept of Computing, University of Massachusetts, Amherst, Ma, U.S.A., 1994.

[27] F. Pereira and D.H.D. Warren. Definite clauses for language analysis. *Artificial Intelligence*, 30, 1980.

[28] F. Pereira and D.H.D. Warren. Parsing as deduction. In *Proc. of the 21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, Ma, U.S.A., 1983.

[29] W. Rounds and R. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proc. of the 1st IEEE Symposium on Logic in Computer Science*, Boston, Ma, U.S.A., 1986.

[30] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Pittsburg, Pa, U.S.A., 1989.

[31] V.A. Saraswat and P. Lincoln. Higher order linear concurrent constraint programming. Technical report, Xerox Parc, Palo Alto, Ca, U.S.A., 1992.

[32] G. Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[33] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Ma, U.S.A., 1989.

[34] L. Vieille. Recursive axioms in deductive databases: the query-subquery approach. In *Proc. of the 1st Conference on Expert Database Systems*, Menlo Park, Ca, U.S.A., 1986.

[35] A. Yonezawa and I. Ohsawa. Object-oriented parallel parsing for context-free gramars. In *Proc. of COLING'88*, Budapest, Hungary, 1988.