PAPER

Collaborative Constraint Functional Logic Programming System in an Open Environment

Norio KOBAYASHI[†], **Mircea MARIN**[††], ***Nonmembers, and*** **Tetsuo IDA**[††], ***Regular Member***

SUMMARY In this paper we describe collaborative constraint functional logic programming and the system called Open CFLP that supports this programming paradigm. The system solves equations by collaboration of various equational constraint solvers. The solvers include higher-order lazy narrowing calculi that serve as the interpreter of higher-order functional logic programming, and specialized solvers for solving equations over specific domains, such as a polynomial solver and a differential equation solver. The constraint solvers are distributed in an open environment such as the Internet. They act as providers of constraint solving services. The collaboration between solvers is programmed in a coordination language embedded in a host language. In Open CFLP the user can solve equations in a higher-order functional logic programming style and yet exploit solving resources in the Internet without giving low-level programs of distributions of resources or specifying details of solvers deployed in the Internet.

key words: *equational solving, functional logic programming, solver collaboration, constraint solving, CORBA*

1. Introduction

In our previous work [12], we presented a constraint functional logic programming system called CFLP, where the higher-order lazy narrowing calculus [11] and built-in constraint solvers collaborate to solve equational goals. Constraint functional logic programming is an integration of constraint programming, and functional and logic programming (FLP for short).*

An important observation to be made here is that FLP is a paradigm of solving equations over the domain of terms. In other words, FLP itself is also constraint programming aiming at solving constraints over domains described by rewrite systems. FLP holds a distinguished position in programming, since it manipulates terms directly without giving special meaning to terms. FLP alone, however, is not sufficient to solve equations modelling practical scientific problems. In real world applications, equations are often defined over several domains with terms interpreted specially in each domain. Each equation defined over a specific domain requires a dedicated constraint solver. Here the paradigm of constraint programming, which was originally proposed to extend the capability of logic programming, naturally comes into play. Constraint FLP is thus general constraint programming, where various constraint solvers interplay.

The language of FLP consists of equational goals and a set of rewrite rules. From programming language point of view, we need no extra linguistic constructs for constraint programming. In addition, constraint FLP requires constructs which specify how the constraint solvers collaborate to solve equations. These constructs add a new dimension to programming, *i.e.*, programming of collaboration of solvers. This fits very well in the advocated slogan [5]: programming = computation + coordination, where in our case coordination is collaboration of constraint solvers. A similar argument focusing on constraint solving has been made in [13].

In this paper, we present a new version of CFLP running in an open environment, where we program collaborations of solvers in addition to FLP. We call the new system Open CFLP since solvers are not fixed a priori and should be obtained dynamically from the open environment. Our work is based on the service model where solvers are distributed over the network and that they are not downloadable to the clients. The reasons for taking this premise are as follows:

- solvers evolve over time; sophisticated algorithms may require long time efforts for their perfection,
- solvers may be big and require specialized resources which are not downloadable,
- solvers are willing to provide services, but not necessarily willing to allow the clients to copy the programs to protect the intelligent and copy rights,
- solvers and the clients that use those services are independent and only communicate with the standardized protocol, and
- solving services may be deployed dynamically, independent of the plan of the potential clients.

The focus of the paper is the programming paradigm with Open CFLP, and on the architecture of Open CFLP. The rest of this paper is structured as follows. In Sect. 2 we describe a programming example that illustrates collaborative constraint programming. In Sect. 3 we describe the languages of our system.

Manuscript received January 4, 2002.
Manuscript revised September 10, 2002.

[†]Doctoral Program in Engineering, University of Tsukuba, Japan.
[††]Institute of Information Sciences and Electronics, University of Tsukuba, Japan.

*Functional and Logic programming is also an integration of functional programming and logic programming, but in this paper we take this integration for granted as the paradigm of functional and logic programming is amply discussed elsewhere [6].

In Sect. 4 and 5 we describe the architecture of Open CFLP. Finally, in Sect. 6 we draw conclusions.

2. Motivating example

We begin by a small motivating example to illustrate how we solve a problem with Open CFLP. The example is taken from [8]. The following presentation may appear a bit contrived to make clear the essence of our collaborative constraint functional logic programming. Let us consider the problem of solving the equations

$$y'(t) = k\ y(t),\ y(0) = 1,\ y(2) = 3,\ y(T) = 5$$

for variables y, k and T. Let us assume that we are functional programmers, and we decide to use the definition of `map` to make the program succinct. We further assume that `map` is not a built-in function, and so we write down its definition.

Our system is built on top of the Mathematica system [14], and we write the program in the language of Mathematica in a Mathematica notebook. Function `FLPProgram`, shown below, is a special function that treats its arguments as a FLP program. The result of the evaluation of the function call is the internal representation of the FLP program. The language of CFLP [12] is that of (conditional) pattern rewrite systems expressed in the syntax of Mathematica. The terms are simply-typed λ terms in $\beta\eta^{-1}$ normal form. The symbols on the left-hand side of the rewrite rules are underlined if they are free variables.

```
(* FLP program declaration *)
R = FLPProgram[
      {map[λ[{t}, f[t]], {}] → {},
       map[λ[{t}, f[t]], [H | T]] →
         [f[H] | map[λ[{t}, f[t]], T]]},
      Signature →
       {DefinedSymbols →
         {map : (ℝ → ℝ) × TyList[ℝ] → TyList[ℝ]}}]
```

Next we specify the goal to be solved. Logically a goal is an existentially quantified formula of conjunction of equations $\exists x_1 \cdots x_m.\, e_1 \wedge \cdots \wedge e_n$, where $e_1, \ldots, e_n$ are equations. In the language of CFLP, we write it as `exists`$[\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{m}\}, \{\mathtt{e}_1, \ldots, \mathtt{e}_\mathtt{n}\}]$. There are three kinds of equations; oriented equation `s` $\triangleright$ `t`, unoriented equation `s == t` and strict unoriented equation `s === t`. The former two are relevant to the present discussion. If the equation `s` $\triangleright$ `t` or `s == t` is in solved form, it is written as `s` $\mapsto$ `t`. The variables occurring in the formula can be type-annotated. Thus, the goal to be solved is as follows.

```
(* Goal specification *)
G = exists[{y : ℝ → ℝ, k : ℝ, T : ℝ},
      {λ[{t}, y'[t]] == λ[{t}, k y[t]],
       map[λ[{t}, y[t]], {0, 2, T}] == {1, 3, 5}}]
```

We now have to find solvers in the open environment. We may have solvers in our local computer, but here let us assume that the necessary solvers are on the Internet. Our computing model is that we are not allowed to download the solvers, but are allowed only to use the service of the solvers. We only know the names of the services that are available somewhere on the network. In order to uniquely identify the solving service, we use URI to name the service. For example, the URI `http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN` is the name of the service of solving equations over the domain of higher-order terms using Higher-Order Lazy Narrowing Calculus HOLN [11]. `FindSolvers`[`"http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"`] will find the solvers that offer the service of HOLN by the help of the broker of Open CFLP (*cf.* Sect. 5.2). Similarly, we will find the solvers for systems of differential equations and systems of linear equations.

```
(* Find solvers *)
aHOLN = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"]
aDeriv = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/Deriv"]
aLinear = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/Linear"]
```

`FindSolvers` returns the entity *elementary collaborative*. An elementary collaborative is a collection of basic solvers that perform the same solving service. In Sect. 5.2, we will explain the mechanism of finding such solvers. An elementary collaborative can be configured by `ConfigSolvers` to work on specific problems more effectively by supplying additional parameters. To some solvers, configuration is essential to equip them with necessary solving power. For example, in the case of HOLN solvers, we need to supply a FLP program `R` to solve equations with respect to `R`. Thus, we configure the HOLN solvers as follows.

```
(* Configure solvers *)
aHOLN = ConfigSolvers[aHOLN, Prog → R]
```

The solvers of other two services need not be configured since we use their standard services.

The next step of programming is the important one in Open CFLP. We program a collaborative solver. In CFLP, collaboration of solvers is fixed and is *hard-wired* to CFLP system. For this example, we define a collaborative solver in which elementary collaboratives `aHOLN`, `aDeriv` and `aLinear` collaborate. First, we want to apply the goal to these elementary collaboratives sequentially in the order of `aHOLN`, `aDeriv` and `aLinear`. This is programmed as `seq`$[\{$`aHOLN`, `aDeriv`, `aLinear`$\}]$. Furthermore, we apply the goal to `seq`$[\{$`aHOLN`, `aDeriv`, `aLinear`$\}]$ repeatedly until the goal reaches the fixed-point. Thus we have a collaborative definition `repeat`[`seq`[{`aHOLN`,

aDeriv, aLinear}]]. The definition is given to the system using a special function NewCollaborative.

```
(* Collaborative specification *)
aCollabo = NewCollaborative[
            repeat[seq[{aHOLN, aDeriv, aLinear}]]]
```

NewCollaborative returns the entity *collaborative*.

Finally, we apply the goal G to the collaborative aCollabo:

```
(* Invoke collaborative *)
ApplyCollaborative[aCollabo, {G}]
```

and obtain the following solution.

$$\{\{\mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}], \mathtt{k} \mapsto \mathtt{Log[3]/2},\\ \mathtt{T} \mapsto 2\ \mathtt{Log[5]/Log[3]}\}\}$$

The following transformation of goals took place during the solving process of the initial goal G.

$$\{\mathtt{G}\} \Rightarrow_{\mathtt{aHOLN}} \{G_1\} \Rightarrow_{\mathtt{aDeriv}} \{G_2\} \Rightarrow_{\mathtt{aLinear}} \{G_3\}\\ \Rightarrow_{\mathtt{aHOLN}} \{G_3\} \Rightarrow_{\mathtt{aDeriv}} \{G_3\} \Rightarrow_{\mathtt{aLinear}} \{G_3\}$$

where †

$$\{G_1\} = \{\mathtt{exists}[\{\mathtt{h}, \mathtt{T_1}\}, \{\mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{h[t]}],\\ \mathtt{T} \mapsto \mathtt{T_1}, \lambda[\{\mathtt{t}\}, \mathtt{h'[t]}] == \lambda[\{\mathtt{t}\}, \mathtt{k\ h[t]}],\\ \mathtt{h[0]} == 1, \mathtt{h[2]} == 3, \mathtt{h[T_1]} == 5, \ldots\}]\},$$

$$\{G_2\} = \{\mathtt{exists}[\{\mathtt{c}, \mathtt{k}, \mathtt{T_1}\}, \{\mathtt{h} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{c\ e^{k\ t}}],\\ \mathtt{T} \mapsto \mathtt{T_1}, \mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{c\ e^{k\ t}}],\\ \mathtt{c} == 1, \mathtt{c\ e^{2\ k}} == 3, \mathtt{c\ e^{k\ T_1}} == 5, \ldots\}]\},$$

$$\{G_3\} = \{\{\mathtt{c} \mapsto 1, \mathtt{T_1} \mapsto 2\ \mathtt{Log[5]/Log[3]}, \mathtt{k} \mapsto \mathtt{Log[3]/2},\\ \mathtt{h} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}], \mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}],\\ \mathtt{T} \mapsto 2\ \mathtt{Log[5]/Log[3]}, \ldots\}\}.$$

Note that the collaborative operates on a list of goals and returns a list of goals and that $\{G_1\}$ can be obtained if we evaluate ApplyCollaborative[NewCollaborative[aHOLN], {G}].

3. Languages of Open CFLP

From the example in the previous section we see that collaborative constraint functional logic programming proceeds in the following steps:

1. Define a FLP program R.
2. Define a goal G to be solved.
3. Obtain sets $C_1, \ldots, C_n$ of elementary collaboratives.
4. Define a new collaborative C by specifying a collaboration of elementary collaboratives $C_1, \ldots, C_n$.
5. Apply the collaborative C to G.

†HOLN introduces extra variables that are irrelevant in this illustration. We have abbreviated their bindings in G_1, G_2 and G_3 by ....

In the scientific problem solving the above process is often interactive. For instance the obtained solution may become new goals after inspecting and editing it. Furthermore even if the FLP program R is completed, the steps 2 to 5 are repeated. This requires the collaboration of solvers to be re-programmed since some combination of solvers may not deliver a desired solution. Therefore, our language should also have the power of modern programming languages such as graphics and interactive debugging. This observation leads to the following design decision.

3.1 Language $\mathcal{M}$

The language of Open CFLP is built on top of Mathematica. We decided to use Mathematica as a base language since we need to make use of its built-in mathematical knowledge and symbolic processing capability. Let us denote our language by $\mathcal{M}$ in this paper. Since the syntax of the language of Mathematica is universal in that a form of functional application like $f[s_1, \ldots, s_n]$ is a basic building block, extending the functionalities that we have shown in the previous section can be made simply by providing special functions to Mathematica. We have seen functions like FindSolvers, NewCollaborative and ApplyCollaborative in our example. Functions FindSolvers, NewCollaborative and ApplyCollaborative are by no means trivially implemented by Mathematica programs. Rather they are realized by sophisticated software components that we have built for Open CFLP.

A collaborative is programmed in a coordination language embedded in $\mathcal{M}$. The language is denoted by $\mathcal{L}$ in this paper. For the definition of a collaborative in $\mathcal{M}$, we use function NewCollaborative whose argument is a collaborative expression in $\mathcal{L}$.

3.2 Coordination Language $\mathcal{L}$

Open CFLP system has a software component called *coordinator*. The coordinator, as the name suggests, coordinates the activities of solvers. The coordinator evaluates the program of $\mathcal{L}$. An element of $\mathcal{L}$ is collaborative. The collaborative C and the list $\overline{G}$ of goals are sent from the user frontend of the system (*cf.* Sect. 4.1) in the form of a function application ApplyCollaborative$[C, \overline{G}]$.

Language $\mathcal{L}$ allows us to define a new solver by combining various solvers using collaboration combinators seq, if, choice and repeat. A collaborative $\mathcal{C}$ is inductively defined as follows:

$\mathcal{C} ::= \mathcal{B}$	elementary collaborative
$\mid \mathcal{X}$	locally defined collaborative
$\mid \mathtt{seq}[\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}]$	sequential
$\mid \mathtt{if}[\phi, \mathcal{C}_1, \mathcal{C}_2]$	conditional
$\mid \mathtt{choice}[\psi, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}]$	concurrency and choice
$\mid \mathtt{repeat}[\mathcal{C}]$	repetition

A collaborative receives a list of goals. All the solutions of the goals are collected and is returned as the solution of the given list of the goals.

We will give informal semantics of the language. Suppose a collaborative C and a list $\overline{G}$ of goals are given initially.

- When C is an elementary collaborative, C is applied to $\overline{G}$ directly.
- When C is a locally defined collaborative, where 'locally defined' means that C is defined as a Mathematica function, the definition of C is interpreted by the user frontend.
- When C is $\texttt{seq}[\{C_1, \ldots, C_n\}]$, we distinguish the following two cases. If $n = 1$, $\texttt{seq}[\{C_1\}]$ is the same as C_1. Otherwise, $\texttt{seq}[\{C_2, \ldots, C_n\}]$ is applied to the result of application of C_1 to $\overline{G}$.
- When C is $\texttt{if}[\phi, C_1, C_2]$, the following takes place. $\phi(\overline{G})$ is computed first. Function ϕ probes the list of goals $\overline{G}$ and checks whether each goal of $\overline{G}$ possesses a certain property, *e.g.*, size or linearity. $\phi(\overline{G})$ returns a list $\{\overline{G_T}, \overline{G_F}\}$, where $\overline{G_T}$ is a list of goals which satisfy the property, and $\overline{G_F}$ is a list of goals which do not satisfy it. Finally the collaboratives C_1 and C_2 are applied to $\overline{G_T}$ and $\overline{G_F}$ respectively.
- When C is $\texttt{choice}[\psi, \{C_1, \ldots, C_n\}]$, the collaboratives $C_1, \ldots, C_n$ are applied to $\overline{G}$ simultaneously and ψ selects one of the results.
- When C is $\texttt{repeat}[C]$, the collaborative C is repeatedly applied to the goal list (initially $\overline{G}$) until no further transformation of the goal list is possible by C.

Language $\mathcal{L}$ is designed to be a small language with primitives for coordination of solvers. It is a very basic language for two reasons. First, $\mathcal{L}$ is a language embedded in a powerful language $\mathcal{M}$. Secondly, additional functionalities that may be desired for full-fledged coordination languages can be provided by $\mathcal{M}$. For example, `repeat` can be easily programmed in $\mathcal{M}$ using the built-in higher-order function `FixedPoint`, as follows.

```
Repeat[collabo_] := Function[GList,
   FixedPoint[Function[x,
      ApplyCollaborative[collabo, x]], GList]]
```

For efficiency reasons, `repeat` is provided as primitive, however.

Similarly, we show that the sequential version of conditional collaboration `CondCollabo` can be defined as a Mathematica function.

```
CondCollabo[Choice_, collabo1_, collabo2_]
   := Function[GList, Union[
         ApplyCollaborative[collabo1,
            Choice[GList][[1]]],
         ApplyCollaborative[collabo2,
            Choice[GList][[2]]]]];
```

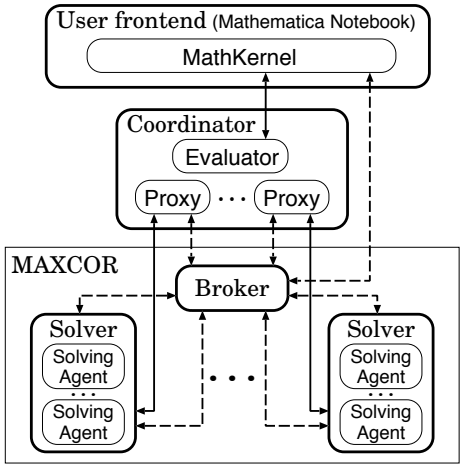


Fig. 1 The architecture of Open CFLP

```
aCondCollabo_M
   := CondCollabo[aChoice, aCollabo1, aCollabo2]
```

delivers the same result as the collaborative defined by

```
aCondCollabo_L
   = NewCollaborative[
        if[aChoice, aCollabo1, aCollabo2]],
```

although $\texttt{aCondCollabo}_\texttt{L}$ is more efficiently executed, because expression $\texttt{aCondCollabo}_\texttt{M}[\texttt{GList}]$ is interpreted by the user frontend, whereas $\texttt{ApplyCollaboarive}[\texttt{aCondCollabo}_\texttt{L}, \texttt{GList}]$ is interpreted by the coordinator.

If the choice function `aChoice` in $\texttt{aCondCollabo}_\texttt{L}$ is provided as the primitive in $\mathcal{L}$, running in the coordinator, the performance difference is even greater. However, there are cases where simplicity and flexibility outweigh the execution efficiency, in which case `CondCollabo` defined as the higher-order function of Mathematica would be preferable. For example, `aChoice` that checks whether each goal is a system of linear equations can be programmed much more easily with Mathematica.

Language $\mathcal{L}$ can be seen as a core of coordination languages that have been studied by several researchers (*cf.* [9], for a good survey).

4. Architecture of Open CFLP

The discussion about the program in Sect. 2 has revealed the following ingredients of Open CFLP:

- user frontend,
- coordinator,
- open framework for solver collaboration, and
- solvers.

These ingredients are realized by software components called *user frontend*, *coordinator*, *broker* and *solver*.

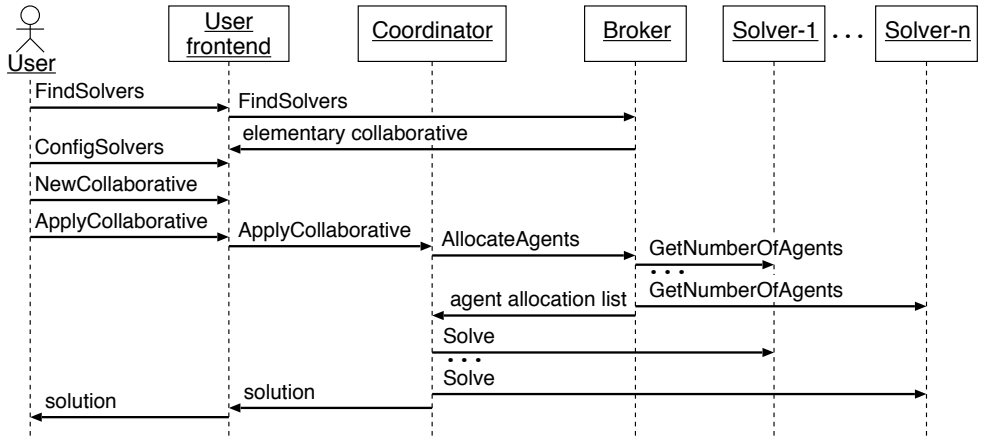


Fig. 2 Interaction between components of Open CFLP

Figure 1 shows the architecture of Open CFLP. Figure 2 is a sequence diagram that shows how the components interact one another. Functionally, the broker and the solvers are grouped together to form a framework called MAXCOR(**MA**th e**X**change for **COR**BA). Since MAXCOR has many functionalities, we discuss the detail of MAXCOR separately in Sect. 5. In the following subsections we will explain each component in more detail.

4.1 User frontend

The user frontend (frontend for short) is a user interface to Open CFLP. It is a Mathematica notebook equipped with the MathKernel and allows the user to define CFLP programs in $\mathcal{M}$, as well as to interact with the Mathematica system. The MathKernel executes locally defined collaboratives as well as Mathematica programs that run with Open CFLP. The interface component called palette is also provided to facilitate the input of CFLP expressions and commands.

4.2 Solver

A solver is a solving service provider. It implements a solving algorithm such as higher-order lazy narrowing, Gaussian elimination method and Gröbner basis algorithm. When the operation `Solve` is invoked by the coordinator (*cf.* Fig. 2), the solver creates (possibly multiple) solving processes. We call the solving processes *solving agents*. The solving agents of a solver execute the same solving algorithm in parallel.

4.3 Coordinator

The coordinator is in charge of evaluating the program of $\mathcal{L}$. The evaluation consists in creating the proxies for the solvers and in interpreting the collaboratives as described in Sect. 3.2. The main component of the coordinator is the *evaluator*. It communicates with the frontend and the broker. When the evaluator receives a collaborative C and a list $\overline{G}$ of goals from the frontend, it creates a buffer U where the solutions are stored and then executes the procedure `Collabo` given in Appendix. When the execution of the procedure `Collabo` is completed, the evaluator fetches the solution stored on U and returns it to the frontend.

We will explain the procedure `Collabo` when C is an elementary collaborative. The procedure in other cases is a straightforward translation of the informal semantics explained in Sect. 3.2.

The important design issue here is how to exploit the parallelism. Note that C is a collection of basic solvers $s_1, \ldots, s_m$ and $\overline{G}$ is a list of goals $G_1, \ldots, G_n$. Each solver s_i can spawn τ_i solving agents. Ideally, n goals are solved in parallel by n solving agents. The procedure `Collabo` will exploit parallelism afforded in the open environment in the following way.

1. Contact the broker and obtain the number τ'_i of currently assignable solving agents for each solver s_i, $i = 1, \ldots, m$ such that $\tau'_i \leq \tau_i$ and $\tau'_1 + \cdots + \tau'_m = n' \leq n$.
2. Create a proxy of each solver s_i if $\tau'_i \neq 0$, for $i = 1, \ldots, m$.
3. Distribute n goals among n' solving agents via the proxies.
4. Configure the solving agents if necessary and then trigger the n' solving agents to solve the given goals via their proxies.
5. Probe the states of the computations via the proxies. Delete the proxies if all the computations are completed, and send the request to the broker to de-allocate all the n' involved solving agents.
6. Return the solution to the frontend.

Here, The procedure `Collabo` exploits n'-fold parallelism. Figure 2 shows the sequence of actions taken by the coordinator, the broker and the solvers.

We illustrate the behavior of the coordinator by the example of Sect. 2. When `ApplyCollaborative`[

aCollabo, {G}] that is the operation defined in the coordinator is evaluated, aCollabo is first evaluated to repeat[seq[{aHOLN, aDeriv, aLinear}]] then ApplyCollaborative[repeat[seq[{aHOLN, aDeriv, aLinear}]], {G}] is sent to the coordinator.

1. The evaluator creates a buffer U_0.
2. The evaluator calls the procedure Collabo with the parameters repeat[seq[{aHOLN, aDeriv, aLinear}]], {G} and U_0.
3. Collabo creates a buffer U_1 and recursively calls Collabo with the parameters seq[{aHOLN, aDeriv, aLinear}]], {G} and U_1.
   3.1. Collabo creates a buffer U_2.
   3.2. Collabo obtains a solving agent HOLN from the broker and creates the proxy of the solver of HOLN.
   3.3. The proxy sends the FLP program R for configuring this solver and sends {G} to this solver by invoking Solve operation.
   3.4. HOLN agent solves {G} and appends the solution $\{G_1\}$ to U_2.
   3.5. Similarly, Deriv agent and Linear agent solve their goals in this order.
4. The result $\{G_3\}$ of Step 3 is appended to the buffer U_1, and the evaluator compares U_1 and {G}.
5. Steps 2–4 are obeyed once more with the parameters repeat[seq[{aHOLN, aDeriv, aLinear}]], $\{G_3\}$ and U_0.
6. Finally, the solution $\{G_3\}$ is appended to U_0.

We assume that a solving agent returns finite number of solutions, since the evaluation is call-by-value, following the default evaluation mode of Mathematica. The coordinator can display the progress of solving. In the case the computation appears be non-terminating, the user can interrupt the computation.

Furthermore, for efficiency the coordinator has a cache of solving agents, although the description of the cache is not explicit in the procedure Collabo. The cached solving agents are reused until the CFLP session is over.

5. MAXCOR

MAXCOR is designed to be a framework which realizes transparent communication between solvers. It consists of object wrappers for those solvers and the broker.

We often want to use various existing solvers such as HOLN and polynomial solvers integrated to Mathematica and integer programming solver CPLEX [2]. These solvers may be heterogeneous, *i.e.*, they are implemented in different programming languages, run on different platforms and have different data formats for describing constraints. Object wrappers are used to hide the heterogeneity of such solvers. To realize this functionality, we need a common data format for communication and a common communication protocol. We adopted MathML for the common data format and CORBA for the communication protocol. An object wrapper thus has the functionalities of data conversion between MathML and the solvers' own data formats, and of mediation of solvers' operations.

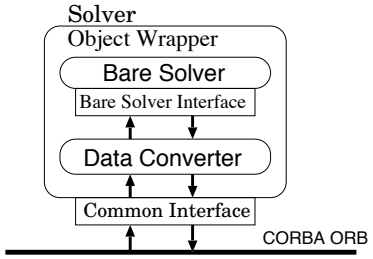


Fig. 3 The architecture of the object wrapper

An object wrapper alone is not sufficient to realize the intended transparent communication with the coordinator. MAXCOR has to provide the facilities for finding the solvers. We are going to discuss in more detail about these components.

5.1 Object Wrappers

Figure 3 shows the architecture of the object wrapper. It provides CORBA compliant common interface for solvers and realizes a solver using a bare solver. The object wrapper transmits MathML data, more specifically valuetype objects of XML DOM (Document Object Model) [1], the standard allowing solvers to transmit mathematical formulas written in MathML on CORBA ORB. The object wrapper is equipped with the data converter between MathML and the bare solver's own internal representation of mathematical formulas. We have implemented object wrappers for Mathematica and CPLEX.

5.2 Broker

The scheme of broker–provider coordination is a well understood design pattern [10], and we follow the scheme for broker–solver coordination. Solvers are providers that offer solving services and the broker finds the appropriate service to its clients. Here the clients are proxies created by the coordinator.

The broker communicates with the frontend and the coordinator. It receives the command FindSolvers[*uri*] from the frontend, where *uri* is a name of the service. Recall that the service of solving a goal is given a unique name in the format of URI. From the coordinator, it receives the request of the process allocation for running the solving agent.

(1) Search of solvers

When the broker receives FindSolvers[*uri*], it accesses

a *service table* and then by simple look-up with *uri* it finds a list of solvers associated with *uri*. This list will be returned to the frontend. The broker maintains the service table with entries of the name of the service and a list of solvers whose agents perform the service. A solver published the availability of service by requesting the broker the registration of the solver with the name of the service in the service table.

(2) Agent allocation

When the coordinator requests n solving agents for the elementary collaborative C by invoking the operation `AllocateAgents`$[C, n]$(*cf.* Fig. 2), the broker returns at most n solving agents in the following way:

1. The broker asks each solver $s_1, \ldots, s_m$ collected in C the number of solving agents it can give to the coordinator by invoking the operation `GetNumberOfAgents`.
2. Solver s_i returns the number τ_i, for $i = 1, \ldots, m$.
3. Based on τ_i, the broker distributes n solving agents among m solvers according to its load balancing policy. Let $n' = \min(n, \tau_1 + \cdots + \tau_n)$. The broker decides τ'_i for each s_i such that $\tau'_i \leq \tau_i$ and $\tau'_1 + \cdots + \tau'_m = n'$.
4. The broker informs each solver $s_1, \ldots, s_m$ that the broker reserves $\tau'_1, \ldots, \tau'_m$ solving agents.
5. The broker returns an agent allocation list $\{\{s_1, \tau'_1\}, \cdots, \{s_m, \tau'_m\}\}$ to the coordinator.

5.3 Features of MAXCOR

MAXCOR is designed as an application of CORBA. The broker and the solvers are CORBA servers, and the frontend and the proxies in the coordinator are CORBA clients. Combined with the features of CORBA, we have achieved the following properties:

- portability—language and platform independence among solvers,
- data interoperability—MathML documents,
- operation interoperability among solvers,
- scalability and modularity—solvers implemented as CORBA servers, and
- location independence of solvers.

Furthermore, we can easily extend our system with rich CORBA common services such as the security service.

In addition, we implemented the broker that CORBA does not offer to achieve dynamic solver deployment. It has the functionalities of searching solvers and allocating solving agents. Furthermore, our broker implements a load balancing policy for the allocation of solving agents, hence realizes efficient use of distributed resources.

6. Conclusions

We have described collaborative constraint functional logic programming and a system that supports this paradigm. The system is open in the sense that it can access via a brokering service the constraint solving resources available in an open environment.

The language of Open CFLP is multi-tiered. The notions of constraint solving, coordination programming and symbolic computation are separable in developing programs, and such a separation leads to a succinct and modular programming style. Yet, each activity is supported by existing programming capabilities. Our languages $\mathcal{M}$ and $\mathcal{L}$ are embedded into the language of Mathematica. This flexibility allows us to make our coordination language $\mathcal{L}$ small. $\mathcal{L}$ is essentially a core of the coordination language BALI [3], [13].

In [3], [13], an implementation model of BALI with MANIFOLD is discussed. Our coordinator and MANIFOLD are based on a control-driven coordination model such as ConCoord [7] and TOOLBUS [4]. The distinctive feature of Open CFLP is that it has been implemented using middleware technology CORBA and further has realized broker-provider scheme needed for open collaborative constraint solving. By this approach, we have achieved openness and extensibility of the system. As far as we know, a collaborative constraint system fully implemented with open technologies, CORBA and MathML, with clear design goals of scientific problem solving is our new contribution in this field of collaborative constraint programming.

References

[1] `http://cgi.omg.org/xml`.
[2] Using the CPLEX Callable Library, CPLEX Optimization, Inc., USA, 1995.
[3] F. Arbab and E. Monfroy, "Coordination of heterogeneous distributed cooperative constraint solving,", Applied Computing Review, SIGAPP. ACM, vol.6, pp.4–17, 1998.
[4] J. Bergstra and P. Klint, "The TOOLBUS coordination architecture," in 1st Int'l Conf. Coordination Models, Languages and Applications (Coordination'96), ed. P. Ciancarini and C. Hankin, LNCS, vol.1061, pp.75–88, Springer-Verlag, 1996.
[5] D. Gelernter and N. Carriero, "Coordination languages and their significance: From theory to practice," Commun. ACM, vol.35, no.2, pp.97–107, 1992.
[6] M. Hanus, "The integration of functions into logic programming: From theory to practice," Journal of Logic Programming, no.19&20, pp.583–628, 1994.
[7] A. Holzbacher, "A software environment for concurrent coordinated programming," in 1st Int'l Conf. Coordination Models, Languages and Applications (Coordination'96), ed. P. Ciancarini and C. Hankin, LNCS, vol.1061, pp.249–266, Springer-Verlag, 1996.
[8] H. Hong, "RISC-CLP(CF) Constraint logic programming over complex functions," Logic Programming and Automated Reasoning, Proc. of the 5th Int'l Conf., LPAR'94, ed. F. Pfennig, pp.99–113, Springer-Verlag, 1994.
[9] A. Omicini and F. Zambonelli,ed., Coordination of Internet Agents, Springer-Verlag, 2001.
[10] M. Klusch and K. Sycara, "Brokering and matchmaking for coordination of agent societies: A survey," in Coordina-

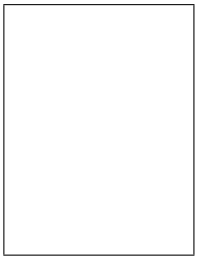
tion of Internet Agents, ed. A. Omicini and F. Zambonelli, pp.197–224, Springer-Verlag, 2001.
[11] M. Marin, T. Ida, and T. Suzuki, "Higher-order lazy narrowing calculus: A solver for higher-order equations," in Proc. 8th Int'l Conf. Computer Aided Systems (EuroCAST 2001), LNCS, vol.2178, pp.478–493, Springer-Verlag, 2001.
[12] M. Marin, T. Ida, and W. Schreiner, "CFLP: Distributed constraint solving system," The Mathematica Journal, vol.8, no.2, pp.287–300, 2001.
[13] E. Monfroy and F. Arbab, "Constraints solving as the coordination of inference engines," in Coordination of Internet Agents, ed. A. Omicini and F. Zambonelli, pp.399–419, Springer-Verlag, 2001.
[14] S. Wolfram, The Mathematica Book, 4th Edition, Wolfram Media Inc. Champaign, Illinois, USA, and Cambridge University Press, 1999.

Appendix: Procedure `Collabo`

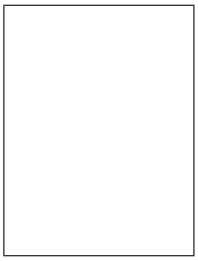
procedure `Collabo`(C, L, U)

{Inputs are collaborative C, a list L of n goals and a buffer U.}

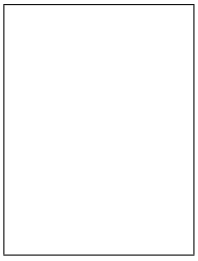
case C is an elementary collaborative
begin
Obtain the agent allocation list $\{\{s_1, \tau_1'\}, \ldots, \{s_m, \tau_m'\}\}$ from the broker, where $s_1, \ldots, s_m$ are solvers of C;
for $i = 1, \ldots, m$ **do**
if $\tau_i' \neq 0$ **then** Create a proxy of s_i;
Distribute L to the proxies;
Send configuration requests if C includes parameters for configuration, and solving requests to the proxies;
Obtain the solutions from the proxies;
Append each solution to U;
Delete the proxies and issue a request to the broker to de-allocate the solving agents;
end;
case C is a locally defined collaborative X
begin
Send L to the frontend;
{The frontend evaluates `X[L]`.}
Append to U the result sent from the frontend;
end;
case $C =$ `seq`$[\{C_1, \ldots, C_k\}]$
if $k = 1$ **then call** `Collabo`(C_1, L, U);
else
begin
Create a buffer U';
call `Collabo`(C_1, L, U');
call `Collabo`$(\texttt{seq}[\{C_2, \ldots, C_k\}], U', U)$;
end;
case $C =$ `if`$[\phi, C_1, C_2]$
begin
Let $\{L_T, L_F\} = \phi(L)$;
call `Collabo`(C_1, L_T, U) and
call `Collabo`(C_2, L_F, U) simultaneously;
end;
case $C =$ `choice`$[\psi, \{C_1, \ldots, C_k\}]$
begin
Create k buffers $U_1, \ldots, U_k$;
call `Collabo`$(C_1, L, U_1), \ldots,$
call `Collabo`(C_k, L, U_k) simultaneously;
Append $\psi(U_1, \ldots, U_k)$ to U;
end;
case $C =$ `repeat`$[C_r]$
begin
Create a buffer U';
call `Collabo`(C_r, L, U');
if the contents of the buffer U' and L are the same **then** Append L to U;
else call `Collabo`(C, U', U);
end;
end of `collabo`;



Norio Kobayashi is currently a doctor course graduate student in engineering, University of Tsukuba. His research interests include scientific equational solving and open computing model for collaborative equational solving. He received the B.Sc. degree in physics from Science University of Tokyo and M.E. degree in information sciences and electronics from University of Tsukuba in 1997 and 1999, respectively.



Mircea Marin is a visiting researcher at the University of Tsukuba. His current research interests include integration of functional logic programming with constraint solving, and coordination models for collaborative constraint solving. He holds a Ph.D. in Computer Science from the Johannes Kepler University of Linz, Austria.



Tetsuo Ida is a professor at the University of Tsukuba, where he leads a research group of symbolic computation (SCORE) in the institute of information sciences and electronics. His research includes distributed symbolic computation, integration of functional and logic programming and term rewriting. He is an editor of the Journal of Symbolic Computation and the Journal of Functional and Logic Programming. He is a member of the IEICE, the IPSJ, the JSSST, the ACM and the IEEE Computer Society. He received a Doctor of Science from the University of Tokyo.