A Soft Trading Service for COTS Components Using Parametric Contracts

Bo Ma, Ralf Reussner, Yi Zhang

1Laboratory for Internet Software Technology
Institute of Software, Chinese Academy of Science
No.4 South Fourth Street, Zhong Guan Cun, Beijing, China
mabo, zhangyi@itechs.iscas.ac.cn

2Software Engineering Group
Department of Computing Science
University of Oldenburg / OFFIS
Escherweg 2, D-26121 Oldenburg, Germany
reussner@informatik.uni-oldenburg.de

Abstract. Currently, various efforts are concerned with the creation of domain-specific components. However, the use of repositories of such components requires efficient methods for component selection. Current component retrieval and trading services are limited when dealing with mismatches between components and architectural requirements. In this paper, we analyze the required features that soft trading services should have, and describe soft trading algorithms used to implement a soft trading service based on parametric contracts. In addition, we validate our approach by presenting soft trading experiments and an analysis of the results. The results show an significant improvement against existing approaches for component selection which do not use parametric contracts.

1 Introduction

Currently, software components are used either (a) as domain-independent and relatively low-level entities of reuse (such as container-classes in the C++ STL [17] or Java util classes [18]) or (b) as plug-in components specifically developed for pre-defined plug-in interfaces (e.g., consider applications such as Mozilla, Adobe Photoshop, etc.) The case (a) can be considered a kind of primitive, low level “bottom up reuse” of components, as components are defined prior to the system construction and their reuse case. Opposed to that, case (b) can be considered as “top-down use” of components, as components are developed after the definition of plug-in interfaces and during or after the development of the system they are reused in. While in the top-down use case rather sophisticated and domain-specific components exist (e.g., the various media plug-ins for browsers), the bottom-up reuse case only created rather low level domain-independent components. However, currently large efforts are

---

1 Supported by the National High-Tech Research and Development Plan of China under Grant No. 2004AA1Z2050
invested in the creation of standardized high-level domain-specific components and
compartment platforms, such as my-SAP [19] or Microsoft’s “Business Solutions” [20].
In addition, companies are also concerned with the setup of in-house repositories of
high-level, domain-specific components (e.g., the Shanghai Baosteel Group
Corporation). These efforts can be seen as an attempt to raise the bottom-up
component reuse approach to more domain-specific and complex components, and
hence to make component reuse more beneficial.

Anyhow, any bottom-up component reuse scenario requires methods to efficiently
deal with component selection [2]. In particular, any component market-place (in-
house or open) needs such mechanisms to enable component trading. Although this
shift to bottom-up reuse of high-level components is far from being completed, one
can notice a shift from development-centric processes towards procurement-centric
approaches [1].

In procurement-centric approaches, an abstract architecture is defined first, which
specifies abstract roles and represents the functional and extra-functional
requirements of the application. These abstract roles are then matched against the list
of concrete components available in the repository. This process produces a list of
candidate components that could form a part of the architecture, (a) because they
provide some of the functions required by the application, and (b) because they fulfill
some of the extra-functional requirements such as security, performance, etc. With
this list, the architecture is re-examined in order to accommodate as much candidates
from the list as possible, and to produce a group of solutions, i.e., the combination of
candidate components. At last, the final composed applications are picked up among
these solutions according to some criteria.

Under this new setting, effective search and selection processes of COTS
components have become one of the cornerstone of any effective COTS component-
based development [2].

The concept of the trader was first formally specified by ANSA [10]. There are a
number of trading systems available, such as the CORBA trader [11], the ANSAware
software model [10], the Aster project [12], the Matchmaking framework [13], the
MAGNET trader [14]. Applying the trading service to CBSD occurs recently, as
proposed in [2], [6], [15], [16].

During the matching process, component adaptations are usually necessary because
in reality mismatches exist (a) between the component requires-interface(s) and the
trading context, which is formed by the combination of candidate components, or (b)
between the component provides-interface(s) and the architecture expectations.
These mismatches come up, if components are re-used in composition contexts for
which they are originally not developed for. However, this is in line with components
as third party developed, deployable and composable units [3]. The trading service
should carry out partial matches and also select those components that may provide
(part of) a required service as candidates. This is proposed in [2], and is identified as
“soft matching”.

However, current component trading services present some limitations when
dealing with soft matching, generally due to the following three main reasons.

Firstly, a component model with static types only leads to the functionality-reuse
problem and the type-extension problem [4]. Actually, these two problems hinder
effective component trading.
Secondly, an overly simplified search and selection criteria gives little or no thought to component adaptation during trading. This can lead to either combination explosion, as the number of component increase if the criteria is too loose, or insufficient combinations if it is too rigorous.

Thirdly, an inflexible architecture specification neglects the negotiation between user requirement and available components in different contexts. This can lead to a low ratio of useful solutions to total solutions. Many if not most solutions are either unclosed solutions, that is, requires-interfaces of some components are not satisfied by provides-interfaces of other components, or none-architecture-compliant solutions which do not respect the architectural structure, that is, the distribution of interfaces does not coincide with architecture specification.

The contribution of this paper is a solution to the above problems which comprises three parts: (1) dynamic component types with parametric contracts [5], (2) prioritized architecture definition, and (3) a group of soft trading algorithms.

The remainder of the paper is organized as follows. Section 2 presents the design of our soft trading service, including the analysis of the trading process, fundamental definitions and a group of related soft trading algorithms. Section 3 presents a case study of using soft trading service to build an example application. Section 4 analyzes the experimental results we gained by comparing with the results of traditional trading services. Finally, section 5 and 6 relate this work to other similar approaches and draw some conclusions, respectively.

2 A Soft Trading Service

2.1 Understanding the Trading Process

Traditional trading processes, as described above, comprise selecting candidate components, matching candidate components against a given architectural specification, and filtering out unclosed and none-architecture-compliant solutions. There are three limitations of this approach:
1. Coarse selection criteria. Given a component C with provides-interfaces \( P \) and requires-interfaces \( R \), an architecture A with functions (which can be represented by a group of interfaces as defined in section 2.2) \( F \), C is considered as candidate component if and only if \( C.P \cap A.F \neq \emptyset \). Thus, all C's provides-interfaces are kept as they are, which can lead to a combinatorial explosion of the state space during the matching process. For instance, consider a small repository contains 5 components with 10 interfaces, which will produce \( 2^{10} = 1024 \) combinations.
2. The ratio of closed solution is too low. As reported in [6], there are only 4 closed in 512 solutions, the ratio is \( 4/512 = 0.78\% \). This is the consequence of a coarse selection: not all provides-interfaces of the candidate component are useful for a given architecture. If all provides-interfaces of a component are kept, then one has to satisfy all the requires-interfaces of this component, because a component can only offer the functionality described in its the provides-interfaces if all the
functionality requested in their requires-interfaces is fulfilled. Not all requires-interfaces can always be satisfied, which makes the solution unclosed. In other words, we have to pay for what we do not need.

3. *Neglect negotiation with user.* Actually, some functions in an application are auxiliary in certain contexts for certain users. We can abandon some non-critical functions if the user agrees, and get a degenerated but usable application. For instance, for a video mail client component, if the sound player component is missing in the environment, the user may choose to use it as an ordinary mail client component instead of abandoning the whole application.

![Fig. 1. The comparison of a traditional trading process (top) and a soft trading process (bottom)](image)

Based on the analysis above, we design a novel soft trading service by extending the traditional trading process by a soft selection facility with provides-interfaces pruning, which prunes away those provides-interfaces that are useless for the current application. This pruning improves the solution’s fit to the architecture. Our soft solution uses requires-parametric-contracts pruning, which degenerates the architecture to obtain a usable application and improves the matching process by provides-parametric-contracts pruning, which prunes away the redundant requires-interface of candidate components. The comparison of traditional (i.e., exact) trading processes and soft trading process is shown in figure 1.

### 2.2 Basic Definitions

In this paper, an interface is a set of named operations that can be invoked by users. A component will have multiple interfaces corresponding to different access points [3]. Furthermore, an interface specification is isolated from any particular component that provides or requires an implementation of such an interface.

**Definition 1 (Component with dynamic type)** A component $C$ is determined by a triple tuple $C = \{P, R, \Pi\}$, the first one with the provides-interfaces of the component $P = \{p_1, \ldots, p_n\}$, and the second one with the requires-interfaces of the component $R = \{r_1, \ldots, r_m\}$, and the third one with the parametric contracts between provides-interfaces and requires-interfaces, which comprise two functions $\Pi_p = P \rightarrow R$, and $\Pi_r = R \rightarrow P$, these two functions link provides-interfaces with requires-interfaces in both directions.

**Remark 1** The definition of a component with a dynamic type forms the foundation of soft trading. The dynamic type allows us to compute the new requires-interfaces of
the component based on the set and types of provides-interfaces are needed by the architecture in selection, and to compute the new provides-interfaces of the component based on the set and types of requires-interfaces that are satisfied by the solution.

**Definition 2 (Role with priority)** A role $K$ is determined by a group of provides-interfaces annotated with priority $C = \{ [p_1, \tau], \ldots, [p_n, \tau] \}$. The priority $\tau$ ranges from zero, means inessential, to 1, means required, in a left open and right close interval. A value between zero and 1 means auxiliary.

**Remark 2** Adding priority to interfaces in roles enable the users to get a usable application by degenerating the architecture when no solution is closed. The priority depends on the specific use case and the user, and is not part of the component itself.

**Definition 3 (Architecture)** An architecture $A$ is determined as a set of roles $A = \{ K_1, \ldots, K_n \}$, which represents a structural partition of the functions.

**Remark 3** The architecture is defined by abstract roles which can be mapped to different concrete components or component compositions under different environments. This separates the functional architectural definition from the physical implementation. The definition of an architecture can also be used to define parts of the application, which needs to adapt to environments during runtime to support the dynamic composition. In this paper, we are only concerned with the functional aspects of the application.

### 2.3 Soft Trading Algorithms

Given the concepts defined above, we can design a soft trading algorithm which comprises three sub-algorithms, (1) selection with provides-interfaces pruning, (2) matching with provides-parametric-contracts pruning, and (3) solution closing with requires-parametric-contracts pruning, with abandoning some non-critical functions.

We first give the overall skeleton of the soft trading algorithm, then deliberate the constitute sub-algorithms, using a java-like syntax:

**Algorithm 1 (Soft trading)**

```java
void trading(components, architecture) {
    closing(
        matching(
            selecting(components, architecture), architecture);
    )
}
```

The cascaded methods form a composition. In fact, the trading service is implemented in component-based manner, with each constitutional part as a pluggable component, which can be replaced, by using to the strategy design pattern [8].
Algorithm 2 (Selecting with provides-interfaces pruning)

Candidates selecting(components,architecture) {
    for all component in components{
        intersection = component[i].P ∩ architecture.P;
        if (intersection ≠ Ø){
            providesInterfacesPruning(intersection)
            candidates = candidates ∪ component[i];
        }
    }
    return candidates;
}

void providesInterfacesPruning(intersection){
    component.P = intersection;
}

Algorithm 3 (Matching with provides-parametric-contracts pruning)

Solutions matching(candidates,architecture){
    processComponents(0,workConf,0,sols);
    return sols
}

void processComponents(compoIdx,workConf,_intfIdx,sols){
    if (compoIdx < candidates.length){
        processComponent(compoIdx,workConf,_intfIdx,sols)
        processComponents(compoIdx+1,workConf,0,sols);
    }
}

void processComponent(compoIdx,workConf,_intfIdx,sols){
    for all _interface in candidates[compoIdx].P{
        if(_interfaces[i]∉ workConf){
            workConf = workConf ∪ _interfaces[i];
            if(workConf ⊇ architecture.P){
                // configuration covers the architecture
                providesParametricContractsPruning(workConf);
                sols = sols ∪ workConf;
            } else { // next interface
                processComponents(compoIdx,workConf,_intfIdx+1);
            }
            workConf = _interfaces[i];
        }
    }
}

void providesParametricContractsPruning(workConf){
    for all component that (component.P ∩ workConf ≠ Ø){
        component[i].R = \prod_{\text{P}(i)} (component[i].P ∩ workConf)
A question about the time to carry out provides-parametric-contracts pruning is whether to execute it during selection? The reason for delaying the pruning to the match process is based on the observation that even if all provides-interfaces of a candidate component are useful for the architecture, not all of them are useful for current solution. This means, provides-interfaces that do not contribute to current solution are useless, and may bring redundant requires-interfaces to the solution. So carrying out provides-parametric-contracts on the unit of solution is clearly preferable.

Algorithm 4 (Closing with requires-parametric-contracts pruning)

```java
ClosedSolutions closing(sols, architecture){
    forall sol in sols{
        if(sol[i].P ⊇ sol[i].R){ // no service gap
            closedSols = closedSols ∪ sol[i];
        } else {
            requiresParametricContractsPruning(sol[i]);
            if(isAcceptable(sol[i])){
                closedSols = closedSols ∪ sol[i];
            }
        }
    }
    return closedSols;
}

// Computes provides-interfaces of sol after pruning
// unsatisfied requires-interfaces
Solutions requiresParametricContractsPruning(sol){
    forall component that (comp.R ⊄ sol.P){// unsatisfied
        // compute provides-interfaces to be pruned
        prunedP = \prod_{i} (component[i].R - (component[i].R ∩ sol.P));
        if(!isPrunable(prunedP)) return;
        component[i].P = component[i].P - prunedP;
        proposedSols = proposedSols ∪ component[i].P;
        // pruning transitive closure of prunedP dependers
        forall component that (component.R ∩ prunedP ≠ \emptyset ){
            transitivePrunedP = \prod_{j} (component[j].R ∩ prunedP);
            if(!isPrunable(transitivePrunedP)) return;
            component[j].P = \prod_{j} (component[j].R - (component[j].R ∩ prunedP));
            proposedSols = proposedSols ∪ component[j].P;
        }
    }
    return proposedSols;
}
boolean isPrunable(intfaces){
    forall intface in (intfaces ∩ architecture.P){
        // code to check if intface is prunable
    }
    return true;  // if all intfaces are prunable
}
```
if (interface.T == 1) return false;
}
return true;

boolean isAcceptable(proposedSolution){
    return (user think it is acceptable) ? true : false;
}

Carrying out requires-parametric-contracts pruning on one component may trigger a chain reaction of pruning all provides-interfaces of other components which depend on the pruned provides-interfaces, thus the transitive closure must be pruned together to achieve closure.

3 An Example Application

In the following, we use data from [6] as the experiment data base to test the effectiveness of our parametric-contract-based soft trading service. This allows us to compare our approach against the results of [6]. We validate three aspects:

1. The effectiveness of component selection with provides-interfaces pruning. This is accomplished by adding additional but unnecessary provides-interfaces to components.
2. The effectiveness of component-architecture matching with provides-parametric-contracts pruning. This is accomplished by defining a concrete form of parametric-contracts at the signature level as interface A depends on interface B if and only if the implementation of A calls interface B. This is simple but is enough for current experiments. This leaves tests of parametric-contracts at behavioral semantic and protocol levels as future work.
3. The effectiveness of closing with requires-parametric-contracts pruning. For that we also use parametric-contracts as defined above.

Table 1 below shows the original data from [6]. Note, based on the observation that the requires-interfaces of roles in [6] are never used, we specify roles with provides-interfaces only and abstract away from actual component implementations as defined in definition 2 in section 2.2.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Candidate components</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIO={cio}</td>
<td>C1=[[cio],[]]</td>
</tr>
<tr>
<td>CAL={cal}</td>
<td>C2=[[cal],[]]</td>
</tr>
<tr>
<td>AG={ag, lis}</td>
<td>C3=[[ag, cio],[cal]]</td>
</tr>
<tr>
<td>MS={ms}</td>
<td>C4=[[lis],[]]</td>
</tr>
<tr>
<td></td>
<td>C5=[[ms,ag],[cio]]</td>
</tr>
<tr>
<td></td>
<td>C6=[[cal,lis],[p]]</td>
</tr>
</tbody>
</table>
The experiment 1 of soft trading uses the original data. As there is only one component which has only one unsatisfiable requires-interface, C6.p, we firstly define that C6.cal depends on C6.p and yield the parametric-contract 1: \{cal ←→ p\}, and secondly make C6.lis depending on C6.p and yield the parametric-contract 2: \{lis ←→ p\}. Thus, the definition of component C6 is \{\{cal,lis\},\{p\},\{cal ←→ p\}\} firstly, and \{\{cal, lis\},\{p\},\{lis ←→ p\}\} secondly as defined in definition 3 in section 2.2.

The results of experiment 1 comprise 4 items, (1) the number of configurations, (2) the number of solutions, (3) the number of closed solution, and (4) the number of architecture-compliant solutions.

The results of the first two items are the same as in [6], (a) because all provides-interfaces contained by components are usable for the given architecture, the effectiveness of provides-interface pruning does not show in this case, and therefore the number of configurations is the same, and (b) because the number of the solutions is determined by (i) the type and number of the interfaces specified in the architecture, and (ii) the type and number of the interfaces contained by components in repository. Because of using the data in [6], we get the same number of solutions as [6].

The number of the closed solutions is increased from 4 to 8 as shown in table 2 below (for brevity, the complete list of configurations is omitted.). This is because the provides-parametric-contracts prune away the interface C6.p if the depending interface (C6.cal in parametric-contract 1, and C6.lis in parametric-contract 2) is not used in the solution. Otherwise the interface would make the solution unclosed.

**Table 2. The results of experiment 1 with parametric-contract 1 (left) and 2 (right).**

<table>
<thead>
<tr>
<th>Exp.1 parametric-contract 1</th>
<th>Exp.1 parametric-contract 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solutions</td>
<td>Solutions</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>1 c2.cal c5.ms c1.cio c4.lis c3.ag C</td>
<td>1 c2.cal c5.ms c1.cio c4.lis c3.ag C</td>
</tr>
<tr>
<td>2 c2.cal c5.ms c1.cio c6.lis c3.ag C</td>
<td>2 c2.cal c5.ms c1.cio c6.lis c3.ag C</td>
</tr>
<tr>
<td>3 c2.cal c5.ms c1.cio c4.lis c5.ag C</td>
<td>3 c2.cal c5.ms c1.cio c4.lis c5.ag C</td>
</tr>
<tr>
<td>4 c2.cal c5.ms c1.cio c6.lis c5.ag C</td>
<td>4 c2.cal c5.ms c1.cio c6.lis c5.ag C</td>
</tr>
<tr>
<td>5 c6.cal c5.ms c1.cio c4.lis c3.ag</td>
<td>5 c6.cal c5.ms c1.cio c4.lis c3.ag C</td>
</tr>
<tr>
<td>6 c6.cal c5.ms c1.cio c6.lis c3.ag</td>
<td>6 c6.cal c5.ms c1.cio c6.lis c3.ag C</td>
</tr>
<tr>
<td>7 c6.cal c5.ms c1.cio c4.lis c5.ag C</td>
<td>7 c6.cal c5.ms c1.cio c4.lis c5.ag C</td>
</tr>
<tr>
<td>8 c6.cal c5.ms c1.cio c6.lis c3.ag</td>
<td>8 c6.cal c5.ms c1.cio c6.lis c5.ag C</td>
</tr>
<tr>
<td>9 c2.cal c5.ms c3.cio c4.lis c3.ag C</td>
<td>9 c2.cal c5.ms c3.cio c4.lis c3.ag C</td>
</tr>
<tr>
<td>10 c2.cal c5.ms c3.cio c6.lis c3.ag C</td>
<td>10 c2.cal c5.ms c3.cio c6.lis c3.ag C</td>
</tr>
<tr>
<td>11 c2.cal c5.ms c3.cio c4.lis c5.ag C</td>
<td>11 c2.cal c5.ms c3.cio c4.lis c5.ag C</td>
</tr>
<tr>
<td>12 c2.cal c5.ms c3.cio c6.lis c5.ag C</td>
<td>12 c2.cal c5.ms c3.cio c6.lis c5.ag C</td>
</tr>
<tr>
<td>13 c6.cal c5.ms c3.cio c4.lis c3.ag</td>
<td>13 c6.cal c5.ms c3.cio c4.lis c3.ag C</td>
</tr>
<tr>
<td>14 c6.cal c5.ms c3.cio c6.lis c3.ag</td>
<td>14 c6.cal c5.ms c3.cio c6.lis c3.ag C</td>
</tr>
<tr>
<td>15 c6.cal c5.ms c3.cio c4.lis c5.ag C</td>
<td>15 c6.cal c5.ms c3.cio c4.lis c5.ag C</td>
</tr>
<tr>
<td>16 c6.cal c5.ms c3.cio c6.lis c5.ag</td>
<td>16 c6.cal c5.ms c3.cio c6.lis c5.ag C</td>
</tr>
</tbody>
</table>

Table 2 shows the solutions obtained by requires-parametric-contract pruning. Note that requires-parametric-contract pruning requires user interactions and is non-deterministic, while provides-parametric-contract pruning is deterministic. For instance, we may apply requires-parametric-contract pruning to solution 5 of exp. 1 with the parametric-contract 1 if the user agrees. This results in the closed solution
If the parametric-contract of C3 is \{ag \rightarrow cal\}, or \{c5.ms, c1.cio, c4.lis, c3.ag\}, if the parametric-contract of C3 is \{cio \rightarrow cal\}. The requires-parametric-contract pruning can be applied to other unclosed solutions, too.

Finally, the architecture-compliancy problem can be distinguished between two cases. Firstly, if the architecture is not distributed, that is, all roles reside in one host. Then, in our example, there is no architecture-compliancy problem. For instance, solution \{c1.cio, c2.cal, c4.lis, c5.ms, c5.ag\} is fine to the architecture, although interfaces ms and ag are distributed in two separate roles. Secondly, in case of a distributed architecture, the solution above is not architecture-compliant. The resulting solutions of experiment 1 are all architecture-compliant, because there is no component implementing more than one role.

For comparison purpose, we run the trading service without parametric-contracts pruning using the same data set. The comparable statistic results of experiment 1 are listed in table 3 below. In table 3, column 2 describes the parametric-contract used, column 3-5 show the numbers of configurations, solutions, and closed solutions respectively. Column 6-7 shows the ratios of the number of solutions to the number of configurations and the number of closed solutions to the number of solutions respectively.

<table>
<thead>
<tr>
<th>Experiment 1</th>
<th>PCDm</th>
<th># confs</th>
<th># sols</th>
<th># closed sols</th>
<th>sols/confs</th>
<th>closed/sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact trading</td>
<td>161</td>
<td>16</td>
<td>4</td>
<td>10%</td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td>Soft trading</td>
<td>1</td>
<td>16</td>
<td>8</td>
<td>10%</td>
<td>50%</td>
<td></td>
</tr>
</tbody>
</table>

By adding additional provides-interfaces and requires-interfaces to components, including unsatisfiable requires-interfaces, we carried out further experiments by running the trading service with parametric-contracts pruning enabled firstly and disabled secondly using the same data set. The statistical data is shown in table 4. The figures of the above tables are visualized in the next section.

<table>
<thead>
<tr>
<th>Experiment 2</th>
<th>Dm</th>
<th># confs</th>
<th># sols</th>
<th># closed sols</th>
<th>sols/confs</th>
<th>closed/sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact trading</td>
<td>1199</td>
<td>288</td>
<td>96</td>
<td>24%</td>
<td>33%</td>
<td></td>
</tr>
<tr>
<td>Soft trading</td>
<td>1</td>
<td>1199</td>
<td>192</td>
<td>24%</td>
<td>67%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1199</td>
<td>144</td>
<td>24%</td>
<td>50%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 3</th>
<th>Dm</th>
<th># confs</th>
<th># sols</th>
<th># closed sols</th>
<th>sols/confs</th>
<th>closed/sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact trading</td>
<td>767</td>
<td>162</td>
<td>54</td>
<td>21%</td>
<td>33%</td>
<td></td>
</tr>
<tr>
<td>Soft trading</td>
<td>1</td>
<td>1999</td>
<td>432</td>
<td>29%</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1999</td>
<td>384</td>
<td>29%</td>
<td>67%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 4</th>
<th>Dm</th>
<th># confs</th>
<th># sols</th>
<th># closed sols</th>
<th>sols/confs</th>
<th>closed/sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact trading</td>
<td>639</td>
<td>108</td>
<td>4</td>
<td>17%</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>Soft trading</td>
<td>1</td>
<td>639</td>
<td>12</td>
<td>17%</td>
<td>12%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>639</td>
<td>12</td>
<td>17%</td>
<td>12%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>639</td>
<td>12</td>
<td>17%</td>
<td>12%</td>
<td></td>
</tr>
</tbody>
</table>
4 Results Analyses

Based on the experiment results presented in section 3, we draw trendline graphs to analyze the effectiveness of our parametric-contracts-based soft trading service. They are presented below, from figure 2 to 5. The horizontal axes in all figures are the number of configurations.

**Fig. 2.** Number of solutions.

**Fig. 3.** Number of closed solutions.

**Fig. 4.** The ratio of sols/conf.

**Fig. 5.** The ratio of closed sols/conf.

Figure 2 shows the comparison of the number of solutions between exact trading (top) and soft trading (bottom) with the vertical axis denoting the number of solutions. The improvement of the number of solutions is between 100% and 300% in the range between 700 and 1000 configurations. Figure 3 shows the comparison of the number of closed solutions between exact trading (below) and soft trading (above) with the vertical axis denoting the number of closed solutions. The improvement of the number of closed solutions is between 200% and 700% in the range between 700 and 1000 configurations. Figure 4 shows the comparison of the ratio of the number of solutions to the number of configurations between exact trading (below) and soft trading (above) with the vertical axis denoting the ratio. The improvement of the ratio of sols/conf is between 30% and 50% in the range between 700 and 1000 configurations. Figure 5 shows the comparison of the ratio of the number of closed solutions to the number of configurations between exact trading (below) and soft trading (above) with the vertical axis denoting the ratio. The improvement of the ratio of closed sols/conf is between 90% and 100% in the range between 700 and 1000 configurations.
Both the results of the experiments and the trendline graphs show that our parametric-contracts-based soft trading service makes a significant improvement over the traditional exact trading.

5 Related Work

The soft match making in trading service for COTS components was first proposed by [2]. While they resort to [9] for the implementation, we use a parametric-contracts-based approach. The work presented in this paper can be seen as an improvement of [6]. Parametric-contracts are usually used for automatic component adaptation and system properties predication [5], while we apply parametric contracts to a component trading process.

The soft trading service presented in this paper can be used as a complementation of task-oriented computing [7]. While task-oriented computing defines the skeleton of infrastructure for pervasive computing, soft trading service can be employed as the task assembly engine.

6 Conclusions and Future Work

In this paper we analyze the limitations of current trading services when dealing with mismatches between components and the target architecture. We analyze the required features of soft trading services in general, and describe the design of a soft trading approach, including its fundamental definitions and algorithms using parametric contracts for computing dynamically context-dependent interfaces. We validate the proposed approach by empirical data gained by experimentation with our implementation. The analysis of the results clearly shows the improvement of the proposed soft trading service to the traditional exact component trading. We argue that the systematic handling of imperfect matches is a requisite for any bottom-up CBSE approach.

A direct application of our work is the definition of "context-specific component substitutability" which relaxes the classical component substitutability by intentionally not taking into account substitutability problems which would be detected by conventional substitutability checks but which cannot occur in specific usage contexts. This naturally leads to a novel component type system which we will investigate as future work.

There are several possible extensions to our work. Parametric contracts have been applied to component interfaces containing finite-state machines for modeling component protocols. This is a good support to extend our work to semantic and protocol level trading. Extra-functional properties and user preference are not covered by our proposal either. These can be used to form heuristic functions for selecting, matching, and closing processes.
References

18. http://java.sun.com/j2se/1.5.0/docs/api/java/util/package-summary.html