Generic Component Lookup

Till G. Bay[1], Patrick Eugster[2], and Manuel Oriol[1]

[1] Chair of Software Engineering, Swiss Federal Institute of Technology in Zürich
CH-8092 Zürich, Switzerland
[2] Purdue University, Dept. of Computer Sciences, West Lafayette, IN 47907, USA

Abstract. The possibilities currently offered to conduct business at an electronic level are immense. Service providers offer access to their attendances through components placed on the Internet; such components can be combined to build applications, which can themselves be used as components by further business units. The final leg of the way to this paradigm has been paved by the advent of service-oriented architectures in general, and Web Services in particular.
With protocols existing for any parties to communicate, the most critical ingredient to the success of a business idea remains the task of choosing one's business partners. At a technical level, this translates to the issue of identifying *which* components represent the most adequate services to build a final application.
While each middleware technology and system proposed in the past has been described with its scheme for "looking up" components, this paper chooses the more difficult approach of trying to distill the fundamentals of component lookup. We propose a first-order model of component lookup — applicable to settings as diverse as tagged sets, classic white pages, or even method dispatch — and its implementation. We illustrate our model through various examples of existing lookup schemes. It turns out that in our generic context the common distiction between name-based and type-based lookup becomes rather artificial.

1 Introduction

The evolution from the Internet to the World Wide Web, more recently boosted by the advent of the semantic web and Web Services, has marked the gradual transformation of a communication infrastructure consisting of bare metal into a mighty platform fostering interaction of business parties.

The possibilities currently offered to conduct business at an electronic level are amazingly vast. Service providers offer access to their attendances through components placed on the Internet; such components can be combined to build applications, which can themselves be used as components by further business units. Web Service technologies typically provide the glue between individual components by proposing safe, efficient, and flexible communication protocols.

The most critical ingredient to the success of a business idea remains the task of setting up interactions, that is, choosing one's business partners. The success – or failure – of an entire business plan can depend on a single participant. At a technical level, the selection of appropriate business partners translates to the issue of identifying *which* components represent the most adequate services to build a final application. In particular, application designers have to face the challenge of specifying their own components,

and choosing foreign components according to potentially several specifications. Coding algorithms to perform such selections is ~~not only~~ an onerous task, but the outcome is usually an ad-hoc solution of limited application scope.

Each middleware technology and system proposed in the past has typically been described with its own scheme for "looking up" components, i.e., seeking, selecting, and connecting to components. In fact, the high number of non-redundant systems for looking up components advocates for a tighter integration or a composition model for such systems. In this paper, we try to distill the fundamentals of component lookup. We propose a first-order model of multiple specifications component lookup based on mathematical formulae, called $\mathcal{COLOS}$ (generic COmponent LOokup based on Specification matching). By differentiating between *explicit* and *implicit* component specifications, as well as between *internal* and *external* ones, our model becomes applicable to settings as diverse as tagged sets, name-based schemes, or even method dispatch. What we propose is thus a framework intended to provide programmers with an infrastructure to use and freely code specifications and their associated components. By offering the possibility of combining specifications by the means of mathematical operators, our system is able to sort the components that match with a *group of specifications* and put first the components that match *best*.

Our way of combining specifications and their matching is robust to distribution as results can be collected in a peer-to-peer manner. Our implementation is itself based on collections of components that can be combined, and includes the foundations for building secure matching as it can also be used for locking.

We illustrate our model through various examples of existing lookup schemes. Quite interestingly, it turns out that in the generic context we consider the traditional differentiation made in the past between value-based ("white pages") and type-based ("yellow pages") is artificial.

Roadmap. Section 2 presents preliminary material, including our model of components and their specifications, and related approaches. Section 3 elaborates on our generic model of component matching. Section 4 illustrates that model through various matching schemes. Section 5 discusses the implementation, deployment, and instantiation of our $\mathcal{COLOS}$ framework based on the model introduced. Section 6 draws final conclusions.

2 Preliminaries

Various systems and models have been described in the past for coordinating components in distributed settings. This section starts by presenting a simple abstract model of lookup, and then relating that model to a set of predating approaches.

2.1 Lookup Model

Components are described towards the outside world by respective *specifications* (see Figure 1). Lookup services basically provide components, on one hand, a means to construct and advertise such specifications, and on the other hand, a mechanism to

query components based on (specification) *templates*. The composition and nature of these specifications and templates, as well as the *matching* between them, vary between approaches.

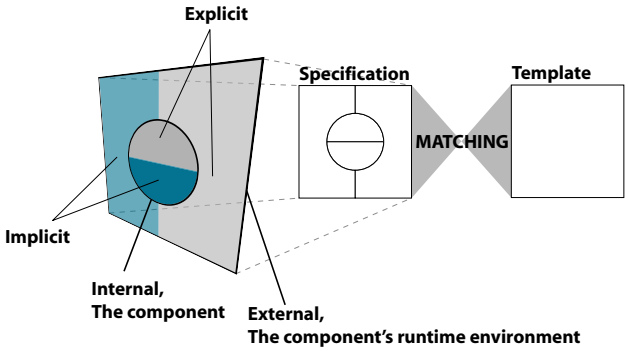

Fig. 1. Component and lookup model

Internal vs. external specification. When viewing ~~such~~ specifications as being based on different properties, one can in a first step distinguish between *internal* and *external* properties. Internal properties are based on the nature of components themselves, i.e., they reflect properties of a given component. External criteria reflect properties which pertain to the surroundings of the component, such as its context or (runtime) environment.

Implicit vs. explicit specification. In a second step, one can distinguish between *implicit* and *explicit* properties. The former kind of criteria reflect intrinsic properties *of the services provided* by a given component; they are not influenced by the nature and set of targeted clients for that component, or the means by which the component is made available to such customers. Explicit criteria in contrast, manifest in the way the component's very design is influenced by the perspective of making it ultimately available to the outside world.

Static vs. dynamic evaluation. Furthermore, the evaluation of the matching can be *static*, i.e., based on attributes of component specifications which are evaluated once and for all when the component is loaded, or *dynamic*, in which case the matching becomes a continuous process (see Section 5.1).

2.2 Examples

We illustrate the above model through a set of well-known lookup services, and overviewing derivatives for each. Results are summarized in Table 1 (due to the sparse occurrence of dynamic criteria in common lookup services the distinction static/dynamic is however omitted).

Domain Name System (DNS). DNS is very likely to be the most frequently used, static, name-based lookup system. Components are IP addresses, the specifications are (internal) host names, the templates are host names as well, and the matching tries to find the component that registers with a given host name (explicitly) and returns it's IP if possible.

Network Information Service (NIS). NIS is one of the oldest type-based, static lookup systems. Components are the entries of the maps (external), the specifications are map names (implicit), the templates are either map names or nicknames (e.g., $passwd$ for $passwd.byname$), and the matching is the result of the `ypcat` command.

CORBA. The Common Object Request Broker Architecture (CORBA) [1] defines both a Naming Service ~~(name-based)~~ and Trading Object Service for name-based and type-based lookup of objects respectively. The Naming Service represents the original means of looking up objects based on a *hierarchical* naming scheme, where an object is registered (explicit) and made available by attaching it (external) a unique name $N_1.\cdots.N_n$ of which each component N_i is a name/kind-pair. In this case, specifications and templates are both defined as sets of such pairs. Java RMI, or regular expressions, are similar in that sense, with $n = 1$.

The Trading Object Service offers rich combinations of means of defining the service type of a component. The most preferred way of attaching a type specification to a component consists in attaching it a name/value-pair. This definition of a component is external and explicit as well: the "type" describes actual properties of the component itself, but is not implicit like the actual classification of a component according to the type system of the considered language/environment.

Note that the OMG has more recently specified the Interoperable Name Service, defining URL-format object references that can be typed into a program to reach services at a remote location, including the Naming Service.

RM-ODP. The Reference Model for Open Distributed Processing (RM-ODP) [2] defines, similarly to CORBA, both a "white pages" (name-based) and "yellow pages" (type-based) lookup service (both explicit and external), going by the names of Relocator and Trader respectively. The latter service describes two roles which interacting components may take: *exporters* of services, and *importers*. A *service description* is an interface (type) and a set of properties attached to it, and a *service offer* binds a service description to a concrete component, which can be a CORBA object or another object. Properties are thus used to describe specifications and templates, the latter ones being more precisely combinations of properties; rules are expressed based on properties and operators (these are called matching criteria).

A novelty of the Trader specification is the description of delegation and collaboration among individual Trader units, which however does not seem to impact the model ultimately perceived by an application programmer, as, expressed in our terminology, specifications are simply cascaded.

UDDI. The *universal description, discovery and integration* (UDDI) [3] specification defines a lookup service for Web Services. Such a *registry* is centered around a *public*

cloud, a set of replica nodes storing white pages (abstract services by ”name”), yellow pages (by ”type”), and green pages (by ”description” and ”location”). Targeting at Web Services, UDDI encompasses a set of XML messages for SOAP-based interaction with registries. Each party is described through a *businessEntity*, several of which can be linked through *publisherAssertions*. A *businessService* is a particular Web Service offered by a business entity. Such a service is described by one or more *bindingTemplates*, which optionally contain textual service descriptions, and URLs for the respective services. Finally, *bindingTemplates* refer to one or more *tModel*s, which contain the pointers to actual descriptions of the services offered, and delineate the interaction protocols with the respective services. All the above-mentioned entities describe a refined pattern for specifications in the sense of our model introduced before-hand. The enforcing of authentication is covered in our model by external explicit criteria (see Section 4.5). The load distribution among nodes forming the public cloud is achieved in our implementation in an efficient manner by distributing the matching, greatly transparently, over a peer-to-peer overlay network (see Section 5.1).

Note that UDDI is a rare example of dynamic lookup, where components can be notified of changes in specifications of other components. Further examples are given by load balancing, or reuse frequency [4].

Service Groups. Sadou et al. [5] introduce a notion of *service group* to mediate between client and server components. These are motivated by the desire for type evolution, e.g., the possibility of adding parameters to methods. Just like in RM-OPD, the approach introduces both a notion of *type* which reflects provided services (i.e., the server side) in the terminology introduced by the authors, and a notion of *role* which represents the needs of customers (i.e., the client side).

At a first glance, one could hence be brought to viewing the types of [5] as specifications in our case, roles as templates, and service groups as defining the matching, respectively. However, the emphasis of [5] consists in making services of a given type available to clients expecting a slightly different type. Service groups are thus a form of glue aiming at expressing *how to pass from a given type to a given role*. They consist in stubs for respective server objects, which transform invocations based on a given role (the expected type) such as to fit the effective type. In our model, this represents explicit, internal component registration, and the specifications are made up of the stubs.

In a sense, HydroJ [6], LuckyJ [7], can be seen as simiiar approaches to service groups, as these are also based on some notion of type. Borrow/Lend [8], a derivative of the Type-based Publish/Subscribe (TPS) abstraction [9], as suggested by the name of the latter paradigm, in contrast, is primarily based on type-based matching of inherent Java object types (implicit, internal), which it however augments by (dynamic) predicate evaluation, and keys (explicit, external).

Coordination Spaces. The Borrow/Lend abstraction can in fact be seen as a variant of the Linda Tuple Space [10] with callback functionalities. The original Tuple Space is a means of setting up connections among distributed components, based on tuples of place holders (types) and values, i.e., a mixture of value-based and type-based matching, where values can also be character strings. This demonstrates how thin the border between types and values is.

Just like Borrow/Lend, Tagged Sets [11] are a variant of Tuple Spaces, where tuple items can also be predicates (leading to a dynamic evaluation), or keys (symmetric or asymmetric). Similarly, SecOS [12], supports the use of keys, with a partial matching. Clearly, any such criterion is explicit and external.

Criteria	Explicit	Implicit
External	UDDI, CORBA Naming, Trading, Java RMI, Linda, Regular Expressions, Tagged Sets, Borrow/Lend, SecOS	Reuse Frequency, Load Balancing, NIS
Internal	HydroJ, LuckyJ, Service Groups, DNS	Method Dispatch, Borrow/Lend

Table 1. Coarse classification of lookup services

2.3 A Note on Values and Types

A distinction that is often made when discussing component lookup is the one between *values* and *types*. This is nicely illustrated by the metaphors of "white pages" and "yellow pages" respectively.

However, component lookup in a distributed heterogenous environment is basically untyped. Matching components for their "type" boils down to matching such components for the *name of their type*, an internal property of these components. The possibility of registering several objects under a same given name, as supported by many systems, illustrates this seamless transition; by doing so, such a name becomes more a type description than a unique identifier. The issue of matching in such a setting becomes essentially a question of *depth*, in a way similar to the issue of object copying/cloning [13]. Any categorical distinction between values and types at this level seems unnatural. This is captured by our abstract notions of specifications and templates, which will become clearer through the matching model presented in Section 3, and illustrations thereof in Section 4.

3 Matching Model

The matching model presented in this section has resulted from the desire of capturing all the different lookup criteria outlined in the previous section.

In our model, the matching of components against requirements builds on the two basic notions introduced in the previous section, namely *specifications* and *templates*. The former roughly represent actual component descriptions (i.e., server-side views of components, see Figure 1), and the latter represent requirement descriptions (i.e., client-side views of components). In our matching model, specifications and templates are related by *matching modules*. Our goal is to be able to combine several specifications and templates into a compact notation and to design a lookup mechanism that sorts the retrived components in a list.

Our solution relies on mathematical formulae containing templates. Such a formula will be evaluated for each component C that has specifications s_0, s_1 and s_2, by replacing each template with a value (the matching value) that is calculated by applying a

matching function ($?_i$) between the specifications of the component and the templates. An example of such a formula follows:

$$t_0 + 3.0 - t_1 * t_2 \ applied \ to(\{s_0, s_1, s_2\}, C) \rightarrow (?_0(s_0, t_0) + 3.0 - ?_1(s_1, t_1) * ?_2(s_2, t_2))$$

For each component, this formula yields its matching value. When a client looks a component up, it is given a list of components sorted by their matching values in descending order. Components for which the matching value is 0 or below are omitted from the list. In the remainder of the section we define the theoretical framework to formalize this intuition using denotational semantics.

3.1 Matching Modules

A *matching module* is a triplet encompassing a set of specifications $\mathfrak{S}$, a set of templates $\mathfrak{T}$ and a matching relation $?$.

$$\begin{aligned} mm ::= & (\mathfrak{S}, \mathfrak{T}, ?) \\ & where \ ? : \mathfrak{S} \times \mathfrak{T} \rightarrow \mathbb{N} \end{aligned}$$

3.2 Specifications

A specification $\mathfrak{S}$ is itself a set of *specification terms* s_i. Informally a specification term is the specification for a component according to a given formalism. A template $\mathfrak{T}$ is itself a set of *templates terms* t_i. Informally a template term delineates a set of components according to a given formalism. The matching relation $?$ is a function that takes a specification term and a template term as arguments and returns a natural number.

In short, we define here what we need for providing ways of matching specifications and templates. Our goal being to integrate several of these modules into a multi-module specification, we do not enter into details but rather give examples of this in Section 4.

3.3 Qualified Specifications

A *qualified specification term* qs is a specification term annotated with a *qualifier*.

$$\begin{aligned} s & \in \mathfrak{S}_i \\ val & ::= n \in \mathbb{N} \\ comp & ::= \ < \mid > \mid \neq \mid \leq \mid \geq \mid = \\ qualifier & ::= \textbf{required} \ comp \ val \mid \varnothing \\ qs & ::= \ s \ qualifier \end{aligned}$$

Qualifiers on specification terms are used as a way for the component provider to order differences in the treatment of the matching. We specify two different types of qualifiers: $\varnothing$ that means that we do not modify the basic mechanism (that we always ommit in practice as a notation abuse) and **required** that allows us to filter and impose a condition on the matching for specific specification terms. This latter qualifier allows us, in particular to envision security constrained matching as shown in Subsection 4.5. Even

if, for now, we only consider the qualifiers **required** and $\varnothing$ we could imagine other qualifiers that modify the infrastructure's behavior accordingly.

A *component specification* CS consists of a set of qualified specification terms that appear at most once in the set of specifications of a given matching module.

$$\begin{aligned} CS \ ::= \ & \{qs_1, \ ..., \ qs_n\} \\ & such\ that\ \forall\ i,\ j\ \in\ [1,n]\ \ s_i \in s_0\ s_j \in s_0\ \Rightarrow\ i = j \end{aligned}$$

A component specification is the way a component provider can describe its components.

3.4 Templates

A *template* $T \in \mathfrak{T}$ consists of a mathematical formula using mathematical operators and *template terms*.

$$\begin{aligned} t\ &\in \mathfrak{T}_i \\ op\ &::=\ +\ |\ -\ |\ *\ |\ / \\ T\ &::=\ n \in \mathbb{N}\ |\ t\ |\ T\ op\ T \end{aligned}$$

The idea is, that unlike qualifed specification terms that are composed in a list to make the component specification, we compose template terms to a mathematical formula in order to allow component seekers to allocate more weight to some specification. It also allows to exclude components that answer to a specification by using subtractions and divisions to lower their matching values and possibly rule them out of the returned list.

3.5 Matching

The *valued matching* of a component specification CS with a template T consists in matching on the specification and calculating its value according to the template definition. It is defined as follows:

$$valuedMatch \quad ::= CS?_v T$$

$$\begin{aligned} &\mathcal{V}[\![\cdot]\!] : valuedMatch \rightarrow \mathbb{Q} \cup \{\infty\} \\ &\mathcal{V}[\![CS?_v n]\!] \quad = \ n \\ &\mathcal{V}[\![CS?_v t]\!] \quad = \ 0\ if\ \nexists\ qs = s_0\ q_0 \in CS \\ &\qquad\qquad\quad such\ as\ \exists\ mm_0 = (\mathfrak{S}_0, \mathfrak{T}_0, ?_0)\ |\ t \in \mathfrak{T}_0,\ s_0 \in \mathfrak{S}_0 \\ &\qquad\qquad\quad ?(s,\ t)\ otherwise \\ &\mathcal{V}[\![CS?_v T_1\ op\ T_2]\!] \ = \ \mathcal{V}[\![CS?_v T_1]\!]\ op\ \mathcal{V}[\![CS?_v T_2]\!] \end{aligned}$$

The intuition behind the matching we describe here is that each template term within the mathematical formula of the template is replaced by the result of the application of the matching relation between the template term and the specification term of the component specification.

The *matching compliance* of a component specification CS with a template T consists in knowing if the required specification terms are matched. It is defined as follows:

$$
\begin{array}{lll}
compliesToMatch & ::= CS?_cT & \\
 & & \\
\mathcal{C}[\![\cdot]\!] : compliesToMatch \rightarrow \mathbb{B} & & \\
\mathcal{C}[\![\{s \textbf{ required } comp_0\ n_0\}?_cT]\!] & = & \text{TRUE } if\ \exists\ t\ in\ T\ s.a.\ \mathcal{V}[\![s?_vt]\!]\ comp_0\ n_0 \\
 & & \text{FALSE } otherwise \\
\mathcal{C}[\![\{s\varnothing\}?_cT]\!] & = & \text{TRUE} \\
\mathcal{C}[\![\{qs_1,\ ...,\ qs_n\}?_cT]\!] & = & \mathcal{C}[\![\{qs_1\}?_cT]\!] \wedge ... \wedge \mathcal{C}[\![\{qs_n\}?_cT]\!]
\end{array}
$$

As a simple explanation, a template complies with a specification if all the required conditions on the specifications are fulfilled by any of the basic templates.

3.6 Component Selection

Finally, we can define the *selection* mechanism built on top of the valued matching and the matching compliance. A component C declares its interface in its component specification CS. The component repository $\mathfrak{C}$ consists in a set of components stored with their specifications. These can be selected using the selection operator $\downarrow$ that returns a list of components for which we show the semantics $\mathcal{E}$.

$$
\begin{array}{lll}
\mathfrak{C} & ::= \{(CS_1, C_1), ..., (CS_n, C_n)\} & \\
lookup & ::= \mathfrak{C} \downarrow T & \\
 & & \\
\mathcal{E}[\![\cdot]\!] : lookup \rightarrow list\ of\ (CS_i, C_i) & & \\
\mathcal{E}[\![\mathfrak{C} \downarrow T]\!] & = & \{(CS'_1, C'_1), ..., (CS'_m, C'_m)\} \subseteq \mathfrak{C} \\
 & & such\ that \\
 & & \forall i \in [1, m], \mathcal{C}[\![CS'_i?_cT]\!]\ and\ \mathcal{V}[\![CS'_i?_vT]\!] > 0 \\
 & & and\ \forall i, j \in [1, m],\ i < j \Leftrightarrow \mathcal{V}[\![CS'_i?_vT]\!] \geq \mathcal{V}[\![CS'_j?_vT]\!]
\end{array}
$$

Intuitively, the final result of a component selection on a respository is a list containing elements from the repository ordered by decreasing matching values. That way, we can obtain the component that is best adapted regarding to the templates we defined. In the next section we show examples of such matching modules and how they can be used.

4 Illustration

This section illustrates our first-order model of component lookup through a succinct set of existing lookup schemes. More examples can be found in a separate technical report [14]. ~~Such examples are based on subtyping (both nominal and structural), or on an extended notion of reuse frequency defined by Bay *et al.* [4].~~

4.1 Unique Identifiers

As a first simple example, we consider the selection mechanism based on a unique component identifier. In that case the matching module can be described by the following

triplet:

$$\begin{aligned} mm_{\mathfrak{UID}} ::= & (\mathbb{N}, \mathbb{N}, ?_{\mathfrak{UID}}) \\ & where\ ?_{\mathfrak{UID}} : \quad \mathbb{N} \times \mathbb{N} \mapsto \{\ 0,\ 1\ \} \\ & ?_{\mathfrak{UID}}(x, y) \ = \quad 1\ if x = y \\ & \qquad\qquad\quad\ 0\ otherwise \end{aligned}$$

As a first example of use, we can imagine a collection of software components that have unique identifiers:

$$\mathfrak{C} = \{(\{1_{\mathfrak{UID}}\}, C_1), ..., (\{1337_{\mathfrak{UID}}\}, C_{1337}), ..., (\{n_{\mathfrak{UID}}\}, C_n)\}$$

Looking up for the component 1337 can be made as follows:

$$\mathfrak{C} \downarrow 1337_{\mathfrak{UID}} = \{(\{1337_{\mathfrak{UID}}\}, C_{1337})\}$$

Note that a variation of this module can be used to describe the DNS.

4.2 Regular Expressions

Among the most widespread and popular descriptions of components are component APIs, and component documentation. It seems conceivable that one might be interested in selecting components based on criteria expressed on their textual description, in addition to other specifications. An example is selecting components according to their author(s), as appearing in the documentation. This constitutes the case of matching regular expressions (note that we use the original regular expressions as defined in Kleene algebra):

$$\begin{aligned} char \quad & ::= a \mid ... \\ string \quad & ::= char \mid string\ string \\ expr \quad & ::= \varnothing \mid char \mid (expr\ expr) \mid (expr + expr) \mid expr* \\ mm_{\mathfrak{regexp}} & ::= (string, expr, ?_{\mathfrak{regexp}}) \\ & \quad where\ ?_{\mathfrak{regexp}} : string \times expr \mapsto \mathbb{N} \\ & \quad ?_{\mathfrak{regexp}}(s, e) = number\ of\ occurrences\ of\ s\ in\ e \end{aligned}$$

Now imagine that a user wants to obtain a component for which John Doe is indicated as the main author of that component in the accompanying documentation and preferably take the component with the unique identifier 1337. A collection including such a component could then be:

$$\begin{aligned} \mathfrak{C} = \{\ & (\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_1), ... \\ & (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337}), ... \\ & (\{n_{\mathfrak{UID}}\}, C_n)\} \end{aligned}$$

Looking up for a component fulfilling at least one of these characteristics would then produce:

$$\begin{aligned} & \mathfrak{C} \downarrow (1337_{\mathfrak{UID}} + \text{“} * author\ :\ John\ Doe * \text{”}_{\mathfrak{regexp}}) = \\ & \{\ (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337}), \\ & \quad (\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_1)\} \end{aligned}$$

Looking up for a component fullfilling both criteria can be made as follows:

$$\begin{aligned} & \mathfrak{C} \downarrow (1337_{\mathfrak{UID}} * \text{“} * author : John\ Doe * \text{”}_{\mathfrak{regexp}}) = \\ & \{\ (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337})\} \end{aligned}$$

4.3 Load Balancing

Another criterion of component linking, is its current load.

$$
\begin{aligned}
mm_{\mathtt{load}} ::= & (\mathbb{N},\ \varnothing,\ ?_{\mathtt{load}}) \\
& where\ ?_{\mathtt{load}} : \mathbb{N} \times \varnothing \mapsto \mathbb{N}^{+} \\
& ?_{\mathtt{load}}(n) = number\ of\ components\ currently\ using\ component\ n
\end{aligned}
$$

Imagine that a user wants to obtain a component that is currently the less used and written by John Doe. Suppose also that some components requires less than 10 clients at the same time. A collection containing such components could then be specified as follows:

$$
\begin{aligned}
& \mathfrak{C} = \{(\{1_{\mathfrak{UJD}}, \text{“}...author : John\ Doe...\text{”}_{\mathtt{regexp}}, C_1), ... \\
& \ (\{1337_{\mathfrak{UJD}}, \text{“}...author : John\ Doe...\text{”}_{\mathtt{regexp}}, \textbf{required}\ 1337_{\mathtt{load}} < 10.0\}, C_{1337}), ... \\
& \ (\{n_{\mathfrak{UJD}}, n_{\mathtt{load}}\}, C_n)\}
\end{aligned}
$$

A programmer wishing to get such a component should perform the following lookup (note that the result is dependant of the number of clients currently connected to both components):

$$
\begin{aligned}
& \mathfrak{C} \downarrow (\text{“} * author : John\ Doe * \text{”}_{\mathtt{regexp}} / (1.0 +_{\mathtt{load}})) = \\
& \{\ (\{1_{\mathfrak{UJD}}, \text{“}...author : John\ Doe...\text{”}_{\mathtt{regexp}}, C_1), \\
& \quad (\{1337_{\mathfrak{UJD}}, \text{“}...author : John\ Doe...\text{”}_{\mathtt{regexp}}\}, C_{1337})\}
\end{aligned}
$$

4.4 Compliance to an Interface

It very often happens that programmers want to obtain components that comply to a given interface. Informally, complying to an interface is usually expressed in terms of a structural subtyping relationship. Suppose that I_1 is compliant to I_2 if and only if I_1 has at least the same procedures as I_2 ~~but may declare more.~~

$$
\begin{aligned}
& p && procedure\ names \\
& t && tyes\ names \\
& procedure && ::= (p, \{t_0, ..., t_n\}) \\
& I && ::= \{procedure_1, ..., procedure_n\} \\
& mm_{\mathtt{comply}} && ::= (Interfaces,\ Interfaces,\ ?_{\mathtt{comply}}) \\
& && \quad where\ ?_{\mathtt{comply}} : Interfaces \times Interfaces \mapsto \{0,\ 1\} \\
& && \quad ?_{\mathtt{comply}}(I_1, I_2) = 1\ iff\ I_2 \subseteq I_1,\ 0\ otherwise
\end{aligned}
$$

Let suppose that some components ~~that~~ offer procedures to set and get internal attributes, the collection of components could be:

$$
\begin{aligned}
& \mathfrak{C} = \{(\{1_{\mathfrak{UJD}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}, decrement\ \{\}\}_{\mathtt{comply}}, C_1), ... \\
& \ (\{1337_{\mathfrak{UJD}}, \text{“}...author : John\ Doe...\text{”}_{\mathtt{regexp}}\}, C_{1337}), ... \\
& \ (\{n_{\mathfrak{UJD}}, n_{\mathtt{load}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}\}\}_{\mathtt{comply}}, C_n)\}
\end{aligned}
$$

Then a program seeking for components that complying to an interface containing set_a and get_a could make the following lookup:

$$\begin{array}{l}\mathfrak{C} \downarrow \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}\}_{\mathfrak{comply}} = \\ \{\ (\{1_{\mathfrak{UID}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}, decrement\ \{\}\}_{\mathfrak{comply}}, C_1), \\ \quad (\{n_{\mathfrak{UID}}, n_{\mathfrak{load}}, \{set_a\ \{Void,\ string\}, get_a\ \{string\}\}\}_{\mathfrak{comply}}, C_n)\}\end{array}$$

Variants of this examples are countless as we could return the number of procedures in common, or the number of lacking procedures etc. However this is the simplest variant and it corresponds to the approach of service groups [5].

4.5 Secure Linking

By specifying a **required** clause, a component *provider* can enforce the matching of a specification as a necessary precondition for handing out any reference to the component. Our current example is presenting encrypted matching and can be considered as a subset of tagged sets [11] or any other matching mechanisms driven or restricted by encryption [12, 8].

We call $E(K, value)$ the encryption and $D(K, value)$ the decryption, for which we give the semantics $\mathcal{S}[\![\cdot]\!]$ that we detail below.

$$\begin{array}{ll}\overline{SKey} & SymetricKeys \\ \overline{AKey} & Asymetric\ Keys\ (private) \\ AKey & Asymetric\ Keys\ (public) \\ value & ::= basic_value \mid value_{\overline{AKey}} \mid value_{SKey} \\ e & ::= value \mid E(SKey, e) \mid E(AKey, e) \mid D(SKey, e) \mid D(AKey, e)\end{array}$$

$$\begin{array}{l}\mathcal{S}[\![\cdot]\!] :\ e \mapsto value \\ \mathcal{S}[\![value]\!] = value \\ \mathcal{S}[\![value]\!] = value \\ \mathcal{S}[\![E(SKey, e)]\!] = \mathcal{S}[\![e]\!]_{SKey} \\ \mathcal{S}[\![E(\overline{AKey}, e)]\!] = \mathcal{S}[\![e]\!]_{\overline{AKey}} \\ \mathcal{S}[\![D(SKey, e_{SKey})]\!] = \mathcal{S}[\![e]\!] \\ \mathcal{S}[\![D(AKey, e_{\overline{AKey}})]\!] = e\end{array}$$

The associated matching module is then:

$$\begin{array}{l}mm_{\mathfrak{Crypt}} ::= (Keys,\ Keys,\ ?_{\mathfrak{Crypt}}) \\ \qquad where\ ?_{\mathfrak{Crypt}} : Keys \times Keys \mapsto \{0, 1\} \\ \qquad\qquad ?_{\mathfrak{Crypt}}(K_1, K_2) = 1\ if\ \mathcal{S}[\![D(K_2, E(K_1, value))]\!] = value \\ \qquad\qquad\qquad\qquad 0\ otherwise\end{array}$$

A collection containing components being locked by an asymetric key $AKey$ could then be :

$$\begin{array}{l}\mathfrak{C} = \{(\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, C_1), ... \\ \ (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, \textbf{required} AKey_{\mathfrak{Crypt}} = 1.0\}, C_{1337}), ... \\ \ (\{n_{\mathfrak{UID}}, n_{\mathfrak{load}}, AKey_{\mathfrak{Crypt}}\}, C_n)\}\end{array}$$

A programmer wishing to know all the components locked with $AKey$ should then make the following lookup:

$$\mathfrak{C} \downarrow \overline{AKey}_{\mathfrak{Crypt}} = \\ \{(\{1337_{\mathfrak{UJd}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, \textbf{required}AKey_{\mathfrak{Crypt}} = 1.0\}, C_{1337}) \\ (\{n_{\mathfrak{UJd}}, n_{\mathfrak{load}}, AKey_{\mathfrak{Crypt}}\}, C_n)\}$$

Implementation-wise, locking a component with a cryptographic key means that the access to the component should be made on the platform where the component is located. Similarly to tagged sets [11], the keys do not need to transit through the network.

5 Putting $\mathcal{COLOS}$ to Work

This section first presents our Eiffel implementation of the model described in Section 3. Thereafter, we present an application scenario for $\mathcal{COLOS}$.

5.1 Implementation

The implementation of the $\mathcal{COLOS}$ framework consists mainly in the specifications, templates and the surrounding component infrastructure. Currently the framework consits of 21 classes with 1700 lines of code. We are extending it to more component models and plan on publishing it opensource.

Specifications. *LL_SPECIFICATION* is a list of *LL_SPECIFICATION_TERM*s. The deferred (abstract) class *LL_SPECIFICATION_TERM* should be subclassed by a programmer who wants to define his own matching module. The only mandatory feature to be implemented returns a *STRING* representing the name of the corresponding matching module. The infrastructure already implements the features to look through the specifications given that the *LL_SPECIFICATION_TERM*s return the correct matching module name. This enables an implementation based on hashtables. Just like for templates, which are described in following subsection, we use the possibility to define our own infix operators for setting constraints on the specifications that describe a component. The Eiffel programming language makes it easy to define these operators and together with automatic conversion functions they allow writing easily readable code.

Templates. To implement our prototype, we relied on two features of the Eiffel language, namely (1) user-defined infix operators and (2) user-defined automatic type conversion. Infix operators allow us to compose templates using the infix operators as defined by the natural mathematical intuition while automatic conversion lets us have valid types for general mathematical operations. Due to the latest definition of Eiffel and the priority of the operators, the canonical priorities apply. The infix operators are coded into *LL_TEMPLATE* and are thus inherited by all templates. The automatic conversion from **DOUBLE** to *LL_TEMPLATE* ensures that we can compose doubles and templates in a same expression containing infix operators. In short the Eiffel compiler (ISE Eiffel 5.7) converts mathematical formulae containing templates by transforming the doubles that they contain into *TEMPLATES*. As an example, the formula

template := 2.0∗*template0*−1.0/(*template1*−*template2*)

is automatically transformed by the compiler into:

template :=
((**create** {*LL_TEMPLATE*}.*make_from_double*(2.0))∗*template0*) −
((**create** {*LL_TEMPLATE*}.*make_from_double*(1.0))/(*template1*−*template2*))

The deferred class *LL_TEMPLATE_TERM*, inherits from the class *LL_TEMPLATE*. A programmer wishing to implement a matching module should subclass it and implement the feature *match* that takes an *LL_SPECIFICATION_TERM* as an argument and he should also provide a feature returning the name of the matching module as mentioned previoulsy. Note that in our infrastructure the only *LL_SPECIFICATION_TERM*s that can be passed as parameter to the *match* feature are the ones actually belonging to the same matching module.

Decentralized lookup. The current matching prototype infrastructure performs centralized component lookup. We are currently in the process of augmenting our implementation for efficient component lookup in peer-to-peer (P2P) settings, which will make our infrastructure available as a service within a peer group of JXTA networks [15].

In order to complete such a decentralized lookup efficiently, it is very useful to be able to "decompose" the matching. The idea can be viewed as a generalization of the problem of content-based event routing in P2P networks, where event contents are viewed as consisting in several properties which are each matched against values, and an overlay network can be built which regroups participants with common interests and whose nodes many perform matching of only subsets of the properties (e.g. [16]).

In order to be able to decompose the matching in the lookup problem, a little help is however required from the programmer. Both specifications and templates have to provide access to a tree-based representation of themselves, akin to abstract syntax trees. The individual tree nodes represent elementary matching operations, and can be performed in a decentralized, yet minimally redundant, manner.

The logical regrouping of several *tModel*s to a *bindingTemplate*, several *bindingTemplate*s to a *businessEntity*, and several instances of latter kind to a *businessService* in UDDI (see Section 2.2), is but an illustration of such a decomposition.

5.2 Instantiating $\mathcal{COLOS}$

In the current state of the implementation of $\mathcal{COLOS}$, one willing to use the component lookup mechanism can simply instantiate the class *LL_COMPONENT_COLLECTION* and the components along with their specifications. By subclassing the two deferred (abstract) classes *LL_SPECIFICATION_TERM* and *LL_TEMPLATE_TERM*, the programmer can implement a matching module. It implies setting two variables and redefining the feature *match*. Note that keeping a reference to the object encapsulating a component with its specification allows revoking parts of the specification dynamically.

In the following example (see Figures 2 and 3) we show how one desribes a component and then uses our lookup mechanism to match requirements against the entire component repository. We see how the specification terms are first declared an enriched

```
uid_specification_term : LL_UID_SPECIFICATION_TERM
regexp_specification_term : LL_REGEXP_SPECIFICATION
load_specification_term : LL_LOAD_SPECIFICATION
...
create uid_specification_term .make ("1337")
create regexp_specification_term .make ("This component...
                    author: John Doe")
create load_specification_term .make (Current.component)

Current. add_specification_term_to_spec ( uid_specification_term )
Current. add_specification_term_to_spec ( regexp_specification_term )
Current. add_specification_term_to_spec ( load_specification_term <10.0)
...
```

Fig. 2. Specification declaration

```
uid_template : LL_UID_TEMPLATE
regexp_tempalte : LL_REGEXP_TEMPLATE
load_template : LL_LOAD_TEMPLATE
component: LL_COMPONENT
components: LL_COMPONENT_COLLECTION
...
create uid_template .makr ("1337")
create regexp_tempalte .make ("*author: John Doe*")
create load_template .make
component:= (components^(( uid_template + regexp_tempalte ) /(1.0+ load_template ))) .
      get_first_component
...
```

Fig. 3. Using the lookup infrastructure

with to corresponding information. Then they are added to the component's specification. Note how the *less* operator is used to impose a constraint on the specification about the component's load. In the second listing (see Figure 3) of the example it is shown how to prepare a component lookup. Instead of specifications we are now preparing templates that are put together to match against the component repository. The ^-operator is used to initiate the matching. In the resulting list the components are ordered according to rating of the matching in respect to the template formula. In this case we are only interested in the component with the highest rating and we are therefore only getting the first component of the resulting list.

6 Conclusions

Lookup mechanisms are a necessary burden in distributed component interaction. Various systems and specifications have been proposed in the literature, each targeting at a specific setting.

We have presented in this paper $\mathcal{COLOS}$, a generic model of component lookup, which can be used to express most predating lookup schemes. $\mathcal{COLOS}$ is based on an abstract model for matching component specifications against templates, based on mathematical formulae. We have described this matching through denotational semantics, illustrated it through various examples, and presented its implementation in Eiffel. The implementation reflects exactly the theory and uses automatic transformations as well as infix operators to obtain extremely compact and intuitive code.

We envision the definition of more "common" matching modules, and intend to implement our framework on top of a fully decentralized peer-to-peer overlay network. Furthermore, we plan to port it to a wider range of programming languages and platforms in order to obtain interoperability among different types of components.

References

1. Group, O.M.: The Common Object Request Broker Architecture: Core Specification, Version 3.0.3. OMG (2004)
2. Blair, G., Stefani, J.B.: Open Distributed Processing and Multimedia. Addison-Wesley (1997)
3. ShaikhAli, A., Rana, O., Al-Ali, R., Walker, D.: Uddie: An extended registry for web services. In: SAINT-W '03: Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops). (2003) 85
4. Pauls, K., Bay, T.: Reuse Frequency as Metric for Dependency Resolver Selection. In: Component Deployment: Third International Working Conference, CD 2005. Volume 3798. (2005) 164–176
5. Sadou, S., Koscielny, G., Mili, H.: Abstracting Services in a Heterogeneous Environment. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). (2001) 141–159
6. Lee, K., LaMarca, A., Chambers, C.: Hydroj: object-oriented pattern matching for evolvable distributed systems. In: OOPSLA '03: Proceedings of the 18th annual ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications. (2003) 205–223
7. Oriol, M., Di Marzo Serugendo, G.: A disconnected service architecture for unanticipated run-time evolution of code. IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution **151**(2) (2004) 95–107
8. Eugster, P., Baehni, S.: Abstracting Remote Object Interaction in a Peer-to-Peer Environment. Concurrency & Computation: Practice and Experience **17**(7-8) (2005)
9. Eugster, P., Guerraoui, R.: Distributed Programming with Typed Events. IEEE Software **2**(21) (2004) 56–64
10. Carriero, N., Gelernter, D.: Applications experience with Linda. ACM Sympos. on Parallel Programming (1985)
11. Oriol, M., Hicks, M.: Tagged Sets: A Secure and Transparent Coordination Medium. In: 7th Int. Conf. on Coordination Models and Languages. (2005)
12. Bryce, C., Oriol, M., Vitek, J.: A Coordination Model for Agents Based on Secure Spaces. In: 3rd Int. Conf. on Coordination Models and Languages. (1999) 4–20
13. Gregono, P., Sakkinen, M.: Copying and Comparing: Problems and Solutions. In: 14th European Conference on Object-Oriented Programming (ECOOP 2000). (2000) 226–250
14. Bay, T., Eugster, P., Oriol, M.: A First Order Model of Component Lookup. Technical report, Swiss Federal Institute of Technology in Zurich (ETHZ) (2006)
15. Oaks, S., Gong, L.: Jxta in a Nutshell. O'Reilly & Associates, Inc. (2002)

16. Eugster, P., Guerraoui, R.: Probabilistic Multicast. In: 3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002). (2002) 313–323