

Querying reflexive component-based architectures

Mourad Alia, Romain Lenglet, Thierry Coupaye, Alexandre Lefebvre
France Telecom R&D, DTL/ASR
28 chemin du Vieux Chêne, 38243 Meylan, France
{alia.mourad, romain.lenglet, thierry.coupaye,
alexandre.lefebvre}@rd.francetelecom.com

Abstract

In the context of large-scale distributed component-based systems, this article motivates the need for and defines a general open query service. This query service allows for retrieving and selecting components from both repositories and deployed running systems. The problem of retrieving components from repositories has already been tackled by many research works whose purpose is to construct systems by assembling reusable components, typically in the context of COTS components. However, the proposed query services allows for an uniform efficient associative access to both repositories and running systems, while being based on distributed database techniques, which is the main contribution in this paper.

1. Introduction and context

Data and services managed by large companies tend to become more and more heterogeneous and distributed. Such networks and information systems as a whole include network equipments such as IP routers and ATM switches, server computers such as authentication servers, web servers and application servers (e.g. for billing), and database systems. Such systems are highly *distributed*, i.e., their elements all interact through network interconnections, and *dynamic*, or *open*, i.e., new elements are added or removed dynamically, are physically or logically moved, and failures may occur. These elements are also *heterogeneous*, since their functions and capacities differ.

Distribution, dynamicity and heterogeneity make a system difficult to develop and deploy, i.e. to install, activate, control and upgrade, by the several kinds of actors that interact with it: commercial representatives, system administrators, network equipment integrators, software developers and maintainers. To perform these tasks, *views* on systems [17] must be offered, i.e., systems can be considered as sources of information. Typically, administrators use views

of remotely running systems that describe their topology and the resources they use, and distributed teams of developers, possibly speaking different languages, share views of system specifications to reuse parts of them in order to instantiate systems. The problem that we address in this paper is the construction of such views.

Generally, many works on component-based software architecture address those problems of distribution, dynamicity and heterogeneity. More specifically, *reflexive* component models offer in addition a means to consider components as sources of meta-data, and as such as small-scale abstract views of a system. In this paper we consider the Fractal reflexive component model [7, 6] because it is well-suited both to design software systems and to reify hardware resources in a uniform manner. It is therefore conceivable to design a whole large company network and information system using exclusively Fractal components.

The rest of this paper is organised as follows. Next section describes the Fractal component model and the meta-data that Fractal components provide. Scenarios are described to motivate a broad scope for the use of component query, in particular the need for querying systems at runtime. Existing works on component retrieval approaches are then presented, focusing mainly on querying component repositories. The last part of this paper describes our proposal of a component retrieval system, currently under development, which prominent feature is to enable queries on both component repositories and running systems. This work is based on several of our existing research works in the area of component-based distributed system architecture on one hand, and in the area of databases and persistence on the other hand.

2. The Fractal model

The Fractal component model The Fractal component model exhibits several interesting properties for the construction of distributed, dynamic and heterogeneous systems:

- *dynamism*: components are runtime entities: they are manifest and can be created, destroyed and (re)configured during execution.
- *encapsulation*: components can interact only through well-defined access points called *interfaces* and explicit *bindings* between interfaces. Bindings are arbitrary communication paths.
- *identity*: components have a well-defined identity, that unambiguously distinguishes one component from another, and allows to interact with them.
- *nested composition*: components can contain components, recursively. Recursion ends up with *primitive* components which have an empty content and may reify hardware resources or directly encapsulate plain software objects (e.g. Java objects). Components that do have a content, i.e., that contain sub-components, are called *composite* components.
- *control*: components transparently provide *introspective* and *intercessive* capabilities (i.e., access and manipulation of metadata respectively) to exercise arbitrary reflective control over their execution. Standard controls in Fractal include adding or removing components from composite components, starting or stopping components, and (un)binding components, at runtime.

Meta-data information As a meta-data source, a typical Fractal component offers at runtime the following information: identity, life-cycle state (started, stopped, etc.), identity of subcomponents (if composite component), identity of components bound to it, functional and control (meta-level) interface names and types.

Fractal components are specified and instantiated programmatically, using the Fractal APIs. In addition, an architecture description language, the Fractal ADL, can be used to statically define *component templates*, which are specifications sufficient to instantiate components. The Fractal ADL is an XML-based language that allows for the specification of the interfaces of components (names and signatures), subcomponents (identified by local names) and the bindings between them in a component template, initial values of component properties, and the implementation of primitive components (e.g., the name of a Java class). Unstructured textual information can be attached to any element. Template definitions have unique names and can be specialized through inheritance in order to factorize and reuse them. More generally, component templates described in the Fractal ADL can be considered as sources of meta-data more general and more precise than that available from running Fractal components. For instance, template inheritance is an ADL-specific concept. However, the use of the Fractal ADL is optional and is only static, i.e., it

describes only the initial state of a configuration of components.

A repository of Fractal component templates would contain both Fractal ADL information and the code for the implementation of primitive components. In addition, it may contain additional unstructured textual information, such as Javadoc information for interface signatures.

Introspecting a system as a whole Reflexive component models such as Fractal therefore offer abstract views of systems, either dynamic (running systems) or static (template repositories). However, these views are generally at a small scale, i.e., at the scale of a component, and do not allow to consider a system as a whole, which is essential for instance when managing a running system. Such introspection mechanisms can therefore be considered only as a basis for general retrieval systems that allow to construct large-scale abstract views of systems. Such views are constructed using what we call *associative introspection*, i.e., queries on the components using introspection as a mechanism to access the component meta-data.

3. Scenarii

In the context of this paper, we consider not only development teams that develop reusable components and assemble systems using component repositories, but also other actors that require manipulating running systems, such as system administrators. This section presents two scenarii that emphasize the need for a broader-scoped component retrieval system that can query running systems through associative introspection.

Case of study We consider a simplified software architecture of an IP router, as a Fractal components composition, as illustrated in figure 1. The *C2* router control component is bound to the *C1* protocol stack component, through two interfaces provided by *C1*: Internet control protocol interface, and a TCP/IP connection interface. In this configuration, *C1* is a composite component that contains two components: *C11* implements the ICMP protocol and provides the Internet control protocol provided by *C1*, *C12* implements the TCP/IP protocol stack and provides the TCP/IP connection interface provided by *C1*. *C12* is in turn a composite component that contains two components: *C122* actually implements the TCP/IP protocol stack, *C121* is a filtering component. *C121* and *C122* both provide the TCP/IP connection interface, and they are bound together so that *C121* is a proxy (according to the Proxy design pattern) to *C122*, to filter connections opened through *C122*. *C11* and *C12* are bound to low-level protocol components, in order to actually receive and emit packets on the network. We con-

sider that most routers in the system have the same software architecture.

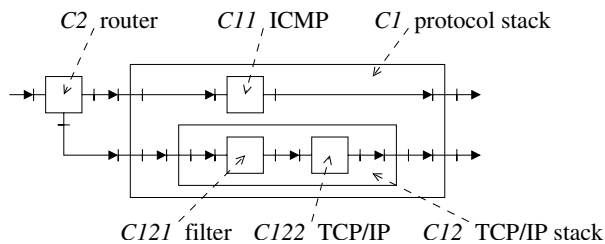


Figure 1. Example router component design

S1) Identify running components that provide the router service

An administrator needs to interact with each router to collect traffic statistical information. Each router control component provides such information through its provided interface. The administrator therefore has to retrieve the set of identities of components deployed on the network that provide the router interface. Using each component identity, she interacts with each router component to gather traffic information. That query feature is close to trading services in object-based systems, as specified in the ODP trading service specification [14] and the CORBA trading service specification, and in management protocols and architectures such as JMX [28]. In the context of the trading of binary heterogeneous COTS components repositories, [12] compares component trading and object trading systems, and proposes a trading architecture. However, in our context, the query system must offer trading features over running component-based systems.

S2) Replace packets filtering components

Following a denial of service attack, a security administrator decides to update the security policy. She therefore has to replace the all packet filtering components in routers. She has to carry out three actions: 1) identify the filtering components to replace, 2) retrieve a component implementation suitable to replace them, from a component repository, and 3) instantiate a component and make the actual replacement. The third step is out of the scope of this paper.

Identify components Packet filtering components are not only used in routers, but also in application servers and web servers, those are all interdependent and interconnected, and all concerned by security policies. The administrator therefore has to limit searching for filtering components in routers only. She then describes unambiguously such components as “primitive components bound to the provided

interface of a composite component that is contained in a composite component bound to a router control component”. Such a query takes into account component configuration information, typically hierarchical composition and bindings in the Fractal model.

Such a feature could be used in other cases, for instance when the administrator detects that a given composite TCP/IP stack component configuration is faulty, or consumes too much resources (e.g. memory leak), without being able to identify which subcomponent is actually faulty. In that case, she must replace the whole composite configuration, by first identifying such a configuration in all routers.

Find templates in a repository In the preceding scenario, in order to replace Fractal components, the administrator uses the standard Fractal reflective control interfaces of components to stop components, instantiate new ones, modify bindings and start the new components. In general, such a reconfiguration problem is complex, because most components have a state, and cannot be easily replaced dynamically. However, the problem of dynamic reconfiguration is out of the scope of this paper, and we consider that the packet filtering components are stateless.

The remaining problem is the construction of queries on a repository according to introspective information given by the running component to be replaced. Hence a component retrieval system must allow the expression of queries on both running systems and template repositories.

4. Related works

As one of the main purposes of component-based development is reusability, retrieving software components has been the center of interest of many research works. The problem that is generally addressed by such works is the retrieving of components that best fulfill requirements from libraries or component repositories. In this section we distinguish between two points of view: 1) general works and theory in *matching components*, where each work considers one homogeneous component model and minimal component specifications, and 2) researchs in *storing and selecting components* from a large number of heterogeneous components in repositories, that require an *external specification* of components that is additional to the one necessary for component matching techniques.

4.1. Component matching

Matching and comparing components concern several levels of component specifications: signature, semantic and behaviour. Typically, component specifications are expressed in an Architecture Description Language. *Signature matching* [25, 29, 13] uses signatures (or types) as a basis

for comparing components. The signature of a component is the set of the signatures of its interfaces, which in turn are sets of operation signatures. For instance, an interface signature is a Java interface in the Fractal / Java component model. Matching component signatures is achieved through the comparison of their sets of interface signatures, recursively. Exactly matching two component signatures implies they are exactly the same sets of interface signatures, and recursively of operation signatures. Relaxed matching is more general, for instance two operation signatures may match even if they declare different return types.

Semantic and behaviour matching [11, 30, 15], also called specification matching, considers more abstract specifications of components and interfaces. Such formal specifications may include for instance assertions (pre-, post-conditions and invariants) or temporal logic models [24].

4.2. Retrieving and storing components

The main purpose of research works in component repository querying is to assemble systems from existing reusable components. Those works have to deal with heterogeneity of components, i.e., those may be implemented in different component models, from different application domains (banking, medicine), etc. Such components are typically COTS-C (Commercial Off The Shelf Components). Two questions must be answered: 1) how to *represent* components in repositories? 2) how to *express and process queries*?

A representation model must deal with the integration of components in different component models and different domains. Several classifications of component information representations [20, 27, 16] have identified simple keyword sets, faceted and natural language specification classification. The expressivity of queries depends on the component representation.

Keyword sets Keywords are defined for each component according to general string keyword sets for each domain [21]. Queries are themselves expressed as simple keyword sets, and are therefore limited in expressivity and semantics because keywords have no intrinsic meaning.

Facets Facets are more elaborated component descriptions that are structured according to well-defined schemas (or ontologies or taxonomies) [23, 31, 4, 19]. Facets may describe a functional area, quality of service, relations between components, or domain-specific information, etc. Queries are expressed according to the underlying schema that structures each considered kind of facet, which may be for instance a relational database schema. It is however difficult to identify and separate the relevant facets for each

component, and to deal with the evolution of their underlying domain. Integrating components from different domains therefore also requires to convert facets from several schemas into a single one, for instance through the use of mediation-based architecture integration [4, 5].

Natural language specifications Natural language queries [10, 27] provide the most semantically rich information, and are the least constraining ones to users. However, querying on natural language specifications requires to use complex techniques to manipulate it, such as morpho-lexical analysis, and lexical resources such as dictionaries and thesauri that are difficult to gather.

4.3. Discussion

The component retrieval works described above consider component specifications at different levels: structural, semantic, behavioural, ontological descriptions, natural language descriptions, etc. These are complementary, i.e., our component retrieval system should allow combining various representations and techniques to improve the quality of queries. For instance, one of our concrete needs is to combine rich specifications from ADL files, and natural language interface specifications from Javadoc files. Furthermore, as in our context we consider the Fractal model as a unique component model, our system should therefore be able to efficiently use component matching techniques. The most important point is however that the scope of related works is different from ours, as generally they aim at assembling systems from reusable components retrieved from repositories. Therefore they do not address the problem of querying running systems, that we have motivated in the scenario, in the context of large-scale systems supervision. We argue that this is the most prominent feature of a general component retrieval system. More precisely, the scenario have highlighted the need for a highly distributed retrieval system that considers likely templates repositories and running systems, and that allows using both functional and topological component matching to construct views. Such a system can therefore be considered as a *middleware service*.

5. A general retrieval service

This section describes our proposal of a middleware component query service, in the form of a distributed component-based software framework, that can be used by the software tools of many kinds of actors: development teams, system administrators, etc. Its purpose is to remotely collect architectural data from introspected systems, and information from template repositories, both being likely considered as data sources. The problem therefore boils

down to the problem of distributed queries in the database management area. The issues of the query framework are then 1) query expression and 2) query evaluation.

5.1. Distributed query service architecture

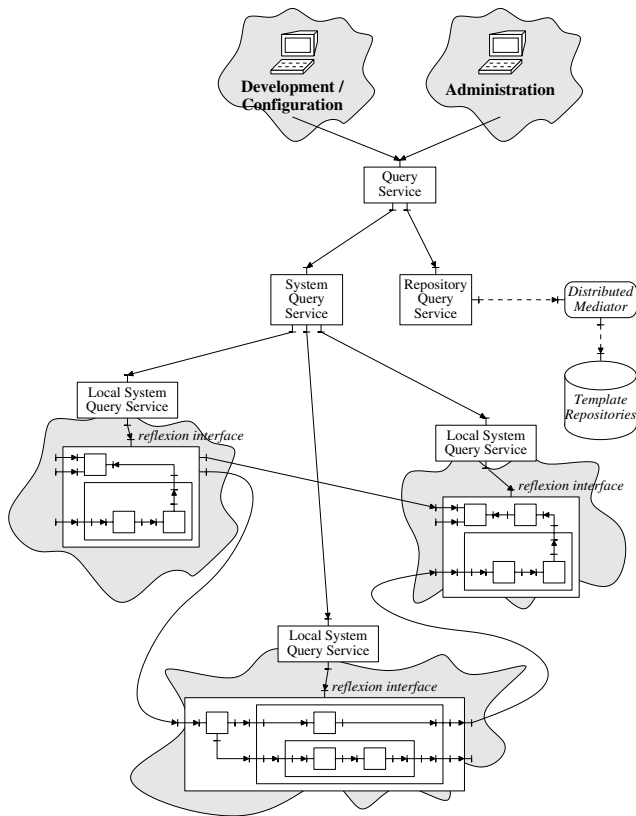


Figure 2. Distributed query service architecture

As the queried component configurations and repositories are highly distributed and numerous, the query service must efficiently evaluate distributed queries. The query service is therefore distributed itself. The query service architecture is designed using the Fractal component model. Figure 2 illustrates a typical configuration.

A Query Service component is the entry point for supervision and development tools to query systems and repositories. A Query Service homogenizes access to both repositories and running systems, by delegating to more specific kinds of components: Repository Query Services and System Query Services. A Repository Query Service translates Query Service queries into the query language of a mediator, which an access point to template repositories: it is therefore functionally comparable to existing related works

in component retrieval from repositories. A System Query Service component interacts with Local System Query Service components that are locally bound to introspection interfaces of components in the system, in order to query their meta-data. A Local System Query Service can recursively introspect the bindings and content of components (cf section 2), using the standard Fractal introspection interfaces, starting from the component to which it is bound. It therefore handles all the queries on a given subpart of the system. The placement and number of Local System Query Service components depends on the number of remote sites, i.e., connected through distributed bindings, and the number of components. Thus it impacts the performance of introspective query evaluation. The proposed modular query service architecture is well adapted to the properties of distribution and dynamicity of systems: Local System Query Service components can be deployed at runtime to tackle with dynamic evolution and the cost of distribution.

5.2. Query expression

Queries can be expressed on either the introspected system, through a System Query Service component, on template repositories represented by mediators through a Repository Query Service component, or on both. For instance, one should be able to query for “component templates in repositories that are similar to a given queried runtime component”. The following section points out that introspecting Fractal component configuration boils down to querying a semi-structured document. As repositories represent also structural information, i.e., component compositions and bindings, though their models contain more detailed information (cf section 2), we adopt the same semi-structured representation to express all queries in the query service.

System Query Service Bindings and composition relationships between Fractal components can be represented as an oriented labeled graph. For instance, figure 3 illustrates partially the graph for the component configuration for the router software layer illustrated in figure 1. The nodes represent architectural elements, i.e., components, bindings, interfaces, interface signatures, etc. Leaf nodes are primitive values such as names of interfaces and methods or operation argument types. Edges represent semantic relationships between these elements. Such a graph representation of data is the same as the OEM (Object Exchange Model) graph representation of semi-structured data [22]: like reflexive components, semi-structured data auto-describe themselves.

Expressing a query for associatively introspecting a system comes down to expressing queries over semi-structured data. Topological and functional expressions are trans-

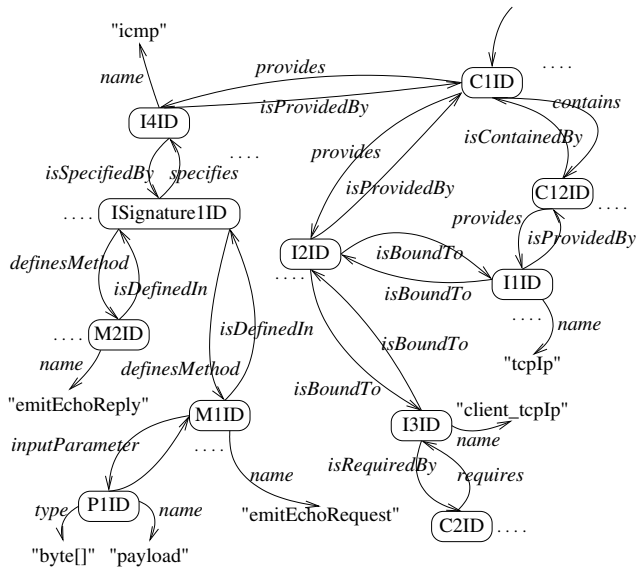


Figure 3. Labeled graph representation of a Fractal component configuration

lated into path expressions over the graph, that can contain regular expressions, predicates and wildcards, as largely described in many active research works in the semi-structured data query field, typically such as in [1]. For instance, to query the identify of ICMP components in routers, as illustrated in figure 1, those can be identified as “components C that provide an interface of name “icmp”, that are contained in a composite bound to a component providing an interface of name “router””:

```
select C
where *.C[provides.name="icmp"]
.isContainedIn.provides.isBoundTo
.isRequiredBy.provides.name="router"
```

In that case, one can say that the matching is *relaxed*, since only the names of interfaces have been used, but it would have been possible to compare interfaces signatures, etc. Similarly, queries of the scenarii can be expressed by such semi-structured query expressions.

Repository Query Service As repositories may have different data schemas and data from different domains, e.g., networking components or banking components, they are integrated by a mediation architecture, typically as in [5, 4], to provide an integrated view of the whole repositories. A Repository Query Service component therefore interacts with a mediator by translating queries into the mediator specific query language. For instance, the following query expression asks for “the component templates that provide

an interface of name “icmp”, and all the templates that extend them”. It should be noted that this query uses “extends” relationships, which are specific to templates in Fractal ADL repositories, and not available once components are instantiated in running systems.

```
select T
where *.T.(extends)*
.provides.name="icmp"
```

As a concrete case we consider that natural language specifications, with their language as an attribute, are associated with each architectural element in a repository, such as templates, interfaces signatures, bindings, etc. For instance, such specifications are extracted from Javadoc files. The following query expression asks for “templates that provide an interface which signature documentation contains a word that is semantically close to the english word “creates”, whatever the language of that documentation”, using the “containsWordCloseTo” natural language operator:

```
select T
where containsWordCloseTo(
"en", "creates",
*.T.provides.isSpecifiedBy)
```

That query will return templates which provide interfaces which signature documentation in English contains the word “created” as morphological analysis is performed to compare the several forms of “create”. It may also match “instanciate”, as it is a synonymous of “create”, and the French words “créer” and “instancier” as they are translations for “create”. This emphasizes the need for rich lexical resources such as thesauri to enhance the quality of results.

Query Service query expression Supervision and development tools interact with a Query Service component through what we call *global queries*, to query either running systems or template repositories, or both. Like in data integration query processing systems [3], each global query is decomposed into two sub queries, one being delegated to a System Query Service component, the other to a Repository Query Service component.

The query language allows for signature matching, behaviour matching and natural language specification matching. Signature matching is achieved through simple classical database schema matching operators, such as “<” and “=”, as signatures are represented as a schema or a graph as illustrated in figure 3 and the query examples above. As natural language specification matching relies on very large lexical resources and complex algorithms, its evaluation is essentially delegated to mediators. Behaviour specifications are more formal and complex: behaviour matching cannot be performed as a simple graph matching. Thus, in order

to perform behaviour matching, a Query Service retrieves behaviour-related data from mediators, before it evaluates it locally.

We consider that operators such as " \leq " that compares components for substitutability [13], are higher-order operators that can be expressed using the operators of our language.

In order to be flexible and adaptable to needs of software that uses the query service (as adding new operators), rather than to impose a query language such as XQuery, queries are expressed by constructing programmatically an algebraic tree, as we proposed in [2].

5.3. Query evaluation

When evaluating a global query, a Query Service component delegates as much as it can to the System Query Service component, and to the Repository Query Service component which in turn delegates to mediators. The non-delegated parts of the global query, such as the behaviour matching operators, are evaluated locally in the Query Service component.

Since we consider large scale and distributed component architectures, the queries passed to a System Query Service are distributed through Local System Query Services. This problem is the same as processing queries in distributed semi-structured databases through *structural recursion* [8]. In that context, [26] proposed an approach and algorithms to evaluate and optimize such queries, that we can use. More precisely, in our context, optimizing queries means that the number of interactions between a System Query Service and Local System Query Service should be constant, and the amount of transferred data should depend only on the size of the queries answers and the total number of introspected components, and not on the size of the whole system (number of components, etc.). To improve the efficiency of query evaluation, views can be cached. However, in this case, cache consistency must be explicitly handled by cache maintainers, since the system is dynamically evolving.

6. Conclusion

This paper argues for and sketches the design of an open component query service that would allow both queries on component repositories and queries on running software systems, which is our main contribution. The proposed architecture allows for such global queries using multiple component matching mechanism such as type matching, behavioural matching and natural language specification matching.

This article advocates the generalized construction of component-based distributed systems so that information

could effectively be queried on these systems uniformly. Component-based architectures would also ease the integration of a query service such as the one introduced in this paper. Indeed, such a query service is just one of the pieces of a complex software systems which involves development, deployment, monitoring and management services and tools.

The implementation of the proposed query service, based on the Julia Fractal-compliant Java platform [6, 9] and on the Medor distributed query processing framework [18], will include the implementation of the several components of the proposed architecture.

Finally, further development would include the study of the generalization of the usage of queries as a means to manage software systems - very much like queries in the database field - though intercession.

References

- [1] S. Abiteboul. Querying semi-structured data. In F. N. Afrati and P. G. Kolaitis, editors, *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18, Delphi, Greece, Jan. 1997. Springer.
- [2] M. Alia, S. Chassande-Barrioz, P. Déchamboux, C. Hamon, and A. Lefebvre. A middleware framework for persistence and querying of Java objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 2004.
- [3] M. Alia, C. Collet, and A. Lefebvre. Systèmes d'intégration de données : une approche à composants. In *Proceedings of Journées Composants (JC'04) and Langages et modèles à objets (LMO'04)*, Lille, France, Mar. 2004.
- [4] R. M. Braga, M. Mattoso, and C. M. Werner. The use of mediators for component retrieval in a reuse environment. In *Proceedings of the Workshop on Component-Based Software Engineering Process (TOOLS-30 USA '99)*, pages 542–546, Santa Barbara, CA, Aug. 1999.
- [5] R. M. Braga, M. Mattoso, and C. M. Werner. The use of mediation and ontology technologies for software component information retrieval. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR2001)*, pages 19–28, Toronto, Ontario, Canada, May 2001.
- [6] É. Bruneton, T. Coupaye, M. Leclerc, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (ICSE2004 - CBSE7)*, Edinburgh, Scotland, May 2004.
- [7] É. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP'02, ECOOP 2002)*, Malaga, Spain, June 2002.
- [8] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on struc-

- tural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, Mar. 2000.
- [9] T. Fractal Project. <http://fractal.objectweb.org/>.
- [10] M. R. Girardi and B. Ibrahim. Using English to retrieve software. *The Journal of Systems and Software*, 30(3):249–270, Sept. 1995.
- [11] D. Hemer and P. Lindsay. Specification-based retrieval strategies for module reuse. Technical report, University of Queensland, Australia, 1999.
- [12] L. Iribarne, J. M. Troya, and A. Vallecillo. Trading for COTS components in open environments. In *Proceedings of the 27th Euromicro Conference 2001: A Net Odyssey (EUROMICRO'01)*, Warsaw, Poland, Sept. 2001.
- [13] L. Iribarne, J. M. Troya, and A. Vallecillo. Selecting software components with multiple interfaces. In *Proceedings of the 28th Euromicro Conference (EUROMICRO'02)*, Dortmund, Germany, Sept. 2002.
- [14] ISO/IEC. ODP trading function. ITU-T Draft Rec. X.9tr I ISO/IEC DIS 13235, ISO/IEC, 1995.
- [15] J.-J. Jeng and B. H. C. Cheng. Specification matching for software reuse: A foundation. In *ACM Symposium on Software Reusability (SIGSOFT)*, pages 97–105, Seattle, Washington, 1995.
- [16] O. Khayati and J.-P. Girardin. Component retrieval systems. In *Proceedings of the Workshop on Reuse in Object-Oriented Information Systems Design (OOIS 2002)*, Montpellier, France, Sept. 2002.
- [17] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative research (CASCON'97)*, pages 144–154, Toronto, Ontario, Canada, Nov. 1997. IBM Press.
- [18] T. Medor Project: Middleware Enabling Distributed Object Requests. <http://medor.objectweb.org/>.
- [19] R. Meling, E. J. Montgomery, P. S. Ponnusamy, E. B. Wong, and D. Mehandjiska. Storing and retrieving software components: a component description manager. In *Proceedings of the 2000 Australian Software Engineering Conference*, pages 107–117, Canberra, Australia, Apr. 2000.
- [20] H. Mili, P. Valtchev, A.-M. D. Sciuillo, and P. Gabrini. Automating the indexing and retrieval of reusable software components. In *Proceedings of the 6th International Workshop (NLDB'01)*, pages 75–86, Madrid, Spain, 2001.
- [21] R. Mili, A. Mili, and R. T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1997.
- [22] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th Conference on Data Engineering*, Taipei, Taiwan, Mar. 1995. IEEE Computer Society.
- [23] R. Pietro-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, Jan. 1987.
- [24] N. Rivierre and T. Coupaye. Observing component behaviours with temporal logic. In *Proceedings of the ECOOP Workshop on Correctness of Model-based Software Composition (CMC'02)*, Darmstadt, Germany, July 2002.
- [25] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 166–173, London, UK, 1989. ACM Press.
- [26] D. Suciu. Distributed query evaluation on semistructured data. In *ACM Transaction Database Systems*, 27(1):1–62, 2002.
- [27] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.
- [28] Sun Microsystems. The Java Management eXtensions (JMX) 1.2 specification.
- [29] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.
- [30] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.
- [31] Z. Zhang. Enhancing component reuse using search techniques. In *Proceedings of the 23rd Conference on Information System Research in Scandinavia (IRIS'23)*, Lingatan, Sweden, Aug. 2000.