HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems

Keunwoo Lee
University of Washington
klee@cs.washington.edu

Anthony LaMarca
Intel Research Seattle
lamarca@intel-research.net

Craig Chambers
University of Washington
chambers@cs.washington.edu

ABSTRACT

In an evolving software system, components must be able to change independently while remaining compatible with their peers. One obstacle to independent evolution is the *brittle parameter problem*: the ability of two components to communicate can depend on a number of *inessential* details of the types, structure, and/or contents of the values communicated. If these details change, then the components can no longer communicate, even if the *essential* parts of the message remain unaffected.

We present HydroJ, an extension of Java that addresses this problem. In HydroJ, components communicate using self-describing, semi-structured messages, and programmers use pattern matching to define the handling of messages. This design stems from two central ideas: first, that self-describing messages reduce dependence on inessential message format details; and second, that object-oriented pattern matching naturally focuses on the essential information in a message and is insensitive to inessential information.

We have developed these ideas in the context of Rain, a distributed, heterogeneous messaging system for ubiquitous computing. To evaluate the design, we have constructed a prototype HydroJ compiler, implemented some Rain services in HydroJ, studied the evolution of an existing Rain service over time, and formalized HydroJ's key features in a core language.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; D.1.5 [**Software**]: Programming Techniques—*Object-Oriented Programming*; D.2.12 [**Software**]: Software Engineering—*Interoperability*

General Terms

Languages

Keywords

pattern matching, dynamic dispatch, XML, semi-structured data, software evolution, object-oriented programming, distributed systems, ubiquitous computing, HydroJ

1. INTRODUCTION

Distributed or persistent systems built from heterogeneous components are growing in importance — today's software systems often run on multiple nodes that differ widely in their underlying implementation technology, and that communicate across the boundaries of space, time, or administrative domains. Examples of such systems include web services [57] and ubiquitous computing [54] systems, as well as systems that process persistent semi-structured data stored in databases [52].

The combination of object-oriented languages and semi-structured data exchange formats are often touted as a good substrate for these kinds of systems (for example, SOAP [7] messaging combines these technologies). In particular, semi-structured data — of which XML [56] is currently the most popular species — has several strengths for this domain. First, the simple and transparent encoding of semi-structured data has enabled widespread cross-platform support, in the form of parsers and other tools.

Second, and perhaps more importantly, semi-structured data mitigates what we call the *brittle parameter problem.* The brittle parameter problem is caused by unnecessary dependence on low-level details of the types, structure, and/or contents of communicated data values. When components depend on these details, they cannot remain compatible in the face of changes to these details as the system evolves. For example, in a typical C-based remote procedure call (RPC) system [5], any change to a procedure's argument or return types requires that both endpoints be recompiled and relinked at once — even if only one endpoint depends on the changes. For example, if a component A originally sent a structure of type τ to component B, but then is upgraded to send a structure τ' which is just τ plus some additional fields, then replacement of A with its upgraded version will generally require that B be modified, recompiled, relinked, and restarted — even if B doesn't care about the extra fields in τ'. The reverse situation will happen if B is upgraded to return additional information that A doesn't care about. To take another example, in Java remote method invocation [51], the object serialization format embeds a hash of the structure of each serialized object's class. This hash serves effectively as a unique identifier — therefore, even the tiniest difference in the implementation or interface of an object sent as a parameter can render two communicating components incompatible.

We call *essential* those parts of the information communicated to a component that the component actually uses; all other parts of the information, format, and type are *inessential*. The brittle parameter problem occurs whenever the information sent from one component changes in a way that affects only features that are inessential to the receiving component, but the two components are still rendered incompatible.

Unlike more rigid data encodings (e.g., XDR [49], which is used in standard RPC), semi-structured data structures are often processed via libraries that permit variation in the exact format and contents of a message. For example, the existence of a subtree of data that the client program does not use can usually be ignored. Therefore, programs that communicate using XML data *can be* relatively indifferent to "inessential" details.

However, existing techniques for processing semi-structured data still leave much to be desired. Typically, programmers rely on libraries that either provide a cumbersome tree-traversal interface, or map semi-structured data to the types in the host language. The former approach sacrifices the convenience of the native programming system, and the latter sacrifices the flexibility of semi-structured data (we discuss these alternatives in more detail in Sections 3.2-3.3).

To address these problems, we propose HydroJ, a language that preserves much of the flexibility of semi-structured data exchange, but also provides compact and natural constructs for processing this data. HydroJ is based on two key ideas: first, software components should communicate via self-describing, semi-structured messages; and second, receiving components should define how to behave upon receipt of such a message via *handler methods*, whose interfaces are defined declaratively through *patterns* over messages rather than the usual method name and argument profile. Patterns naturally characterize the essential features of messages and ignore the inessential features.

We believe that this communication foundation will make HydroJ components significantly less brittle in the face of independent evolution of components in distributed or persistent systems. It will also provide a context in which other mechanisms that enable software evolution — such as support for dynamic updates to program code and state [25], external composition languages [1], and implicit or late-bound invocation [22, 2, 14, 30] — have more opportunities to update components without introducing incompatibilities.

Our specific technical contributions, relative to previous languages with similar features (e.g., XDuce [28, 27], which we discuss in Section 4, along with other related languages), are as follows:

- an "object-oriented" dispatch semantics in which message values dispatch to the *most specific* matching case in a set of patterns;
- the combination of ordered ("list-like") and unordered ("record-like") patterns in a single pattern language for semi-structured data with structural subtyping; and
- integration of these features into a mainstream programming language (Java), and an initial evaluation of their utility in the context of an experimental ubiquitous computing system.

Of course, even when two components exchange *syntactically* compatible data, many other factors may render them *semantically* incompatible. This holds true even for the most brittle communication mechanisms, such as RPC or RMI. For example, previous work by Deline [16], Garlan et al. [20], and Walker and Murphy [53] has catalogued various kinds of structural incompatibility beyond those we describe here. Our work strives to maximize the potential for syntactic compatibility in message data, leaving deeper semantic compatibility checking to separate mechanisms more appropriate to that task.

The rest of this paper is organized as follows. In the following section, we informally introduce HydroJ's syntax and semantics through a series of concrete examples. In Section 3, we describe an implementation and evaluation of HydroJ in the context of the Rain distributed messaging system [31]. Finally, in sections 4, 5, and 6, we describe related and future work and conclude.

2. HYDROJ LANGUAGE DESIGN

In this section, we describe the key features of HydroJ informally through a series of examples. At a high level, HydroJ is a Java extension that adds four principal features:

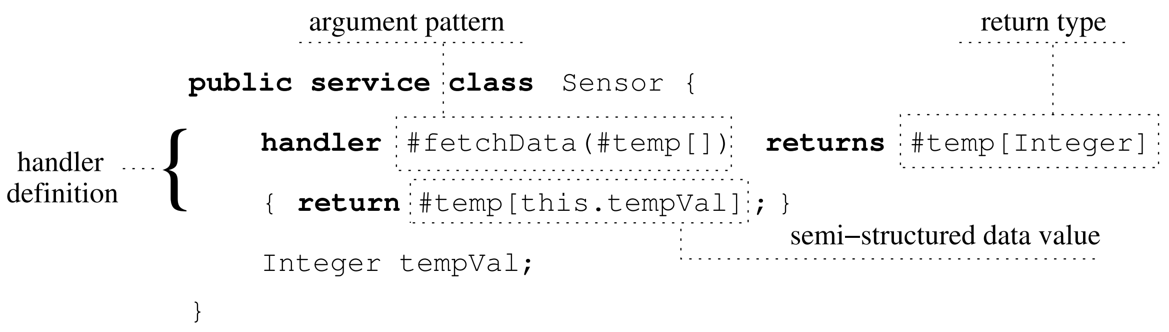
Semi-structured data To the Java roster of primitive, array, and class types, HydroJ adds a new kind of type for semi-structured, self-describing data. Informally, these data are arbitrary-arity trees whose non-leaf nodes are tagged with symbols ("tag names"). HydroJ semi-structured types abstract the salient features of XML. They are described further in Section 2.1.

Patterns and handlers HydroJ defines a special pattern language for describing semi-structured data. Patterns are principally used to specify the set of semi-structured messages accepted and returned by special *handler* methods. They also are used to specify the types of semi-structured values. The pattern sublanguage comprises HydroJ's most important extension to Java. We describe patterns, types, and handlers in Section 2.2.

Service classes Service classes[1] define the types of components of a HydroJ distributed system. Service classes are Java classes marked with an additional keyword `service`; classes so marked may define handlers. Instances of service classes — henceforth called *services* — dispatch incoming semi-structured messages using their handlers, rather than their ordinary methods. A simple example service is shown in Fig. 1. We describe service classes in Section 2.3.

Type-based discovery At runtime, service class instances publish their interface types with a public *discovery service*. Services connect to each other by sending queries to the discovery service; a query describes a service class interface, and the result consists of a list of currently published service class instances that match the requested type. We describe this discovery service in Section 2.4.

[1]The term "service" was chosen for historical reasons; see Section 3.1.



```
public service class Sensor {
    handler #fetchData(#temp[]) returns #temp[Integer]
    { return #temp[this.tempVal]; }
    Integer tempVal;
}
```

Figure 1: A simple service class, with one handler. The handler takes a message matching the pattern `#fetchData(#temp[])`, and returns a value of type `#temp[Integer]`.

In Sections 2.1-2.4, we present each of the above features in turn. In Section 2.5, we discuss pragmatic issues specific to the integration of HydroJ with its host language, Java. Finally, in Section 2.6, we briefly describe Core Hydro, a formalization of HydroJ.

In addition to the above features, HydroJ also has some constructs to support concurrency and distributed messaging. These constructs are non-novel and orthogonal to our central ideas, so we do not discuss them further.

In earlier work, we developed a "pure" Hydro language as a simplified vehicle for experimenting with the present ideas. Pure Hydro closely resembles the Core Hydro formalization; the only data types in Core Hydro are services and semi-structured data. HydroJ is therefore the embedding of (Pure) Hydro in Java. We believe Hydro's pattern language could be embedded in any statically typed general-purpose language, but we chose Java because its portability and rich libraries enabled us to rewrite actual Rain ubiquitous computing services in HydroJ, as discussed in Section 3.

2.1 Semi-structured data

The basic semi-structured data type in HydroJ is the *tagged tree*. A tagged tree can be constructed in HydroJ using the following syntax:

```
#tagName[child1, ..., childN]
```

where `tagName` is the root node's tag name, and `[child1, ..., childN]` are expressions evaluating to the children of this node. A tree with zero children is written as `#tagName[]`. As expected, a child of a tagged tree node can be another tagged tree. In addition, we assume that there is a fixed set of base types serializable into XML, such as `String` and `Integer`, whose values can be children of a tagged tree node. So, for example, one might represent a temperature sensor reading with the following tree:

```
#reading[#temperature[68], #humidity[1000]]
```

The children of a tagged node are ordered, so the above tree value is distinct from the tree

```
#reading[#humidity[1000], #temperature[68]]
```

Naturally, a programmer may wish to write code that is indifferent to the above distinction; we describe how HydroJ supports this in Section 2.2.3.

Sometimes it is useful to manipulate references to a semi-structured list directly; such values are constructed by using brackets without a tag name:

```
#[child1, ..., childN]
```

We store the child list of a tree node as a collection of object references, so all elements of a semistructured value must be of object type. Therefore, unboxed Java primitive values (`int`, `boolean`, etc.) cannot be directly stored in semi-structured types. However, we provide automatic boxing on tree construction and some syntactic sugars for unboxing during pattern matching.

All semi-structured values are immutable, to simplify typing of references to these values.

2.2 Patterns, types, and handlers

Pattern matching is a powerful language mechanism that compactly specifies both how a value should be structured and how pieces of that value should be extracted and bound to names. In HydroJ, a pattern is used to define the "profile" of a *handler* method, which is responsible for processing semi-structured messages. A handler's pattern is analogous to a regular method's name and formal parameter list.

Handlers in HydroJ have the following syntax:

handler ::= `handler` P `returns` τ { *stmt*$*$ }

The pattern P defines the set of values to which this handler is applicable, i.e., its argument type. The type τ gives the handler's return type.

The syntax of semi-structured types and patterns is given by Fig. 2. A type describes a set of values. The pattern syntax corresponds exactly to the type syntax, augmented with optional name binding constructs. A pattern matches a value if and only if its corresponding type (i.e., the type derived by erasing all name binding constructs) includes that value.

Each feature in the pattern language has a practical motivation; the combination of these features in a single language is one of the key technical contributions of this work. In the rest of this section, we motivate and describe patterns in more detail using examples.

2.2.1 Simple types and patterns

The `any` type describes the set of all values, making it the equivalent of `Object` for semi-structured types. At the opposite extreme, the type `void` describes the empty set of values. It may appear only as the return type of handlers that do not return values.

A base type *baseType* describes the set of values of that type. The current HydroJ implementation supports `String`

Types

τ	::=	`any` | `void`	any value, no value
	|	*baseType*	value of base type
	|	*baseValue*	particular value
	|	*tagName* Λ	tree with tag *tagName* and children of type Λ
	|	Λ	bare list
Λ	::=	`[`$\rho_1, \ldots, \rho_n$`]`	list (ordered)
	|	`(`$\rho_1, \ldots, \rho_n$`)`	bag (“unordered list”)
ρ	::=	τ | `*`τ	single and repeated child

Patterns

P	::=	`any`	any value
	|	*baseType*	value of base type
	|	*baseValue*	particular value
	|	*tagName* L	tree with tag *tagName* and children matching L
	|	L	bare list
	|	*id*`=`P	match P and bind to name *id*
L	::=	`[`Q_1`,` ... `,`Q_n`]`	ordered list
	|	`[`Q_1`,` ... `,`Q_n`,`*id*`=..]`	ordered list, binding rest
	|	`(`Q_1`,` ... `,`Q_n`)`	bag (“unordered list”))
	|	`(`Q_1`,` ... `,`Q_n`,`*id*`=..)`	bag, binding rest
Q	::=	P | `*`P | *id*`=*`P	single and repeated child

Tag and type names

tagName	::=	`#`*id*
baseType	::=	*id*
	|	*packageName*`.`*baseType*

Figure 2: Syntax of HydroJ semi-structured data types and patterns

and `Integer` (as well as `int`, which is an `Integer` that's automatically unboxed), although there is in principle no obstacle to supporting more. A literal value of a base type can also be used as a type, describing the singleton set of just that value.

Recall that a pattern matches a value iff the value is a member of the type corresponding to the pattern. Thus, the `any` pattern matches any value, the *baseType* pattern matches any value of that base type, and a literal value pattern matches only that value.

2.2.2 Trees and ordered lists

A tree type $\#id\ \Lambda$ describes the set of all trees with tag *id* whose children are of list type Λ. One kind of list type, the ordered list type, `[`τ_1`,` ... `,`τ_k`]`, describes the set of all semi-structured lists of length $\geq k$ whose first k children have types τ_1,...,τ_k. Lists matching a pattern of this form can have more than k elements; any extra elements are ignored. This feature, *width subtyping* for ordered list types, is one important way that HydroJ's patterns avoid dependence on inessential information: a sender can evolve by adding additional trailing “parameters” to a tree of information, without affecting communication with unchanged receivers.

For example, a service might have the following handler to handle polling requests:

```
handler #poll[] returns #ack[] { ... }
```

The `#poll[]` pattern matches any message that's a tagged tree whose root tag is `poll`. The child list pattern `[]` matches any list with ≥ 0 children, i.e., any list at all. Therefore, a message `#poll[#timestamp[123456]]` would match this pattern as well. An alternate, more specific handler:

```
handler #poll[#timestamp[int]] returns #ack[]
  { ... }
```

would also match the following messages:

```
  #poll[#timestamp[123456]]
  #poll[#timestamp[123456,any],#sender[String]]
```

(width subtyping is used twice, at different tree levels, in the second tree), but not the following:

```
  #poll[]
  #timestamp[123456]
  #poll[#sender[String],#timestamp[123456]]
```

(in the last tree, the `#timestamp[123456]` subtree is not the *first* child, so it doesn't match the ordered list pattern).

2.2.3 Bags (unordered lists)

The order of children in a list may not be important to the receiver of a message. This is particularly true if all essential elements of the list are distinctly tagged trees, as in the examples above. For example, a handler may wish to match any `#poll[...]` message that has a `#timestamp[int]` subtree as any child, not just as the first subtree. Intuitively, `#poll` may be a “record-like” tree, in which the child order does not matter, just their “field names”, i.e., tags.

To meet this need, HydroJ provides unordered list types, called *bag* types, written with round parentheses `(...)` instead of square brackets `[...]`. A bag type of the form `(`τ_1`,` ... `,`τ_k`)` describes the set of all semi-structured lists of length $\geq k$ that have children with types τ_1,...,τ_k *in any* *order, not necessarily consecutively.* As with ordered list patterns, list values matching a bag pattern of this form can have more than k elements; any extra elements are ignored. This more flexible kind of width subtyping for bag types is a second important way that HydroJ's patterns avoid dependence on inessential information: a sender can evolve by inserting parameters or reordering parameters, without affecting communication with unchanged receivers.

For example, a more flexible polling handler could be written as follows:

```
handler #poll(#timestamp[int]) returns #ack[]
  { ... }
```

This handler would accept the following messages:

```
  #poll[#timestamp[123456]]
  #poll[#timestamp[123456],#sender[String]]
  #poll[#sender[String],#timestamp[123456]]
```

but not the following:

```
  #poll[]
  #timestamp[123456]
  #poll[#info[#timestamp[123456]]]
```

Both ordered list and (unordered) bag patterns can match against the same underlying lists (whose elements always appear in some order). Therefore, when constructing a value, a programmer need not decide *a priori* whether to view the lists within it as ordered or unordered. It is only a pattern's *view* of data that is ordered or unordered.

To ensure non-ambiguity of pattern matching for bag patterns, we require that the child patterns in a bag pattern be *disjoint*, i.e., that each child pattern match a non-overlapping set of values from every other child pattern. Intuitively, this captures the fact that bag patterns are "record-like:" in a conventional language, the fields of a record are disjoint by virtue of having distinct field names. Our requirement generalizes this notion. In the common case (including all the examples above), child patterns will have disjoint root tags. However, our disjointness rule also makes patterns like following legal:

```
  #data(#reading[#humidity[int],
        #reading[#temp[int]]])
```

The two subpatterns above, `#reading[#humidity[int] and #reading[#temp[int]]`, cannot match the same value, even though they have the same root tag `#reading`. Therefore, they are permitted. We can statically check whether a bag pattern has disjoint child patterns. At runtime, we can implement pattern matching against a bag pattern in a single left-to-right pass through the list value's children, without backtracking.

2.2.4 Repetition ("star") types and patterns

The list types above are analogous to tuple or record types in a traditional language: they have a fixed number of (essential) elements, each with its own type. It is also important to describe the analogue to a traditional array type: a variable-length list of elements, all of which have the same type. In keeping with previous work in XML processing languages, such as XDuce [28], we support this notion not with a new kind of list type, but rather with a new form of list child: the repetition type.

Repetition types have the syntactic form $*\tau$, and they can appear only in place of a child in a list (ordered or unordered). The type $*\tau$ describes a sequence of zero or more children, all of type τ. The syntax echoes the Kleene star operator of regular languages, and we will informally call these *star types.* We will call the type τ beneath a star type $*\tau$ its *base type.*

For example, a handler could match against a sequence of `reading` values that contain a temperature value by using the following pattern:

```
handler #data[#timestamp[int],
            *#reading[#temp[int]],
            #sender[String]] returns void
  { ... }
```

This handler accepts all of the following messages:

```
  #data[#timestamp[1234],
        #sender["sensor3"]]
  #data[#timestamp[1234],
        #reading[#temp[12]],
        #sender["sensor3"]]
  #data[#timestamp[1234],
        #reading[#temp[12]],
        #reading[#temp[23]],
        #reading[#temp[34]],
        #sender["sensor3"]]
```

Star types are permitted as children of bag types and patterns as well. In a bag pattern, a star pattern matches all children that match its base pattern, regardless of the order of those values. For example, consider the following handler:

```
handler #data(#timestamp[int],
            *#reading[#temp[int]],
            #sender[String]) returns void
  { ... }
```

This handler accepts of all the messages above, plus the following:

```
  #data[#sender["sensor3"],
        #timestamp[1234]]
  #data[#reading[#temp[12]],
        #timestamp[1234],
        #sender["sensor3"]]
  #data[#sender["sensor3"],
        #reading[#temp[12]],
        #reading[#temp[23]],
        #timestamp[1234],
        #reading[#temp[34]]]
```

A single list pattern may contain several star patterns, each of which matches disjoint subsequences of children. For example, the following handler accepts sequences of temperature and humidity readings, in any order:

```
handler #data(#timestamp[int],
            *#reading[#temp[int]],
            *#reading[#humidity[int]],
            #sender[String]) returns void
  { ... }
```

This handler accepts of all the messages above, plus the following:

```
#data[#reading[#humidity[65]],
      #sender["sensor3"],
      #reading[#temp[12]],
      #reading[#humidity[87]],
      #reading[#temp[23]],
      #reading[#humidity[98]],
      #reading[#humidity[76]],
      #timestamp[1234],
      #reading[#temp[34]]]
```

As with bag patterns, unrestricted star patterns can introduce ambiguities in pattern matching. To ensure non-ambiguity of pattern matching in the face of star patterns, and to enable matching to be implemented by a single left-to-right non-backtracking pass through the list value's children, we impose some disjointness restrictions. In ordered list patterns, we require that each star pattern child be disjoint with the following child pattern (if any); this ensures that it is unambiguous when the subsequence of children matching the star pattern ends and the next child pattern (which might itself be another star pattern) begins. In bag patterns, the stricter rule that all children are disjoint suffices even in the face of star patterns.

For example, the pattern

```
#readings[*#info(#temp[int]),
          *#info(#humidity[int])]
```

is not legal, because the children of `#readings` overlap on values of type

```
#info(#temp[int],#humidity[int])
```

In other words, if the `#readings` tree begins with a child with the above type, it is unclear whether this child should be matched against `*#info(#id[int])`, or instead against `#info(#temp[int])`.

In contrast, the slightly different pattern

```
#readings(*#info[#temp[int]],
          *#info[#humidity[int]])
```

is legal, because the two subpatterns `#info[#temp[int]]` and `#info[#humidity[int]]` do not overlap (since the `#info` tags use ordered list rather than bag patterns). At run-time, it is always clear which if any of `#info[#temp[int]]` or `#info[#humidity[int]]` a child value matches.

2.2.5 Name binding

When matching a semi-structured value against a handler's pattern, names can be bound to parts of the value for later reference in the handler body; these bound names play the role of formal parameters for handlers. For example, a pattern of the form *id*=*P* matches the same values that *P* matches, but also binds the identifier *id* to the value that matched *P*. Name binding patterns can occur anywhere within a larger pattern. A given identifier can be bound at most once in a handler's pattern.

For example, suppose the programmer wished to *use* the timestamp value in the `#poll[]` message, rather than merely matching against it. The programmer could then write the following:

```
handler #poll[#timestamp[ts=int]] returns #ack[]
  { ... ts ... }
```

The `ts=int` subpattern specifies that the value matching `int` should be bound to the name `ts` for use in the handler body.

We can also bind the result of a star pattern to a name:

```
handler #data[#timestamp[ts=int],
              infos=*#reading[#temp[int]],
              #sender[sender=String]] returns void
  { ... ts ... infos ... sender ... }
```

In the above code, `infos` will be bound to the entire sequence of consecutive values matching the star pattern. The type of `infos` will be `*#reading(#temp[int])`. Accessor operations are provided on sequences to get individual elements (e.g., `infos.getFirst()`) and to iterate through the sequence (see Section 2.5.2).

Binding a name *beneath* a star pattern requires special treatment. Consider the following:

```
handler #data[#timestamp[ts=int],
              infos=*#reading[#temp[t=int]],
              #sender[sender=String]] returns void
  { ... }
```

In this pattern, `t` should be bound not just once, but once for each element of the enclosing `infos` sequence. To store these bindings, HydroJ augments each element of `infos` with a field named `t` of type `int`. These fields can be accessed in the body of the handler, for instance:

```
  int firstTemp = infos.getFirst().t;
```

2.2.6 Rest patterns

Through width subtyping, list patterns can match list values with additional children beyond those explicitly listed in the pattern. To enable the programmer to conveniently access these additional children, HydroJ allows list patterns to optionally end with a special "rest pattern" *id*=`..` (two dots). If present, a rest pattern binds a name to all values in the list not matched by any previous child pattern.

For example, consider the following handler:

```
handler #poll[#timestamp[t=int], rest=..]
  returns #ack[] { ... t ... rest ... }
```

When matched against the value

```
  #poll[#timestamp[1234],
        #auditKey[3456], #timeout[5678]]
```

the name `rest` would be bound to the list value

```
  #[#auditKey[3456], #timeout[5678]]
```

Rest patterns may be used in bag patterns as well; they match any children not matched by explicit child patterns. For example, consider the following handler:

```
handler #poll(#timestamp[t=int], rest=..)
  returns #ack[] { ... t ... rest ... }
```

When matched against the (reordered) value

```
  #poll[#auditKey[3456],
        #timestamp[1234],
        #timeout[5678]]
```

the name `rest` would be bound to the list value

```
  #[#auditKey[3456], #timeout[5678]]
```

The rest pattern affects name binding only; its presence or absence in a list pattern does not affect the set of values that the list pattern matches.

2.3 Service classes

HydroJ service classes are Java classes marked with the keyword `service`. Instances of service classes, called services, are prepared to send and receive semi-structured messages from other services. A service class may contain handler methods, which define the set of semi-structured messages the service is prepared to receive (i.e., the service's *service type*) and how incoming semi-structured messages should be dispatched to handler bodies. A simple service was shown in Fig. 1; some extended examples are given in Section 3.

2.3.1 Message dispatch

When a semi-structured message is received by a service, the service attempts to match the incoming message against each of its handlers' patterns. Of the handlers that matched successfully (called the *applicable handlers*), the *most specific* handler is chosen for execution. One handler is more specific than another if its pattern is more specific, meaning that it matches a strict subset of values (the corresponding type is a strict subtype).

In general, two handlers may be incomparable, neither being more specific than the other. As a result, there may not be a single applicable handler that is more specific than all other applicable handlers. (These issue are analogous to those found in languages with multiple inheritance or multiple dispatching.) Likewise, there may be no applicable handler for an incoming message. It is the task of static type checking of service classes to ensure that neither of these eventualities can occur, discussed below.[2]

Once a unique, most-specific applicable handler has been chosen, its body is executed in an environment where all the names bound by the selected handler's pattern are available.

2.3.2 Service types and type checking

The type of a service class consists of two parts: its Java type and its service type. The Java type includes the class's regular Java methods and fields. The service type is the set of *handler types*, each of the form `argumentType -> returnType`, for all of the service's handler methods. The argument type of a handler's handler type is just the type corresponding to the handler's argument pattern. The service type defines the interface to the service, i.e., the set of semi-structured messages that the service is prepared to receive and the result messages that will be returned in response.

A key component of type checking of a service verifies that the service's handler types are *unambiguous* and *conforming.* This checking ensures that, for every message that may legally be sent to a service, there will exist a unique most-specific handler, and moreover that the message returned by this handler conforms to the return type expected by the sender. The set of legal messages and their corresponding expected return types are given by the service's service type.

To verify *non-ambiguity* among handlers, it is sufficient to compare each pairwise combination of handler argument types. In each pair, if one argument type is strictly more specific than the other, then this pair is unambiguous. Otherwise, the intersection of the types is computed. If this intersection is empty, then the types are disjoint and hence unambiguous. Alternatively, if one or more other handlers have argument types that both cover this intersection and are more specific than at least one of the ambiguous argument types, then the ambiguity is resolved by these other "overriding" handlers. Otherwise, the intersection represents a potential ambiguity, and an error is reported statically.

To verify that return type declarations *conform* to the sender's expectations, we verify that, whenever one handler is more specific than another (i.e., that overrides the other), the more-specific handler's return type is equal to or a subtype of the less-specific handler's.

The following is an example of a class that triggers an ambiguity error during type checking:

```
service class Amb {
  handler #a(#b[]) returns void { ... }
  handler #a(#c[]) returns void { ... }
}
```

The intersection of these two handlers' argument types is `#a(#b[],#c[])`; any tree of this type will be matched by both handlers, but neither handler is more specific than the other. To fix this error, the programmer must add an overriding handler that resolves the ambiguity:

```
  handler #a(#b[], #c[]) returns void { ... }
```

The following is an example of a class that has a conformance error:

```
service class Amb {
  handler #a()     returns #ack[] { ... }
  handler #a(#b[]) returns void   { ... }
}
```

The second handler overrides the first, but its return type is not a subtype of the first's. This could cause an error for a client that sent a message of static type `#a()` (which might at runtime be the value `#a[#b[]]`) and expected a value of type `#ack[]` back. The second handler's return type should be changed to `#ack[]` or any subtype of `#ack[]`.

This "implementation-side type checking" is similar to algorithms used to check non-ambiguity of a set of multiply dispatched methods [8] or patterns in object-oriented extensions of functional languages such as EML [39]. More details are presented in our accompanying technical report [32].

2.3.3 Inheritance

A service class may extend another service class, in which case it inherits its superclass's handlers. The semantics of handler inheritance are mostly straightforward. The only complication is that handlers dispatch on a non-receiver argument — the message — whereas ordinary Java methods dispatch on the receiver only. This means that HydroJ services implement a form of multiple dispatch, which introduces additional opportunities for method ambiguity. Consider the following two service classes:

```
service class A {
  handler #a[#b[]] returns void {}
}
service class B extends A {
  handler #a[] returns void {}
}
```

[2]In fact, when a message is received, handlers are conceptually checked for applicability in order of most-specific to least-specific. As soon as an applicable handler is found, it can be chosen for execution, since static type checking guarantees that this handler will be the most-specific one.

If a B instance receives a message #a[#b[]], should it dispatch to the handler defined in B or the one defined in A? We solve this problem by considering such cases to be static ambiguities. We require that the user define overriding handlers for the ambiguous cases, just as with ambiguously defined handlers within a single class. For example, the user could add the following method to the class B:

```
handler #a[#b[]] returns void {}
```

Since this handler is at least as specific on the argument type, and strictly more specific on the receiver type, it strictly overrides the handler defined in A.

2.4 Channels and the discovery service

The previous subsections described how services define the *receipt* of semi-structured messages. Services also need a mechanism to connect to each other and *send* semi-structured messages. The main way a service finds out about other services is through a *type-based discovery service.* When a service is created, it is automatically registered with the discovery service, indexed by its service type. A service can find out about other registered services using a discovery expression of the form discover *serviceType*, where *serviceType* is written as a semicolon-separated list of handler types enclosed in curly braces. For example, the following is a legal discovery expression:

```
discover { #ping[] -> #ack[]; #pong -> #back[] }
```

The result of evaluating a discovery expression is a sequence of *channels* to all registered services whose service types are equal to or subtypes of the specified service type. (Subtyping between service types follows the usual rules for record and function subtyping: the order of handler types within a service type doesn't matter, the service subtype can have more handler types than the service supertype, and one handler type is a subtype of another if the subtype's argument type is at least as general as the supertype's and the subtype's result type is at least as specific as the supertype's.)

To send a message to a service, a program selects one of the channels returned by a discovery expression and uses the semi-structured send operator, <-:

```
(discover { #ping[] -> #ack[] }).getFirst()
  <- #ping[];
```

This sends the #ping[] message to the first service returned by the given discovery query.

The type of the channels returned by a discovery expression is the service type used in the discovery query. This type enables static checking of the semi-structured send. A semi-structured send is type-correct as long as there is at least one handler type in the receiving channel's type whose argument type covers (i.e., is a supertype of) the message being sent. This check justifies the assumption when type checking service classes that only legal messages will be received. The type of the result of a semi-structured send is the return type corresponding to the covering argument type.[3] (Type checking of service classes ensures that this assumption about the type of the result is valid.)

[3]It is possible for more than one handler type to cover a given message send, in which case the send's result type is return type of the most specific matching handler. We apply the same ambiguity rules to channel types as to service types, so there can be at most one most-specific handler.

2.5 Java integration

Up to this point, our presentation of HydroJ's features has been fairly insulated from the Java host language. In this section, we describe the portion of HydroJ that deals specifically with Java. The main issues are (a) what Java types are used to manipulate semi-structured values, (b) what operations are supported on those types, and (c) how uses of these types are statically type checked.

2.5.1 *Java types for semi-structured values*

The pattern of a handler can bind names to parts of the received semi-structured message. These names are to be used in the body of the handler, which is written in regular Java code (augmented with the constructs for discovery, semi-structured sends, and semi-structured value creation described above). But to refer to these bound names, they need to have proper Java types. These Java types will define the operations that can be performed within handler bodies. Handlers should also be able to pass semi-structured values to any other Java code, including the Java standard library.

Accordingly, there is a bidirectional mapping between non-void Hydro types and a set of Java types. Each kind of semi-structured Hydro type has a corresponding class in the package hydroj.lang: trees belong to the class XTree, lists belong to the class XList, sequences (star types) to XSequence, and so on.

However, we do not wish to lump all different kinds of e.g. trees into a single Java type, because this would lose the static type information in the Hydro type about the structure of the tree. Such a loss would not allow differently structured values to be kept distinct through Java code, and it would prevent static type checking of the semi-structured value returned by a handler. Therefore, we *parameterize* each of the Java classes for semi-structured data by the corresponding Hydro type. This enables us to statically track the Hydro types through Java code and then recover the Hydro types when returning semi-structured values from handlers.

The following table gives the relationships between the Hydro and Java forms of each kind of type:

Hydro type	corresponding Java type
any	hydroj.lang.XValue
baseType	*baseType*
baseValue	Java type of *baseValue*
tagName Λ	hydroj.lang.XTree<*tagName* Λ>
Λ	hydroj.lang.XList<Λ>
$*\tau$	hydroj.lang.XSequence<$*\tau$>
$\{handlerType; \ldots\}$	hydroj.lang.XChannel< $\{handlerType; \ldots\}$>

Parameterized types may appear in HydroJ code anywhere that a regular Java type may appear. For example:

```
XTree<#foo[String]> t = #foo["hi"];
```

The parameterized type is optional, and users can (implicitly or explicitly) cast away the type parameter:

```
XTree u = #foo["hi"];
```

Because all HydroJ values are self-describing, we can also allow runtime casts to recover the type parameter:

```
XTree<#foo[String]> v = (XTree<#foo[String]>)u;
```

This is semantically equivalent to matching the value against the requested type. Any Hydro type can be used as the parameter, and the cast will succeed as long as the semi-structured value is a member of the given type. If the match fails, a runtime cast exception is thrown.

To reduce the burden of writing type parameters, a simple form of local type inference is provided: when assigning to a final reference from a semi-structured value with more specific type, the type of the initializer is propagated to the reference. For example, in the code fragment

```
String s = ...;
final XTree t = #foo[s];
```

the second line is equivalent to

```
final XTree<#foo[String]> t = #foo[s];
```

2.5.2 Operations on semi-structured values

Each of the `hydroj.lang.*` classes described above provides a set of operations for manipulating semi-structured values of that type. For example:

- `XTree` provides two accessor methods, `getTag()` and `getChildren()`; the former returns a `String`, and the latter returns an `XList` instance of appropriately parameterized type.

- `XSequence` provides an `xiterator()` method, which returns an instance of the special class `hydroj.lang.XIterator`. `XIterator` instances are parameterized by the Hydro type of the sequence's underlying base type, and allow type safe extraction of elements of a sequence:

  ```
  handler #temperatures[ts=*#temp[Integer]] {
    // ts is of type XSequence<*#temp[Integer]>
    XIterator<#temp[Integer]> i = ts.xiterator();
    while (i.hasNext()) {
      XTree<#temp[Integer]> nextT = i.next();
      ... // code dealing with #temp[Integer]
    }
  }
  ```

  Note the absence of a cast when assigning to `nextT` in the loop body.

- `XList` provides an accessor, `iterator()`, which returns a `java.util.Iterator` over the list. Because `XList` classes can be heterogeneous, it is not possible to provide more specific type information about values returned by this iterator than that they are `Object`.

2.6 Formalization: Core Hydro

In order to gain confidence in our design, and to guide our implementation, we have formalized the interesting features of HydroJ's pattern matching and type system in a core language, Core Hydro. Although we do not yet have proofs of soundness, simply stating the language semantics precisely and formally has already yielded considerable benefits for our understanding. For space reasons, we omit the formalization from this paper; a complete presentation appears in our accompanying technical report [32].

3. IMPLEMENTATION AND EVALUATION

The HydroJ language is implemented via source-to-source translation into Java using the Polyglot extensible compiler infrastructure [43]. The current HydroJ implementation, including the compiler extension and runtime libraries, adds about 16,000 lines of code (written mostly in MultiJava [9]) to Polyglot's 45,000 lines of Java code.

In the remainder of this section, we describe the use of HydroJ in the context of the Rain distributed system. First, in Section 3.1, we briefly describe Rain itself. Next, in Sections 3.2 and 3.3, we describe two Rain services, `LightControl` and `FusionService`, originally written in Java. Adapted from ubiquitous computing applications developed by our colleagues at Intel Research and the University of Washington, these services' Java implementations use two alternative approaches to handling semi-structured messaging, which we call *manual disassembly* and *host type mapping*, respectively. We explain the drawbacks of these approaches, and contrast them with hypothetical implementations in HydroJ.

These services' interfaces are relatively simple, and they do not exercise all the flexibility of HydroJ's type system; we present them here primarily to provide some assurance that the features of HydroJ would yield real benefits when writing components in a system like Rain.

In Section 3.4, we describe the evolution of `SqlService`, a third Rain service whose development history we examined in order to understand whether the HydroJ's flexibility would actually yield benefits in realistic evolution scenarios.

Finally, in Section 3.5, we briefly discuss the performance of HydroJ dispatch.

3.1 Context: Rain

Rain is a messaging infrastructure originally developed for ubiquitous computing research [31]. Participants in a Rain system are called services; this is the origin of our term "service class". Rain services communicate using messages encoded in XML and sent over HTTP [29]. Because of this lowest-common-denominator data encoding and network transport, Rain services may be written in any language; in this sense, the Rain message transport resembles XMLRPC [55], SOAP [7], and .NET messaging [38]. Nonetheless, most Rain services to date have been written in Java.

Rain services "publish" their existence by registering with a unique shared discovery service.[4] Rain services find other services by sending a query to this discovery service, which returns a list of registered services that match the query. The details of Rain's discovery service are unimportant to our work; we note only that services are registered and queried by name, and that (unlike in HydroJ) a service's advertisement is not checked against its interface type.[5]

Incidentally, message sends in Rain are asynchronous, and return immediately rather than waiting for a result. In order to expose this control over concurrency to the programmer, HydroJ message sends are also asynchronous, and evaluate to a *future* object [18] that, when touched, blocks waiting

[4]The discovery service is conceptually a single entity, but there exist well-known algorithms for reliable distributed lookup services, e.g., INS [2] and Chord [50].

[5]More precisely, services register an advertisement encoded as an XML tree. Queries return all services whose advertisements match a given XPath expression [58].

```
public class LightControl
  extends com.intel.research.rain.Service
{
  public void process(Message m) {
    Element el = m.getElement("Light");
    if (el == null) return;

    Element el2 = el.getElement("Location");
    if (el2 == null) return;
    String location = el2.getText();

    Element el3 = el.getElement("Operation");
    if (el3 == null) return;
    String op = el3.getText();

    if (op.equals("ON")) {
       ... // handle ON messages
    } else if (op.equals("OFF")) {
       ... // handle OFF messages
    } else if (op.equals("FLASH")) {
       ... // handle FLASH messages
    } else {
       ... // (raise error)
    }
  }
}
```

Figure 3: Java version of the `LightControl` Rain service. The code extracts components of the received message using Rain's XML library, then performs different operations based on the extracted `op` value.

for the reply to arrive. In this paper, we have omitted this detail from the discussion of HydroJ, because futures are a well-studied language mechanism that is orthogonal to the present work.

3.2 `LightControl` and Manual Disassembly

Our first example is `LightControl`, a service responsible for remotely controlling light switches in a house. Despite its simplicity, this example was part of an actual application (a demo of an "assisted living" system for the mentally or physically impaired), and it illustrates a common strategy for processing semi-structured data.

When the Rain runtime system delivers a message addressed to a service written in Java, it invokes that service's `process` method. A fragment of `LightControl`, including its `process` method, is shown in Fig. 3. `Element` is the class that represents a tagged tree in Rain's XML library. `Element` corresponds roughly to HydroJ's `XTree`. The `getElement` method of `Element` takes a string, and returns the first child whose tag name equals the string; if no such child exists, it returns `null`. The `getText()` method returns the tree's first child of class `String`.

We call the message processing technique employed here — i.e., ad hoc library calls — the *manual disassembly* style. Note this style is quite flexible with respect to independent extensibility: a component implemented this way requires nothing of its peers except that they include the proper tags in the proper structure. The user gets subtyping "for free" — if peers include more information than this component needs to do its work, it will not break.

```
public service class LightControl {
  handler #Light[ #Location[loc=String],
                  #Operation["ON"] ]     {
    ... // handle ON messages
  }
  handler #Light[ #Location[loc=String],
                  #Operation["OFF"] ] {
    ... // handle OFF messages
  }
  handler #Light[ #Location[loc=String],
                  #Operation["FLASH"] ] {
    ... // handle FLASH messages
  }
}
```

Figure 4: A HydroJ approximation of code in Fig. 3. The single `process` method has been replaced by three handlers, which declaratively specify the message formats understood by this service.

However, this flexibility comes at great cost. Writing such code is tedious and error-prone. The programmer must expend extraordinary effort programming defensively: branches must be inserted to check that all message components are present, and to manually dispatch the message to behavior depending on the message content. To understand what message structure is expected, one must reason about complex imperative control flow. These problems would grow far worse in a service with more complex message dispatch needs.

The HydroJ code for `LightControl` is shown in Fig. 4. This code retains most of the flexibility of the prior code, and the explicit message disassembly and dispatching code has been eliminated.

On the other hand, the code in Fig. 4 does not preserve *all* the flexibility of the code in Fig. 3. In Fig. 4, the `#Location` and `#Operation` trees must be the first two children respectively of the `#Light` tree, whereas in Fig. 3 these children may occur anywhere in the child list of `#Light`. This observation gives us an opportunity to demonstrate one of the features of HydroJ. Suppose we alter the handlers to use bag patterns instead of list patterns (we show only the `ON` and `OFF` cases):

```
handler #Light( #Location[loc=String],
                #Operation["ON"] )    { ... }
handler #Light( #Location[loc=String],
                #Operation["OFF"] )   { ... }
```

Now, the `#Location` and `#Operation` tags will match anywhere in the child list of `#Light`. However, a problem arises. Recall that the children of a tagged tree value are an ordered list. The data in a child list are not guaranteed to be unique. What if more than one child matches `#Operation` — in particular, what if both `#Operation["ON"]` and `#Operation["OFF"]` are present? This message would match both handlers, and it is unclear which to invoke; and, in fact, HydroJ will flag the above handlers ambiguous on argument type:

```
#Light( #Location[String],
        #Operation["ON"], #Operation["OFF"] )
```

In HydroJ, the absence of this case is an ambiguity error. The programmer will be forced to write a handler to cover

```
public abstract class LocationMsg {}

public class MeasurementMsg extends LocationMsg {
  protected long timestamp;
  protected String objectData;
  protected String sensorName;
}

public class ObjectNameQuery extends LocationMsg {
  protected String typeName;
}
```

Figure 5: Java code for Location Stack message classes: The abstract class `LocationMessage`, and two subclasses.

this case — this handler might log an error and do nothing, or it might signal the sender and request that it disambiguate between the two requests. In any case, the programmer is forced to consider this case. By contrast, in the original code from Fig. 3, the service would silently execute whichever `#Operation` child came first, because of the semantics of the `getElement()` method. This ambiguity — probably a programming error — could have gone unnoticed indefinitely.

On the other hand, rather than requiring that the implementor provide the covering case above, one might prefer to require that clients never send both the `"ON"` and `"OFF"` operations in the same message. HydroJ's type system does not currently allow the user to express this restriction. One approach to providing this expressiveness would be to enable the implementor to state that some tag name must be unique in a child list. We discuss this further, as possible future work, in Section 5.

3.3 `FusionService` and Host Type Mapping

`FusionService` is part of the Location Stack abstraction layer, which integrates physical location information from multiple data sources [26]. We present it here because its Java implementation illustrates *dynamic host type mapping*, a library-based alternative to HydroJ's language constructs.

The Location Stack handles XML messages by using a generic XML-to-Java serialization library. Figs. 5 and 6 show the classes that implement `FusionService`. The code has been simplified for presentation purposes, but retains its essential character.[6] In Fig. 5, the program defines several simple classes that serve purely as message data — they have no methods, only fields. In Fig. 6, we show a fragment of `FusionService`, the main class responsible for integrating sensor data from various sources.

`FusionService` must deal with a wide array of sensor sources, often refining its behavior based on small differences between various kinds of messages. Unlike the `process` method of `LightControl`, `FusionService`'s `process` method does not use ad hoc calls on the `Message` object and its

[6]In addition to more general simplifications, we have translated this code, which was originally written in MultiJava, into plain Java. We expect that readers will be more familiar with the latter. MultiJava does not solve the problems with host type mapping that we discuss in this section, but it does allow the programmer to define message handlers in a more declarative fashion.

```
public class FusionService
  extends com.intel.research.rain.Service {
  public void process(Message m) {
    // Deserialize XML
    XMLReader reader = new XMLReader(m.getXML());
    Object msg = reader.readObject();

    // Process this message
    if (msg instanceof ObjectNameQuery) {
      ObjectNameQuery q = (ObjectNameQuery)msg;
      List results;
      if (q.typeName == null) {
        results = this.queryAll();
      } else {
        results = this.queryType(q.typeName);
      }
      ... // code to return results

    } else if (msg instanceof MeasurementMsg) {
      MeasurementMsg m = (MeasurementMsg)msg;
      ... // code to store this measurement

    } else {
      System.err.println("Unexpected: "+msg);
    }
  }
}
```

Figure 6: Java code, adapted from original Location Stack `FusionService`.

`Element` components to dispatch this message to its behavior. Instead, it passes the message to an `XMLReader` instance; the `XMLReader` class belongs to a third-party library that uses Java reflection to find an appropriate class instance, instantiate an object of that class, and initialize the object's fields based on the message data. In this approach, messages are mapped at runtime to locally available classes in the host language, which is why we call it dynamic host type mapping.

After deserializing the message object, `FusionService` uses `instanceof` tests and casts to dispatch various classes of received `LocationMessage` objects to some appropriate handler. Although the interface is still encoded in hand-coded control flow, the programmer no longer needs to reason manually about traversing the tree of message data. It is relatively simple to refine cases to handle more specialized data — if the programmer defines a subclass of `MeasurementMsg` named `TempMeasurementMsg`, for example, it is easy to add another `instanceof` test:

```
} else if (msg instanceof TempMeasurementMsg) {
  TempMeasurementMsg t = (TempMeasurementMsg)msg;
  ... // handle this message
```

3.3.1 *Drawbacks of dynamic host type mapping*

The safety and flexibility of dynamic host type mapping is somewhat deceptive. First, dynamic host type mapping does not really provide static safe typing — like any library, it throws a *runtime* error when it receives a value for which it can find no matching host type. HydroJ provides that, for those components written in HydroJ[7], messages will only be sent to services that understand them.

Second, host type mapping may reintroduce brittle dependencies on the host type system. In particular, for languages like Java, the type system employs by-name subtyping. The library must decide how to map this subtyping relationship into XML; fundamentally, it must make certain generic decisions on how to encode class and field names, and whether to encode inheritance links. For example, consider the following Java classes:

```
class Point { float x, y; }
class ColoredPoint extends Point { String color; }
```

Should the `ColoredPoint` class be encoded as a tree with tag `#Point`, or with `#ColoredPoint`? Whichever choice the library makes, some clients will be rendered incompatible. If the library chooses to use `#ColoredPoint` as the serialized tag name, then clients that possess only the `Point` class definition will be unable to interpret the data as an instance of a `Point` subtype. On the other hand, if the library chooses to use `#Point`, the resulting serialization of `ColoredPoint` will be incompatible with clients that define a `ColoredPoint` class directly:

```
class ColoredPoint { float x, y; Color c; }
```

In dynamic host type mapping, therefore, the system must either forego subtype extensibility entirely, or force all clients in the network to share the same by-name subtyping hierarchy. Either choice renders the system brittle in the face of evolution.

HydroJ addresses this problem by encoding all subtyping as structure, and exposing that encoding directly to the programmer. Because the programmer controls the encoding, there are no hidden decisions about encoding tag names; and because HydroJ supports structural subtyping, it does not force the programmer to use a new name in order to define a subtype (as Java does). The "natural" extensibility of semi-structured data is therefore preserved.

3.3.2 *Translation to HydroJ*

We give one possible translation of `FusionService` into HydroJ in Fig. 7. As in all our previous examples, the message handlers are specified declaratively and separately. The code also happens to be slightly more compact, although this is not the major benefit. The major benefit is that the HydroJ version remains unencumbered by ties to classes in the host (Java) type system — it is effectively as flexible as a manual disassembly approach.

One other notable change, relative to the previous version of `FusionService`, is that the `ObjectNameQuery` handler has been split two cases — the non-null branch of the imperative `null` test has been factored into a declarative specialization of an overriding handler case. If the `#typeName` tag is present, its case will be invoked and its contents will be used; if not, the less specific empty case will be invoked.

3.4 Message evolution in `SqlService`

A reader may wonder whether the flexibility of HydroJ is likely to yield any benefits in practice. In realistic systems,

[7]HydroJ services can freely interoperate with non-HydroJ Rain services through the Rain transport layer. However, sends from foreign language components will not benefit from HydroJ's static type checking.

```
public service class FusionService {
  handler #MeasurementMsg(
              #timestamp[ts=long],
              #objectData[data=String],
              #sensorName[name=String] ) {
    ... // code to store this measurement
  }

  handler #ObjectNameQuery() {
    List results = this.queryAll();
    ... // code to construct and send reply
  }

  handler #ObjectNameQuery(#typeName[name=String]) {
    List results = this.queryType(name);
    ... // code to construct and send reply
  }

  ... // other cases, as separate handlers
}
```

Figure 7: A HydroJ translation of Figs. 5 and 6. The `instanceof` tests and message classes and have been replaced by handlers and patterns.

do components' interfaces evolve to structural subtypes, i.e., by extending the data in subtrees or sublists? In order to investigate this question, we decided to examine the revision history of a real application, written independently of HydroJ. By examining the changes occurring "in the wild", we hoped to understand how this application's message formats evolved, and whether this evolution would be expressible in ways compatible with HydroJ's structural subtyping.

For this purpose, we chose Oasis [47], a distributed peer-to-peer database system developed at Intel Research Seattle.[8] The Oasis code was placed under version control (CVS) starting at an early stage of its development. We studied the version history of the `SqlService` class, which was the principal class in the implementation of Oasis's server. Among the Rain services written to date, `SqlService` seemed a promising example because it has a relatively large interface — the current version exposes roughly a dozen distinct actions to clients.

In the following subsection, we first describe in more detail the methodology we used to examine the evolution of `SqlService`. Next, in Section 3.4.2, we summarize the results and give some concrete examples extracted from the history of `SqlService`. Finally, in Section 3.4.3, we draw a few conclusions about the evolution we observed.

3.4.1 *Methodology*

Our approach was to inspect the evolution of `SqlService` in detail, reviewing the code before and after each revision to determine whether the message formats sent and received by `SqlService` changed between revisions. We examined both "argument types" (the handling of incoming messages) and "return types" (replies sent in response to requests).

[8]Although the second author of the present paper participated in the development of Oasis, Oasis was written in Java, and the other Oasis developers never had contact with HydroJ. We can reasonably assert that the Oasis code developed wholly independently of HydroJ.

Where message formats changed, we tried to characterize those changes by translating only the interface types into a HydroJ-like pseudocode (we did not use strict HydroJ types, for reasons we discuss below). Finally, we sorted the changes into several categories, which we describe in the next section.

This process presented several challenges. First,the code of `SqlService` uses the manual disassembly style of XML processing described in Section 3.2; its interface is defined implicitly by the branching in its control flow, which spans many functions. The message formats were undocumented, and even developers intimately familiar with the code could not necessarily remember the exact differences between any two arbitrary versions. Therefore, the "interface type" of `SqlService` had to be reverse-engineered by close inspection of the code itself. Upon examination, certain details of the implementation could have been either features or bugs: for example, some versions allowed clients to send a message with either a subtree tagged `#sql_internal` (for "internal" messages, used by the replication algorithm), or a subtree tagged `#sql_statement` (for "client" messages, e.g. database queries), or both. The last of these cases — a message with both subtrees — was never intended or required by the programmer. Therefore, the inspection process inherently required the exercise of some subjective judgment.

Second, `SqlService` uses XML *attributes*, a feature of full XML that is not currently supported by HydroJ's pattern language.[9] In full XML, a tag may include an optional set of name/value pairs, distinct from the regular child list; these are called attributes. The current `SqlService` code frequently encodes distinct requests using attributes when tag names would suffice just as well — for example, rather than using distinct tags `#sql_internal_acquire` and `#sql_internal_release`, the code uses a single tag `#sql_internal` with two distinct string values `"acquire"` and `"release"` for its `internal_type` attribute. A translation of `SqlService` to HydroJ would have used tag names instead of attributes. We have assumed this transformation in our study, and in the following presentation.

Finally, we note the following weakness in this experiment's input set: versions stored in a developer's repository do not necessarily reflect the character of versions that would coexist in a deployed system. Presumably, the former contains many more unstable revisions than the latter, particularly when one includes revisions occurring early in the project's development cycle. Additionally, the version control repository reflects phases of development — such as the initial "ramp-up" (during which the code may evolve more monotonically than in mature development) and the addition or removal of debugging instrumentation — that occur with less frequency in deployed systems.

3.4.2 Results and examples

In this subsection, we describe the evolution observed in `SqlService`. We first give an overview of the revision history; then we give specific examples of message type evolution extracted from the system.

At the time of this writing, `SqlService`'s revision history consists of 37 commits, made by 3 different developers and spanning more than 5 months, from 10 January 2003 to 18 June 2003. The earliest version of the `SqlService.java` source file comprises 164 lines, and the latest version comprises 1433 lines.[10]

We call changes *trivial* when they *only* alter whitespace, comments, local names (e.g., changing the package to which a class belongs), purely local debugging statements (e.g., adding `System.err.println` call), or other code that never affects `SqlService` itself. Out of 36 revisions (after the initial commit), 7 were trivial. For revisions containing non-trivial changes, the size of a Unix `diff -B` between subsequent versions ranges from 4 lines to 842 lines, with an arithmetic mean of about 112 altered lines per revision. Of course, `SqlService` calls methods in other classes (which also evolved, but are not included in these counts), so these numbers only approximate the relative size of changes in each revision.

We call a non-trivial change a *non-messaging change* when it changes the code in a way that affects observable behavior, but does not alter the format of messages seen by other nodes in the distributed system. For example, a non-messaging change might be a change to the Rain discovery advertisement, or a Java `synchronized` block inserted to fix deadlocks or race conditions. Of the 29 non-trivial revisions, we categorized 13 as containing only non-messaging changes.

This leaves 16 revisions in which the format of one or more messages changed; we call these *interface evolution* revisions. We discovered that, in all but 4 of these revisions, the changes consisted solely of evolution to structural subtypes. Space prevents us from describing each of these evolutionary steps here; instead, we summarize and give examples of a few interesting cases (more details appear in our technical report [32]). We group message format changes into three kinds: *additions* of new message types, *subtype evolution* of existing messages, and *non-subtype evolution* of existing messages. More than one kind of logical change may occur in a single source revision. We explain and give examples in turn.

Simplifying somewhat, the basic form of a `SqlService` message is a `#msg` root tag with a `#db[]` tag:

```
handler #msg(#db[db=String]) returns ... {
  ... // commands for database named by db
}
```

The other children of `#msg` specify what action should be performed on the named database. One common form of interface evolution consists of adding a new "type" of message — that is, the addition of an entirely new action that, conceptually, might correspond to a method in a standard object-oriented language. For example, the fifth version contained the first implementation of the weighted-voting replica management algorithm [47]; this entailed adding a series of new message types:[11]

```
handler #msg(#db[db=String], #internal_acquire[])
   returns ... { ... }
handler #msg(#db[db=String], #internal_update[])
   returns ... { ... }
... // etc.
```

[9]In the current HydroJ implementation, the user can access attribute values, but attributes are not part of the pattern language. This limitation is not fundamental; see Section 5.

[10]For all line counts in this section, we conservatively count a line of source code or diff result only if it contains a semicolon or an open curly brace.

[11]As previously noted, the original message formats use XML attributes, but the encoding is trivially isomorphic to the ones we describe here.

Nodes in the database use these new message types to send internal management messages to each other.[12]

Addition of wholly new message types, in this fashion, occurs in 8 out of the 16 interface evolution revisions. Aside from the implementation of the replica management algorithm, other additions include message types that control debugging and performance profiling, that enable clients to query lock status (previously, clients could only acquire or release locks), and that create or delete databases.

The large number of evolution steps of this kind argues for some support for service subtyping. Surely, for example, in a peer-to-peer distributed database, it seems desirable to be able to incrementally deploy nodes that support debugging messages, without updating the whole system; nodes that support such messages should simply be transparently substitutable for nodes that do not. Clients that never send these new message types will simply be unaffected by them.

However, the addition of new message types does not, by itself, necessarily argue for the flexibility of more general structural subtyping. To evaluate the need for that, we must examine message evolution. In 12 out of the 16 revisions where the interface evolved, existing message types evolved in some way that affected the format of their messages. Of these, 8 appear to be instances of solely *subtype evolution* — i.e., the addition of data to a subtree of an argument or return type. Of the remaining four versions, three contain only non-subtype changes, and one contains a mixture of both. In the remainder of this section, we describe several of the subtype cases in detail, as well as two of the non-subtype cases.

One simple instance of evolution occurred in the fourth version, when the return value from a query message added fields describing the number of rows returned and the name of the table from which they were extracted. In other words, the query handler's type evolved roughly from the first to the second of the following (for clarity, we elide most of the handler pattern):

```
handler ..., #queryCmd[String], ...
  returns #results(*#row[*column[]]) { ... }

handler ..., #queryCmd[String], ...
  returns #results(*#row[*column[]],
                   #rowcount[Integer],
                   #tablename[String]) { ... }
```

The new return type is a subtype of the old one.

A more interesting case of evolution occurred in the 10th commit, which added support for updating metadata (related to replica versioning). Metadata updates had the following format:

```
#metadatas[
  *#meta[#db[String], #version[Integer],
         ... /* data for replication protocol */]]
```

[12] Incidentally, had HydroJ been used, the handlers shown here would have been ambiguous — the user would have to provide for the case where both `#internal_acquire` and `#internal_update` tags appeared. This is an annoyance, but we claim that a HydroJ programmer would have used slightly different message formats that encoded the same information in an unambiguous manner. In any case, this fact is tangential to our aim in this section, which is to investigate how messages tend to evolve, not to demonstrate that HydroJ would have sufficed to encode exactly the message formats used in this application.

Metadata update requests could optionally be included with any query. Therefore, for example, each handler like the following:

```
handler #msg(#internal_acquire[], ...)
  returns ... { ... }
```

effectively evolved to the *pair* of handlers

```
handler #msg(#internal_acquire[], ...)
  returns ... { ... }

handler #msg(#internal_acquire[], ...,
                 #metadatas[*#meta[...]])
  returns ... { ... }
```

Once again, structural subtyping would render the latter pair of handlers compatible with the former single handler.

Incidentally, notice that, had the programmer not chosen to wrap the `*#meta[]` sequence inside a `#metadatas[]` tag, the single hander

```
handler #msg(#internal_acquire[], ...,
                 *#meta[...])
  returns ... { ... }
```

would have sufficed. This is an example of how, as previously mentioned, a programmer in HydroJ might have chosen to do things differently. On the other hand, one might prefer to express this idiom using an optional child pattern (having zero-or-one occurrences, rather than zero-or-more), as the original developers did. HydroJ does not currently support this feature, but it might be worth exploring; see Section 5.

The final subtype evolution example we will describe in this paper occurs in the 28th version. In this revision, the `#acquire_lock` message was augmented with an optimization called "query piggybacking", wherein the client could bundle an SQL query in the `#internal_acquire` message; the message evolved approximately from the first to the second of the following:

```
handler ..., #internal_acquire[], ...
 returns #reply(#lock_acquired[])
   { ... }

handler ..., #internal_acquire[],
             #piggysql[query=String], ...
 returns #reply(#lock_acquired[], #piggyres[...])
   { ... }
```

This modified message format indicates that the receiver should first acquire a lock, then execute the "piggybacked" query, then immediately release the lock and return the result. For this common usage pattern, query piggybacking reduces the network overhead to one round-trip. A client that did not understand query piggybacking could still use `#piggysql`-supporting services, oblivious to this extra functionality. Therefore, this evolution step would be supported by structural subtyping.

We conclude this section by briefly describing two non-subtype examples, in which `SqlService` evolved to a type that was not a structural subtype of its prior version.

A typical case is given by the 35nd revision, which removes an obsolete type of update database message (this type was superceded by other types, introduced in earlier revisions). The removal of a message type does not result in a subtype of the older type.

A different, and perhaps more interesting, example of non-subtype evolution occurs in the 32nd revision, which altered the format of messages that enable performance profiling. In version 31, the message enabling collection of performance data was a separate top-level option, roughly the following:

```
handler #msg(#perf_data[])
  returns ... { ... }
```

In version 32, this tag gets sunk down into a subtag of SQL messages:

```
handler #msg(#SqlRequest(#perf_data[]), ...)
  returns ... { ... }
```

In other words, performance monitoring is now only a semantically valid option for `#SqlRequest` messages, not for all messages handled by `SqlService`.

3.4.3 Evaluation

One might be surprised, as the authors were, that such a large fraction of the evolutionary steps in the development of `SqlService` were classifiable as structural subtype evolution.

Intuitively, we believe that this is partly due to the portion of the development cycle that we studied. Over the course of the 5 months surveyed, Oasis mostly *grew* in sophistication, acquiring improved protocols and debugging functionality. The developers were primarily "building up" to a certain level of functionality, rather than "tearing down" obsolete code or "renovating" existing code. On the other hand, even some of the evolution steps that occurred late in the development process — for example, query piggybacking — were instances of structural subtype evolution.

Evolution in software systems remains a thorny problem, and further empirical data would be needed to draw a firm conclusion. Overall, however, we tentatively conclude that some meaningful fraction of evolution "in the wild" does consist of structural subtype evolution.

3.5 Performance

Our current compilation strategy for message dispatch performs virtually no optimization. To get a sense of the overhead compared to hand-coded message dispatch (i.e., dispatch code hand-written in the manual disassembly style), we performed the following experiment.

First, we extracted a trace of 5,000 messages from a benchmark of the Oasis database. The messages are a representative sample of real messages sent by the Guide ubiquitous computing application [24]. The workload includes a variety of messages, including reads, writes, acquisition and release of locks, and metadata exchange. Next, in order to isolate the *dispatching* cost only, we manually extracted a subset of `SqlService`'s "dispatch skeleton" — those statements that only dispatch and disassemble incoming messages, as opposed to actually manipulating the database back-end. Then, we ported this skeleton to HydroJ, transforming the dispatching implicit in the control flow into HydroJ patterns. Finally, for both the Java and HydroJ versions, we constructed a test harness that timed the overhead of processing all messages in the trace 100 times each, for a total of 500,000 message dispatches. The test harness passed pre-constructed message objects directly to the classes' internal dispatcher methods, bypassing the Rain runtime system; therefore, the timings did not include any overhead from XML parsing or network communication (which would be suffered by any messaging substrate that used XML over HTTP, not just HydroJ).

On an otherwise idle 700 Mhz Pentium 3, using the Sun JDK 1.4.1, we found that the HydroJ dispatcher took 0.108 ms per message, whereas the hand-written dispatcher took 0.037 ms per message.

Previous experiments with Oasis [47], using the same message profile, have measured a mean overhead of 3.5 ms for sending a message one-way between two machines on the same segment of 100Mb/s switched Ethernet. Therefore, we judge that HydroJ's increased dispatch overhead is relatively insignificant. On the other hand, we believe that the dispatching overhead could be reduced significantly by optimization, if necessary.

4. RELATED WORK

In this section, we discuss four principal flavors of related work: programming language mechanisms for dispatch; XML processing systems; component-based systems for software reuse; and, finally, adaptive programming.

4.1 Pattern matching and dynamic dispatch

In the 1970's, pattern matching as a programming style was pioneered in the Prolog [11] and ML [41] languages. Around the same time, object-oriented dynamic dispatch was developed in Simula [15] and Smalltalk [23]. Various forms of pattern matching *without* static type checking have also seen extensive use in Lisp, scripting languages, and libraries.

Both object-oriented dispatch and pattern matching bind behavior to data based on types. HydroJ builds on past work in predicate dispatching [17] and Extensible ML [39], which unify pattern matching with object-oriented dispatch. HydroJ shares with these languages the key feature that data dispatches to the *most specific* matching handler; we view this as the essence of object-oriented dispatch. HydroJ differs from these languages in that it employs purely structural subtyping, and its data model is based on semi-structured XML data.

HydroJ's pattern language was inspired partly by XDuce [28, 27], which we discuss in more detail below.

4.2 XML processing

4.2.1 Schema languages

XML languages from the database community have focused primarily on the specific domains of queries [60] and schema definitions [59, 45] rather than general-purpose programming languages. There are some broad similarities between schema languages and HydroJ's type system, but most XML schema languages are more expressive and more complex to type check than HydroJ types. For example, the RELAX-NG schema language allows an "interleaving" type, in which elements of a child list must alternate sequentially between two different types. We believe that this level of expressiveness, although perhaps desirable for database schema, entails too much complexity for a programming language type system.

4.2.2 Libraries

Many libraries exist for XML processing. Some of these libraries are essentially generic programmatic interfaces for tree traversal, and are completely untyped. Use of these libraries to handle messages is verbose, awkward, and error-prone, as we showed in Section 3.2.

Other libraries provide a form of dynamic typing through schema — as an XML value is constructed, it is validated at runtime against an XML schema type. However, library approaches in general suffer from the problem that the programming language does not "understand" the types that are being constructed. Libraries cannot assure static type safety; they can only throw runtime errors when type safety is dynamically violated.

An alternative to explicit XML tree construction is an XML serialization library, which maps XML trees to values in the host language type system and vice versa. We called this the *dynamic host type mapping* approach, and discussed its drawbacks in Section 3.3.

A final library-based approach is to generate code for some general-purpose language automatically, perhaps based on a schema. For example, the Relaxer library [46] generates Java classes based on a RELAX-NG schema. We call this approach *static host type mapping*, because it maps an XML schema onto statically generated host type code. Static host type mapping regains static type checking for message construction — the static types in the generated code can be defined so as to prevent invalid tree construction statically. However, this approach is cumbersome — the programmer must separately generate code from the schema, and then program using that code. Furthermore, static host type mapping alone does not solve the problem of dispatching messages to behavior, as HydroJ does. Finally, static host type mapping does not allow a single XML value to be "viewed" at different types by different pieces of code — the static type of a given XML tree maps to exactly one host type in any given code base.

4.2.3 XML programming languages

Notable XML-related languages from the programming languages community include XDuce [28, 27], XMλ [34], `<bigwig>` [6], and CDuce [4]. Our technical contribution relative to these languages is the use of most-specific-match (rather than first-match) for message dispatch, and a type system that combines ordered and unordered tree children with repetition patterns.

The XDuce family of languages merit some further discussion here, as XDuce directly inspired some of our work, and some of its successors contain features comparable to those in HydroJ.

XDuce's type system and pattern language describe only ordered child lists. As a result, XDuce's type system translates into an elegant formalism based on tree automata [12], which gives it the full expressive power of regular tree languages [21]. However, the encoding does not extend naturally to unordered children of a tree.[13] On the other hand, XDuce's attribute-element constraint extension [27] augments XDuce's ordered *element* children with unordered *attribute* children. Since attributes and elements are distinct, but not orthogonal, mechanisms, attribute-element constraints do not subsume the functionality of our bag types, which allow *elements* to be viewed as unordered.

[13] A combination of union, wildcard, and difference types can be used to encode some features of our bag patterns, but the encoding is cumbersome and incomplete. The type representation takes factorial space, and certain restrictions cannot be encoded at all.

Xtatic [19] embeds the XDuce type system in the object-oriented language C#. Unlike HydroJ, Xtatic does not target distributed messaging, and retains some of XDuce's "functional" flavor. For example, pattern matching in Xtatic retains XDuce's "functional-style" dispatch, in which messages dispatch to the first matching case of a function rather than the most specific matching case. Xtatic currently exists as a prototype interpreter for a core language; its creators are pursuing several extensions to the language and its implementation, including interoperability with full C#.

CDuce [4], another language inspired by XDuce, is a general-purpose programming language with some features that superficially resemble features of HydroJ. For example, CDuce contains both a list type (for which child order matters) and a record type (whose child order does not matter). However, CDuce record and list types are stamped with their type at creation, and belong to disjoint kinds. By contrast, in HydroJ all semi-structured data are ordered (bag types are statically introduced by matching ordered data against a bag pattern, but dynamically the underlying data remains ordered) and ordered types subtype unordered types. Also, more fundamentally, CDuce is not object-oriented. Unlike HydroJ, CDuce does not support inheritance, and retains XDuce's "functional-style" dispatch semantics.

4.3 Component connection and reuse

Researchers and practitioners have proposed that "component-oriented" programming facilitates the construction of independently extensible software. The precise definition of "component" is widely debated, but the central feature of most component systems seems to be a mechanism for externally composing separately developed software abstractions.

For example, the key feature of the Piccola [1] composition language is its mapping of component composition to process composition in the π calculus [40]. In recent work by Mezini and Ostermann, components declare *collaboration interfaces*, which are linked together by an external *binding* mechanism [37]. ML's parameterized module system [33] uses module-level function application to compose modules. ArchJava [3] unifies architectural description with implementation; it augments Java with *component classes*, which have interfaces called *ports*, and provides a `connect` construct to bind ports together.

All these mechanisms are mostly orthogonal to HydroJ, even though they also address aspects of the independent extensibility problem. HydroJ addresses the question of *which* components are compatible — specifically, it tries to make more components compatible with each other through a novel form of interface flexibility. This issue is distinct from the issue of *how* compatible components should be connected, which is the central concern of component systems.

The one aspect of HydroJ that is not entirely orthogonal to component systems is the typed discovery service. HydroJ services connect to peers through discovery queries; discovery therefore serves as the intermediary that enables separately developed components to cooperate. Type-based discovery can be viewed as an application of component signature matching [61], although most past work on signature matching has focused on searching libraries of *static* source or object code, rather than *running* service instances. Christiansen et al.'s "type management" concept [10] operates over running instances, but its type language differs considerably from HydroJ's.

HydroJ's choice of connection mechanism does not rule out the use of other component connection mechanisms. In fact, HydroJ's type system might complement such mechanisms. For example, by using HydroJ patterns as the interface types of ports, HydroJ's flexibility could be brought to an ArchJava-like system. It seems possible to combine most other component connection mechanisms with HydroJ types in a similar fashion.

Component systems have seen widespread use in industrial practice; we mention only a few here. CORBA [13] is a widely used multi-language component system; CORBA interfaces employ a form of by-name subtyping, although some language mappings may complicate or obscure this subtyping. In Microsoft's Component Object Model (COM) [44, 48], components' interfaces are identified by a globally unique identifier (GUID), and hence COM cannot accommodate structural subtyping. Microsoft's more recent *assemblies* [35] component system also uses GUIDs, although programmers can specify explicitly that assemblies with different GUIDs may be compatible. HydroJ's subtype-based interfaces are therefore more flexible, with less programmer effort, than the interfaces in CORBA, COM, or assemblies.

4.4 Adaptive programming

Adaptive programming [36] and the HydroJ type system share a similar motivation: namely, to enable components to be written in a way that makes them relatively robust to changes in their peers. Adaptive programming, however, is considerably more radical than HydroJ: it insulates clients from changes to the structure of object's reference graph, by providing "traversal patterns" that mediate between clients and graphs of objects. The XPath [58] query language can be viewed as an XML analogue of adaptive programming patterns: it also decouples the client from the details of traversing and extracting parts of an XML value. HydroJ does not incorporate adaptive programming constructs, but combining semi-structured data with expressive traversal patterns at the programming language level might yield interesting results.

5. FUTURE WORK

We have noted, in passing, at least three situations in which one might desire a more expressive type system than HydroJ currently supports. First, HydroJ does not allow the programmer to specify that clients may not send more than one instance of a tag in a given list. The result is that, as mentioned in Section 3.2, implementors sometimes have to resolve burdensome ambiguities that might be better addressed by restrictions on the client.

Second, as mentioned in Section 3.4.1, HydroJ does not currently support XML attributes, which are name-value pairs that can be associated with tree nodes. XML attributes are highly non-orthogonal — they overlap in functionality with child tags; only atoms (not arbitrary subtrees) can be stored as attribute values; and only one occurrence of a given attribute name may occur for a tree. However, despite these aesthetic problems, practical factors indicate that one might like some principled way of specifying attributes in HydroJ patterns.

Third, as mentioned in Section 3.4.2, one would like to support some form of "zero-or-one occurrences" matching (in addition to the "zero-or-more occurrences" matching provided by star patterns). Optional subtrees appear to be a common idiom in existing semi-structured data.

All three of the above features seem expressible if the type system allowed finer control over cardinality of child list members and/or tags, a feature supported by some of the previous XML processing languages mentioned in Section 4.2. Optional children and uniquely named child tags seem to translate straightforwardly into constraints on cardinality. XML attributes could be mapped to (specially named) tags with exactly-one cardinality — for example, the constraint that tag has an attribute named `foo` could be mapped into a child pattern `#attr.foo[]` that may appear only once in a child list.

6. CONCLUSION

The evolvability and independent extensibility of software are among the most important problems in modern software engineering. The *brittle parameter problem* — wherein communicating components depend on inessential features of the data they use to communicate — is a basic obstacle to evolvability. As long as components are sensitive to inessential changes in their peers' interfaces, components can only be independently replaced or upgraded with great difficulty.

In the present work, we attack the brittle parameter problem through the use of *pattern matching* over *semi-structured* data types. Our central idea is that this combination of language features in the HydroJ language encourages programmers to express only the essential features of an interface, i.e. those features that the programmer actually uses. By doing so, we believe HydroJ broadens the opportunities for other dimensions of support for system evolvability.

We have studied a few systems that use semi-structured messaging, and implemented a compiler for the HydroJ language design. Our initial evaluation of the HydroJ language shows promise. As we write larger systems and maintain them over longer intervals, we hope to gain more experience with system evolution, and to explore the potential benefits of integrating our ideas with more flexible type systems and other forms of software evolution support.

7. ACKNOWLEDGMENTS

The authors would like to thank the following for their support, cooperation, and suggestions: Gaetano Borriello and Intel Research Seattle; Sorin Lerner, Todd Millstein, Matthai Philipose, and other members of the Cecil Research Group; and the anonymous reviewers for OOPSLA.

This research has been supported in part by DARPA contract N66001-99-2-8924 and NSF grant CCR-0073379.

8. REFERENCES

[1] F. Achermann, M. Lumpe, J.-G. Schneider, O. Nierstrasz. Piccola: A Small Composition Language. Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches, H. Bowman and J. Derrick, eds., pp. 403-426. Cambridge Univ. Press, 2001.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proc. 17th ACM Symp. on Operating Systems Principles*, pp. 186-201, Charleston SC, 1999.

[3] J. Aldrich, C. Chambers, D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proc. 24th Int'l Conf. on Software Engineering*, pp. 187-197, Orlando FL, May 2002.

[4] V. Bénzaken, G. Castagna, A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proc. ACM Int'l Conf. on Functional Programming*, 2003.

[5] A. D. Birrell, B. J. Nelson. Implementing Remote Procedure Calls. ACM Trans. on Computer Systems, 2(1):39-59, Feb. 1984.

[6] C. Brabrand, A. Møller, M. I. Schwartzbach. The `<bigwig>` project. In *ACM Trans. on Internet Technology*, 2002.

[7] D. Box et al. Simple Object Access Protocol (SOAP) 1.1. W3C Note, 8 May 2000. `http://www.w3.org/TR/SOAP/`.

[8] C. Chambers, G. T. Leavens. Typechecking and Modules for Multimethods. ACM Transactions on Programming Languages and Systems 17(6):805-843, Nov. 1995.

[9] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 130-145, Oct. 15-19, 2000.

[10] B. O. Christiansen, M. Münke, K. Müller-Jones, W. Lamersdorf. Type Management: A Key to Software Reuse in Open Distributed Systems. In *Proc. First Int'l Enterprise Distributed Object Computing Workshop*, Gold Coast, Australia, Oct. 1997.

[11] A. Colmerauer. An introduction to Prolog III. Communications of the ACM 33(7):69-90, July 1990.

[12] H. Common, M. Dauchet, Rémy Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. Tree Automata Techniques and Applications. Draft manuscript, Aug. 2003. `http://www.grappa.univ-lille3.fr/tata/`

[13] The Common Object Request Broker: Architecture and Specification, Version 3.0. Object Management Group, Inc. July 2002. `http://www.omg.org/cgi-bin/doc?formal/02-06-01`

[14] A. Carzinaga, D. S. Rosenblum, A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Trans. on Computer Systems, 19(3):332-383, Aug. 2001.

[15] O.-J. Dahl, K. Nygaard. SIMULA — an ALGOL-Based Simulation Language. Communications of the ACM 9(9):671-678, Sept. 1966.

[16] R. Deline. A Catalog of Techniques for Resolving Packaging Mismatch. In *Proc. 1999 Symp. on Software Reusability*, pp. 44-53, Los Angeles CA, May 1999.

[17] M. Ernst, C. Kaplan, C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *Proc. 12th European Conf. on Object-Oriented Programming*, pp. 186-211, Brussels, Belgium, 1998.

[18] C. Flanagan, M. Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proc. 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp. 209-220, Jan. 1995.

[19] V. Gapeyev, B. C. Pierce. Regular Object Types. In *Proc. 17th European Conf. on Object-Oriented Programming*, LNCS 2743, Darmstadt, Germany, July 2003.

[20] D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. IEEE Software, 12(6):17-26, Nov. 1995.

[21] F. Gécseg, M. Steinby. Tree Languages. Handbook of Formal Languages, 3:1-68. G. Rozenberg and A. Salomaa, eds. Springer-Verlag, 1997.

[22] D. Gelernter. Generative Communication in Linda. ACM Trans. on Programming Languages and Systems, 7(1):80-112, 1985.

[23] A. Goldberg, D. Robson. Smalltalk-80: The Language. Addison-Wesley, Jan. 1989.

[24] Guide: Understanding Daily Life via Auto-Identification and Statistics. Project web site. `http://seattleweb.intel-research.net/projects/guide/`

[25] M. Hicks, J. T. Moore, S. Nettles. Dynamic Software Updating. In *Proc. ACM SIGPLAN'01 Conf. on Programming Language Design and Implementation*, pp. 13-23, Snowbird UT, 2001.

[26] J. Hightower, B. Brumitt, G. Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *Proc. 4th IEEE Workshop on Mobile Computing Systems & Applications*, pp. 22-28, Callicon NY, June 2002.

[27] H. Hosoya, M. Murata. Validation and Boolean operations for Attribute-Element Constraints. In *Informal Proc. of Workshop on Programming Language Technologies for XML: PLAN-X 2002*, pp. 1-10. Pittsburgh PA, Oct. 3, 2002.

[28] H. Hosoya, B. Pierce. Regular Expression Pattern Matching for XML. In *Proc. 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp 67-80, London, United Kingdom, 2001.

[29] Hypertext Transfer Protocol — HTTP/1.1. IETF RFC 2616. June 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616.txt`

[30] J.-H. Kang, M. Philipose, G. Borriello. River: An Infrastructure for Context Dependent, Reactive Communication Primitives. In *5th IEEE Workshop on Mobile Computing Systems & Applications*, Monterey CA, Oct. 9-10 2003.

[31] A. LaMarca, D. Koizumi, M. Lease, S. Sigurdsson, G. Borriello, W. Brunette, K. Sikorski, D. Fox. PlantCare: An Investigation in Practical Ubiquitous Systems. Intel Research, IRS-TR-02-007, Jul. 2002.

[32] K. Lee, A. LaMarca, C. Chambers. HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems. Technical Report UW-CSE-03-08-01, University of Washington, 2003. Forthcoming. `http://www.cs.washington.edu/research/projects/cecil/pubs/hydroj.html`

[33] D. MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symp. on Lisp and Functional Programming*, pp. 198-207, Austin TX, 1984.

[34] E. Meijer, M. Shields. XMLambda: A Functional Programming Language for Constructing and Manipulating XML Documents. Drafr manuscript, 2002. http://www.cse.ogi.edu/~mbs/pub/xmlambda

[35] E. Meijer, C. Szyperski. Overcoming Independent Extensibility Challenges. Communications of the ACM, 45(10):41-44, Oct. 2002.

[36] M. Mezini, K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 97-116, Vancouver, Canada, 1998.

[37] M. Mezini, K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proc. 17th ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Seattle WA, Nov. 2002.

[38] Microsoft Corporation. .NET Framework Developer's Guide: XML and SOAP Serialization. http://msdn.microsoft.com/library/en-us/cpguide/html/cpconserialization.asp

[39] T. Millstein, C. Bleckner, C. Chambers. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. In *Proc. Seventh ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 110-122, Pittsburgh PA, Oct. 2002.

[40] R. Milner. The Polyadic π-Calculus: A Tutorial. Logic of Algebra and Specification, F. L. Buaer, W. Brauer, H. Schwichtenberg, ed., pp. 203-246, Springer-Verlag, 1993.

[41] R. Milner, M. Tofte, R. Harper, D. MacQueen. The Definition of Standard ML (Revised). Cambridge: MIT Press, May 1997.

[42] P. Muckelbauer, V. F. Russo. Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems. In *Proc. 1995 USENIX Conf. on Object-Oriented Technologies*, Monterey CA, June 1995.

[43] N. Nystrom, M. R. Clarkson, A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12th Int'l Conf. on Compiler Construction*, Warsaw, Poland, April 2003.

[44] R. Pucella. Towards a Formalization for COM Part I: The Primitive Calculus. In *Proc. 17th ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Seattle, WA, USA, Nov. 2002.

[45] RELAX NG Specification. Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/committees/relax-ng/

[46] Relaxer home page. http://www.relaxer.org/

[47] M. Rodrig, A. LaMarca. Decentralized Weighted Voting for P2P Data Management. In *3rd Int'l ACM Workshop on Data Engineering for Wireless and Mobile Access*, San Diego, CA, USA, Sept. 2003.

[48] D. Rogerson. Inside COM. Redmond, WA: Microsoft Press, 1997.

[49] R. Srinivasan. XDR: External Data Representation Standard. IETF RFC 1832. Sun Microsystems, Inc., August 1995. http://www.ietf.org/rfc/rfc1832

[50] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 149-160, San Diego CA, 2001.

[51] Sun Microsystems, Inc. Remote Method Invocation web site. http://java.sun.com/products/jdk/rmi

[52] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pp. 204-215, Madison, WI, 2002.

[53] R. J. Walker, G. C. Murphy. Implicit Context: Easing Software Evolution and Reuse. In *Proc. 8th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pp. 69-78, San Diego CA, 2000.

[54] M. Weiser. The Computer for the 21st Century. Scientific American, 265(3):94-104, Sept. 1991.

[55] XML-RPC Home Page. http://www.xmlrpc.com.

[56] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Second Edition). T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, eds. W3C Recommendation. 6 October 2000. http://www.w3.org/TR/REC-xml

[57] World Wide Web Consortium (W3C). Web Services Architecture. W3C Working Draft, August 2003. http://www.w3.org/TR/ws-arch

[58] World Wide Web Consortium (W3C). XML Path Language (XPath), Version 1.0. J. Clark and S. DeRose, eds. W3C Recommendation. Nov. 1999. http://www.w3.org/TR/xpath

[59] World Wide Web Consortium (W3C). XML Schema. W3C Recommendation. May 2001. http://www.w3c.org/XML/Schema

[60] World Wide Web Consortium (W3C). XQuery: A Query Language for XML. W3C Working Draft, June 2001. http://www.w3.org/TR/xquery

[61] A. M. Zaremski, J. M. Wing. Signature Matching: A Tool for Reusing Software Libraries. ACM Trans. on Software Engineering and Methodology, 4(2):146-170, Apr. 1995.