A Delta-Driven Execution Model for Semantic Computing

Roly Perera
Dynamic Aspects
14 King Square
Bristol BS2 8JJ
United Kingdom
+44 (0)117 924 8915
roly.perera@dynamicaspects.com

Jeff Foster
Dynamic Aspects
14 King Square
Bristol BS2 8JJ
United Kingdom
+44 (0)117 924 8915
jeff.foster@dynamicaspects.com

György Koch
Dynamic Aspects
14 King Square
Bristol BS2 8JJ
United Kingdom
+44 (0)117 924 8915
george.koch@dynamicaspects.com

ABSTRACT
We describe (and demonstrate) the execution model of a computing platform where computation is both *incremental* and *data-driven*. We call such an approach *delta-driven*. The platform is intended as a delivery vehicle for semantically integrated software, and thus lends itself to the semantic web, domain-driven development, and next-generation software development environments. Execution is transparent, versioned, and persistent. This technology - still at an early stage - is called **domain/object**.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *interactive environments*; D.2.11 [**Software Engineering**]: Software Architectures – d*ata abstraction, patterns*; D.2.12 [**Software Engineering**]: Interoperability – *data mapping*; D.3.2 [**Programming Languages**]: Language Classifications - *data-flow languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *constraints*; D.3.4 [**Programming Languages**]: Processors – *code generation, incremental compilers, run-time environments*.

General Terms: Design, Languages

Keywords: Delta-driven execution, incremental computation, adaptive functions, lazy memoization, relational programming

1. OVERVIEW
Domain/object is a functional, persistent, versioned, transparent, incremental, reactive execution environment embedded in Java. Data values are "live" and update automatically in response to changes in dependent values, like cells in a spreadsheet. The design philosophy places a premium on simplicity and emphasises elegance over featurism. Our goal is to enable a new breed of application where components are integrated in a way only possible today with the help of large amounts of manual boilerplate. Suitable applications include domain-driven development, the semantic web and next-generation software development tools.

The execution model of domain/object is a radical departure from that of most programming languages and virtual machines in use today, in that execution takes place solely by the propagation and interpretation of structural deltas. When a data value changes, the effects of that change are propagated recursively to all dependent data values, meaning that data values are "live" [32], not snapshots like variables are in traditional programming languages. Domain/object is thus a kind of *dataflow* language, and abstractly more similar to a spreadsheet than a standard programming language.

When changes propagate across domains, the change propagation process plays the role of an *interpreter* in the traditional computer science sense, incrementally translating changes in the "source" language as they are received into changes in the "target" language. Domain/object is therefore ideal for hosting interactive applications whose architectures are best considered to be sets of interconnected domain models.

The plan of this paper is as follows. In section 2 we briefly describe the background and motivation for our research. Section 3 describes the broad principles which underly the platform, such as the unification of compile-time and run-time, and key behavioural characteristics such as incrementality and "liveness". We also discuss in detail various design decisions and their relevant context. Section 4 presents a short case study of a "domain-driven" implementation of Java called domain/j, which was the primary motivation for the development of domain/object. Section 5 summarises some of our experiences of embedding a functional dataflow language in Java. In section 6, we attempt to summarise the large body of related work. Finally, section 7 closes with some thoughts on the many exciting possibilities for future research.

2. BACKGROUND AND MOTIVATION
Domain/object began life as a core infrastructural component of a new Java programming environment called **domain/j** which follows in the footsteps of research projects such as Harmonia [17], the Mjølner system [22] and Self [33]. Domain/j is a language-centric implementation of Java where the user interacts directly with the Java language itself rather than a separate "IDE". As such it relies heavily on the availability of various views or models of the user's program, including a "physical" or syntactic view, a more abstract "logical" view (sometimes, somewhat confusingly, known as a "semantic" model), dataflow and control-flow models of the program, and so on. These views must be **live**, in other words kept up-to-date as the user edits code; **incremental**, so the user does not have to continually rebuild or refresh; **bidirectional**, in that the user can in principle make changes at any level of description (e.g. textually or semantically) and have the models remain synchronised; and **transactional**, so that changes can be safely rolled back if a failure occurs deeply nested within a compound edit and so that flexible granularities of view synchronisation can be obtained.

Designing an application with these traits using existing programming languages involves the development of a considerable amount of infrastructure for managing change discovery and notification. Moreover, one might interpret many areas of growing interest in the field of software development – including domain-driven development [14], aspect-oriented programming [21], intentional programming [30] and generative programming [6] – as

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

to some extent facets of an emerging and compelling paradigm where any application can be thought of as a set of intrinsically connected models or views.

We refer to this broad emerging paradigm as *semantic computing* because it emphasises the fact that computation is about **semantic interpretation** [29] – the translation of one domain into another. Truly semantic computing requires a much tighter connection between software components, such that merely to express (declaratively) the relationship between *x* and *y* is to ensure that *x* is automatically re-synchronised as *y* changes.

3. KEY PRINCIPLES OF DOMAIN/OBJECT

3.1 Liveness

Liveness is the property by which data values always appear up-to-date and is the essence of the "connectedness" required for true semantic computing. Liveness eliminates in quite a profound way any distinction between compile-time, in the traditional sense, and runtime. For example all test assertions are "live"; a test fails as soon as the programmer makes a change which violates the assertion. It also dispenses with the notion of *control flow*, leaving *dataflow* as the only mechanism by which work gets done. Continuing with the testing example, tests do not need to be written in the "make a change; make an assertion; make a change; make an assertion" style common with Java or C#. Test assertions must hold generally, not just when "control" flows over them. This is a significantly simplified programming model for reasons argued for in detail by other authors [10].

3.1.2 Programs as mutable queries

In simple terms the domain/object execution model can be described as the incremental modification of structure interwoven with the incremental reactive change of any dependent structure. To understand the incremental execution model, we first need to consider the structure of the domain/object "universe".

Domain/object represents all data and programs ultimately as **functions**. A function is a left-univalent relation, a set of ordered pairs with unique left-hand sides. Certain species of function which satisfy particular properties are of significance in domain/object. For example a *tree* is a function where there is exactly one path between any two objects. There is a tree-like function called *contains* (or where grammatically convenient, *containment*), and we define a *domain* to be any maximal set of objects which is closed under containment.

Functions, however are not mutable but rather are static, purely extensional entities. What we traditionally think of as a program is in domain/object a live projection of this extensional structure, consisting of a graph of mutable references each of which points to an underlying constant or function application (see Figure 1 below). A reference represents the "selection" – either explicitly, via the user, or implicitly, via the reactive propagation of updates to the program – of a particular argument for some function *f*. Mutating the reference selects which part of *f*'s extension is being "looked at" by the reference. It is quite accurate therefore to describe the program as a mutable query of the underlying extensional structure, and computation as the synchronisation of the output of the query in response to changes to the inputs to the query. To interact with a domain/object program – either as a programmer or an end-user – is simply to mutate one or more of the references in the graph and observe the propagation of changes to the graph of references in response to that change. The approach is reactive [8] in that the system is passive in the absence of external changes and responds to any such changes by propagating the changes to all dependent values.

References roughly subsume the roles of *expressions* and *variables* in traditional programming languages. The main difference is that they are live, rather than snapshots taken at a particular point in the control flow, like traditional procedural variables. Although references are somewhat like variables, there is nothing that corresponds directly to the traditional notion of assignment. Instead, one relates references to other references by connecting them together into applications of functions. These applicative structures are the structures through which data flows, responding to deltas on their "source", or input, references by generating the appropriate deltas in their "target" references, or outputs. An applied function is either *primitive*, in which case its workings are completely hidden, or *aggregate*, in which case its workings are transparent, the functions being realised by a graph of internal references mediating between the argument references and the output references. Functions return values by writing to an output reference.

Since there is no way to mutate an object other than by applying a delta to it, there need be no separate difference discovery and notification mechanism, whereby modified objects inspect their state in response to external requests and fire a description of the resulting change to all dependent objects. Instead the same delta serves (in the future tense) as a request for change and (in the past tense) as a description of the resulting change. The Observer pattern [11] thus comes "built in".

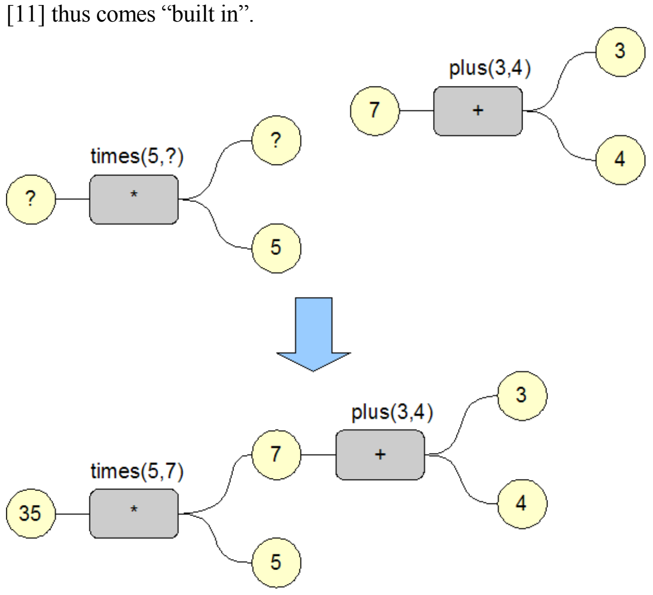

Figure 1: Application as the replacement of an unbound reference by another reference.

Finally, we should note that domain/object structures are *algebraic*, in other words uniquely built out of applications of primitive constructors. Effectively, an object is nothing more than the canonical delta that builds it from the null object. Since everything that happens in domain/object happens ultimately by way of the translation of one delta sequence into another, this simple algebraic notion of object is sufficient for all purposes.

The algebraic approach unifies construction with discovery, merging the Composite and Visitor patterns [11] into one. If two non-empty objects o_1 and o_2 are connected semantically for the first time, it is as though o_2 were empty at the point of connection and then immediately populated with its sub-structure. Equivalently we can say that o_1 and o_2 were always connected but that the actions describing the construction of o_2 were batched up and published in one go rather than propagated incrementally.

3.1.3 Application and partial application

To apply a domain object function to an argument reference is to copy its prototypical definition into the calling context, splicing in the argument reference in place of one of the unbound inputs of the function. This is shown in bottom half of Figure 1 above, where the function *times()* is shown applied to to *5* and *plus(3,4)*.

The question marks in Figure 1 indicate a reference whose referent is the primitive constant *indeterminate.* This represents either an unbound or an unknown input, or an output whose value cannot be determined (say because insufficient inputs are determinate). Indeterminacy of references is a general enough concept in domain/object to serves a number of purposes, including support for non-strictness, trivalent logic, and partial application.

Partial application is intrinsically supported because all functions are curried. The iterated application of an *n*-ary function *f()* to *m* arguments (where $m \leq n$) yields an (*n*-*m*)-ary function. (The partial application *times(5)* appears in the top half of Figure 1.) Under our prototype-based model of application (where to apply is simply to bind an unbound input), composition is just a special case of partial application. Composing *f()* and *g()* is the same as applying *f()* to *g()*; in both cases, the output of *g()* is passed as the input to *f()*. Composition is just a name we might give to application when *g()* has unbound inputs, as Figure 2 below shows:

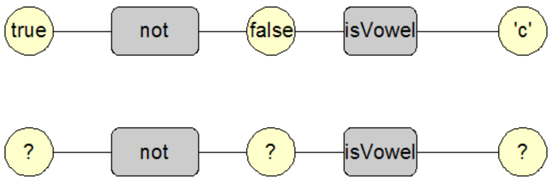

Figure 2: Application as composition

In the figure, the upper graph corresponds to the application of *not()* to the expression *isVowel(character('c'))*, and the lower graph corresponds to the composition of *not()* with the function *isVowel()*.

As a convenience for embedding domain/object code in Java, the constant *unbound*, which is an identity element of *apply()* for any function (*apply(f,unbound) = f* for any *f*), can be used to carry out a positional form of partial application. For example the fact that *lessThan()* is not commutative might lead to the following programming error:

```
lessThanFive = lessThan(number(5));
assert_(lessThanFive(number(3))); // fails!
```

which could then be remedied as follows:

```
lessThanFive = lessThan(unbound(),number(5));
assert_(lessThanFive(number(3)); // OK
```

The use of *unbound* allows *number(5)* to be bound to the second rather than the first argument of *lessThan()*. (Primitive types such as integers and characters are denoted via an equivalent primitive function, e.g. *number()* or *character()*. Sometimes these functions are omitted from examples for brevity.)

3.1.4 Recursion and non-strictness

Since to call a domain/object function by name is to copy its body into the calling context, a function which directly or indirectly calls itself induces an infinitely deep structure, at least conceptually. (In this respect domain/object differs from dataflow languages which model recursion with *cyclic* structures, distinguishing invocation context by associating a suitable label with each dataflow token, e.g. [34], [15]).

We avoid instantiating an infinite structure by taking the approach of Subtext [10] and ignoring infinitely deep parts of the structure which are contained within dead branches of conditionals. This emulates the behaviour of non-strict lazy functional languages when a non-terminating but unused argument is passed to an otherwise terminating function, and is achieved by making the actual copying of the function body demand-driven (lazy). When a calling function *f()* connects to the output of another function *g()*, as in *f(g())*, the body of *g()* is not actually spliced into *f()* unless *f()* actually requires the value of *g()*. Certain primitives such as *and()*, *or()* and *if()* are non-strict in their arguments and will ignore arguments whose values they are not concerned with. *if()* for example only queries the value of *either* the "then" branch or the "else" branch, leaving the other branch effectively dead.

3.2 Incrementality

Incrementality is another fundamental characteristic of the domain/object system. Incrementality makes domain/object functions very flexible and naturally suited to interactive applications such as user interfaces, document editors and programming environments. In some cases, the incremental computation is actually faster overall than batch computation, because it can take advantage of intermediate results, particularly with problems that are amenable to the "strength reduction"-type optimisation common in optimising compilers [25]. Full incrementality is clearly slower than batch computation for certain other kinds of problem, however; an area for future investigation is allowing non-incremental optimisations to be added transparently to a domain/object application.

Incrementality has two components: **memoisation** and **adaptivity**. *Memoisation* [27] is the caching of the value of a function for a given argument, and trades storage overhead and lookup time for compute time. The fact that certain domain/object primitives such as *if()* are non-strict in their arguments means that our memoisation scheme must not force arguments to be evaluated; work on this aspect of the memoisation design has only just begun and will probably be based on Hughes' work on lazy memoisation [18]. *Adaptivity* [1] is a strategy which ensures that only dependent data values are re-evaluated when change occurs and trades the cost of maintaining explicit dependency information for the benefit of avoiding unnecessary computation. Domain/object functions are inherently both adaptive and memoised. ([2] discusses how these features can be combined non-orthogonally.)

3.2.1 Memoisation

Memoisation is a technique that retains previously calculated values so that they can be used again without having to be recomputed. Memoisation takes place the first time a function is applied to a given argument. If a function has already been instantiated with the supplied inputs (at any version of the system), the output value of the existing invocation is used to set the output reference. Otherwise, the body is instantiated as normal and used to determine the value of the output reference for the given inputs, and the result memoised. The memoisation table is persisted between sessions, which means that once any value is calculated, it never has to be calculated again, modulo any limits on offline storage capacity. The abstraction is that computation is lazy discovery of the extensions of functions.

3.2.2 Adaptivity

Adaptivity is fundamental to the domain/object design since evaluation can only occur in one of two says:

- in a data-driven fashion, in response to a dependee value changing and causing a dependent value to resynchronise (and if a delta is in turn implied in the dependent value then *its* dependents will be recursively re-evaluated, and so on);
- in a demand-driven fashion, when a dependent data value is connected for the first time (or more generally, reconnected after a period of disconnection), which is semantically equivalent to a composition of data-driven updates (this is described in more detail in section 3.3)

Compound functions derive their incrementality from the primitive functions of which they are ultimately comprised, and therefore do not themselves need to be written in an explicitly incremental style.

3.3 Transactions

Real-world applications rarely operate in a closed universe. Users add new queries or connect new client applications. Developers add new tests and new application code. Even when the scope of the dependency closure is known in advance, it is typically too large to require it to be instantiated entirety up front and part of the "live" reactive structure of the application. It is critical therefore to support the dynamic addition of new dependent references.

What this means is that it is legal for a reference *f* to be instantiated even though one of its *dependent* references *g* (for example an argument reference of a caller-to-be) has not yet been instantiated. This violates the incremental dataflow model, where changes propagate as soon as they are available, since changes to *f* cannot be seen by *g* if *g* does not exist. Instead, we allow *g* to be instantiated at a later time and only at that point receive all the changes that have happened to *f* since it was created. We call this process of "catching up" with the state of a dependee reference **synchronisation**. Support for reference divergence and synchonisation is an essential scalability feature of the platform.

With fully versioned references, this pattern of "just-in-time synchronisation" for dependent references generalises to a transactional model of change propagation with much broader utility. Under this more general model, we allow dependent references to disconnect for arbitrary periods of time. Reconnection with a dependee *f* causes synchronisation with respect to the net delta in *f* since the references were last connected, with connection for the first time becoming just a special case of this. Since different dependent references may have been disconnected at different points in the history of *f*, this facility requires read access to all previous versions of *f*, i.e. requires *f* to be at least *partially persistent* in the sense of [2]. Domain/object references are actually *fully persistent* in the sense of [7] and therefore are *a fortiori* capable of supporting this paradigm.

Any business process, collaboration model or interaction model which requires independent activity with well-defined synchronisation points - such as shared document editors, team-based software development environments and incremental parsers as well as more traditional "business" applications such as an online shopping - could be take advantage of this very general scheme. The approach allows an application to choose any point on the spectrum between full incrementality and explicit batch-mode synchronisation and indeed to combine the two arbitrarily. The self-similarity of the architecture means that the same transactional mechanism could apply equally well to fine-grained concurrency problems as to the larger-scale business-related examples just mentioned. This is an important area for future research.

3.4 Bidirectionality

Bidirectionality – the ability to modify the output of a function and have its inputs adjust accordingly, as well as the other way around – is central to the ultimate vision for the platform and also critical to its current intended use. Since bidirectionality, which is probably best understood more generally as a requirement to support non-deterministic computation, is fraught with both theoretical and practical issues, and intersects with many interesting areas of computer science research, it is likely to remain an active area of investigation for many years. For now our goal is to enable bidirectionality to an extent sufficient for our current purposes without the introduction of excessive ad hocery. See section 7 for some thoughts on future directions.

A simple example of bidirectionality is the *filter()* function which selects only those elements of a list which satisfy a given predicate. The **result** of a *filter()* can also be modified directly, causing its argument to be modified in such a way that the definition of *filter()* is satisfied.

When two references are connected via a mapping which does not support bidirectionality, the target reference is effectively immutable. That is to say, an attempt may be made to apply a primitive action to that reference but the action will be rejected by the mapping function. The action cannot be applied not because the reference itself is immutable, but because the action cannot be interpreted in terms of the source domain. Since one broken link can undermine the bidirectionality of the containing expression, our current support requires all primitive functions to define a sensible backwards mapping. The general heuristic for backwards mapping is "minimal change". For example, for the backwards mapping for the boolean functions *and()* and *or()*, we make a minimal number of modifications to the inputs such that the desired output is obtained, such as only setting one input to *false* if that is sufficient.

A compound function's inverse is simply the composition of the inverse of its constituent functions. It should be obvious that even a small amount of local indeterminacy rapidly amplifies into the realms of the intractible on a global scale. There are a number of complementary strategies we can use to manage this problem:

- lazy generation of alternatives (alternatives only materialised when they are explored)
- support for hand-coded inverses for specific functions
- simple general-purpose heuristics
- allow user interaction, when possible
- prefer paths the user has visited previously

To understand how the backwards mapping of a compound function might work, consider a simple example, the function *size*() for lists, whose body is:

```
if_(
    empty(list),
    integer(0),
    add(integer(1), size(tail(list)))
);
```

Assume that the input list contains a single element, and that we want to set the size to zero, as in:

```
Reference list = cons(character('a'), nil());
Reference size = size(list);

assert_(identical(size, integer(1)));
set(size, integer(0));
assert_(empty(list));
```

A suitable interpretation of setting the output of *size()* to zero is that we are changing the output of the *if()* function from the "else" branch to the "then" branch. A straightforward inverse mapping for this is to modify the condition of the *if()* from *false* to *true*. This in turn requires the backwards mapping for the *empty()* function. The definition of *empty()* is simply:

```
identical(list, nil());
```

So to make *empty()* true, we must change the output of *identical()* from *false* to *true*. This inverse mapping can be relatively easily achieved by simply setting the *list* argument to *nil()*. (Setting *identical()* from *true* to *false*, by contrast, might be achieved by setting one of the inputs to *indeterminate*.)

This example, although simple, suggests that bidirectional mappings are perhaps achievable in practice with a well-chosen combination of techniques, such as brute-force search and heuristic pruning, while remaining intractable in a more theoretical sense.

One reason to take the bidirectionality requirement seriously in the longer term is that today's applications are rich in behavioural redundancies which are rarely noticed, let alone discussed, mainly because they are obscured by complex notification schemes and irregular language mechanisms. For example any application which allows one object which is derived from another to be mutated in such a way that changes eventually end up being "written back" to the dependee object - a pattern common in user interfaces and database applications - must generally duplicate some logic in order to make the mapping bidirectional. And similarly, any application which allows structures to be deconstructed as well as constructed must contain redundancy to the extent that deconstruction is just construction running in reverse. Domain/object's highly systematised approach makes these redundancies much more explicit and ultimately more amenable to elimination through automation.

4. CASE STUDY: A "DOMAIN-DRIVEN" DEVELOPMENT TOOL

Figure 3 below gives the overall flavour of the incremental approach in the context of domain/j, a Java development tool where the user interacts directly with a number of interconnected domain models representing aspects of the Java language. Two distinct domain models are shown, **Java** and **JavaPhysical**. Java is the abstract, "logical" view of Java (consisting of entities such as packages, types, methods, and so forth), whereas JavaPhysical is the more concrete syntactic view, consisting of characters, syntax nodes such as class and method declarations, source folders, and so forth. Each is a separate domain in that it is closed under containment.

The diagram shows what happens as change occurring in the JavaPhysical domain causes incremental update of the Java domain. The JavaPhysical structure on the left shows part of a Java class declaration, where each yellow box represents a Java syntax object and each grey box represents the value of *contains()* for that object. The Java structure on the right shows part of a Java class along with some of the functions it participates in, including *implements()*, which is the set of Java interfaces it implements.

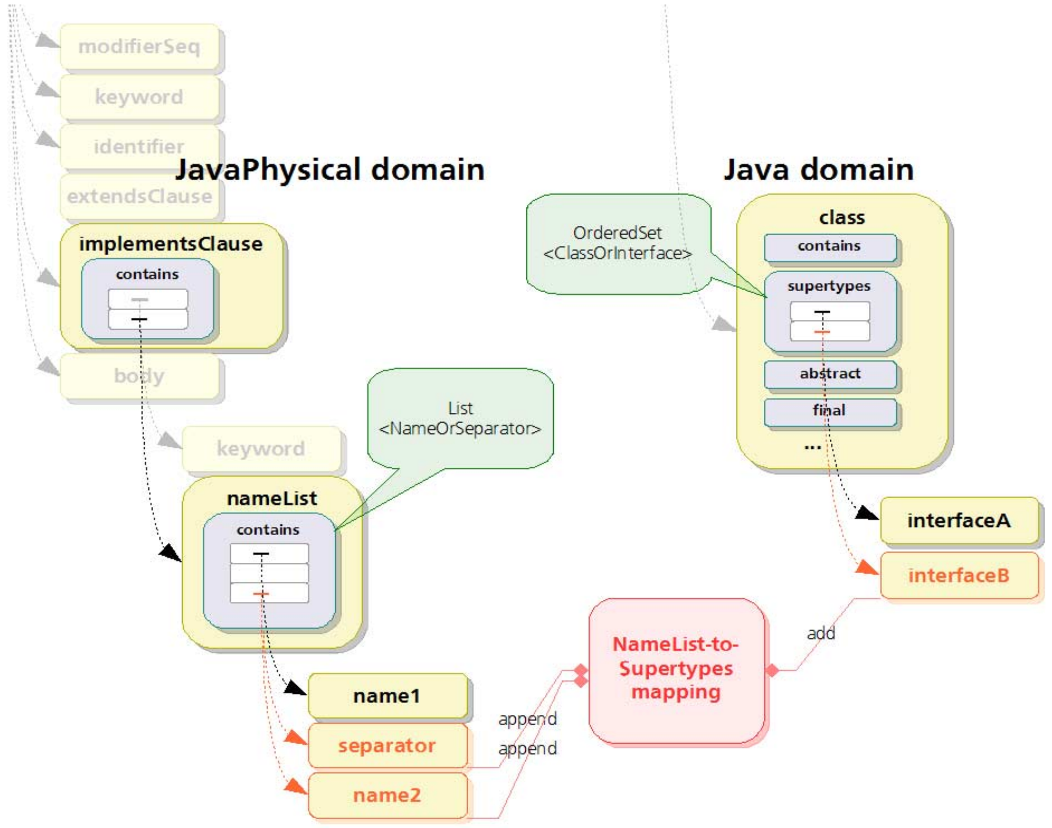

Figure 3: Incremental synchronisation of domain models

The sequence of events is roughly as follows:

1. A "separator" object (representing a comma character, in this case) is appended to the *contains()* list of the comma-separated list of names in the class declaration's implements clause.
2. The mapping function which incrementally derives the *supertypes()* set of the class does nothing in response to this particular event.
3. A "name" object (whose referent happens to be the interface B) is appended to the *contains()* list of the name list.
4. The mapping function responds to this action by adding the referent of the newly-appended name to the *implements()* set of the class.

This glosses many details. First, it is important to remember that *all* the reactive structures involved change incrementally. Thus, when the name is added to the name list, it appears empty at first and then its immediate children are added recursively. The mapping must therefore be capable of responding to the insertion of an empty name, and also to the insertion of an individual identifier or "." into that name. (In Java a *name* is a period-delimited sequence of *identifiers* [13].) Even when whole objects are moved, perhaps as part of a cut-and-paste operation, the runtime decomposes the operation into the incremental movement of sub-structure with the mapping receiving and responding to each primitive action. Although this is somewhat slower in a batch-mode style of operation, it is optimal in interactive mode, which is the primary mode of operation.

This example also glosses what actually happens inside the mapping. In this case, the mapping code would look something like this:

```
implements = orderedSet(
    map(
        filter(
            nameList.contains,
            instanceOf( Name )
        ),
        referent
    ),
    simpleNameComparator
)
```

The various functions named here - *orderedSet*, *map*, *filter*, *instanceOf*, *referent* and *simpleNameComparator* - are themselves incremental and respond to individual atomic changes in their input to produce the corresponding incremental change in their output. The *orderedSet* function for example inserts a single element into a list representing an ordered set in a position which respects the ordering defined by the comparator. Structural induction guarantees that the list elements are always correctly ordered. In a similar fashion *map* incrementally responds to insertion of a new element into its input list by applying its functional argument to it and inserting the result into its output list. The incremental version of *map* relates to the batch version of *map* by satisfying the following:

$$\begin{aligned}&\text{current} = \text{map}_{\text{batch}}(\ \text{list},\ f\ ) \Rightarrow\\ &\quad \text{map}_{\text{incremental}}(\ \text{action},\ \text{current},\ f\ ) =\\ &\quad \text{map}_{\text{batch}}(\ \text{apply}(\ \text{action},\ \text{list}\ ),\ f\ )\end{aligned}$$

where *action* is a primitive list operation (the insertion or removal of an element).

In all cases the input and the output of these incremental functions are references. One therefore can think of references as expressions which have been "lifted" into a purely reactive evaluation scheme. Bearing in mind that domain/object is implemented in Java, the execution of the Java code equivalent to the above pseudo-code only sets up the meta-level dependency structure between the references *qua* sub-expressions. It does not do any actual application-level computation. Computation is purely reactive, in response to a change being initiated somewhere in the system, typically via user input.

This domain/j example also sheds some light on the ultimate utility of bidirectional interpretative mappings. In the context of the relationship between the Java and JavaPhysical domains that we have just seen, the inverse mapping – in this case the mapping from Java to JavaPhysical – corresponds to the traditional notion of "forward engineering". The Java domain is more "abstract" than the JavaPhysical domain to the extent that it contains less information. Making the Java domain mutable with respect to JavaPhysical therefore means making various lossy mappings invertible, by adopting various strategies and techniques for restoring the lost information. (The lost information in this case includes information about particular syntactic forms used for various Java semantic concepts, layout/whitespace information, etc.) These "range restricting" strategies, whose job it is to make a non-deterministic inverse deterministic, all amount to quality-of-implementation decisions for particular mappings and are not intrinsic to the domain/object platform itself. But the more support provided by the runtime environment itself for bidirectionality, the easier it is for domain implementors to provide high-quality, robust, customisable inverse mappings. Some of these opportunities are discussed in section 7.

5. EMBEDDING DOMAIN/OBJECT IN JAVA

The domain/object platform is implemented on top of the Java 5 platform and takes advantage of certain new language features in Java 5 such as static import declarations and "vararg" argument lists. Although these new features of Java are mainly syntactic sugar, they make it somewhat easier to implement an embedded language. Varargs in particular make it relatively easy to support partial application and composition in a syntactically tidy fashion.

The Java 5 generic type system turned out to be impractical for our purposes and so domain/object is currently untyped. Section 7 discusses some possibilities for using domain/object's own features to augment the language with a modern type system like that of Haskell.

The combination of varargs and static imports allow Lisp-like, human-readable domain/object code to be embedded in a Java program. The following shows the definitions of the two functions which comprise a naive insertion sort:

sort (list, comparator):

```
if_(
    empty(list),
    nil(),
    insert(
        head(list),
        sort(tail(list), comparator),
        comparator
    )
);
```

insert (object, list, comparator):

```
if_(
    empty(list),
    cons(object, nil()),
    if_(
        apply(comparator, object, head(list)),
        cons(object, list),
        cons(
            head(list),
            insert(object, tail(list), comparator)
        )
    )
);
```

Currently the above definitions require some supporting scaffolding in the form of Java classes, although it is likely that these can be eliminated with further effort.

One scalability issue still to resolve is avoiding stack overflow during the demand-driven traversal of large recursively-defined structures. Similar problems are faced by implementations of languages such as Scheme on the Java virtual machine (e.g. Kawa [4]). The solution will probably involve explicitly managing our own stack in the Java heap.

Domain/object code can be interfaced to regular Java code via the *set()/get()* methods on a reference and via standard notification schemes along the lines of the Observer pattern [11].

6. RELATED WORK

The **Mjølner** system [22] and the **Harmonia** project, previously known as **Ensemble** [17], were the original inspiration for our work and perhaps explain why the first application to be delivered on the domain/object platform is a fully incremental software development tool. In emphasising simplicity and regularity over baroque language features, the programming languages **Self** [33] and **Beta** [26] contributed much of the philosophy.

There are several more recent influences too. We have adopted the SDF2 meta-syntax, used on the **Stratego** program transformation project [31] for the purpose of defining syntactic domains. **Squeak**'s [19] meta-circular user interface and "horizontal inheritance" paradigm is similar in several ways to domain/object. Most recently, **Subtext** [10], as has already been mentioned, shows some striking similarities to domain/object, particularly in its reactive model of computation and its elimination of the distinction between runtime and compile-time. Subtext has also suggested some interesting avenues for future research.

There is a large body of related work in the literature of visual programming languages (e.g., [32]) and dataflow languages such as Lucid [34], which have not been a direct influence, although there is considerable common ground. **Forms/3** [5] is a declarative, spreadsheet-based visual programming language. The developer directly places cells on a form, and defines formulae relating cells to other cells via a graphical user interface. The Form/3 notion of "time travel" [3] closely corresponds to domain/object's versioned runtime.

SCIL-VP [23] is a visual programming language which allows users to combine arbitrary high-level functions into a dataflow graph, which can be visualised for debugging and optimisation.

Lucid [34] was initially developed as a language in which it would be straightforward to prove assertions about programs. Lucid is a non-imperative dataflow language; computation is precipitated by *eduction*, which is simply demand-driven dataflow. Under this model, when the value of an object is requested, then if it is available in a cache it is returned; otherwise it is computed thorough other means, recursively applying the same pattern. In domain/object the mechanism for eduction is the demand-driven means by which values are calculated, and the memoisation table plays the role of a cache.

7. FUTURE DIRECTIONS

7.1 Types and abstract interpretation

One exciting possibility is how a type system might be integrated into domain/object. In eliminating the distinction between compile-time and runtime, domain/object might ironically lend itself better to a powerful type system, if we understand a type system as an abstract interpretation [9] and consider that abstract interpretation itself is a perspective on program analysis where "static" analysis and the detection of "actual" runtime conditions lie on a continuum.

In a research context, the approximation of runtime behaviour comprising the type system being experimented with would be a reactive structure that was updated live as the researcher tweaked the definition of the type system. The distinction between test assertions, aspects and type systems would eventually blur away with the freedom for any particular query of the runtime structure of the program to migrate between a live query in the form of an aspect, an application-level test, or a hard language constraint in the form of a component of a type system.

7.2 Relational programming

As discussed in section 3.4, the domain/object requirement to support bidirectional functions makes explicit a degree of redundancy which is usually hidden in today's applications and languages. For example, for some query or view derived from a list, the *remove* mapping is generally the inverse of the *insert* mapping, yet this usually needs to be implemented manually. The reason is simply that the *remove* mapping is non-deterministic if the *insert* mapping is lossy. For exactly the same reason, the reverse mapping required to make a query or view mutable must also usually be hand-coded. Consider for example the *filter()* example mentioned earlier whose inverse is non-deterministic with respect to the insertion point in the underlying list.

The ultimate problem here is that the inverse of a function is itself only a function if the original function is *bijective*, i.e. 1-1, and defined for every member of its range. Otherwise the inverse is a relation or at best a partial function. This suggests that a possible future development for domain/object would be the incorporation of some techniques from the relational programming field. Relational programming is a marriage of functional programming and logic programming which explicitly supports non-determinism in the form of *choice* and *cut* operators. A relational runtime for domain/object could exploit this non-determinism to allow for example the *remove* mapping for a list to be simply the *insert* mapping running backwards, even if the *insert* mapping was lossy. A deterministic result could be obtained by range-restricting the inverse using some of the techniques mentioned in section 3.4, such as default user preferences or explicit user interaction.

7.2 Modal logic

There is a potentially interesting relationship between the relational programming model, modal logic, and domain/object's versioned runtime. In their "primary" (forward) direction, domain/object functions are genuine functions: they are always left-univalent, i.e. uniquely defined for a given set of inputs. One way of thinking about bidirectionality is to imagine that applying the inverse of a function only injects *one possibility* of the (non-deterministic) inverse into the "current" version, placing all alternatives of the inverse mapping into alternate versions. It would be natural then to think in terms of a "possible worlds" semantics [22], and say that the user inhabits a single actual world at any point in time, and computation is the traversing a path through the space of possible worlds. The significance of this, if any, is far from clear.

7.3 Pattern matching
The algebraic nature of domain/object structures suggests a natural fit with the decompositional, pattern-matching style common in functional programming languages. Structural pattern matching - of the incremental kind - will therefore probably be part of any eventual higher-level language for expressing domain/object functions.

7.4 Concurrency
Finally, dataflow languages naturally lend themselves to concurrency, but we have largely dodged this important topic until now. The following very simple example illustrates the kind of issue we currently face. Given:

```
c = true();
b = not(c);
d = or(c,b);
```

The reference graph for the example is shown in Figure 4 below. Assume that we now wish to set c to *false*. Clearly we should not observe any change in the value of *d*; it should always appear *false* to an observer external to the system. This can only be achieved if it is not possible to observe any values in the system until the initial action (setting *c* to *false*) is complete in the sense that all dependent values are updated.

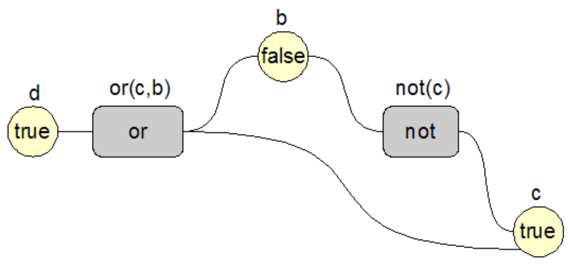

Figure 4: Setting c to false should not cause a change in the observed value of d

When *c* is set to *false*, notification is sent along two edges to *or(c,b)*, which becomes *false*, and *not(c)*, which becomes *true*. If change then propagates from *or(c,b)* to *d* **before** it propagates from *not(c)* to *b* and thence to *or(c,b)*, then *d* will temporarily become *false*. This interim state of *d* arises because it does not see its inputs change atomically from (*false*, *true*) to (*true*, *false*), but instead sees (*false*, *true*), (*false*, *false*), (*true*, *false*). The observability of the interim state is problematic if for example it is associated with an assertion which will fail immediately should *false* be observed as a value of *d*.

Domain/object's transaction support and versioned runtime will probably play central roles in the platform's future treatment of concurrency. Dataflow languages arose from research into concurrent computing [35], and our future efforts will no doubt leverage the substantial research already undertaken in this area.

8. CONCLUSION
The "integration of disparate systems" is the number one priority facing internet-centric businesses today, according to IBM [20]. Yet as an industry we have spent precious little time understanding how to make software really *connect*. While much remains to be done, domain/object represents an important move in the direction of truly semantic computing.

9. REFERENCES
[1] Acar, U., Blelloch, G. and Harper, R. (2002). **Adaptive Functional Programming.** POPL '02.

[2] Acar, U., Blelloch, G. and Harper, R. (2004). **Adaptive memoisation.** Carnegie Mellon Technical Report.

[3] Atwood, J., Burnett, M., Walpole, R., Wilcox E. and Yang, S. **Steering Programs via Time Travel.** (1996). In *IEEE Symposium of VIsual Languages*, Boulder, Colarado USA.

[4] Bothner, P. (2003). **Kawa, the Java-based Scheme system.** http://www.gnu.org/software/kawa/.

[5] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J. and Yang, S. (2001). **Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm.** In *Journal of Functional Programming* 11(2), pp. 155-206.

[9] Cousot, P. **Types as Abstract Interpretations.** (1997). In *Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages,* pp. 316-33. ACM Press, New York.

[6] Czarnecki, K., and Eisenecker, U. (2000). **Generative Programming - Methods, Tools, and Applications.** Addison-Wesley.

[7] Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E. (1989). **Making Data Structures Persistent.** In *Journal of Computer and System Sciences*, vol. 38, no. 1. Academic Press.

[8] Edwards, J. (2005). **Structure-Oriented Programming.** http://alarmingdevelopment.org/index.php?p=9.

[10] Edwards, J. (2005). **Subtext: Uncovering the Simplicity of Programming.** OOPSLA 2005 (forthcoming).

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). **Design Patterns.** Addison-Wesley.

[12] Giraud-Carrier, C. (1994). **A Reconfigurable Data Flow Machine for Implementing Functional Programming Languages.** In *ACM Sigplan Notices*, vol. 29, no. 9.

[13] Gosling, J., Joy, W., Steele. G. and Bracha, G. (2005). **The Java Language Specification (3rd Edition).** Addison-Wesley Professional.

[14] Graham, P. (1993). **On Lisp.** Prentice Hall, New Jersey.

[15] Gurd, J. R., Kirkham, C. C. and Watson, I. (1985). **The Manchester Prototype Dataflow Computer.** In *Communications of the ACM*, vol. 28, no. 1. Association for Computing Machinery.

[16] Harel D. and Pnueli A. (1985). On the **Development of Reactive Systems.** NATO ASI Series F, vol. 13. Springer-Verlag.

[17] **Harmonia Research Project.** http://harmonia.cs.berkeley.edu/harmonia/index.html.

[18] Hughes, J. **Lazy memo-functions.** (1985). In Jouannaud, J. (ed), *Functional Programming Languages and Computer Architecture*, no. 201 in *Lecture Notes in Computer Science*, pp. 129-146. Springer-Verlag.

[19] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S. and Kay, Alan. (1997). **Back to the future: the story of Squeak, a practical Smalltalk written in itself.** In *Proceedings of the*

12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Association for Computing Machinery, New York.

[20] Karpinski, R. (2002). **IBM's New Tools Complete Web Services Plunge.** Infoweek.

[21] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. (1997). **Aspect-oriented programming.** In Akşit, M. and Matsuoka, S. (eds), *ECOOP '97 --- Object-Oriented Programming 11th European Conference*, vol. 1241, pp. 220-242. Springer-Verlag.

[22] Knudsen, J. L., Madsen, O. L., Magnusson B. and Lofgren M. (1993). **Object-Oriented Software Development Environments: Mjølner Approach.** Prentice-Hall.

[23] Koelma, D., van Balen, R. and Smeulders, A. (1992). **SCIL-VP: a multi-purpose visual programming environment.** In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's*, pp. 1188-1109.

[24] Kripke, S. **Naming and Necessity.** (1980). Harvard University Press.

[25] Liu, Y. A. (1999). **Efficient Computation via Incremental Computation.** Pacific-Asia Conference on Knowledge Discovery and Data Mining.

[26] Madsen, O. L., Moeller-Pedersen, B. and Nygaard K. (1993). **Object-Oriented Programming in the BETA Language.** Addison-Wesley.

[27] Michie, D. (1968). "**Memo" functions and Machine Learning.** *Nature*, vol. 218.

[28] Object Management Group. (2001). **Unified Modelling Language, v1.5.** http://www.omg. org/cgi-bin/doc?formal/03-03-01. Object Management Group, Inc., Needham, Massachussetts.

[29] Rapaport, W. J. (1999). **Implementation is Semantic Interpretation.** In *The Monist*, no. 82.

[30] Simonyi, C. (1995). **The Death of Computer Languages, the Birth of Intentional Programming, The Future of Software.** University of Newcastle-upon-Tyne, England, Department of Computing Science.

[31] **Stratego: Strategies for Program Transformation.** http://www.program-transformation.org/Stratego/WebHome.

[32] Tanimoto, S. L. (1990). **VIVA: A Visual Language for Image Processing.** In *Journal of Visual Languages and Computing*, vol. 1, issue 2.

[33] Ungar, D. and Smith, R. B. (1987). **Self: The Power of Simplicity.** In *SIGPLAN Notices*, Vol. 22, No. 12. Association for Computing Machinery, New York.

[34] Wadge, W. W. and Ashcroft, A. (1985). **Lucid, the Dataflow Programming Language.** Academic Press.

[35] Whiting, P. and Pascoe, R. (1994). **A History of Data-Flow Languages.** In *IEEE Annals of the History of Computing*, volume: 16, Issue 4, pp. 38-59.

[36] Zhanyong W. and Hudak P. (2000). **Functional reactive programming from first principles.** In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 242-252.