An Active Functional Intensional Database

Paul Swoboda John Plaice
School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{pswoboda, plaice}@cse.unsw.edu.au

2004

Abstract

We introduce a new kind of functional database that unifies concepts from the realms of publish-subscribe middleware, pervasive computing, and intensional programming. The AFID (Active Functional Intensional Database) Project allows the distribution of both pervasive context and related, versioned content, and offers the means of effecting a client's interaction with both. The AFID data model builds on existing infrastructure from the Intense project for the efficient manipulation and networked distribution of intensional context, adding the ability to encode multiple versions of any complex entity, where each version may vary in both content and structure, at any level of granularity. Further, the system ensures that clients may listen to structured, minimal changes in *specific logical versions* of encoded entities, as they occur, whenever any change is made to the total entity encoded in the database.

1 Introduction

This paper introduces a first attempt at the development of an active database that supports communities, as proposed by Buneman and Steedman [1]:

> Now is the time for a radical revision of the way we publish data on the Web. The traditional view of a database as an inert structure, only to be observed through the machinery of a query language or browser, is no longer tenable. In the future our databases and Web documents are going to become vehicles of communication between their communities of users. The challenge is to support and enable this development.

The AFID project is comprised of an *active, pervasive context server*, which doubles as a functional database, an intensional context (§2.1) and versioning library, and a client/server protocol and associated APIs for the distribution of context. This active database allows the storage and distribution of both pervasive context and contextually versioned entities, which can themselves have multiple, versioned components and arbitrary, versioned structure. Further, AFID allows both networked clients and the content on which they operate to be simultaneously versioned in a medium that blends the notions of distributed context and distributed intensional content. The client is versioned at all levels using the distributed context as a mechanism for pervasive runtime configuration management, whereas the versioned content on which the client operates can be stored in the context server itself.

The AFID system addresses another issue visited by Buneman and Steedman, that of the importance of communication via annotation:

> Our claim is that annotation is becoming a new form of communication, and that understanding and managing this new medium presents a major challenge.... Databases are rather rigid structures and typically do not provide *room* for annotations.

With AFID, distribution of both context and content uses *context operators*, a structured form of annotation, akin to the deltas of software configuration. AFID goes even further, allowing entities in the database to be *interpreted* according to an adjacent context, so that context operators applied to either the total entity or the context of interpretation result in *interpretive context operators* describing the *interpretive annotation* under interpretation.

2 Background

2.1 Intensional Programming

Intensional programming [2] refers to programs that are sensitive to a global multidimensional runtime context, which the program itself can modify. In the general case, an intensional program contains a number of versioned entities, each of which has a single valid instance under a particular global context. The set of (*versioned entity*, *current context*) pairs is termed an *intension*, whereas a specific, resolved version of the entity is termed an *extension*. As the program executes, intensional entities are resolved to single versions against the global context. With intensional languages, a versioned entity consists of a set of pairs of context-tagged extensional values (E_x, C_x), termed *versions*. The process of determining which of these versions should be used is termed a *best-fit*: a partial order called *refinement* [3] is defined over the set of all contexts; the context tags of an entity are compared to the global reference context C_r. The set of all tags in a versioned entity is termed a *context domain*.

The distinction between an intensional program and one that merely uses a context or pervasive data source as a polled reference for runtime configuration is that the semantics of an intensional program varies implicitly with the context, often with language-level support for versioned control flow. In the extreme case, anything with an identifier can exist in multiple versions, including all functions and data structures, as with the Perl-like language ISE [5].

2.2 Contexts and Context Operations

The Intense library [4, 6] was developed to provide a comprehensive set of generic tools for C++, Java, and Perl, to perform fast intensional versioning of arbitrary software entities with language-level constructs. In the C++ implementation, for example, templates are provided to version any existing typed entity, with support for a thread-safe pervasive runtime context. The context model used by Intense is tree-structured, with arbitrary depth and arity at each node. Each node in a context is indexed by a string *dimension*, unique to the parent node, and may optionally contain a *base value*, a scalar which can be used to reference any entity. A similarly-structured *context operator* is used to modify a context — each node in an operator may contain a new base value or a base value clearing flag ("`-`"), and a pruning flag ("`--`"). The latter prunes all subcontexts in the target context node which are not present at the corresponding node in the operator. An example application of an operator to a context is as follows:

```
<dim0:<"base0"+dim01:<"base01>+dim02:<"base02">>
["newbase"+dim0:[-+--+dim02:["newbase02"]+dim03:["newbase03"]]]
```

```
<"newbase"+dim0:<dim02:<"newbase02">+dim03:<"newbase03">>>
```

In the above, the operator (delimited, in canonical text format, by `[ ]`) inserted a new base value `"newbase"` at the root dimension of the context (delimited by `< >`), pruned and cleared the base value under dimension `dim0`, changed the base value for dimension `dim0:dim02` to `"newbase02"`, and inserted a new subcontext `dim0:dim03:<"newbase03">`. A key benefit of encoding context operators with the same structure as contexts is that we can apply them in a single recursive pass; another benefit is the fast serialisation of operators for transmission in a networked environment. The context operators provided by Intense are fully associative, and can

be applied efficiently to one another, to yield an operator whose effect is equivalent to applying its component operators to a context in succession.

2.3 The Æther

An *æther*, first introduced in the Intense project, is a context which *participants*, connected at any node, can use both as a contextual reference and as a structured medium for contextual discourse. The latter is achieved by sending and receiving context operators to and from the æther, modifying the æther in the process. A participant receives all operators that are applied to its own node, or to descendants of its own node. In the latter case, an operator appears to the participant as an operator relative to the participant's node. For example, if we treat the context above as an æther, participants registered at the following dimensions in the æther would see the following operators:

```
dim0:       [-+--+dim02:["newbase02"]+dim03:["newbase03"]]
dim0:dim03: ["newbase03"]
dim0:dim02: ["newbase02"]
dim0:dim01: [-+--]              -- pruning
dim1:       []                  -- no change
```

Tree manipulation aside, the important innovation here is that, as a reference data structure for efficient intensional versioning, the æther and context may be used interchangeably. A typical use of an intensional participant is to recompute best-fit entities on each notification of an operator; the operator itself may be considered by the participant to see if recomputation is actually necessary. For example, a participant may consider a number of sub-æthers under its node to be intensional reference contexts. In this case, recomputation of a best-fit for an entity controlled by one of the reference contexts must only be performed if an operator affects the entity's particular context.

2.4 Æther Protocol

The Intense project includes a protocol (AEP, for Æther Protocol) and associated server and client APIs for the networked sharing of context between distributed participants. The æther server, `aepd`, exists in a threaded C++ implementation and is geared towards scalability in the number of networked æther participants, as well as in traffic volume, i.e., the number of context operators per second.

The latest version of AEP allows multiple participants to use the same connection, where a single connection corresponds to a single remote æther. A context operator may be optionally ignored by its originating participant, or by all participants in the same connection.

In a very large AEP-based system, context operator frequencies can occur in the range of hundreds to bursts of tens of thousands per second. In this case, it is often not feasible to rebroadcast all operators to listening participants. Instead, the æther server makes use of the full associativity of context operators, applying successive operators to one another between *associative fences*, prior to rebroadcast of the resultant operator to listeners. To this end, a participant is able to label a submitted operator as a fence, meaning it cannot be added to previous operators, or have successive operators added to it. The æther server employs a number of internal operator queuing mechanisms to handle the application of operators to one another.

To facilitate the processing of context operators on both the client and server sides of AEP, integer sequences are employed. The AEP client API generates a unique sequence for every client/participant connection, and the æther server maintains a sequence for all client-server operations. In this manner, a client is able to tell if a rebroadcast operator is or contains an operator which the client submitted, i.e., if the server sequence for the rebroadcast operator is equal to or greater than the server sequence number given to the client's operator.

The AEP design is abstract, in that it requires a concrete implementation in some host protocol. The concrete implementations supplied with Intense include a shared (threaded) implementation that simply passes protocol token objects between client and server threads. Also

provided are a number of stream-based implementations; these include, in order of efficiency, implementations using direct binary serialisation of protocol token objects, XDR (from Sun RPC, architecture-independent) serialisation, and AETP, a textual, stream-parsing implementation for maximum client-server compatibility. With stream-based AEP connections, a mechanism is provided to negotiate the fastest of these modes. Where necessary, it will be possible to encapsulate AEP in OORPC transport layer such as CORBA.

3 AFID: Motivation and Intuition

To date, no-one has created a database that can be used both as an active context server *and* as an active storage mechanism for intensional entities, i.e., a service that can deliver updates to clients of changes in intensional entities, *as they occur*. Some networked directory services such as LDAP and various publish/subscribe middlewares do come close with respect to context distribution, although few provide active contextual updates to clients and almost none do so in the form of structured context operators or deltas. Certainly, no previously existing directory service has attempted to provide a concrete means for the interpretation of structured directory data itself as an intensional context. Tools such as XQuery do allow us to perform functional data extraction from trees, but the infrastructure on which they operate (i.e., XML "databases") is by no means suitable for efficient intensional computing.

The inability to use existing database infrastructure in implementation lends us a bit of freedom in design. The desired system should not only encapsulate the context used to version intensional clients, as well as the intensional content, but it should use the same medium of interaction to effect changes to both. Further, a client should be able to listen to a networked context either as an interpreted intensional entity, or as a pure context (an uninterpreted, atomic entity). Since the shared context is active, and both content and pure context are encoded in it, changes in either can be used to drive the execution of the participants. In the general case, an intensional participant should only have to deal with changes to the particular logical version of itself or the data on which it operates. The existing Æther Protocol and associated æther server have been modified to achieve these purposes.

The central intuition for AFID is that the set of (E_x, C_x) version pairs that comprise a context domain can be encoded in the Intense context structure, as subcontexts of a node representing the domain. Under a domain node are nodes for each version in the domain; each version node contains one or more nodes for its tag contexts, as well as a node for its versioned content.

Given a reference context, a domain node is *interpreted*, which means that one or more of the version nodes below it will be selected, and the content nodes below each of the selected versions will be used as the content. The interpretation process continues recursively on the content nodes, until uninterpretable (pure context) nodes are reached.

A second intuition is that we can aggregate the components of a compound intensional entity, simply by using further dimensions in or above the same context to define structure and ordering of various components, as is common practice in structured markup languages such as XML.

A straightforward means of aggregating ordered components using the same context node is through the concatenation of the interpretations of lexicographically ordered dimensions, termed a *join*. Figure 1 shows an entity composed of multiple intensional components, each of which is named by a single dimension, and is versioned with the dimensions `lang`, `size`, and `rev`.

The final and most important intuition is that we can encode the structure of an entity in a hierarchical fashion, using nodes to denote the concatenation of ordered subcomponents, as well as to denote best-fits of intensional subcomponents. In this manner, we can easily encode entities whose content *and structure* can be simultaneously versioned by the same adjacent reference context. Under one logical version of an encoded document, a section might be omitted, added, or rearranged, and certain elements of content might appear in a different format or language. Significantly, due to the fact that a single logical version of an entity is represented by a portion of the context encoding the entity, the *difference* between two logical versions can be directly represented with a *context operator*. The resulting system can be used as an infrastructure for

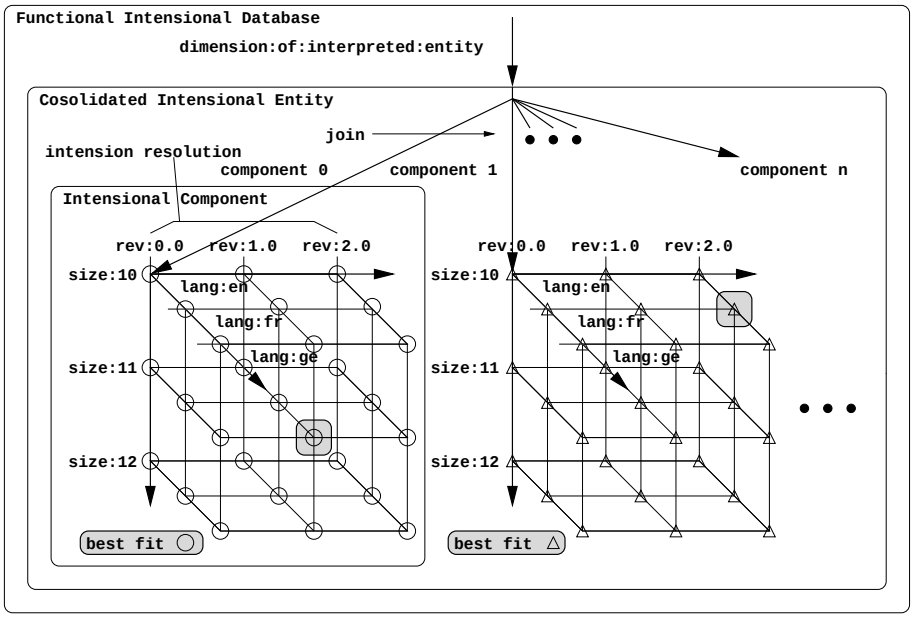

Figure 1: Accessing Sequenced Components of an Intensional Entity Under Lexicographically Ordered Subdimensions

arbitrary intensional tools, with several key benefits — the abstraction of intensional content into a structured database, the networked distribution of the content with the possibility of contextual collaboration between interacting clients, and the simultaneous versioning of both the application and content with the same medium.

4 AFID: A Structural Overview

The AFID server is derived from the Intense æther server codebase, and adds one key abstraction to the æther server: the *context interpretation*. This is the mapping of a reference context to specific portions of an interpreted context or æther, as discussed in Section 3, where the æther has been encoded with a functional mapping at each interpreted node. An important feature of AFID's context interpretation is that the result of an interpretation is another context that *parallels* the context rooted at the interpreted node; i.e., the interpretation has its root at the same dimension in a parallel context. While the result of an interpretation is a context, the semantics of each interpretation is list-valued, where the elements of the list are references to descendent nodes in the interpreted context, as shown in Figure 2. This allows the result of an interpretation to be passed to a parent interpretation, while also allowing the result to denote a portion of the context rooted at the interpreted node.

The key interpretations in the first version of AFID are *literal*, *join*, and *best-fit*. A literal interpretation yields a single-element list comprised of a reference to the interpreted context node, and does not require any special encoding in the interpreted node. A join interpretation concatenates the lists resulting from the interpretation of its argument parts. A best-fit interpretation is a best-fit selection of a single value from the interpretations of all subcontexts of the dimension `versions`, where each version is comprised of an interpreted *value* branch and an interpreted list of *tag contexts*, comprising the context domain.

For the transmission of context operators to an interpreting client, the server should only send branches of the interpreted entity that correspond to actual content. Further, we stipulate that reinterpretation of the partial entity at the client end must produce the same results as the initial interpretation in the server. For example, in a best-fit interpretation, it is unnecessary to send the

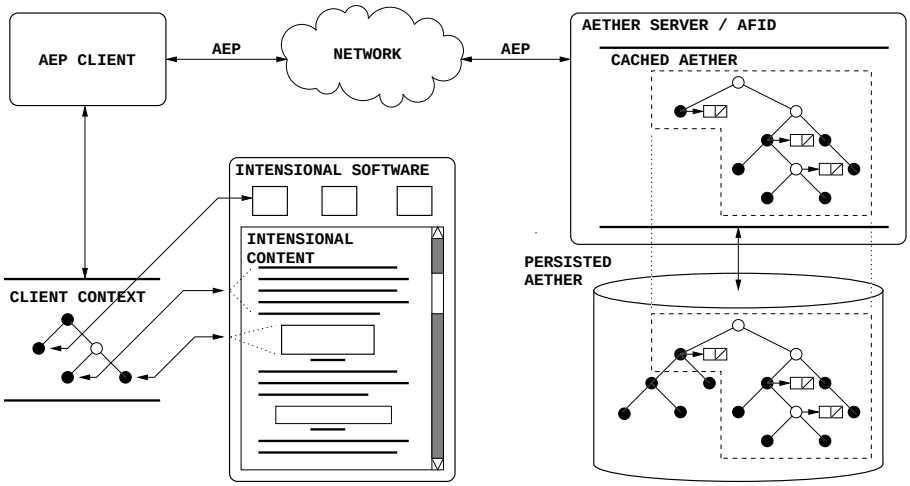

Figure 2: The AFID Architecture

tag branches under each version, or to send any part of non-best-fit versions. If an interpretation changes structurally, we generate a context operator that prunes removed branches. In general, a reinterpretation creates a context operator that transforms the previous interpretation to the new one.

We can often avoid total recomputation of an encoded entity, when changes are made to a single portion of the entity in the database. There are two basic varieties of changes we must consider, (extensional) changes made to the total encoded entity, and changes in the reference context used for interpretation.

In the first case, where some portion of the encoded entity is modified, recomputation need only be propagated up the context until either the root node of the interpretation is reached, or until an interpretation node is reached, whose result remains unchanged. The difference between a previous interpretation and the result of this smaller recomputation may still be sent to a listening interpreter in the form of a minimal context operator.

In the second case, where changes are made to the intensional reference context, reinterpretation must of course be performed again from the root of the encoded entity; however, since only intensional interpretation nodes are affected by a change in context, we need only recompute those branches of the entity that contain intensional nodes. In each interpretation node, we use a

Table 1: AFID Interpretations

Function	Semantics
$I(C)$	identity — equal to a single-element list $\{C\}$
$J(C)$	a list of the joined interpretations of all subcontexts of C, in lexicographic order of their dimensions
$B(C_{ref}, C)$	an empty or single-element list $\{C_{best}\}$, where C_{best} is the interpretation of the `value` dimension under the best-fit subcontext of the `versions` dimension. The tags of the context domain are formed from the interpretation of each of `versions:`seq_i`:tag`, and the best-fit is performed with respect to C_{ref}.
$N(C_{ref}, C)$	a possibly-empty list of near-fit contexts $\{C_i\}$, where the context domain for the near-fit is formed as per $B(C_{ref}, C)$.
$F(C_{ref}, C)$	an empty or single-element list consisting of the first element returned by $N(C_{ref}, C)$.

count of all context-sensitive interpretations at or beneath the node; where sub-interpretations are considered, only those branches containing a non-zero intensional node count are reinterpreted.

The first version of AFID supports the interpretations shown in Table 1. We add two new intensional interpretations, *near-fit* and *first-fit*.

5 Applications

5.1 Networked Intensional Desktop Environments

A clear application of the functional intensional database is in the simultaneous versioning of applications and content in a local-area setting. The corporate and classroom environments could both benefit tremendously from such infrastructure, with fine-grained runtime configuration management of applications. Because the Intense framework is based on intensional logic, desktop users can set personal preferences in local contexts, that may or may not be overridden by controlling applications on servers. A teacher might invoke changes in the context controlling a document being viewed by an entire class, for example advancing a presentation to the next section, with individual students following the document in another language, or without images, or in a certain font. Likewise, a document or database posted in AFID on a corporate intranet could support interpretations specific to the department or profession of the user who views it.

5.2 Collaborative Content Authoring

Using AFID, collaborative annotation, as suggested by Buneman and Steedman, is directly realisable for both concrete and abstract or programmatic content.

By *concrete content*, we mean content *processed* by applications, such as structured textual documents, database records, images, and multimedia content. Concrete content can be annotated in realtime by any number of authors, each of whom may listen to changes in either the raw content, or to changes in specific logical versions (contextual interpretations) of the content. The annotations of one or more authors, in the form of context operations, may or may not be perceived as deltas in the interpretations of any specific listener.

By *abstract content*, we mean content that, while encoded in the AFID mechanism with structure, requires further interpretation on the part of a listener, to be rendered to concrete content. As with the latter, annotations can be submitted collaboratively, but are perceived selectively under different interpretations.

By *programmatic content*, we mean manipulation of structured, AFID-encoded software constructs, which can be used by interpreting clients, and may or may not govern the execution of a client. A simple example would be the encoding in AFID of an algorithm as an object-oriented data structure. This is very simple to achieve, but has powerful applications — multiple clients can thus annotate the encoded algorithm, while differing interpreters are free to perceive and subsequently apply/use it, under differing contexts.

5.3 Versioned Software Repositories

An obvious use of AFID is as a repository for *entire* software projects, where each file in the project can be versioned with arbitrary dimensionality. Changes to individual files, components of files, and the additional, removal or reorganisation of directory structure can all be effected with context operators; aspects such as (ordinary) revision control, branching, and release tagging *come for free*.

6 Future Directions

We leave open the possibility for interpretations that operate on arguments other than entire subcontexts (for example, programmatic case-statement functions, that depend either on the reference

context or on specific base values in the interpreted context).

Although the primary use of context interpretation is the extraction from the encoded entity of a set of literal-interpretation "leaf" contexts, there exists the possibility of *higher-order context interpretation*, as is understood in the functional programming sense. Since the `interpret` dimension in an interpreted node can itself contain an arbitrarily complex subcontext, we can conceive of best-fit interpretations, or in the general case, interpretations encoding interpretations.

An abstraction currently being developed is the notion of a *disjunct interpretation*, essentially an adjacent function definition for context interpretations. With this scheme, the intensional qualities of an interpretation node can be specified by supplying arguments; the interpretation can be referenced by dimension from other interpretations with the equivalent of a function call.

A generic, server-side scripting API is also being developed to allow manipulations of interpretations, using Perl as the scripting framework. Such code fragments would be stored in the context directly, and are compiled on insertion into the æther, to be executed efficiently during interpretation.

References

[1] Peter Buneman and Mark Steedman. Annotation – the new medium of communication. *Grand Challenges for Computing Research*. `http://www.nesc.ac.uk/esi/events/Grand_Challenges/panelb/b2.pdf` . October 2002.

[2] A.A. Faustini and W.W. Wadge. Intensional programming. In J.C. Boudreaux, B.W. Hamil and R. Jenigan, eds., *The Role of Languages in Problem Solving 2*, Elsevier North-Holland, 1987.

[3] John Plaice and William W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering* 19(3):368–276, March 1993.

[4] Paul Swoboda. *A Formalisation and Implementation of Distributed Intensional Programming*. Ph.D. Thesis, The University of New South Wales, Sydney, Australia, 2004.

[5] Paul Swoboda and William W. Wadge. Vmake, ISE and IRCS: General tools for the intensionalization of software systems. In M. Gergatsoulis and P. Rondogiannis, eds., *Intensional Programming II*, World-Scientific, 2000.

[6] Paul Swoboda and John Plaice. A new approach to distributed context-aware computing. In A. Ferscha, H. Hoertner and G. Kotsis, eds., *Advances in Pervasive Computing*. Austrian Computer Society, 2004. ISBN 3-85403-176-9.