Possible Worlds Versioning

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

2008

Abstract

We present a history of the application of the possible-world semantics of intensional logic to the development of versioned structures, ranging from simple software configuration to the networking of distributed context-aware applications permeated by multiple shared contexts.

In this approach, all of the components of a system vary over a uniform multidimensional version, or context, space, and the version tag of a built version is the least upper bound of the version tags of the selected best-fit components. Context deltas allow the description of changes to contexts, and the subsequent changes to components and systems from one context to another. With æthers, active contexts with multiple participants, several networked programs may communicate by broadcasting deltas through a shared context to which they are continually adapting.

1 Introduction

This paper presents the development of *possible-worlds versioning*, from 1993 to the present, and its application in the areas of software configuration, hypertext, programming language design and distributed computing. This research has followed naturally from the seminal article by author Plaice and William (Bill) Wadge, "A New Approach to Version Control" [28], which demonstrated that software configuration could be understood, and greatly simplified, using the possible-worlds semantics of Richard Montague's intensional logic, and which presented the vision of a flexible Unix-like system called *Montagunix*.

The application of possible-worlds versioning has been used successfully — as will be seen in this article — to add versions to C programs, electronic documents, file systems, Linux processes and programming languages, among others, and to create collaborative software for shared browsing of Web sites. In the 2004 *ACM Hypertext* conference, Doug Engelbart told the authors that this collaborative work [25] was "exactly what we were trying to do in the 1960s" [9].

The intuition driving the above research has changed over time, even though the technical solutions have been remarkably stable. Initially, when possible-worlds versioning was simply being applied to the problem of software configuration, the science fiction-like intuition of possible worlds sufficed: in each world, there is a completely new copy of every entity. This copy may be similar to corresponding copies in other worlds, but it is a complete copy. The technical details and implementations then followed from this vision.

Subsequently, as the problems being addressed became more dynamic in nature—as, for example, in shared browsing of a Web site—the possible world was more easily understood as a physical medium permeating a system and every component therein, as water does to cells in a living body. Any change to the medium can affect instantaneously any part of the system; conversely, any part of the system may change the medium, indirectly affecting other components of the system. In Marshall McLuhan's words, "Environments are not simply containers, but are processes which change their content entirely" [22, p.275].

The problems needing to be resolved today will require a more refined understanding of this permeating medium. If one looks at the Web today, one can see the rise of many different kinds of *communities*, for sharing ideas, software, entertainment, etc. This Web is neither a sea of atomised individuals nor one giant shared space: there are *many* shared spaces, and each individual might be participating in a number of different, overlapping spaces.

This complex arrangements of spaces is also reminiscent of science fiction, but on a higher plane, where one can pass from one virtual space to another, while still retaining some part of the original space: the potentialities are mindboggling, especially when there are many overlapping *universes* of possible worlds, so that one may be simultaneously in different worlds in different universes. This vision, of course, corresponds exactly to present-day needs for the interactions between mobile and ubiquitous computing.

Bill Wadge has been a key player in developing these intuitions. Of course, many of the projects described below have been undertaken by students under his supervision. More importantly, however, he has always been present for the discussions about the future of the research. It was he who discovered the relevance of the ideas of visionaries such as Marshall McLuhan and Ted Nelson — as well as the ancient philosophical debate between atomism and plenism — to our work.

The article is arranged as follows. We begin (§2) with an exposition of intensional logic, possible-worlds semantics and the interplay between intension and extension. We then show (§3) that possible-world semantics is applicable to the structure of programs. We continue with experiments in electronic documents (§4) and in versioned programming (§5). The idea of *intensional communities* leads naturally to the æther and its applications (§6). The concluding remarks propose extending these solutions, using synchrony, to wide-scale distributed computing.

2 Background: Possible-Worlds and Intensional Logic

In 1987, Bill Wadge and Tony Faustini published an article entitled "Intensional Programming" [10], which showed that the possible-worlds semantics of intensional logic was directly applicable to the semantics of Lucid, especially its multidimensional variants. The semantics of a variable is an intension, a mapping from possible worlds to specific values. In each possible world, when one refers to the values of other variables, one is referring to the values within the current world, *unless* some other world is explicitly mentioned.

In this section, we present a brief history — inspired directly from Bill Wadge's article, "Intensional Logic in Context" [35] — of the ideas of possible-world semantics and of intensional logic.

Ever since the beginnings of logic, it has been understood that there is a difference between sentences that are necessarily true because of the nature of logic and sentences that just happen to be true because of contigent factors. For example, the sentence:

$$\textit{Nine is a perfect square.} \tag{1}$$

and the sentence:

$$\textit{The number of planets is nine.} \tag{2}$$

were once both true, but the nature of the truth in the two sentences is different, and they cannot be substituted equivalently into other sentences, as in:

$$\textit{Kepler believed that nine is a perfect square.} \tag{3}$$

and:

$$\textit{Kepler believed that the number of planets is nine.} \tag{4}$$

Given that Johannes Kepler (1571–1630) was a gifted mathematician and remarkable astronomer, and that only six planets were known during his lifetime, likely (3) is true and (4) false.

Furthermore, in August 2006 the International Astronomical Union (IAU) re-classified Pluto as a "dwarf planet," reducing the number of classic planets to eight. In the light of this statement, the sentence deduced from sentences (1) and (2):

$$\textit{The number of planets is a perfect square.} \tag{5}$$

is no longer true either.

The existence of this general problem was recognised by Aristotle (384–322 B.C.E.), *The Logician*, "the first to state formal laws and rules" for logic [2, p.19]. In his writings on formal logic, collectively known as the *Organon*, Aristotle introduced modal logic and distinguished between two modes of truth: *necessity* and *contigency* [2, pp.55-6].

Because of the comprehensive nature of Aristotle's work, his writings dominated all study of logic in the Western world until the development of mathematical logic, beginning in the late seventeenth century. Nevertheless, developments did take place in the understanding of modalities, often with respect to theological arguments as to the nature or existence of God and the world.

In particular, John Duns Scotus (1265/6–1308), a Franciscan scholar, introduced the concept of *possible worlds*. In opposition to Thomas Aquinas, Scotus asserted the primacy of *Divine Will* over *Divine Intellect*: the existing world does not exist because of some moral necessity, but, rather, because of a divine choice. The world we live in could be different and it is just one of numerous logically consistent possible worlds. Gottfried Wilhelm von Leibniz (1646–1716), the founder of mathematical logic [3, p.258], put forward that "a necessary truth must hold in all possible worlds" [6, p.10].[1]

As mathematical logic post-Leibniz developed, with a strong anti-Aristotelian bias, the modalities were pushed aside, at least temporarily, and sentences were designated as being either true or false. The focus of attention was placed entirely on mathematical rigor. In so doing, the distinction between *intension* and *extension* surfaced.

Bertrand Russell wrote (1903) [3, p.361]: "*Class* may be defined either extensionally or intensionally. That is to say, we may define the kind of object which is a class, or the kind of concept which denotes a class: this is the precise meaning of the opposition of extension and intension in this connection." But, he believed "this distinction to be purely psychological." For Russell, the difference between extension and intension is quantitative, not qualitative: the intension is only needed because one cannot write down infinite classes.

However, the distinction between the two is qualitative as well. Already in the 3rd century, Porphyry of Tyre noted the difference between *genus* and *species* in his *Isagoge* [3, pp.135,258].[2] More recently, the *Logique de Port-Royal* (Antoine Arnault et Pierre Nicole, 1662), distinguished *compréhension* and *étendue*, and Leibniz retained these terms. Leibniz called the "*comprehension* of an idea the attributes which it contains and which cannot be taken from it without destroying it." He called "the *extension* of an idea the subjects to which it applies" [3, p.259].

Rudolf Carnap (1891–1970), in *Meaning and Necessity*, made explicit the connection between Leibniz's possible worlds and the intension-extension duality. He began with a first-order system S_I with standard connectives and quantifiers. A *state description* is a class of sentences in S_I "which contains for every atomic sentence either this sentence or its negation, but not both, and no other sentences." The description "obviously gives a complete description of a possible state of the universe of individuals with respect to all properties and relations expressed by predicates of the system. Thus the state-descriptions represent Leibniz' possible worlds" [6, p.9].

Carnap then distinguished between *truth* and *L-truth*. *Truth* simply means truth with respect to a specific state description. *L*-truth means truth in all state descriptions. Using the conventions that two *predicators* (predicate symbols) have the same extension if, and only if, they are equivalent and the same intension if, and only if, they are *L*-equivalent.

To give an overall semantics to this process, Carnap stated that "an assignment is a function which assigns to a variable and a state-description as arguments an individual constant as value" [6, p.170]. As a result, an intension becomes a mapping from the state-descriptions to its extensions. According to Dowty [8, p.145], Carnap's "intension is nothing more than all the varying extensions (denotations) the expression can have, put together and *organised*, as it were, as a function with

[1] Nevertheless, he too used logic to participate in theological arguments. In his *Essais de Théodicée sur la bonté de Dieu, la liberté de l'homme et l'origine du mal* (1710) he stated that notwithstanding the many evils of this world, "We live in the best of all possible worlds." François Marie Arouet de Voltaire (1694–1778) replied in his *Candide ou l'optimisme* that "If this is the best of all possible worlds, what then are the others?"

[2] These terms are kept in today's classification of life: *genus* being a taxonomic grouping of organisms and containing several *species*, the latter designating a group of organisms capable of interbreeding.

all possible states of affairs as arguments and the appropriate extensions arranged as values."

Saul Kripke (1940–) went further and developed possible world semantics for modal logic, in which the possible worlds are indices. "The main and the original motivation for the *possible world analysis* — and the way it clarified modal logic — was that it enabled modal logic to be treated by the same set theoretic techniques of modal theory that proved so successful when applied to extensional logic. It is also useful in making certain concepts clear" [13, p.19].

For Kripke, a possible world is "a little more than the miniworld of school probability blown large" [13, p.18]. Kripke does not attempt to define a "complete counterfactual course of events" arguing further that there is no need to do so: "A practical description of the extent to which the 'counterfactual situation' differs in the relevant way from the actual facts is sufficient." This will prove particularly useful later on when describing and formalising only the changes to the context (and not the entire context) in a running system, changes that need "broadcasting."

According to Dowty [8, p.145] "with the advent of Kripke's semantics for modal logic (taking *possible worlds* as indices), it became possible for the first time to give an unproblematic formal definition of *intension* for formalised languages." Soon after, Richard Montague (1930–1971) created his *intensional logic*, culminating in his paper "The Proper Treatment of Quantification in Ordinary English" [34].

All this was generalised by Dana Scott in his 1969 paper "Advice on Modal Logic" [30]. He assumed a nonempty set I of reference points that do not require any accessibility relation, just like in possible worlds semantics. The truth value of a sentence ϕ at a particular point is the *extension* of ϕ at that point. The intension of ϕ is an element of 2^I, a function that maps each point i to the extension of ϕ at i. A (unary) *intensional operator* is a function mapping intensions to intensions, i.e., an element of $2^I \to 2^I$. Bill Wadge quoted the following prescient passage:

> This situation is clearly situated where I is the context of time-dependent statements; that is the case where I represents the instants of time. For more general situations one must not think of the $i \in I$ as anything as simple as instants of time or even possible worlds. In general we will have
>
> $$i = (w, t, p, a) \tag{6}$$
>
> where the index i has coordinates; for example w is a world, t is a time, $p = (x, y, z)$ is a (3-dimensional) position in the world, a is an agent [*this is 1969!*], etc. All these coordinates can be varied, possibly independently, and thus affect the truth of statements which have indirect references to these coordinates.

The simplicity of Scott's approach is directly applicable to computing. It has been used, and continues to be used, with success for the development of the Lucid languagei [26]. In this article, we take the idea and apply it to systems whose structure and behaviour change as the context in which they are placed changes.

3 Possible-Worlds Versioning

In 1993, Bill Wadge and author Plaice published an article entitled "A New Approach to Version Control" [28], which applied the use of possible-world semantics[3] to the creation of software families. We summarise the results here.

3.1 Introduction

In his 1974 seminal paper on software configuration [23], David Parnas described a need for software families. In his vision, a family should contain many different pieces of software, all slightly different. These are now called *variants*.

[3]Originally, the term used for this work was called *intensional versioning*. However, in the software configuration community [7], intensional versioning has become a consecrated term for any form of automatic variant selection, but without any reference to possible-worlds semantics. For this reason, we now refer to *possible-worlds versioning*.

Wadge and Plaice simply took the intuition of *possible world*, in which there is a *complete state of affairs*, and assumed that in each such world, there was a version of every component and every system. From this intuition, it follows naturally that a software family is an intension and the individual variants are extensions.

However, a basic software engineering principle is to have a single canonical copy of every entity and to avoid duplicate entities (i.e., their variants) in order to prevent unnecessary branching, a very error-prone process. The question remained: How could possible-world semantics be used to provide maximum sharing of code across the variants?

The solution provided was to define a *uniform version space*, shared by all components of a system, and to define thereon a partial order defined over that space. All components are understood to vary conceptually across the entire version space. Physically, on the other hand, any given component may only come in a very limited number of versions, thereby avoiding the unrealistic supposition that the latest version has been developed for each and every component.

When a specific version of the system is to be built, then the *most relevant*, or *best-fit* version, with respect to the partial order, of each component is selected for the build process. This approach is called the *variant substructure principle*. By default, the refined versions inherit from coarser versions; this is called *version inheritance*.

Components here mean not only pieces of codes, but also any other software (or hardware) component, such as splash screens, drop-down menus, build files, configuration files, documentation, i.e, whatever might constitute part of the final deliverable.

3.2 Basic Definitions (1)

The definitions in this section and in section 4.2 do not correspond exactly to the actual definitions used in the articles being summarised. Rather, the key ideas of *context*, *versioned object* and *best-fit version* are being defined so that the discussion below can be understood.

Definition 1 *A* context $\kappa \in \mathbb{K}$ *is a mapping*

$$\kappa \quad ::= \quad (d : v)^+$$

where d *is a* dimension *and* v *is a* value.

To access the value associated with a dimension d, we write $\kappa(d) = v$. If $\kappa(d)$ is undefined, we write $\kappa(d) = \bot$. The empty context is written ϵ. The domain of κ, written dom κ, consists of the set of dimensions for which $\kappa(d) \neq \bot$.

Definition 2 *Context* κ *is* less refined *than context* κ'*, written* $\kappa \sqsubseteq \kappa'$*, when* $\forall d \in \text{dom } \kappa, \kappa(d) = \kappa'(d)$.

Definition 3 *Contexts* κ *and* κ' *are* consistent *if* $\forall d \in \text{dom } \kappa \cap \text{dom } \kappa', \kappa(d) = \kappa'(d)$.

Definition 4 *The* join *of two consistent contexts* κ *and* κ'*, written* $\kappa + \kappa'$*, is the union of* κ *and* κ'.

Definition 5 *Let* $\mathbb{A}$ *be a set of objects. A* versioned object $\mathcal{A}$ of type $\mathbb{A}$ *is a set of pairs* $\mathcal{A} = \{(\kappa_1, \alpha_1), \ldots, (\kappa_n, \alpha_n)\} \subseteq \mathbb{K} \times \mathbb{A}$.

The domain of $\mathcal{A}$, written dom $\mathcal{A}$, is the set $\{\kappa_1, \ldots, \kappa_n\}$. If $(\kappa, \alpha) \in \mathcal{A}$, we write $\mathcal{A}(\kappa) = \alpha$.

Definition 6 *Let* $\mathcal{A}$ *be a versioned object and let* κ_{req} *be a context. Then the* best-fit version of $\mathcal{A}$ with respect to the *requested context* κ_{req} *is* $\big(\kappa_{\text{best}}, \mathcal{A}(\kappa_{\text{best}})\big)$*, where:*

$$\kappa_{\text{best}} = \max\{\kappa \in \text{dom } \mathcal{A} \mid \kappa \sqsubseteq \kappa_{\text{req}}\}$$

The version tag is kept alongside the component in order to be able to compute the version tag of higher-level components, as seen below.

3.3 Experiment: Software Configuration

Plaice and Wadge validated this approach by taking a C programming environment, Sloth [29], and adding versions transparently to create Lemur. With Lemur, creating new variants of a piece of software was radically simplified, in comparison to the standard methodology at the time.

The Sloth system is used to build C *modules* and each is held in a directory. A number of component files within the directory are used to automatically generate a C file named `prog.c` for that module. Any component file can come in several versions, each encoded as a *tag* at the filename level in the filename. When a version of the module is requested, the most relevant version of each component file is chosen. In the end, the actual built version of `prog.c` is the least upper bound of the chosen versions of the component files.

A Sloth C module can import other modules using an *import list*. This import list can itself be versioned, using the mechanisms described above. But it is possible to go further: for each named component in the import list, a new requested version may be attached. For each of these components, Sloth proceeds with the new requested version, and returns the best possible version of the built subsystem. The versions of these subsystems then contribute to the version tag of the whole system.

We present a cleaned-up version of Lemur. A system contains a number of directories at the same level, and each directory contains one module, whose name is that of the directory. In each directory, there are three kinds of file:

- `import_`κ are tagged import list files;
- `header_`κ`.i` are tagged header files;
- `body_`κ`.i` are tagged body files.

Each import file contains a sequence of names of other directories. When version κ_{req} of module m_0 is requested, files `prog_`κ_{m_i}`.c` and `dep_`κ_{m_i} are created in each of a set of directories $\{m_0, m_1, \ldots, m_n\}$, as follows.

Let m be a requested module and κ_{req}. Then there will be:

- a best-fit import list file `import_`κ_{i_m}
- a best-fit header file `header_`κ_{h_m}`.i`
- a best-fit body file `body_`κ_{b_m}`.i`

If file `import_`κ_{i_m} is empty, then $\kappa_m = \kappa_{i_m} + \kappa_{h_m} + \kappa_{b_m}$, and file `dep_`$\kappa_m$ contains:

```
#include header_κ_{h_m}.i
```

and file `prog_`κ_m`.c` contains:

```
#include dep_κ_m
#include body_κ_{b_m}.i
```

The file `dep_`κ_m will be used by modules importing module m.

If file `import_`κ_{i_m} is non-empty, then there will be a list of modules $p_1, \ldots, p_r$ in that file. For each p_j, files `dep_`κ_{p_j} and `prog_`κ_{p_j}`.c` are created. Then $\kappa_m = \kappa_{i_m} + \kappa_{h_m} + \kappa_{b_m} + \sum_{j=1..r} \kappa_{p_j}$. File `dep_`$\kappa_m$ merges the r dependency files `dep_`κ_{p_j} and adds at the end:

```
#include header_κ_{h_m}.i
```

File `prog_`κ_m`.c` contains:

```
#include dep_κ_m
#include body_κ_{b_m}.i
```

Once all of the files `prog_`κ_m`.c` are created, then they must be compiled and linked together. The Lemur system worked remarkably well, with the caveat that a system could only include a single version of each module that was included. This was not so much a restriction of the versioning process but, rather, of the fact that the C code would have had to be created dynamically to ensure that there would be no name clashes for different versions of the same module.

3.4 Discussion

Possible worlds versioning tremendously simplifies configuration of any system (or of parts of it) in which the components have high variability. Configuration of the system at any requested instant (i.e., at any specific version) is automatic, since version κ of a compound entity is the result of combining version κ of each of its parts. A system can be an application with many components, whose version at any time can be assembled without human interaction. And since the configuration is demand-driven, along with version inheritance, it is possible to have very large version spaces without tying up resources by creating and storing unwanted versions. Version inheritance makes the uniform version space practical, since it allows different versions of the system components to share code and data transparently.

This section emphasised software families, where the components and subcomponents are mainly software objects, whose version space is limited. As should already be clear from the discussion, the components are not limited to software objects and the final system can be anything, even Web pages. Assuming appropriate syntax and run-time support, any entity can be versioned and *adapt itself* to any context, changing its internal structure as the context it resides in changes.

4 Navigation through Possible Worlds

In the mid-1990s, Bill Wadge and his students Taner Yildirim, Gordon Brown and Monica Schraefel applied the ideas of possible worlds versioning to the navigation of the Web and to the creation of families of interlinked Web pages [37, 38, 4, 16]. We summarise the results here.

4.1 Introduction

The original article on possible worlds versioning, as discussed in the previous section, was published in March 1993, when the World Wide Web was becoming widely known. At that time, most Web sites consisted of purely static pages, all produced manually. However, there were a few innovative sites that did things differently: By using clickable images and other dynamic forms of linking, pages could be generated on the fly with new or modified information. These generated pages were really "families of pages," equivalent to the families of software just described.

These pages were reproducing a phenomenon little known outside specialised circles at the time: an electronic document is *recreated* every time it is viewed. Just as the philologist claims that a text is different every time it is read because each reader's approach creates a new reading context, the electronic document is different every time it is viewed because it will be rendered for a specific viewing context. It is therefore natural to consider an electronic document to be an intension, and that the extensions are the particular renderings.

What is new with the Web and other hypertext environments is that one is *continually* changing the context. Every click on a button, a clickable region or a hyperlink creates a new context, and the page must be changed accordingly. Furthermore, hyperlinks to neighbouring pages must be regenerated, because the neighbours are all slightly different, to take into account the new context.

This new situation is more dynamic than that in the previous section. One is no longer building isolated entities for a specific possible world but, rather, building entities for a new requested world, given an existing current world, where the new world can be accessed by changing the values of some of the parameters, or dimensions, defining the current world.

4.2 Basic Definitions (2)

The key new concept defined here is the *context delta*, used to manipulate contexts, and the *delta-versioned object*.

Definition 7 *A* context delta $\delta \in \mathbb{D}$*, or* delta *for short, is an operator for changing contexts:*

$$\delta ::= \kappa \mid \overline{\kappa} \tag{7}$$
$$\overline{\kappa} ::= (d : \overline{v})^{+} \tag{8}$$
$$\overline{v} ::= \texttt{clear} \mid \texttt{set}(v) \tag{9}$$

When $\delta = \kappa$*, it is* absolute*, and when* $\delta = \overline{\kappa}$*, it is* relative*. The application of* δ *to* κ_0*, written* $\kappa_0\delta$*, is given by:*

$$\delta = \kappa: \quad (\kappa_0\delta)(d) = \kappa(d)$$
$$\delta = \overline{\kappa}: \quad (\kappa_0\delta)(d) = \begin{cases} v, & \overline{\kappa}(d) = \texttt{set}(v) \\ \bot, & \overline{\kappa}(d) = \texttt{clear} \\ \kappa_0(d), & \textit{otherwise} \end{cases}$$

Definition 8 *Let* $\mathbb{A}$ *be a set of objects. A* delta-versioned object $\overline{\mathcal{A}}$ of type $\mathbb{A}$ *is a set of pairs* $\overline{\mathcal{A}} = \{(\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n)\} \subseteq \mathbb{D} \times \mathbb{A}$*.*

Delta-versioned objects are a generalisation of versioned objects. The domain of $\overline{\mathcal{A}}$, written dom $\overline{\mathcal{A}}$, is the set $\{\delta_1, \ldots, \delta_n\}$. If $(\delta, \alpha) \in \overline{\mathcal{A}}$, we write $\overline{\mathcal{A}}(\delta) = \alpha$. Best-fitting with delta-versioned objects takes place with respect to a current context, by creating a versioned object first.

Definition 9 *Let* $\overline{\mathcal{A}} = \{(\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n)\}$ *be a delta-versioned object and let* κ_{cur} *be a context. The* versioned object $\mathcal{A}$ generated from $\overline{\mathcal{A}}$ in the current context κ_{cur} *is given by:* $\mathcal{A} = \{(\kappa_{\text{cur}}\delta_1, \alpha_1), \ldots, (\kappa_{\text{cur}}\delta_n, \alpha_n)\}$*.*

4.3 Experiments: Intensional HTML

Taner Yildirim designed the original IHTML as part of his MSc work [38] by adding *versioned links* to HTML. These links are analogous to the versioned imported components in Sloth modules: they specify a version to use on the file specified by the URL. The versioned links of an IHTML1 page are of the form:

$$\texttt{<a href=}\textit{URL}\texttt{?}\ s_1 = v_1\ \texttt{\&} \ldots \texttt{\&}\ s_n = v_n\texttt{>} \tag{10}$$

where s_i are string-valued dimensions and the associated values v_i are scalars, either strings or integers.

The IHTML1 implementation was done entirely using CGI, passing the context encoded in the URL to a Perl script. The context becomes an associative array mapping dimensions to values. When the Web page is requested for a particular context, the server converts the IHTML page into an ordinary HTML page, rendered on the fly and adapted to the current context (specified in the URL). If one of the versioned links in the rendered page is followed, then the new context will be the current context *modified* by replacing the values of dimensions that are designated in the versioned link. Unlike in Sloth import lists, links are *relative* to the current context.

As an example, Yildirim produced a family of Web pages in which one can vary background and foreground color as well as presentation language (English or Turkish). The implementation was quite simple and the family of Web pages was reachable through a fixed main page. Nevertheless, the ideas proved fruitful.

Gordon Brown took this work and for his own MSc redesigned and reimplemented IHTML as a plug-in for the Apache server under Linux, and created IHTML2 [4]. This version allowed for far more complex manipulation of the context in a page because he changed the server-side processing, implementing a module called httpd for the Apache Web server.

Brown's robust implementation made possible some ambitious sites, the most complex (at 4 million pages) being the French sentence builder designed by Bill Wadge and his wife, Christine Wadge, who teaches in the French Department of the University of Victoria, Canada.

At the markup level, IHTML2 includes versioned forms of the HTML tags referring to other documents or files: `a`, `img`, `form` and `frame`, each allowing attributes `version` (absolute delta) or `vmod` (relative delta). These links are called *versioned hyperlinks*.

For anchor tags, the `href` attribute becomes optional, and can be used to link to a different version of the same page. For example:

```
<a vmod="language=French">
```

is a link to the current page, where the current context has been modified so that the `language` dimension is set to `French`. On the other hand

```
<a version="language=French">
```

links to the same page, with a completely new context. In both cases, when this anchor is clicked upon, the requested delta δ_{req} is applied to the current context κ_{cur} to create a new requested context $\kappa_{\text{req}} = \kappa_{\text{cur}}\,\delta_{\text{req}}$.

Unlike in the Lemur software configuration case, IHTML2 also provides the opportunity to change the *structure* of a Web page according to the context. There are two additional tags. The first is `iselect`:

```
<iselect>
  <icase vmod="κ1"> text1 </icase>
  ...
  <icase vmod="κn"> textn </icase>
</iselect>
```

(Of course, `versions`'s could also be present.) If the requested context is κ_{req} and an `iselect` construct is encountered, a corresponding versioned object is created from this delta-versioned object, and then only the code of the *best-fit* `icase` is chosen. The second construct, `icollect`, is similar:

```
<icollect>
  <icase vmod="κ1"> text1 </icase>
  ...
  <icase vmod="κn"> textn </icase>
</icollect>
```

(Of course, `version`'s could also be present.) If the requested context is κ_{req} and an `icollect` construct is encountered, a corresponding versioned object is created from this delta-versioned object, and the code of all of the acceptable `icase`'s is chosen, maintaining a stable order.

A single IHTML source file can specify a whole multiversion family of Web pages. This was different from HTML, in which a link could point just to a unique page. Nowadays, most Web programming is done through scripts, but these are still *not* intensional programs: they do manage *some* variance, but not arbitrary dimensionality.

In IHTML, the link points to a *family* of pages, all held in a single source, and the HTML page produced is calculated based on the current context on the fly. The resulting page is a *version of the Web page*, giving hypertext a new *dimension*. This way of constructing the rendered Web page can be understood as having all the components of the page (images, sections of text, local links, plug-ins, scripts) react simultaneously to the change of the context.

For her PhD work, Monica Schraefel [16] used the IHTML infrastructure so that the dimensions did not correspond to technical attributes but, rather, to more subjective parameters that a reader of a novel might be interested in. She created a multidimensional document discussing *Wuthering Heights*, where the dimensions corresponded to level of detail, perspective, bibliography format, which characters were to be examined, and so on.

4.4 Discussion

These developments led Bill Wadge, author Plaice and their colleagues to study more carefully the seminal works on hypertext by Vannevar Bush and Ted Nelson. Bush wrote in 1945 [5] about the *Memex*, a machine in which one can read documents at different speeds or with different levels of detail. Nelson, who coined the term *hypertext* in 1965, also invented the terms *stretchtext*,

plytext and *poptext*, where, depending of the actions of the user, text looks different. By using the standard intensional programming technique that any problem can be solved by adding a new dimension,[4] Nelson's stretchtext, plytext and poptext are all easily implemented.

Nevertheless, further examination of Nelson's writings [21] showed a significant difference between his view of hypertext and the intensional view. Nelson envisages that any reader can take bits and pieces of other documents to produce a hypertext system with two-way links (to ensure strict copyright control). For him, a hypertext family of documents consists solely of juxtaposing extensions. His family of documents is not an intension, but a collection of extensions. So as concluded in [21], "Hypertext, if it is to be meaningful, can only mean *intensional hypertext*."

At the same time, Yannis Stavrakas, Manolis Gergatsoulis and Panos Rondogiannis continued with the approach of intensionalising markup languages. They developed Multidimensional XML [31], along with an intensionalised version of XSLT. In the long run, their work should be highly relevant, but currently there is no distribution of any software based on their ideas.

5 Remote Navigation across Possible Worlds

In the late 1990s, under the direction of Bill Wadge, Paul Swoboda developed a full context-aware sequential programming language, in which every aspect, including data structures, control structures and functions, was context-aware [32]. We summarise the results here.

5.1 Introduction

Possible-worlds versioning allowed the creation of structures that were context-dependent. Possible-worlds navigation allowed the creation of context-dependent structures and their automatic recreation as one moved from one context to another. The next step was to integrate these ideas directly into a programming language and to add therein context-dependent mechanisms:

- Context-dependent variables, functions and blocks of code.
- Context deltas upon entry of blocks of code.
- Context deltas upon calling functions.

The mechanisms required for adding these features are similar to those described in the previous section. The additional difficulty is that when a best-fit block of code or a function body is to be executed, the system must determine what should be the new running context. It turns out that there are several choices.

5.2 Experiment: Intensional Sequential Evaluator

The Intensional Sequential Evaluator (ISE) language developed by Paul Swoboda is a fully versioned scripting imperative language whose syntax was inspired by Perl4, along with the references (pointers) of Perl5. When an ISE program is called, an execution context is initialised either from an environment variable called `VERSION` or from a command line argument `--version=`*version*. Every aspect of the program execution is based on the context: interpretation of variables, functions, control flow, system calls, etc. During execution, the current context can be modified by applying a delta to it.

In ISE, assignments are versioned, as in:

```
$<lgIn:en>createMapValue = "Create Map";
$<lgIn:fr>createMapValue = "Créez la carte";
$<lgIn:es>createMapValue = "Crear el mapa";
$<>createMapValue        = $<lgIn:en>createMapValue;
```

[4] A variation on Butler Lampson's well known dictum that any problem in computer science can be solved with an extra level of indirection.

`createMapValue` is defined in four versions: the English, French and Spanish of the language interface dimension, and the default version, defined to be the same as the English version.

Deltas follow the scoping rules of the program structure, and can be restricted to individual blocks. Hence, in:

$$\texttt{do } \delta \texttt{ \{}B\texttt{\}} \tag{11}$$

the delta δ is applied to the current context before entering block B to create a new current context; upon exit, the previous context is restored. Similarly, in:

$$\texttt{while (}C\texttt{) } \delta \texttt{ \{}B\texttt{\}} \tag{12}$$

so long as the conditional expression C is true, block B is executed with delta δ. Slightly more complicated is:

$$\begin{array}{l}\texttt{if (}C_0\texttt{) } \delta_0 \texttt{ \{}B_0\texttt{\}} \\ \texttt{elif (}C_1\texttt{) } \delta_1 \texttt{ \{}B_1\texttt{\}} \\ \ldots \\ \texttt{else } \delta_n \texttt{ \{}B_n\texttt{\}}\end{array} \tag{13}$$

If C_i is the first true expression, block B_i is executed with delta δ_i.

The most complicated structure in ISE is the function call. Like variables, functions — but not their types — can be defined in several versions. This means that there are several function definitions, each tagged with a version, and that the function calls are themselves versioned. The form of a function call is:

$$f \textit{ modifier } \delta_{\text{req}}\, \delta_{\text{exec}}\, (\textit{args}) \tag{14}$$

where the *modifier* is empty, `?`, or `!`. If we suppose that the current context is κ_{cur}, then in all three cases, the best-fit version of the function is chosen: $(\kappa_{\text{best}}, f_{\text{best}})$. The three different modifiers allow one to determine whether the best-fit body of the function, f_{best}, should be evaluated by applying δ_{exec} to:

- the current context κ_{cur} (empty),
- the requested context $\kappa_{\text{req}} = \kappa_{\text{cur}}\, \delta_{\text{req}}$ (`?`), or
- the best-fit context κ_{best} (`!`).

As a scripting language, ISE was well suited for CGI programming. ISE was used in the first attempts to building a mapping server with a Web interface in an intensional environment ISE [17, 18, 19]. ISE introduced, for the first time, contexts as first-class values. Contexts can be assigned, stored, passed as arguments to functions or even to other processes, and so on. Swoboda [32] used the same runtime system to build VMake and iRCS, versioned variants of make and rcs, as well as to reimplement IHTML so that they all use the same version space.

5.3 Experiment: More Programming Languages

Wadge developed a markup language with troff syntax called Intensional Markup Language [36]. IML macros are used to generate ISE programs, which, given a multidimensional context, will produce the appropriate HTML page. Using IML notation, the source for multidimensional Web pages becomes substantially more compact than using IHTML or raw ISE.

The idea of versioning commonly used programming languages was studied by two undergraduate student groups at UNSW (Australia) under author Plaice's supervision. Ho, Su and Leung [12] developed an approach to versioning C. The Linux Process Control Block was adapted so that each process had a version tag, and system calls were created for manipulating this tag. Versioned C functions were implemented using function pointers. Balasingham, Ditu and Hudson [1] experimented with versioning of C++, Eiffel and Java. With these developments, it became clear that versioning could be applied to all sorts of different programming tools.

6 The Evolution of the Possible Worlds

In 2003, in his PhD thesis [33], supervised under Plaice and Wadge, Swoboda created the *æther*. The æther is an active context which, upon being changed, broadcasts the relevant changes to registered participants. We summarise the results here.

6.1 Introduction

The previous three sections showed progressively more sophisticated ways in which a single entity could adapt itself to a single context. But what about the context itself? What is its nature?

Originally, the context was viewed simply as a useful mechanism to hold on to user preferences. But there is another, very fruitful view: the context is *immersive*, or, in other words, the possible world is a *medium* permeating the program. When the context is changed, then like the water passing through our cells, every last subcomponent is affected by the change. As Marshall McLuhan said, "The medium is the message."

This *plenist* viewpoint — as opposed to the *atomist* viewpoint implicit in object-oriented programming — becomes most useful when we consider that *several* objects might share the same context, and that all of these objects are themselves capable of adapting to and modifying the context. In this *intensional community*, a term coined by author Plaice and Kropf [15, 24], the context can be used as a means for broadcasting to all of the other objects sharing this context.

With the *æther*, the active, explicit context, the intensional community can be distinguished from a multi-agent system, which assumes that the agents are simply communicating between themselves through a vacuum, in a point-to-point manner. In an intensional community, programs can communicate either directly, through some communication channel, or *indirectly*, via the context. Changing the context or some part thereof is equivalent to a radio broadcast: all those listening hear; those not listening hear nothing.

An æther contains a context and a set of active participants, each registered at some point in the æther's context. When the æther is modified, deltas are set to all the participants.

6.2 Experiment: ISE Æthers

The original æther was developed by Swoboda as a networked server to which ISE programs could connect in order to manipulate named contexts. The server being multithreaded, one ISE process could wait for one of these named contexts to be changed. Should another process change that context, then the waiting process was immediately notified of the new value for the context.

This infrastructure was illustrated with a Web page supporting collaborative browsing [27]. An ISE script was written to present some of the best known paintings of the Louvre, the famous French museum. The result was an intensional Web page with five dimensions: text detail level, text language, image size, painting school and painting reference number. This simple interface was much more intuitive to use than the original Louvre Web page upon which it was based. An additional dimension (to follow, with possible values yes or no) was added to allow collaborative browsing. By adding a single line to the page-generation script, and writing a 20-line wrapper ISE script, people browsing this site could choose to follow what someone else is viewing, while maintaining their own personal preferences, forming in practice an intensional community. However, the ISE æther was more a container for contexts, rather than an active entity.

6.3 Experiment: libintense

For his PhD thesis, Swoboda developed a complete suite of programming tools to support æthers as active contexts. These tools include: libintense, an industrial-quality body of code in C++ and Java; and libaep, which supports a new AEPD using active æthers and that allows processes to transparently access æthers over a TCP/IP network. In addition, he extended this basic infrastructure in a number of different directions, including a Perl interface and ijsp, an extension to the Java libintense for the building of intensionalised JSP pages served by an Apache Tomcat server.

The æther is a reactive machine containing a context, and the interface is set up as a subclass of context. Therefore, deltas are applied to æthers in the same manner that they are applied to contexts, with the added side effects of *notifying* the participants of the appropriate deltas. Any process can register a *participant* at a specific node. A participant is a piece of code that is executed when the æther's context is modified at that node or below. Upon context change, the participant is executed, being passed a single argument, namely its relevant delta.

At each instant t, an æther contains:

- κ_t, the current context of the æther;
- P_t, the current set of active participants;
- $D_{p,t}$ (for each participant $p \in P_t$), the set of dimensions being listened to by participant p.

At instant $t+1$, one, and only one, of the participants may undertake an action:

- p.`connect`(D):
  - $\kappa_{t+1} = \kappa_t$;
  - if $D = \emptyset$, then $P_{t+1} = P_t - \{p\}$;
  - if $D \neq \emptyset$, then $P_{t+1} = P_t \cup \{p\}$ and $D_{p,t+1} = D$.
- p.`apply`(δ):
  - $\kappa_{t+1} = \kappa_t\, \delta$;
  - for each $p \in P_t$, a delta $\delta_{p,t+1} = \delta | D_{p,t}$ is generated, where $\delta | D_{p,t}$ is the restriction of δ to $D_{p,t}$.

6.4 Experiment: Anita Conti Mapping Server

As part of her PhD thesis [20], author Mancilla developed the Anita Conti Mapping Server (ACMS), which provides an intensional Web page, itself containing a intensional map, the two varying with separate context. Each context can be manipulated separately, and can be sensitive to an æther, thereby allowing both the Web page and the map being shown to be changed through the actions of another user, or, conversely, local changes can be broadcast to others. As a result, both the interface and the content of the Web page can be shared with other users.

The ACMS software is interesting because of the nature of its *content*. *A map* is intrinsically and naturally a multidimensional entity because it depends on a large set of parameters or dimensions of many kinds. We consider an electronic map to be an intension, and that the specific maps generated on demand are the extensions. However, we believe that intensional maps are in some sense more complex than the intensional documents described before, simply because a map is a *visual presentation* of a context — part of the globe's surface — along with certain features linked to that space. Unlike text or software, maps are not representing a sequence of discrete entities like words, characters or glyphs, but, rather, a continuous physical area: a map is not a graph. Magnification of the image should bring in new data from new sources, as appropriate, and in so doing, influence the production of the map, its coloring, contents, frame, titles and labels.

The ACMS was the first full-fledged piece of software that included the `libintense` libraries. Demonstrations given in many different settings were very well received. Nevertheless, the server could not scale, not because of lack of resources but, rather, because there was no means for defining the aggregate semantics of an æther and a set of registered participants.

7 Future Research: Synchronised Possible Worlds

The problem outlined in the previous section essentially comes down to unclear timing issues. When there are multiple components sharing the same permeating context, and the components

can—in addition to communicating indirectly through the context—communicate directly among themselves, the possibilities for livelocks, deadlocks and other such undesirable phenomena is enormous.

One possibility is to give a completely synchronous semantics to the interaction between æther and participants. Each "instant" would be broken into two:

1. The æther would receive a delta, possibly empty, from each participant, and would then merge these deltas into a single delta, which would then be applied to its context. Should there be inconsistent delta for certain dimensions, then some resolution mechanism would have to be applied, possibly an error, a no-op, or a combination of the different deltas. The æther would then send the relevant deltas to all of the participants.

2. The participants would receive their respective deltas, adapt accordingly, and then proceed to undertake their activity, possibly communicating directly among themselves. Once this activity is finished, then each would send its delta to the æther.

This idea could be extended so that several communities could be hooked up together, all using the same rhythm of alternating æthers/components in action. The intuition is simple, but the details are not: certain dimensions might be controlling critical resources, and would require an appropriate protocol. Similarly, clash resolution needs to be addressed.

8 Conclusions

This special issue focuses on the work undertaken by Bill Wadge or by people he has influenced in topics as diverse as descriptive set theory, logic programming, programming language semantics, hybrid logics, game theory, and other formal work. But Bill's work has always been guided by real, practical problems. In this paper, we have focused on his interest for providing a possible-world semantics for software configuration management, and the surprisingly rich and unexpected consequences of this — on the surface — mundane research topic.

When the original possible-worlds versioning article was submitted to the *IEEE Transactions on Software Engineering*, two of the reviewers made sarcastic comments that Plaice and Wadge wished to do *science fiction*, in order to have spaceships move between possible worlds! In hindsight, we can see that the science-fiction vision of possible worlds has been key in the creation of very useful software.

It is precisely with visions of future possible systems that the ideas described in this paper have been developed. For example, the original article on possible-worlds semantics put forward the idea of a *Montagunix*, a fully versioned Unix-like system; although this idea is now partially available with the development of various flexible Linux distributions, the building process of the latter is clearly not as flexible as the vision of Montagunix.

Similarly, the idea of the intensional communities was created by Plaice and Kropf when they were working on the idea of the *Web Operating System* (WOS), a distributed operating system where the behaviour of every component, and their interaction, was fully versioned.

We believe that with the rise of context-aware computing, mobile computing, pervasive computing and ubiquitous computing, that the relevance of possible-worlds versioning will be increasingly felt. We need simple but powerful techniques for adaptation of both the structure and behaviour of software.

References

[1] Varunan Balasingham, Gabriel Ditu, and Simon Hudson. Intensionality and the object-oriented paradigm. Honours Thesis, Computer Science, UNSW, 2001.

[2] I. M. Bocheński. *Ancient Formal Logic*. North-Holland, Amsterdam, 1951.

[3] I. M. Bocheński. *A History of Formal Logic*. University of Notre Dame Press, Notre Dame, Indiana, 1961. Translated from the German *Formale Logik*.

[4] Gordon D. Brown. Intensional HTML 2: A practical approach. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1998.

[5] Vannevar Bush. As we may think. *Atlantic Monthly*, July 1945.

[6] Rudolph Carnap. *Meaning and Necessity*. University of Chicago Press, 1956. Seventh Impression 1975.

[7] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[8] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics*. D. Reidel, Dordrecht, Holland, 1981.

[9] Doug Engelbart. Personal communication, 2004.

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

[11] Manolis Gergatsoulis and Panos Rondogiannis, editors. *Intensional Programming II*, volume II. World Scientific, Singapore, 2000. Based on the papers at ISLIP'99.

[12] Lapyu Kenneth Ho, Ian Su, and Patrick Leung. Multidimensional Linux library system. Honours Thesis, Computer Science, UNSW, 2001.

[13] Saul A. Kripke. *Naming and Necessity*. Harvard University Press, Cambridge, Mass., 1980.

[14] Peter Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors. *Distributed Communities on the Web*, volume 1830 of *Lecture Notes in Computer Science*. Springer, Berlin, 2000. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[15] P.G. Kropf and J. Plaice. WOS communities - Interactions and relations between entities in distributed systems. In *Distributed Computing on the Web, DCW'99*, pages 163–167, Rostock, Germany, June 1999.

[16] m. c. schraefel. *Talking to Antigone*. PhD thesis, University of Victoria, Victoria, Canada, 1997.

[17] Blanca Mancilla. A new approach to on-line mapping. In *Proceeding of IWHIT'00, Seventh International Workshop on Human Interface Technology 2000*, pages 19–23. University of Aizu, Aizu-Wakamatsu, Japan, November 2000.

[18] Blanca Mancilla. A new approach to interactive mapping. In Michitake Hirose, editor, *Human-Computer Interaction — INTERACT'01*, pages 644–647. IOS press for IFIP, 2001.

[19] Blanca Mancilla. Omega for multilingual mapping. In Bolek and Przechlewski, editors, *Proceedings of XIII European TEX Conference April 29–May 3, 2002, Bachotek, Poland*, pages 60–63. GUST, 2002.

[20] Blanca Mancilla. *Intensional Infrastructure for Collaborative Mapping*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2004.

[21] m.c. schraefel, Blanca Mancilla, and John Plaice. Intensional hypertext. In Gergatsoulis and Rondogiannis [11], pages 40–54. Based on the papers at ISLIP'99.

[22] Eric McLuhan and Frank Zingrone, editors. *Essential McLuhan*. House of Anansi Press, Toronto, 1995.

[23] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[24] John Plaice and Peter Kropf. *Intensional communities*, pages 292–295. Volume II of Gergatsoulis and Rondogiannis [11], 2000. Based on the papers at ISLIP'99.

[25] John Plaice and Blanca Mancilla. Collaborative intensional hypertext. In *HYPERTEXT 2004, Proceedings of the 15th ACM Conference on Hypertext and Hypermedia, August 9-13, 2004, Santa Cruz, California, USA*, pages 91–92. ACM, 2004.

[26] John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Mathematics in Computer Science*, This volume, 2008.

[27] John Plaice, Paul Swoboda, and Ammar Alammar. Building intensional communities using shared context. In Kropf et al. [14], pages 55–64. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[28] John Plaice and William W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–276, March 1993.

[29] John Plaice and William W. Wadge. A UNIX tool for managing reusable software components. *Software—Practice and Experience*, 23(9):933–948, September 1993.

[30] Dana Scott. Advice on modal logic. In Karel Lambert, editor, *Philosophical Problems in Logic*, pages 143–173. D. Reidel Publishing Company, Dordrecht, Holland, 1970.

[31] Yannis Stavrakas, Manolis Gergatsoulis, and Panos Rondogiannis. Multidimensional XML. In Kropf et al. [14], pages 100–109. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[32] Paul Swoboda. Practical languages for intensional programming. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1999.

[33] Paul Swoboda. *A formalization and implementation of distributed Intensional Programming*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2003.

[34] Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[35] William W. Wadge. *Intensional logic in context*, pages 1–13. Volume II of Gergatsoulis and Rondogiannis [11], 2000. Based on the papers at ISLIP'99.

[36] William W. Wadge. Intensional Markup Language. In Kropf et al. [14], pages 82–89. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[37] William W. Wadge, Gordon D. Brown, m.c. schraefel, and Taner Yildirim. Intensional HTML. In E.V. Munson, C. Nicholas, and D. Wood, editors, *Principles of Digital Document Processing*, volume 1481 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[38] Taner Yildirim. Intensional HTML. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1997.