VMAKE AND ISE
GENERAL TOOLS FOR THE INTENSIONALIZATION OF
SOFTWARE SYSTEMS

P. SWOBODA AND W. W. WADGE
Department of Computer Science, University of Victoria, Victoria, BC, CANADA
E-mail: swoboda@uvic.ca, wwadge@esc.uvic.ca

Many intensional programming systems that have been developed thus far provide
a strong means of applying versioning to specific areas of programming, but lack
generality of application. We propose a system for applying run-time versioning
directly to the semantics of common scripting languages, enabling efficient con-
struction of tools whose behaviors vary under context.

Vmake is a rule-based make program that allows its identifiers to be versioned,
namely rule names and file names. The rule tree for vmake is identical to that of
traditional make programs, with the addition that every node is merely an inten-
sion, including the underlying (leaf-node) files. With a syntax that is similar to
most other make tools, vmake allows the arbitrary packages to be intensionalized
at the time of compilation, with no restriction as to the nature of the source lan-
guage, development tools, or compilation process.

ISE is an imperative scripting language with a syntax similar to that of Perl. As
with vmake, all identifiers in ISE are intensions, referring to all possible versions
of the identified entity; conversely, anything that can be identified can also be
defined under multiple versions, including all variables and functions. In addition
to providing the mechanisms for the intensionalization of standard language
constructs for flow control and data representation, ISE provides the means to
precisely control intension resolution across function-call boundaries, and to ma-
nipulate versions as arbitrary expressions. The resulting intensional-imperative
language can be used to build a wide range of scripted, intensional tools.

1 A Practical Versioning System

For use in the ordinary software, a runtime versioning system has to be
fast, versatile and accessible. To these ends, a system has been devel-
oped for vmake and ISE, which extends the system developed by Gord
Brown, in his implementation of IHTML2. Brown's system assumes a ver-
sion to be a collection of string-valued (identifier) dimensions and their
string-valued indexes, or values. An empty version is termed "vanilla" (de-
noted e) and dimensions and their values are syntactically separated by
colons. dimension-value pairs are separated by the symbol "+". This
scheme is adequate for most software-versioning purposes, but does not al-
low for the grouping of dimensions with common prefixes or semantics.
For example, the dimensions "haircolour" and "hairlength" could be bet-
ter described as the sub-dimensions “colour” and “length” of the dimension “hair”. The new versioning scheme allows precisely this, and treats all dimension values as sub-versions. For example, a version using the same dimensions could be written as hair: (colour: green+length: 5), or as hair: colour: green+hair: length: 5. This system treats a complete version as the value of an implicit, base dimension, and allows each dimension in a version to have a single “base-value”, corresponding to the dimension values of Brown’s system, as well as any number of other sub-dimensions and their values. The nesting of sub-dimensions can be carried to an arbitrary depth and allows for a fairly comprehensive means of versioning software. Also, an entire version is allowed a single base-value, for symmetry:

\[
\begin{align*}
\text{version-string} & : \rightarrow \text{dimension-list} \\
\text{version} & \rightarrow \\
& \rightarrow \text{dimension-pair} \\
& \rightarrow "(\" \text{dimension-list} \")" \\
\text{dimension-list} & \rightarrow \text{dimension-pair} \\
& \rightarrow \text{dimension-pair} \ "+" \text{dimension-list} \\
\text{dimension-pair} & \rightarrow \text{ID} \\
& \rightarrow \text{ID} \ "::" \text{version}
\end{align*}
\]

1.1 Version Refinement and Intensional Best-Fits

Refinement is a concept used by Lemur and IHTML, where it was defined as a partial ordering of versions, denoted \( \subseteq \). The expression \( V_1 \subseteq V_2 \) is pronounced “\( V_1 \) refines to \( V_2 \)” or “\( V_2 \) refines \( V_1 \)”, and means that \( V_1 \) is a similar, but more generic, version of \( V_2 \). Refinement under Brown’s IHTML was defined as follows, and can be applied directly to versions in the new system, using an expanded form:

\[
\begin{align*}
\epsilon & \subseteq V \\
V & \subseteq V \\
(V_1 \subseteq V_2) \land (V_2 \subseteq V_3) & \rightarrow V_1 \subseteq V_3 \\
d_1 : i_1 \subseteq d_1 : i_1 + V_2 \\
m, n \in \mathbb{Z}^+, m \leq n & \rightarrow d : m \subseteq d : n
\end{align*}
\]
As used in IHTML, a best-fit algorithm provides a means of determining a unique, most-refined (best-fit) version \( V_b \) from a set \( S \) of versions defined with the new grammar, given a requested version \( V_r \). \( V_b \) must not refine to any other version in \( S \), unless that version does not refine to \( V_r \):

\[
V_i \in S \land V_i \subseteq V_{req} \rightarrow V_i \subseteq V_{best}
\]

2 Vmake

2.1 Introduction

Vmake is a make language where the rule names, like the files beneath them, can be defined in multiple versions; whenever a rule is executed as a dependency of some other target, an intensional best-fit is made, based on current context, to determine which version of the named rule or file will be used. The resulting intensional tree represents all possible ways to build a target. At all points in its execution, vmake maintains a “current version”, being the context under which best-fits are made. This version can be altered across rule-resolution boundaries (i.e. differing levels of the make tree), so that different levels of rules can behave differently, and differing versions of the same rule can be used in the same make.

Vmake was originally intended to be a versioned file-server, for use under IHTML-driven web sites, but has since progressed to a full make language for general software construction. The following sections describe a subset of its functionality.

2.2 Version Expressions

Versions in vmake are denoted as angle-bracket delimited strings, with the same content-syntax as described in section 1. So-called version modifiers (the same as in IHTML2) are denoted with square-brackets, and use the current execution-version of vmake as the base version, to which the expression’s contents are added as a vmod. Thus, the “current version” is always available in the empty-vmod expression, [].
2.3 Variables and File-Inclusion

Like other make utilities, vmake allows the assignment of dependency-lists and strings to variables, which can then be used in the dependency-list and command portions of rules, or in the definition of other variables.

Vmake can perform run-time, versioned file-inclusion and execution. This is effected with standard include- and exec-statements, including the content of (or the output from) the argument file as vmake code. Precise control of file versioning is allowed and the argument file may be contained in an ircs archive, causing a best-fit version of the file to be accessed.

2.4 Rule Definitions

Rules in vmake use the following general syntax:

\[ \text{rule-name version-list: dependency-list} \]
\[ \text{shell-commands} \]

where the rule’s version-list can be empty (in which case its single version is vanilla), and each element of the dependency-list consists of the dependency name, followed by a possible list of explicit versions. If the dependency-name is preceded by a “?”, the dependency will be run under the requested version (possibly explicit); if it is preceded by a “!”, it is run under the best-fit version. By default, a rule is executed under the current-version of the rule that invoked it, preserving global context throughout the make-tree.

2.5 The vmake2daVinci Utility

The daVinci directed-graph visualization system has a straightforward interface language for describing arbitrary, directed graphs. This language is used by the vmake2daVinci tool to generate a concise tree describing node-names and requested- and best-fit versions for each node in a given vmake tree. The tool takes a target name and an optional version-switch (as with vmake) as command-line arguments, and using the same source-file as vmake, generates the graph output on stdout. This output can then be viewed directly with the daVinci application.
3 ISE

3.1 Introduction

ISE, for "Intensional Sequential Evaluator" is an interpreted language with a syntax very similar to that of Perl. Unlike in Perl, which is a normal imperative scripting language, all identifiers and flow-control structures in ISE are versioned. This includes all variables and functions, as well as non-identified blocks of code and common loops and conditional-statements. A best-fit algorithm is used to resolve intensions at run-time, so that one section of code can behave differently, under varying context.

The following sections describe only a small portion of the available functionality in ISE, but should serve to give a general idea of the language.

3.2 Version Expressions

Versions in ISE follow the same format as those used in vmake for pure versions and version modifiers, with the exception that versions in ISE can be arbitrary expressions. If the content of a version or vmod expression matches the regular expression for version-string literals, the entire expression is processed into a version-datastructure at parse-time. A few examples are as follows:

- `<green+a:1>`
  A version literal, evaluated at parse-time.

- `[generate_vmod()]`
  A version modifier; the returned value from the function generate_vmod() is processed as a version and added as a vmod to the current version, to yield the value of the expression.

At any point during an ISE program’s execution, there are three main contexts available to, and governing the actions of, the user’s code. Those are the current version, always available as the vanilla vmod ([]), the requested version for the currently-executing routine (?), and the best-fit version for the currently-executing routine (!).
3.3 Variables

Loosely following the syntax of Perl, variables in ISE have the identifying type character ($, @, %, *, for scalars, arrays, hashes, and version-hashes, respectively), followed by possible version-expressions, followed by the variable's identifier. Used as an lvalue, a variable expression can have no versions, in which case the variable is assigned-to under the current version, or with one or more explicit versions, in which case all the given versions of the variable are assigned the same rvalue. Used as an rvalue (or in an arbitrary expression), a variable can have no versions, in which case the best-fit to the current version of the variable is used, or some explicit version expression, to which a best-fit is also made.

Unique to ISE is the version-hash (vhash) type, which is similar to Perl's associative-array hash-type, but maps versions-expressions to values by intensional best-fit, rather than using strings as keys. This uses the syntax *vhash_id for entire vhashes and $vhash_id<version expr> for accessing individual elements. When combined with the power of references (as element values), this type allows the construction of some fairly complex, intensional datastructures.

Some examples of variable expressions and their uses are as follows:

- `$s1 = "Hello World!\n";`
  Assignment of a string literal to the current version of the scalar variable s1.

- `$<green><red>a1[5] = $<green>s1;`
  Assignment of the contents of the best-fit version to "green" of the scalar s1, to the element with key 5, of both the green and red versions of the array a1.

- `[$purple]vh1<a:1> = $h1{"abc"};`
  Assignment of the contents of the element with key "abc" of the best fit to the current version of the hash h1, to the element with the (best-fit) version-key "a:1" of the vmod-version "purple" of the vhash vh1. (quite a mouthful!).
3.4 Function Definitions

Unlike most other scripting languages, functions in ISE are defined with formal arguments, under a fixed number of version expressions. Function definitions are stripped out of the parse tree, prior to execution, allowing functions to be called in the user’s source, before the location of the definition. The following is an example of an explicitly versioned function definition:

```plaintext
# Versions <hot> and <cool> of function f1():
function f1<hot><cool>($arg1, $arg2)
{
    print("In function f1!\n");
}
```

3.5 Function Calls

Because function calls represent an important, contextual boundary in the language, ISE provides a mechanism for precisely controlling the context of execution over function calls. The syntax for calls is of the general form:

```plaintext
filename<requested-version><execution-version>(arg1, ...)
filename!<requested-version><execution-version>(arg1, ...)
filename??<requested-version><execution-version>(arg1, ...)
```

where either the execution version or both version expressions can be omitted. By default, the function is executed under the context of the caller, but inclusion of the characters ? and ! cause the function to be executed under the requested or best-fit versions, respectively. If an explicit execution-version is given, a ? or a ! causes that version to be evaluated under the requested or best-fit versions.

3.6 References and Intensional References

Like versions 5.0 and greater of Perl, ISE has the capability to take scalar-valued references to variables, functions, blocks of code, and evaluated expressions, allowing the construction of arbitrary datastructures. ISE also allows so-called intensional references to be taken to versioned entities, whereby dereferencing causes a best-fit to be performed, to an appropriate version of the referred-to entity. Dereferencing function-references (intensional or not)
causes the referenced function to be called, possibly with arguments and with
the same call-versioning syntax as regular function-calls. The same is true of
references to blocks of plain code. In addition, all variable (de)references may
be used as lvalues, with the same assignment-versioning semantics as regular,
variable-expression lvalues. Readers are referred to the online ISE manual 6
for a detailed description of reference syntax.

3.7 Versioned Flow Control

Most traditional, imperative flow-control constructs are represented in ISE,
with the addition that controlled blocks can take an optional, explicit block-
version. This expression is evaluated before the block is executed (and in
the case of loop-blocks, prior to each execution of the block), and is used
as the current version during the block's execution. If a block version is
omitted, the current version used (as if the version were an empty vmod).
The following example shows a foreach loop with a contextual modifier (the
expression [$version]), where each of the three versions in the argument
array is used in the context-modifier:

```plaintext
foreach $version ({{ <a:1>, [green], <blue> }}) [$version] {
    print("context: ", [], ", ", $version: ", $version, ".\n");
}
```

If the context prior to this loop were, say, "red", the loop would print the
following:
```plaintext
context: <red+a:1> $version: a:1
context: <green> $version: green
context: <blue> $version: blue
```

3.8 The Vsswitch Expression

A convenient means of applying intensionality to a set of blocks of code in
ISE is the vsswitch expression, which allows the run-time construction of a
best-fit version-hash, mapping version expressions to blocks of code. The
vsswitch expression has the same type of "call" syntax as function calls, so
precise control over execution-, requested- and best-fit versions is provided.
The general form of the statement is as follows:
```plaintext
vsswitch<requested-version><execution-version> {
    version-list { block }
```
version-list { block }

...
is the Intraspeak 7 French-sentence generator of Bill and Christine Wadge; in "pure" IHTML, this relatively complex page used file-inclusion to build its components from 1000 lines of code in 40 IHTML source-files and took anywhere up to several seconds to execute. A reformulation of this site in a single, 500-line ISE script now takes only 0.2 seconds to execute, producing identical results. The script can be viewed on-line at the following URL:

http://i.csc.uvic.ca/intra32.ise

Context is inherited by ISE from the IHTML server; the current-, best-fit and requested-versions are initialized for all CGI scripts in environment variables, based on the version given in the request URL and the existing versions of the script file. Indeed, context is passed from ISE and vmake in the same manner, whenever they execute sub-processes; in this manner, these intensional tools can be combined, to perform complex versioning tasks.

References

   http://i.csc.uvic.ca/ise_docs/index.html, University of Victoria, Canada (1999).