Meta: Extending and Unifying Languages

Wade Holst
Department of Computer Science
University of Western Ontario
London ON Canada
wade@csd.uwo.ca

ABSTRACT

$\mathcal{M}eta$ is an ambitious research project whose overall purpose is to increase the utility and expressive power of a wide range of existing languages. $\mathcal{M}eta$ provides augmented versions of existing languages and guarantees support for aspects, components, language interoperability, visualization, reflection, various inheritance models, and many other extensions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, constraints, inheritance, modules, patterns, polymorphism*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

General Terms

Languages, Design, Documentation

Keywords

object-oriented, aspect-oriented, component-oriented, XML, multi-methods, reflection, language interoperability

1. INTRODUCTION

$\mathcal{M}eta$ is a large research project whose primary goal is to augment and unify the syntax and semantics of existing languages. Although $\mathcal{M}eta$ provides features useful for arbitrary languages, this material focuses primarily on the support provided by $\mathcal{M}eta$ to object-oriented programming languages. This support consists of a collection of mandates: ensuring support for a standard set of object-oriented features, support for aspect-oriented and component-oriented programming, support for multi-method dispatch and other inheritance-related extensions, providing (optional) runtime typing-checking in non-statically-typed languages, numerous statement-level syntactic extensions, support for language interoperability, support for automated 2D and 3D visualization of programs (UML, etc.), and a much higher degree of reflection than is available in most languages, to name only a few. $\mathcal{M}eta$ is designed to support and extend a very wide range of object-oriented languages, and the set of supported languages are referred to as the *base languages* of $\mathcal{M}eta$. Extended versions of these base languages are defined by $\mathcal{M}eta$, with names like $\mathcal{M}eta\langle$Java$\rangle$, $\mathcal{M}eta\langle$C++$\rangle$ and $\mathcal{M}eta\langle$C#$\rangle$.

Language neutrality is a central concept in $\mathcal{M}eta$; the mandates of $\mathcal{M}eta$ are implemented in as language-neutral a manner as possible. Such implementations do not rely on the details of a particular base language, and have the obvious advantage of providing a particular capability in all base languages at the same time. This means that adding a new base language to $\mathcal{M}eta$ requires significantly less work than would otherwise be the case.

```
CLASS Person SCOPE<C++> {
  FIELD height : float;
  FIELD weight : float;
  INITIALIZER (h:float,
               w:float) {
    this->setHeight(h);
    this->setWeight(w);
  };
  METHOD bmi() : float {
    float w = getWeight();
    float h = getHeight();
    return w/(h*h);
  };
} RUN {
  Person * p =
    new Person(1.8,85);
  cout << p->bmi()
       << endl;
};
ASPECT PerAsp SCOPE<C++> {
  before ADVICE () :
    "call(*.b*():float" {
    cout << "Starting"
         << endl;
  };
};
```

```
host% metac -b cc -c
host% metacpp Person
```

```
CLASS Person SCOPE<Java> {
  FIELD height : float;
  FIELD weight : float;
  INITIALIZER (h:float,
               w:float) {
    this.setHeight(h);
    this.setWeight(w);
  };
  METHOD bmi() : float {
    float w = getWeight();
    float h = getHeight();
    return w/(h*h);
  };
} RUN {
  Person p =
    new Person(1.8,85);
  System.out.println(
    p.bmi());
};
ASPECT PerAsp SCOPE<Java> {
  before ADVICE () :
    "call(*.b*():float" {
    System.out.println(
       "Starting");
  };
};
```

```
host% metac -b java -c
host% metajava Person
```

Figure 1: Sample Meta$\langle$C++$\rangle$ and Meta$\langle$Java$\rangle$ Code

The language-neutral philosophy of $\mathcal{M}eta$ extends even to its syntax, as shown in the small Meta$\langle$C++$\rangle$ and Meta$\langle$Java$\rangle$ programs in Figure 1. $\mathcal{M}eta$ introduces a concise, uniform, powerful syntax (and associated semantics) that replaces the high-level syntax of existing languages. In its most common form, $\mathcal{M}eta$ acts as a language template, providing only a partial definition of the syntax of a language. It is when $\mathcal{M}eta$ syntax is applied to an existing object-oriented language like C++, Java, C#, etc. that a fully-defined language ($\mathcal{M}eta\langle$C++$\rangle$, $\mathcal{M}eta\langle$Java$\rangle$, Meta$\langle$C#$\rangle$) is formed. $\mathcal{M}eta\langle$C++$\rangle$ uses $\mathcal{M}eta$ syntax above the level of statements, and C++ syntax at and below statement level. $\mathcal{M}eta\langle$Java$\rangle$ uses $\mathcal{M}eta$ syntax above the level of statements, and Java syntax at and below statement level. The high-level syntax of $\mathcal{M}eta\langle$C++$\rangle$ is identical to that of $\mathcal{M}eta\langle$Java$\rangle$ and any other $\mathcal{M}eta\langle$L$\rangle$ language.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2. BENEFITS OF META

There are two levels at which the contributions and benefits of $\mathcal{M}eta$ can be discussed. First, at the *programmer level*, the augmentation and unification of existing base languages has a significant software engineering impact, substantially increasing the ease with which a development team can design and implement applications.

Second, at the *research level*, $\mathcal{M}eta$ is a vehicle for exploring new object-related technologies (like aspects, components, multi-methods, automated visualization, etc.) and identifying synergisms between these concepts.

The following subsections briefly describe a few of the benefits provided by $\mathcal{M}eta$. For a more exhaustive list of benefits, the reader is referred to [1].

2.1 Programmer-level Benefits

Augmented Languages: The most immediately benefit provided by $\mathcal{M}eta$ is the augmentation of the base languages its supports. $\mathcal{M}eta\langle$C++$\rangle$ provides numerous extensions on top of `C++`, $\mathcal{M}eta\langle$Java$\rangle$ significantly extends `Java`, etc.

Concise Syntax: A $\mathcal{M}eta\langle$L$\rangle$ source file is almost always smaller than its corresponding base language `L` source file. A $\mathcal{M}eta\langle$L$\rangle$ program says more with less.

Reduced Learning Curves: The structured nature of $\mathcal{M}eta$ syntax allows the important concepts (constructs) to be quickly understood, and allows the details (attributes) to be incrementally learned as needed. Furthermore, if a programmer already knows $\mathcal{M}eta\langle$L$\rangle$, learning $\mathcal{M}eta\langle$L'$\rangle$ is easier than learning `L'` (only statement-level syntax needs to be learned).

Implicit code: Each $\mathcal{M}eta$ field results in numerous accessors being defined. Every object can be serialized and printed. Every class has an associated entry point and test harness.

Popularizing New Features: Many new object-oriented concepts are not seen by programmers because such features are usually added into new languages (and programmers rarely see experimental new languages). By providing an environment that augments and extends popular existing languages, $\mathcal{M}eta$ can act as a mechanism for introducing concepts like aspects, components, multi-methods, reflection, etc. to a much wider audience. For example, $\mathcal{M}eta$ provides a language-neutral meta-object protocol that provides for convenient, fine-granularity, intuitive introspection on each atomic constituent of every high-level syntactic entity in the base language, as well as efficient, convenient, intuitive execution-style intercession on meta-objects. The interface provided by $\mathcal{M}eta\langle$L$\rangle$ is almost always more concise and efficient than that provided by `L`, and the same interface exists for every base language supported by $\mathcal{M}eta$. As another example, $\mathcal{M}eta$ provides wide-ranging support for intra-process (as opposed to inter-process) language interoperability issues. This includes support for addressing the legacy code problem, using multiple languages in the same application, hiding the esoteric details of base-language native code implementations, and allowing rapid prototyping in a development language followed by rapid migration of source-code to a production language.

Many other extensions: Multi-methods, mixins, interfaces, Beta-style inheritance, Smalltalk-like class variables, and many other features are available to $\mathcal{M}eta\langle$L$\rangle$ programmers, regardless of whether base language `L` provides direct support for them.

2.2 Research-level Benefits

Extending Mandates: $\mathcal{M}eta$ has numerous mandates, like support for aspects, components, multimethods, etc. Initial implementations of these mandates are often built from existing base-language implementations, appropriately generalized where possible to provide language-neutrality. However, many of these mandates are still developing, and $\mathcal{M}eta$ can play an active role in establishing how they should evolve. For example, although the aspect-oriented paradigm is maturing, numerous extensions are possible. Even more significant, the component-oriented paradigm, although anticipated for decades, has only very recently made significant progress towards truly independent software components that can be swapped out and replaced as easily as hardware components. There are a small number of existing component-oriented languages and component frameworks available, but the paradigm is very far from being mature. $\mathcal{M}eta$ can play an active role in component-oriented research.

$\mathcal{M}eta$ and XML: The structure of $\mathcal{M}eta$, including its self-defining properties, are similar in nature to those of XML, and both $\mathcal{M}eta$ and XML benefit greatly from the implement-once-use-often nature of their design. However, $\mathcal{M}eta$ can also be used as a vehicle for XML and XSLT related research. To see why, note that the implementation of aspects in $\mathcal{M}eta$ currently relies on converting $\mathcal{M}eta$ syntax to XML and using XSLT to implement the aspect weaving, followed by additional XSLT to create base-language source code. Numerous strengths and weaknesses of XSLT as a production-level compiler have been identified during this implementation. Future research will be looking at whether a $\mathcal{M}eta$-level language ($\mathcal{M}eta\langle$XML$\rangle$) can augment XML (and XSLT) by addressing these limitations. One immediate benefit would be a significant reduction in syntactic burden (a problem with XML and XSLT).

Synergism between mandates: Although all of the above research contributions are significant in and of themselves, the most interesting research-level contribution of $\mathcal{M}eta$ is in the identification and use of synergisms between the various $\mathcal{M}eta$ mandates. For example, although there are many research projects addressing aspects, or components, or multi-methods, etc., there are no research projects looking at all of them at the same time. $\mathcal{M}eta$ is designed to provide an environment in which such interactions can be explored. Numerous synergisms have already been identified (augmented reflection helps implement almost all other mandates, design pattern support may be implementable using aspects, language interoperability is significantly influenced by $\mathcal{M}eta$ syntax).

3. CONCLUSION

$\mathcal{M}eta$ is an ambitious project spanning a large number of research areas related to software engineering and programming language design and implementation. The existing $\mathcal{M}eta$ compiler has varying levels of support for C++, Java and Perl, and numerous other languages and features will be aded as the research develops.

Details on $\mathcal{M}eta$, including papers discussing individual languages (Meta$\langle$C++$\rangle$, Meta$\langle$Java$\rangle$, etc.) and individual mandates (reflection, aspects, components, language interoperability, etc.) can be found at:

`http://meta.csd.uwo.ca/Meta`