BEST Names for Semantic Units to Support Reproducible Modeling

Laurence Loewe and Seth A. Keel
Laboratory of Genetics and Wisconsin Institute for Discovery, University of Wisconsin-Madison
loewe@wisc.edu, skeel@discovery.wisc.edu

The ability to understand the meaning of identifiers in the source code of programs is important for reproducible research that aims to apply existing methods to new contexts. An ideal name easily conveys to most readers the core meaning of an identifier without disturbing the flow of reading by triggering the need to look up the meaning or taking up too much space. These constraints reflect a systemic tension between beginners (who prefer speaking names to minimize interrupts caused by looking up names) and experts (who often prefer brevity for faster coding). A similar tension surfaces in simulation models, where the same semantic units have a full technical name that is rarely used and often abbreviated in different ways: semi-speaking names in code are complemented by extremely brief names in mathematical equations of associated research papers. Associated manual conversions can be error prone and substantially add to the effort required to reproduce the corresponding research.
Here we present an approach to naming that addresses this problem. It reduces the inherent tensions between the Brief and Speaking names required by different audiences and can substantially reduce the effort required to avoid naming conflicts by automatically checking for their occurrence. It allows programmers to give four types of names to each semantic unit: a "**Brief**" Name for equations and experts with a good memory, an "**Explicit**" Name for programmers who prefer some reminder of relevant semantics, a "**Speaking**" Name that makes the code more readable for beginners, and a "**Technical**" Name that provides a potentially complex, but technically precise name of the semantic unit. In addition to the main first entry to each of the **Brief-Explicit-Speaking-Technical** (BEST) Names, an arbitrary number of synonyms can be registered for each of these dialects. All respective BEST Names and their synonyms are mapped to a Unique Semantic Unit that can also be accessed directly, can be linked to a stable ontology, and shall use Semantic Versioning for making it easy to track semantic changes in the code as sources evolve. The approach presented here is currently being implemented in Evolvix, a new programming language designed to make rigorous modeling easier. While the BEST Names approach is general and can be implemented in many programming languages without the need for modifying the language itself, maximal modeling ease and efficiency is easier to achieve by designing BEST Name abilities directly into the core of a language.

Keywords: naming, variable names, function names, namespaces, model reuse, semantic versioning, semantic stability, unique semantic unit, reproducibility, readability of source code

Introduction

If the ability to access and execute code is one of the first steps towards reproducible research, then the next step is the ability to understand what the code does. This is essential for properly applying and reusing existing code in new contexts. Understanding code consists of (i) understanding the syntax and semantics of the programming language the code is written in and (ii) understanding the purpose and meaning of the variable, function and module names of the code. Many programming languages are documented in more detail than most programs written in them, so (i) appears less problematic – assuming that language documentation remains accessible and is semantically rigorous. But unless the implementers of a program invest additional effort for writing documentation, it can be difficult to determine the precise meaning of variable and function identifiers.

reproducibility@XSEDE workshop, 2014-07-14,
XSEDE14 Annual Conference, Atlanta, GA, USA
https://www.xsede.org/web/reproducibility

Here we present an approach to improve a reader's understanding of the meaning of those identifiers. The approach specifies some of the infrastructure necessary for such documentation and for ensuring it remains in a consistent state. A core concept of this approach is to provide "BEST Names" for important identifiers. These names map Brief, Explicit, Speaking, and Technical synonyms to a unique semantic unit at the root. Automated checks can ensure the consistency of these names and hence reduce the time modelers need to compare various versions of models under investigation.

The Problem

BEST Names were developed to solve naming difficulties in the following use-cases that all revolve around the dilemma of choosing longer, more readable names vs shorter, more cryptic names:

1. **Beginners vs Experts**
   Often, both beginners and experts need convenient access to the same underlying semantic units. Beginners and infrequent users

often prefer names that include more information about the semantics of an identifier to make it easier to understand code. Experts or specialists who work with the same concepts frequently tend to abbreviate or rename them in the interest of coding speed – even at the risk of making it practically impossible for outsiders to understand them. Both, speed and readability are legitimate concerns in naming, but they can rarely both be satisfied at the same time if identifiers can carry only a single name.

2. **Code vs Math**
   Ideally, naming follows the priorities and culture of the environment the names are used in. Thus, names in source code will ideally reflect at least some aspects of their semantics, even if abbreviated. Names in mathematical equations are usually reduced to single letters, often with subscripts, to maximize compactness and reduce anything that might interfere with a quick grasp of the mathematical structure of a given equation. These conventions create a tension when the same model is described and analyzed in code *and* equations. Working with complex models frequently requires converting from one set of names to another or actively working with both, which is inherently error prone. Different modelers have different strategies for reducing conversion errors that create problems for reproducibility, but the manual work involved could be much reduced by providing automated support (e.g. for detecting inconsistencies).
3. **Compare Model Implementations**
   Sometimes independent researchers generate different codes that intend to implement essentially the same model, yet using different names. In the absence of automated support, a precise comparison of the different names in these codes can cost substantial amounts of time for complex models.
4. **Combining different models with interactions**
   Sometimes independent researchers generate different models for different aspects of a bigger problem and later decide to combine these into a single bigger model. For example, one group may model one biochemical pathway in a yeast cell, whereas another group may model another pathway, in the same yeast. To study interactions between both pathways in the same cell these models will need to be thoughtfully combined, such that (i) molecules that are different in yeast cells get different names in the combined model, even if they had the same name in their original, more limited model (eg.: if 'Km1' refers to one molecule in Model A and to another in Model B, it should get mapped to different names) and (ii) molecules that are the same in yeast cells get the same name, even if they had different names in the original models (eg. the identifiers 'H2O' and 'Water' should map to a common identity).
   While it is easy to adjust these identifiers manually in small models, this process becomes increasingly cumbersome and error prone in larger models, so automated support is desirable.
5. **Documentation of a foreign code base**
   While venturing into the code for a new model, it is often useful to be able to annotate the meaning of identifiers in a way that improves the quality of that code, but only if there is a guarantee that the code itself will not be affected.

Scientific software is often implemented in contexts that provide little time for documentation, so good name choices for identifiers in code become even more important. However, as noted [1,2]:

> *There are only two hard things in computer science: cache invalidation and naming things.* *Phil Karlton*

Incidentally, both problems are related, especially in research, where (i) multiple researchers often work together for extended periods of time before the nature of a new phenomenon becomes clear enough to give it a non-confusing name (requiring many potentially confusing updates to local 'brain-caches' along the way). Also, (ii) the same discovery can be made by more than one researcher, as often seen in biology; the resulting different names create more 'cache-invalidation' work for future nomenclature committees. While there is little that can be done about these fundamental difficulties of naming, we submit to the reader that the stellar memorizing, sorting, and comparing capabilities of computers might be used to make the task easier for us humans. While the Integrated Development Environments of modern compilers substantially simplify the navigation of unknown codebases, we see room for innovation, potentially improving the transfer of information from an implementer's brain to the brain of someone trying to understand or reproduce their work. We next present an approach that can ease the naming problem. The following design has the Semantic Version 0.7.0 (see [3]) and uses keywords *must* = *shall* = *required*, *should* = *recommended* and *may* = *optional* to indicate requirement levels as specified in RFC2119 [4]. We capitalize words with special roles in our system.

BEST Names

An ideal name conveys to readers the essence of a semantic unit and allows them to maintain their train of thought by avoiding what might disturb their flow of reading, such as the need to look up meanings or excessive visual clutter from explanations. Readers at differing levels of familiarity with the underlying concepts will require different meta-information for code to maintain its readability. These conflicting demands are impossible to meet with one representation, as long as there is a diverse user base. Thus, we have given up on that goal. Instead,

we will use 'BEST Names' that explicitly enable communication between diverse audiences by allowing them to use an appropriate *Dialect* corresponding to their natural usage and topic familiarity. As all Dialects *must* [4] unambiguously map to identical semantic roots there *must* be no doubt about what a particular identifier might mean at any given point in time. Key Dialects define (see Fig.1):

Brief Names for experts or specialists who use certain functions so much that they value the option to memorize more and type less. The Brief Dialect values brevity over readability. An effort *should* be made to use common abbreviations, associations and mnemonics, but that only goes so far in a world, where TLAs (Three-Letter-Acronyms) have multiple meanings. All BEST Names value semantic unambiguity in their defined namespace above all other features. Examples: math equations, Perl1liner.

Explicit Names are for programmers who prefer something that is both short *and* memorable. They do not need introductory information as they are past the beginner's stage and now look for more efficient coding. Any Explicit Naming scheme *should* exploit existing opportunities for combinatorial regularity as much as possible to make learning Explicit Names easier. It *should* combine the strengths of Brief and Speaking Names. Examples: many Python libraries.

Speaking Names are for beginners who often need to learn related underlying concepts as well. Thus Speaking Names *must* be as didactic, precise, and explanatory as possible and *should* be field-tested on beginners to improve their clarity. The emphasis of Speaking Names *must* be an introductory presentation and *should not* include unnecessary jargon.

Technical Names are for technical experts and *should* reflect the name of the corresponding semantic unit in an appropriate ontology. Technical Names *should* enable a technically literate reader to quickly identify key concepts at hand, even if she does not know about details irrelevant to her problem domain (eg. syntax of an implementation language).

Synonyms can be defined in arbitrary quantities for any BEST Name, as long as they do not create naming conflicts (an identifier pointing to two different semantic units). The position of "first synonym" is special in that it *must* hold the default identifier that will be used when a representation of its semantic unit is requested in the corresponding Dialect.

Mixing *must* be possible at all times for BEST Names, such that any identifier from any Dialect can be used in any sequence that would be permitted

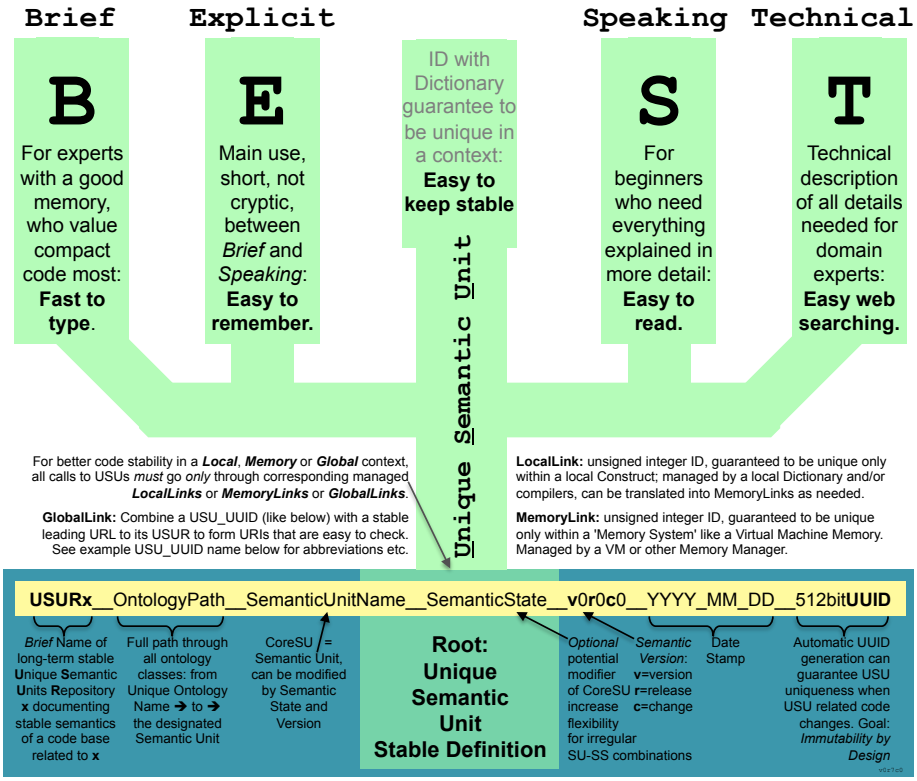

Fig.1: *BEST Names simplify naming. Dictionaries translate Names ←→ Unique Semantic Units, must use Semantic Versioning* [3]*,may use RDF,OWL*[5,6].

when using only one Dialect. The integrity of identifiers *must* be respected by any implementation in the sense that identifiers can occur as fractions of other identifiers without causing name-clashes.

Defining new Dialects for special purposes *may* be allowed, if they do not compromise other required functionality, and if names of new Dialects are unambiguous and are not 'B','E','S','T', 'Best', 'Explicit', 'Speaking' 'Technical', or an obvious variant of these.

Unique Semantic Unit (USU): represents the same meaning described by all its BEST Names. USUs can map to terms from an ontology or be defined by the local context of the code that uses them. A code base can define its own ontology by listing all its USUs and defining their meaning in associated info texts. All USUs can be linked to for quick access without BEST Names. Links can be *LocalLinks*, *MemoryLinks* or *GlobalLinks* (see below or Fig.1; think: array index).

Texts *must all* be encoded in UTF-8 UniCode. Without explicit relaxing specifications, BEST Names *must* be restricted to the first 127 UniCode code points only (avoid character swaps that are very hard to detect). USUs may provide an *OneLineSummary* and a *Details* Text which *should* be able to use the full UniCode set and *should* use ReStructuredText formatting unless explicitly specified otherwise. A One*LineSummary* briefly sums up a USU meaning to enable scanning many of them in auto-compiled lists. *Details* provide the power of reStructuredText [7] to explain as many USU details as needed.

Translating between Dialects *must* be easy for any system implementing BEST Names, such that any Synonym from any Dialect enables access to (i) all Synonyms from all Dialects, (ii) all data and functions in its USU, and (iii) at least one form of linkable ID that facilitates quick USU access for external code.

Managing BEST Names: Constructs

'Constructs' manage BEST Names and connect them to something known locally. They are defined here as any code (eg. class, namespace, database table, collection,...) that can manage a namespace and related content by providing these services:

- **translating Names** between Dialects and faster ID Links for USU using code (using a **Dictionary**)
- **checking for Name clashes**: 1 Name ➔ >1 USU
- **adding / removing Names** but not 'changing' them
- **importing a new USU**: *all* its Names must be free of clashes with *any* exiting USU Name
- **importing a known USU**: *all* its Names get added as Synonyms to its target USU *and* all *must* be free of clashes with *any* existing Names in its Dictionary
- **storing data of its USUs** in internal data structures, or by appropriate Links; data can be any *content*, *variable*, *function, code*, *type*, *message*, *context*, etc or *Link* to another *Construct* or data.
- **checking for existence** of USUs (using Dictionary)
- **providing read and write access to local USUs** as requested by outside code (via Link or Name)
- **accessing Constructs inside** and their content
- **managing access restrictions** (if any)
- **adding, removing, resetting USUs**, managing their Names and the underlying memory as needed
- **linking to outside USUs** in other Constructs using appropriate Links, when local access is required
- **calling functions** defined by USUs as requested by outside code (may be defined at compile time)
- more to follow, including bulk functions for speed.

Some of this functionality may execute at compile time, some at run-time, depending on model needs.

In short: A *Construct is a collection of USUs that manages access to them with the help of a BEST Name Dictionary and the ability to produce and interpret fast Links to its USUs. Each defines a namespace in which the semantic uniqueness of its BEST Names is guaranteed.*

Links contain a respective ID for fast access; if a Construct is too big, a NameServer or a hash-based storage scheme may be associated with the Construct to facilitate lookups of remote content. Three types of Links *shall* allow implementers of Constructs to trade scope vs speed and size:

LocalLinks: Fastest, smallest, but *only valid inside* of Constructs. Good for edges in large networks.

MemoryLinks: Fast, standard size; valid everywhere inside of a given 'Memory' (typically the RAM of a VM Memory Manager), but only for the Memory's lifetime.

GlobalLinks: Less fast, large, but guaranteed unique with some readability built in (see Fig.1). Hashes and NameServers enable speedy identification anywhere.

Unique Semantic Unit Dictionary

A dedicated generic and flexible Dictionary collection *shall* be implemented that can manage all aspects of BEST Names and alleviates Construct implementers of said generic task to help them focus on aspects of the Construct that are specific to the problem domain. Implementers *shall* at most select an appropriate Dictionary for a Construct to start benefitting from BEST Names and increased semantic stability. For consistency, availability and performance reasons, all operations at the level of a *single* Construct/Dict *must* be as atomic, complete, fast and reliable as possible.

Efficient, powerful dictionaries can be implemented in modern data structures like Ternary Symbol Tries or HAT Tries [8,9]. To reduce the cache invalidation problems of large Constructs, it is *recommended* that BEST Name Dictionaries contain only USUs that *immediately* reside in a given *Construct* and thereby exploit data locality. Dictionaries and Constructs too large for single nodes can use algorithms from distributed file systems and NoSQL DBs [eg. 10]. Different sizes of Construct namespaces require Dictionary implementations with different Link types and data structure trade-offs. Nevertheless, their core API *shall* be the same. It *should* make it easy for users to choose a trade-off by requiring minimal or no code changes beyond specifying the choice.

Future Directions

We are currently implementing the first BEST-Names prototype for Evolvix [11], a new modeling language designed to make it easy for biologists to describe the systems they study in mathematically precise terms. We are also refining the BEST Names design to remove the danger of crippling absolute names, to cleanly separate *pure* from *impure* names [12-15], as well as improve support for associated info texts, meta-data and Literate Programming [7,16]. Writing readable code is an art [17]. We believe tools like BEST Names can improve the state of this art.

Acknowledgements, References

Thanks to K. Scheuer, K. Ehlert, I. Flores-Lorca, T. Engbretson, J. Goldfinger, and others for helpful discussions of Evolvix, NSF for CAREER award 1149123 to LL, NIH Training Grant Genetics T32GM007133 for supporting SAK, and WID for general support.

1. http://martinfowler.com/bliki/TwoHardThings.html
2. http://rossduggan.ie/blog/programming/naming-things/
3. Semantic http://semver.org
4. http://www.ietf.org/rfc/rfc2119.txt
5. http://www.w3.org/RDF/
6. http://www.w3.org/OWL/
7. ReStructuredText Specification: http://docutils.sourceforge.net/rst.html
8. Sedgewick R, Wayne KD (2011) "Algorithms". TSTs: p.746-752:
9. Askitis,Sinha'07 HAT-trie:http://crpit.com/confpapers/CRPITV62Askitis.pdf
10. Knecht M, Nicola, CU (2013) A space- and time-efficient Implementation of the Merkle Tree Traversal Algorithm IMVS Fokus Report, S. 35-39
11. Loewe et al. Evolvix Home, Prototype Evolvix 0.2.0. http://evolvix.org
12. Needham RM(1993)"Names" In:"*Distributed Systems*"p315 ACM
13. Krakowiak (2009) Naming: http://proton.inrialpes.fr/~krakowia/MW-Book/
14. Pike R et al. (1995) "Plan 9 from Bell Labs." *Computing Systems*, Vol. 8 (3):221–254 http://plan9.bell-labs.com/sys/doc/9.pdf and.../doc/names.pdf
15. Dolstra E, Löh A, Pierron N. NixOS: A Purely Functional Linux Distribution. J. Functional Programming, Vol. 20 (5-6) pp 577-615 http://nixos.org/~eelco/pubs/nixos-jfp-final.pdf
16. Knuth DE, Levy S (1994) The CWEB system of structured documentation : version 3.0. Reading, Mass.: Addison-Wesley.
17. Boswell D, Foucher T (2011) "The Art of Readable Code". Sebastopol, CA, O'Reilly.