Ensuring Consistency between Designs, Documentation, Formal Specifications, and Implementations

Joseph R. Kiniry and Fintan Fairmichael

School of Computer Science and Informatics and
CASL: The Complex & Adaptive Systems Laboratory,
University College Dublin,Belfield, Dublin 4, Ireland
kiniry@acm.org and fintan.fairmichael@ucd.ie

Abstract. Software engineering experts and textbooks insist that all of the artifacts related to a system, (e.g., its design, documentation, and implementation), must be kept in-sync. Unfortunately, in the real world, it is a very rare case that any two of these are kept consistent, let alone all three. In general, as an implementation changes, its source code documentation, like that of Javadoc, is only occasionally updated at some later date. Unsurprisingly, most design documents, like those written in UML, are created as a read-only medium—they reflect what the designers *thought* they were building at one point in the past, but have little to do with the actual running system. Even those using formal methods make this mistake, sometimes updating an implementation and forgetting to make some subtle change to a related specification. The critical problem inherent in this approach is that abstraction levels, while theoretically inter-dependent, are actually completely independent in semantics and from the point of view of the tools in pervasive use. Entities in different layers have no formal relationship; at best, informal relations are maintained by ad hoc approaches like code markers, or code is generated once and never touched again. This paper presents a new approach to system design, documentation, implementation, specification, and verification that imposes a formal refinement relationship between abstraction levels that is invisible to the programmer and automatically maintained by an integrated set of tools. The new concept that enables this approach is called a *semantic property*, and their use is discussed in detail with a set of examples using the high-level specification language EBON, the detailed design and specification language JML, and the Java programming language as the implementation language.

1 Introduction

Ad hoc constructs and local conventions have been used to annotate program code since the invention of programming languages. The purpose of these annotations is to convey "extra" programmer knowledge to other system developers and future maintainers. These comments usually fall into that grey region between completely unstructured natural language and formal specification. For example, an ad hoc convention promoted by Eclipse are the `FIXME`, `TODO`, and `XXX` task tags that cause errors or warnings to appear in Eclipse's *Problems* view.

Invariably, such program comments rapidly exhibit "bit rot". Over time, these comments and the implementation to which they refer diverge to an inconsistent state (a process often referred to as erosion or drift [1]). Unless they are well maintained by documentation specialists, rigorous process, or other extra-mile development efforts, they quickly become out-of-date. They are the focus for the software engineering mantra: an incorrect comment is worse than no comment at all.

Recently, with the adoption and popularization of lightweight documentation tools in the literate programming tradition [2,3], an ecology of semi-structured comments is flourishing. The rapid adoption and popularity of Java primed interest in semi-structured comment use via the Javadoc tool. Other similar code-to-documentation transformation tools have since followed in volume, including Jakarta's Alexandria, Doxygen, and Apple's HeaderDoc. SourceForge reports dozens of projects with "Javadoc" in the project summary, and FreshMeat reports several dozen more, with some overlap.

While most of these systems are significantly simpler than Knuth and Levy's original CWEB, they share two key features.

Firstly, they are easy to learn, since they necessitate only a small change in *convention* and *process*. Rather than forcing the programmer to learn a new language, complex tool, or imposing some other significant barrier to use, these tools actually *reward* the programmer for documenting their code.

Secondly, a culture of documentation is engendered. Prompted by the example of vendors like Sun, programmers enjoy (or, at least, do not abhor) the creation and use of the attractive automatically-generated documentation in a web page format. This documentation-centric style is only strengthened by the exhibitionist nature of the Web. Having the most complete documentation is now a point of pride in some Open Source projects—a state of affairs unguessable a decade ago.

The primary problem with these systems, and the documentation and code written using them, is that *even semi-structured comments have no semantics*. Programmers are attempting to state (sometimes quite complex) knowledge, but are not given the language and tools with which to communicate this knowledge. And since the vast majority of developers are unwilling to learn a new, especially formal, language with which to convey such information, a happy-medium of *informal formality* is necessary.

That compromise, the delicate balance between informality and formality, is the core principle behind *semantic properties*, the core conceptual contribution of this paper.

Informally, semantic properties are nothing more than what today's programmers call *annotations*, and what are known as *pragmas* in the non-OO world. Javadoc tags, gcc's `#defines`, and C#'s XML comments are three examples of (potential) semantic properties. Of course, none of these specific technologies were created with semantic properties in mind—support for them is engineered into the design of semantic properties. This decision is taken to ensure that programmers do not need to learn new technologies or languages to use and take advantage of semantic properties.

More formally, semantic properties are *general-purpose* annotations that have a *domain-independent* formal semantics. For a given domain (e.g., a programming or specification language), semantic properties' semantics are mapped into the semantic domain of their application via a refinement relation. These refinement relations are *semantics preserving*—they are defined in such a way that their meaning is preserved between refinement levels.

As mentioned earlier, semantic properties look just like comments, annotations, or design notes—they are used as if they were normal semi-structured documentation. But, rather than being ignored by compilers and development environments as comments typically are, they have the attention of augmented versions of such tools. Semantic properties permit one to embed a tremendous amount of concise information wherever they are used without imposing the often insurmountable overhead seen in the introduction of new languages and formalisms for similar purposes.

Semantic properties are not a newly-invented idea. They have been used in several research groups, in the classroom, as well as in corporate settings for over a decade. We are now using them also in a software product lines setting. Over the years, semantic properties, via its underlying logic, called kind theory, have integrated several formal methods in practical, pragmatic fashion, unbeknownst to collaborators, students, and employees. In the following pages several examples of semantic properties are described. We will focus this explanation via the use of a concrete example.

2 Running Example

Our running example comes from an advanced tutorial on JML from 2006 [4]. The focus of that tutorial is the detailed specification and verification of an alarm clock using the Java Modeling Language (JML) and Java. The refinement relationship between JML and Java in this example is depicted diagrammatically in Figure 1. We extend this example to include high-level specifications written in the Extended Business Object Notation language (EBON), for which we are responsible. BON is a textual and graphical domain-independent specification language akin to UML [5], but which focuses on all software engineering stages from domain analysis to architecture to contract-centric component design. EBON is simply the BON language plus semantic properties.

To illustrate the use of semantic properties in more detail, consider the task of modeling a logical clock, like those used in concurrent and distributed algorithms.

The first stage of our verification-centric software development process involves concept identification. (A full description of this process is found elsewhere [6].) Concepts, called *classes* (as in *classifiers*) in EBON, are named, described with single sentences, and their high-level relations (*is-a*, *has-a*, and *is-a-kind-of*) are identified.

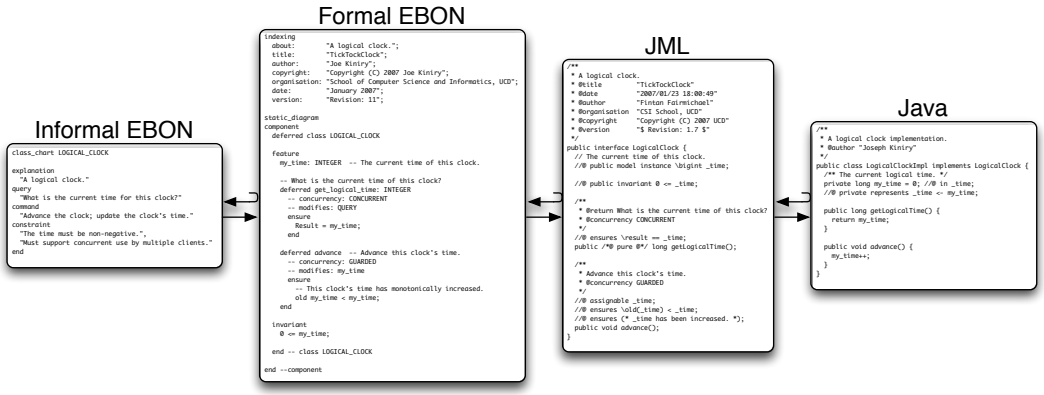

Figure 1: A diagrammatic representation of refinement from EBON to Java.

```
static_diagram CONCEPTS_AND_RELATIONS
  component
    deferred class LOGICAL_CLOCK
    deferred class ALARM
    effective class CLOCK persistent
    effective class ALARM_CLOCK persistent

    ALARM_CLOCK inherit CLOCK
    ALARM_CLOCK inherit ALARM
  end
```

Listing 1: An EBON static diagram describing the core concepts of the running example.

In this example, the concepts identified through domain analysis are *alarm*, *alarm clock*, and *logical clock*. Their relationships are summarized in the EBON static diagram `CONCEPTS_AND_RELATIONS` in Listing 1. Their definitions are elided in this example.

Each concept is summarized with an *informal diagram*. An informal diagram describes the concept and its interfaces in terms of *queries*, *commands*, and *constraints*. Queries and commands are collectively known as *features*.

For example, the logical clock must store a time value and, in EBON terminology, support a *query* to determine the current time stored in the clock. A *command* is also necessary to monotonically advance the time stored in the clock. Furthermore, a *constraint* states that the time stored in the clock is always non-negative. Finally, the logical clock must also behave correctly while being used by multiple concurrent clients.

```
class_chart LOGICAL_CLOCK
  explanation
    "A logical clock."
  query
    "What is the current time of this clock?"
  command
    "Advance the clock; update the clock's time."
  constraint
    "The time must be non-negative.",
    "Must support concurrent use by multiple clients."
end
```

Listing 2: An EBON class chart for `LOGICAL_CLOCK`.

This interface and requirements are expressed using an EBON informal chart. Like most requirement languages, informal EBON uses structured English to denote analysis concepts and requirements. The EBON class chart shown in Listing 2 captures this information.

Classical software engineering *requirements* are expressed as *constraints* in EBON. Likewise, *features* (as in formal feature models from the area of software product lines [7]) are expressed via EBON features.

```
indexing
  about:        "A logical clock.";
  title:        "TickTockClock";
  author:       "Joe Kiniry";
  copyright:    "Copyright (C) 2008 Joe Kiniry";
  organisation: "School of Computer Science and Informatics, UCD";
  date:         "January 2008";
  version:      "Revision: 11";

static_diagram
component
  deferred class LOGICAL_CLOCK

  feature
    my_time: INTEGER  -- The current time of this clock.

    -- What is the current time of this clock?
    deferred get_logical_time: INTEGER
      -- concurrency: CONCURRENT
      -- modifies: QUERY
      ensure
        Result = my_time;
      end

    deferred advance  -- Advance the clock; update the clock's time.
      -- concurrency: GUARDED
      -- modifies: my_time
      ensure
        -- This clock's time has monotonically increased.
        old my_time < my_time;
    end

  invariant
    my_time >= 0; -- The time must be non-negative.

  end -- class LOGICAL_CLOCK

end --component
```

Listing 3: An EBON formal specification for `LOGICAL_CLOCK`.

This model is refined, mapping informal specifications into something more formal and concrete, as seen in Listing 3. For example, the constraint "The time must be non-negative." is refined to an invariant of the form: $my\_time \leq 0$. A feature *time* of type *INTEGER* is defined that represents the current time of this clock. The clock's query and command are also refined into appropriate features (function types). Also, note that the concept "time" is refined to a property having the EBON type *INTEGER*, a mathematical integer.

This diagram contains uses of two semantic properties, one called *concurrency* and the other *modifies*. In order to achieve our desired concurrency property, the informal constraint is refined by annotating the features with the *concurrency* semantic property whose labels denote the concurrency semantics of their feature. The query `get_logical_time` is labelled `CONCURRENT` (multiple calls may proceed at the same time), and the command `advance` as concurrency `GUARDED` (additional calls block until the original call has completed). This specification models a standard multiple reader, single writer pattern.

Additionally, a frame condition is stated for these two features. Frame conditions specify what parts of the model may be changed when a function is invoked. These two annotations make explicit that the function `get_logical_time` is indeed a `QUERY`, and the function `advance` may only modify the value of the field `my_time`.

For the reader familiar with JML, this formal specification of `LOGICAL_CLOCK` looks syntactically familiar. A JML refinement of this EBON class is found in Listing 4.

A JML specification is a JML-annotated Java module (a class or an interface). `LogicalClock.java` contains a Javadoc-annotated Java interface which contains two methods. Each method is, in turn, also annotated with Javadoc comments. Some of the Javadoc tags are standard (e.g., `@return` and `@author`),

```
/**
 * A logical clock.  This realization uses a integral representation,
 * rather than a continuous one.
 * @title         "TickTockClock"
 * @date          "2009/01/23 18:00:49"
 * @author        "Fintan Fairmichael"
 * @organisation  "CSI School, UCD"
 * @copyright     "Copyright (C) 2009 UCD"
 * @version       "$ Revision: 1.7 $"
 */
public interface LogicalClock {
  // The current time of this clock.
  //@ public model instance \bigint _time;

  //@ public invariant (* The time must be non-negative. *);
  //@ public invariant 0 <= _time;

  /**
   * @return What is the current time of this clock?
   * @concurrency CONCURRENT
   */
  //@ ensures \result == _time;
  public /*@ pure @*/ long getLogicalTime();

  /**
   * Advance the clock; update the clock's time.
   * Note that time may increase by more than one.
   * @concurrency GUARDED
   */
  //@ assignable _time;
  //@ ensures \old(_time) < _time;
  //@ ensures (* _time has been increased. *);
  public void advance();
}
```

Listing 4: A JML formal specification of the EBON class `LOGICAL_CLOCK`.

and others are not. All of the tags, standard and non-standard (`title`, `date`, `organisation`, `copyright`, `version`, and the aforementioned `concurrency`), are all semantic properties.

The JML specification also contains, of course, JML annotations. In this particular case, these annotations capture the formal meaning of the idea of a logical clock, as embodied by this Java type. In particular, a model (specification-only) field called `_time` of type `\bigint` (the type representing mathematical integers, i.e., $\mathbb{Z}$) is defined, complemented by an invariant stating that the value of that field is always non-negative.

Furthermore, both methods have contracts. The contract `getLogicalTime` states that the value returned by the method is always identical to that held in the model field `_time`. The contract for `advance` states that the method may only change the value of the model field `_time` (and *nothing* else) and that calling this method causes time to move monotonically forward, as embodied by the formal postcondition `\old( _time ) > _time` (the new value of `_time` is strictly greater than its old value before the call). The informal postcondition, contained in the JML comment block `(* ... *)`, reminds us of the meaning of its sister specification.

One possible implementation of this type is in Listing 5. Time is represented by a `long` field that refines the corresponding model field in the JML specification. This means that the invariants of the model field apply to the concrete one through the refinement, as denoted by the `represents` clause, where the arrow denotes functional data refinement.

2.1 Modifications

We will now consider some short examples of ways that we could modify our example, causing the design and implementation to become inconsistent.

One oft-performed change is renaming a class or feature in a system. If we were to perform a rename at one abstraction-level of our system the other levels will not be in-sync, with regards to naming. This is of course easily remedied by performing a renaming on the related artifacts over all levels. Note that the real

```
/**
 * A logical clock implementation.
 * @author "Joseph Kiniry"
 */
public class LogicalClockImpl implements LogicalClock {
  /** The current logical time. */
  private long my_time = 0; //@ in _time;
  //@ private represents _time <- my_time;

  public synchronized long getLogicalTime() {
    return my_time;
  }

  public void advance() {
    my_time++;
  }
}
```

Listing 5: A Java implementation of the EBON class `LOGICAL_CLOCK`.

name of an entity is independent of a particular naming style. For instance the names `LOGICAL_CLOCK` (Eiffel style) and `LogicalClock` (Java style) are equivalent. Naming styles can easily be applied or removed when mapping to or from particular domains.

Another interesting change to our system would be to modify the `advance` feature of the logical clock to take an integer argument, a measure of the amount with which to increase the time. This change might involve adding an argument of type `int` to the `advance` method signature in the `LogicalClock` interface and `LogicalClockImpl` class, as well as adding a parameter of type `INTEGER` to the `LOGICAL_CLOCK` formal model. The postcondition must also be changed to reflect that the time has increased by the provided amount, but we will ignore this for the moment. Again, if we make the change at one refinement level, the levels will become inconsistent. To maintain consistency the relevant changes must be made at all levels that detail the parameters for the `advance` feature. Thus, the signature of the `advance` feature in the formal EBON specification changes from `deferred advance` to `deferred advance -> INTEGER`.

Consider the case where the postcondition of the `advance` feature/method is to be strengthened, such that the time cannot increase by more than 100 from its old value. As a JML `ensures` clause, one might write this as `_time-\old(_time)<=100`. Once more, a change to the system at one level (JML specification) causes other levels to become inconsistent. The required change to the formal EBON model is the addition of the `ensure` clause `my_time-old my_time<=100`.

There are a many more changes that could be made to our example that require modifications on other abstraction levels to maintain consistency. The levels affected can be above and/or below the original modification's level. There are also changes that do not affect other levels, for instance a change to the author property.

3 Subtleties of Refinement

There are a number of subtleties to examine in this simple example. They relate to the support for inheritance and structure in the refinement semantics of semantic properties.

3.1 Inheritance

First, note that the implementation contains far less documentation that the previous examples. For example, there is no denotation of project or version particulars, and methods do not have comments or contract clauses. This is the case because many semantic properties have a *semantics for inheritance*. Thus, some of the semantic properties that annotate the higher-level specifications, like the Javadoc comments in the `LogicalClock` interface or those in the EBON formal specification, are automatically inherited by their children. The realization of a semantics for inheritance is sometimes quite simple and other times is not. Regardless, it is always concretely shown to the user in a simple fashion.

For example, compare the comments on the feature `advance` in the JML and EBON formal specifications, or the class comments in both specifications. The content of the EBON annotation is exactly the *first sentence* of the content of the JML annotation. The inheritance semantics of natural language comments (which are simply semantic properties with no tag) is *structural* in nature and relies upon the structure of natural language.

Other documentation refinements are subtle. For example, notice that the annotation on a query in formal EBON maps to the `@return` annotation in Java. Likewise, the annotation on a EBON command maps to the first line of the Java method comment. Likewise, the informal constraint "`The time must be non-negative.`" is refined into an informal JML comment. As a reminder to the reader: there is a formal semantics for all of these refinements (discussed below) and consistency is automatically maintained between all of these artifacts.

Another example is seen in the realization of the `getLogicalTime` method. This method is a query in the informal model (since it is in the `query` section), it is marked as a `QUERY` in the formal EBON model, and it is realized as a `pure` method in the JML model.

Concurrency is topic that makes for more interesting semantics. Since the `get_logical_time` feature in the EBON formal model was annotated with a `CONCURRENT` tag, its realization in Java is a normal method that is threadsafe. On the other hand, the `GUARDED` annotation on `advance` indicates that the method must be `synchronized` in its realization.

Type refinement is also interesting. A set of built-in base types exist in EBON. Value types like `INTEGER` and `REAL` are unsurprisingly mapped to mathematical integers and reals in JML (respectively). But other mappings are more subtle. EBON includes a set of basic mathematical abstractions like `SET`, `SEQUENCE`, and `RELATION`. Each of these is mapped to the appropriate JML model (e.g., `org.jmlspecs.models.JMLSet`), and thence to Java, usually via Java collections. The validity of data refinements is maintained by EBON and JML's refinement semantics—for example, the refinement from the `VALUE` type to `INTEGER` in EBON, or the `bigint` model field to the `long` concrete field in JML.

Finally, note that refinement of formal specifications need not be syntactic equality when mapped across syntaxes, but instead semantic equivalence. Note, for example, the difference between the formal invariant in the EBON formal diagram and the JML formal specification to which it refines.

3.2 Structure

Structural refinement has its share of straightforward and subtle aspects as well. Annotations, like those seen in the indexing block in Listing 3 and in the class Javadoc comment in Listing 4 are substructures of the EBON and JML specification respectively. These substructures are maintained in structural refinement between levels, in this case EBON and JML, and can be augmented, refined, replaced, or deleted as one moves down the refinement chain. For example, the `@title` semantic property must match exactly across refinement levels, whereas the `@author` property need not. Likewise, the class comment must match, but only partially—the first sentence of the refinement (the Javadoc class comment) must be identical to the contents of the `about` property.

Naming is another structural property. For example, the standard convention in EBON is to name classes using capitalized underbar-separated words, much like the standard convention used to name constants in Java programs, whereas the standard convention for Java class names is to capitalize the first character of each word, with no separators between words. The substructures of identifiers are thus automatically mapped in refinement, as see in the example: `LOGICAL_CLOCK` refines to `LogicalClock`. Likewise, there is support for refining EBON features to Java constants, fields, and methods (as appropriate).

Feature ordering is a seemingly uninteresting structural property that sometimes has a non-trivial impact on refinement. In particular, in some development groups the declaration order of method calls and fields is tightly constrained and checked with tools like CheckStyle. If this is the case, the refinement relationship between a Java class and, say, an EBON specification, must respect these constraints. For example, when one adds a feature to a EBON formal chart, the automatic addition of the appropriate method at the Java level must occur at exactly the right position in the source so as to respect local conventions.

Another obvious structural relationship maintained by refinement is between classes and features. Obviously, if a Java class contains a method that may be used by clients, it must be captured in the refinement up the chain (in the JML, EBON, etc.). What is less clear is what happens when visibility comes into play.

EBON has a notion of feature visibility. Each feature is either public, and visible to any client class, or is restricted, and available only to a certain group of other classes. The semantics of feature restriction in EBON are more rich than that available in Java, where one only has *public*, *protected*, *package*, and *private* class and method visibility. Consequently, each of Java's visibility levels is mapped to the appropriate EBON selective export specification.

Note that the inverse refinement, from EBON to Java, is not total: some selective export specifications in EBON do not naturally map to Java's visibility constructs. This situation, that of a higher-level being *more* expressive than a lower-level one occasionally happens and our current solution is to detect and flag such situations as an error.

A similar situation exists with regards to naming. EBON, much like Eiffel, supports feature renaming during inheritance—Java does not. Thus, any use of renaming in an EBON specification that relates to a JML or Java refinements triggers an error.

3.3 Semantics and Tools

As one can see, there are subtleties in the interplay between refinements of inheritance and structures and the preexisting tools that operate on artifacts at the various refinement levels. *We must be careful to ensure that all refinements respect tool semantics.* For example, if a refinement of documentation from EBON to Java contradicts the standard use of Javadoc, then the refinement is not very useful.

In the current EBON/JML/Java-centric system there are seven different kinds of tools that we must respect. All of these tools are integrated into the Mobius Program Verification Environment (PVE), which is discussed in more detail later in this paper.

1. Documentation tools (e.g., Javadoc and Doxygen) interpret semantic properties in the *documentation* and *usage* categories.
2. Specification tools (e.g., the JML tool suite, ESC/Java2) interpret the semantic properties in the *contract* category.
3. IDEs (like Eclipse and Emacs) interpret semantic properties in the *process* category.
4. Bug/feature trackers (we use Trac, GForge, and Bugzilla) understand some of the semantic properties in the *meta-info* and *process* categories.
5. Configuration management and revision control tools (e.g., CVS and subversion) process semantic properties in the *meta-info* category.
6. The Java compiler and some static checkers interpret Java annotations that are refinements of the *inheritance* category of semantic properties.
7. And finally, static checkers (like CheckStyle, FindBugs, PMD, and ESC/Java2) interpret a mixed subset of semantic properties across many categories.

By virtue of the manner in which we specify semantic properties' semantics below, and coupled with the precise way that we configure and use the aforementioned tools, it is guaranteed that tools' behavior does not contradict the automatic upkeep of refinements in our system.

4 Expressing Semantics

The meaning of semantic properties is expressed in a formalism called *kind theory* [8]. As kind theory is a relatively new, very rich formal method unfamiliar to most readers, its full form is not used here. Instead, the aspects of kind theory most relevant to this work are explained in terms familiar to most readers, with an emphasis on capturing the important facets of the formalism. The curious reader will find the full details of the original kind theoretic formalization of generic semantic properties in the aforementioned dissertation, and a modernized full explanation of the concrete realization of semantic properties for EBON, JML, and Java in a forthcoming technical report and an undergraduate student research thesis [9].

Kind theory lets one describe general purpose reusable assets. Software artifacts and mathematical systems are two kinds of assets that are described, and about which we reason, using kind theory. In its simplest form, kind theory lets one describe *type-like structures* (called *kind*) and explain their interrelationships. The relationships that one describes are *structural*, *subtyping*, *equivalency*, *composition* and *decomposition*, *realization*, and *refinement*.

From a type theoretical point of view, kind theory permits one to describe *multiple type systems* and their interrelationships. The theorems of kind theory support reasoning at the object and the kind level, much like one can reason about types and typed objects in type theory.

A full description of kind theory and its use requires an entire dissertation [8]. For space reasons we only give a flavor of the means by which the semantics of semantic properties is specified here.

$$\frac{\Gamma \vdash K <_p L}{\Gamma \vdash K < L}\ \text{(Parent Is-a)} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash K < K}\ \text{(Is-a Refl)} \qquad \frac{\Gamma \vdash K < L \quad \Gamma \vdash L < M}{\Gamma \vdash K < M}\ \text{(Is-a Trans)} \qquad \frac{\Gamma \vdash K < L \quad \Gamma \vdash \gamma(\bot(K \equiv L)|)}{\Gamma \vdash \gamma(\bot(L < K)|)}\ \text{(Is-a Asym)}$$

Figure 2: Example rules written in kind theory.

A few examples of kind theory subkinding rules to help the reader get comfortable with reading them are found in Figure 2. The rule *Parent Is-a* states that, if the parent of kind K is the kind L ($K <_p L$) then the kind K *is-a* L ($K < L$). This is akin to subtyping, where L is the immediate supertype of K. *Is-a Refl* states that every kind is a subkind of itself (subkinding is reflexive); *Is-a Trans* that subkinding is transitive. Finally, *Is-a Asym* states that, if $K < L$ and K and L are *not* equivalent ($\gamma(\bot(K \equiv L)|)$) then one can prove that L is not a subkind of K ($\gamma(\bot(L < K)|)$). This last rule hints at the fact that kind theory supports reasoning about proof systems as well as proof artifacts, since proofs and evidence are first-order notions in the theory.

The key foundational axiom of kind theory that supports this refinement-centric work is that properties are preserved under interpretation. Interpretation is simply a (possibly computable) relation between kind, and consequently, between objects that realize those kind. Thus, refinements between BON, JML, and Java in this work are modeled as computational interpretations in kind theory. Such property-preserving relations are described using commutative diagrams, as kind theory is a logic with a categorical feel.

Theorem 1 (FullInterp Part-of)

$$\frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash U \rightsquigarrow P} \qquad \frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash P \subset_p W}$$

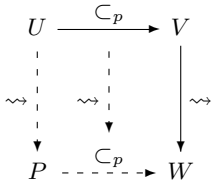

Figure 3: The Theorem Diagram for (FullInterp Part-of)

Consider the diagram in Figure 3. It captures the essential elements of the proof of the *FullInterp Part-of* theorem, as seen in the two rules in Theorem 1.

Proof. Since V contains U, and $\rightsquigarrow$ fully interprets V to W, then this interpretation also acts upon U. Call the object resulting from the full interpretation P. Since full interpretation is structure-preserving, and $\subset_p$ is a component of that structure, then necessarily $P \subset_p W$. □

What this theorem tells us is that substructures are preserved under full interpretations and, as refinements from BON to JML, and JML to Java, are full interpretations, then substructure relationships in BON are preserved under refinement into JML, etc.

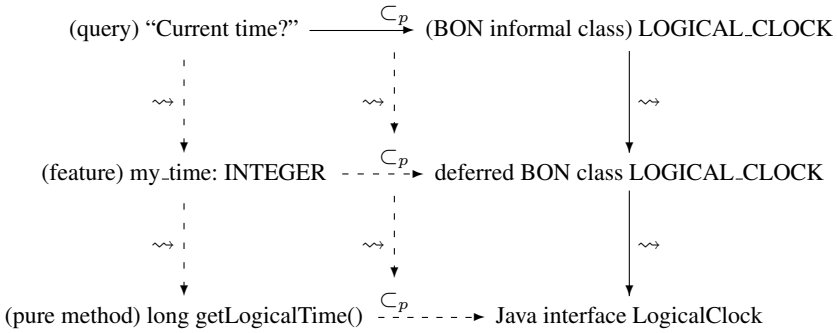

Figure 4: Full Interpretation of an Informal Query to a Formal (Pure) Feature

This property is made more clear if we replace this generic commutative diagram with a particular instantiation for our running example. Consider Figure 4, which is an instantiation of this theorem when applied to the single query of the running example.

Because the high-level design contains this query, (i.e., the BON class chart `LOGICAL_CLOCK` contains the query), then according to this theorem, the BON formal specification of the refinement of the class must also contain this query. *Moreover*, all properties of the substructure (the query) at the less refined level (the BON informal level) must be maintained by the more refined level (the BON formal level). In this case this means that (i) the feature must be of non-`VOID` type (thus, it is a query), (ii) it must be pure (i.e., its postcondition must not mention any frame conditions), and (iii) the documentation of the formal feature must refine the documentation of the informal query (in this case, they are equivalent, but that's specific to this example).

5 Properties and their Classification

Thirty-five semantic properties have been identified and defined[1] . All semantic properties are enumerated in Table 1.

Due to space reasons, only a handful of the more interesting properties and their semantics that we have used in a number of software engineering projects, large and small, over nearly the last decade are discussed. Also, the following descriptions are written entirely from the point of view of a *user* of semantic properties (i.e., a software developer), not a *creator* of new semantic properties (which requires some knowledge of kind theory).

To derive our core set of semantic properties, the existing realizations that we have used in two languages for many years were abstracted and unified. First, the set of predefined Javadoc tags, the standard Eiffel indexing clauses, and the set of basic formal specification constructs were identified and made self-consistent (duplicates were removed, semantics were weakened across domains for the generalization, etc.). The resulting set of unique properties are the *core set* of semantic property kinds.

These properties were then classified according to their general use and intent. The classifications are: *meta-information*, *process*, *contracts*, *concurrency*, *usage*, *versioning*, *inheritance*, *documentation*, *dependencies*, and *miscellaneous*. This classification is represented using kind theory's inheritance operators,

[1] The original specification of these properties was defined in the Caltech Infospheres Java Coding Standard (http://www.infospheres.caltech.edu/). That standard has since been refined and broadened. The most recent version is available via the KindSoftware Research Group's website (http://kind.ucd.ie/).

Meta-information author bon bug copyright description history license title	**Contracts** ensures generates invariant modifies requires	**Versioning** deprecated since version
	Concurrency concurrency	**Documentation** design equivalent example see
Dependencies references use	**Usage** exception param return	
Inheritance hides overrides	**Process** idea review todo	**Miscellaneous** guard space-complexity time-complexity values

Table 1: The full set of semantic properties.

e.g.,:

$$\text{METAINFO} <_p \text{SEMANTICPROPERTYCLASSIFIER} \qquad \text{AUTHOR} <_p \text{METAINFO}$$
$$\text{ENSURES} <_p \text{CONTRACTS} \qquad \text{DEPRECATED} <_p \text{VERSIONING}$$

Many of these semantic properties are used solely for documentation purposes. For example, the `title` property documents the title of the project with which a file is associated; the `description` property provides a brief summary of the contents of a file. These kinds of properties are called *informal* semantic properties.

Another set of properties are used for specifying non-programmatic semantics. By "non-programmatic" we mean that the properties have semantics, but they are not, or cannot, be expressed in program code. For example, labelling a construct with a `copyright` or `license` property specifies some legal semantics. Tagging a method with a `bug` property specifies that the method has some erroneous behavior that is described in detail in an associated bug report. We call these properties *semi-formal* because they have a semantics, but outside of the domain of software (at least for the moment).

Finally, the remaining properties specify structure that is programmatically testable, checkable, or verifiable. Basic examples of such properties are `requires` and `ensures` tags for preconditions and post-conditions, `modifies` tags for specifying frame conditions, and the `concurrency` and `generates` tags for expressing concurrency semantics. These properties are called *formal* because they are realized by a formal semantics.

The KindSoftware Research Group coding standard summarizes the current set of semantic properties and is regularly updated to reflect newly identified properties [10]. Each property has a syntax, a correct usage domain, and a natural language summary. As mentioned before, the formalization of semantic properties is found elsewhere [8].

5.1 Context

Each property has a legal scope of use, called its *context*. Contexts are defined in a coarse, language-independent fashion using inclusion operators in kind theory. Contexts are comprised of *files*, *modules*,

features, and *variables*. Contexts are structured hierarchically; the scope of a property encompasses the context for which it is defined, as well as all sub-contexts.

Files are exactly that: data files in which program code resides. The scope of a file encompasses everything contained in that file.

A *module* is some large-scale program unit. Modules are typically realized by an explicit module- or class-like structure. Examples of modules are classes in object-oriented systems, modules in languages of the Modula and ML families, packages in the Ada lineage, etc. Other words and structures typically bound to modules include units, protocols, interfaces, etc.

Features are the entry point for computation. Features are often named, have parameters, and return values. Functions and procedures in structured languages are features, as are methods in object-oriented languages, and functions in functional systems.

Finally, *variables* are program variables, attributes, constants, enumerations, etc. Because few languages enforce any access principles for variables, their semantics vary considerably.

Each property listed in Table 1 has a legal context [10]. The context *All* means that the property may be used at the file, module, feature, or variable level. Additional contexts can be defined, supporting new programming language constructs that need structured documentation with properties. For example, now that JSR 305 (Annotations for Software Defect Detection) and JSR 308 (Type Annotations) have been finalized the introduction of new contexts may be necessary.

For each concrete language there is a mapping for these contexts/scoping levels. For example, in Java the valid scoping levels are source files, classes/interfaces, methods and variables. Some of the kind theory rules that specify context for a Java method declaration and its documentation are as follows:

$$\textsc{JavadocMethodDescription} \supset \textsc{Return} \oplus \textsc{ParamList} \oplus \textsc{ExceptionSet}$$
$$\textsc{JavaMethodSignature} \supset \textsc{ReturnType} \oplus \textsc{MethodName} \oplus \textsc{FormalParameterList} \oplus \textsc{ThrowsClause}$$
$$\textsc{JavaMethodDeclaration} \supset \textsc{JavaMethodSignature} \oplus \ldots$$
$$\textsc{JavaMethodSignature} \rightsquigarrow \textsc{JavaMethodDescription}$$

These rules formally define the structure of a Javadoc method description (i.e., the fact that it can contain Javadoc tags like `@return` and `@param`) and the type signature of a Java method. The first line is read, "A Javadoc method description is made up of the composition of a return, param list, and exception set." The third line states that a Java method declaration must contain a method signature, and perhaps more (the elision is not part of kind theory, we are simply ignoring the rest of the formula).

The final line, which is the most interesting one, states that a refinement relationship exists between the Javadoc specification of a method and its declaration. It is read, "One can interpret any Java method signature into a Java method description without any loss of information." This refinement relationship captures two things: (1) a Java method declaration *must* have a Javadoc method description, and, more specifically, (2) how each substructure within a method declaration must be documented in Javadoc. This first properly is generally enforced by several basic tools we use in our software development process (e.g., Javadoc and Eclipse). The second, more specific, property is enforced by our customization of tools like PMD and CheckStyle.

Other structural rules of this form are not enforced by customized versions of third-party tools, but instead by our own, as discussed below.

5.2 Visibility

Visibility is a key notion discussed earlier. In languages that have a notion of *visibility*, a property's visibility is equivalent to the visibility of the context in which it is used, augmented by domain-specific visibility options expressed in kind theory.

Typical basic notions of visibility include *public*, *private*, *children* (for systems with inheritance), and *module* (e.g., Java's *package* visibility). More complex notions of visibility are exhibited by C++'s notion of *friend* and Eiffel's class-based feature scoping.

Explicit visibilities for semantic properties are also used to refine the notion of specification visibility for *organizational*, *social*, and *formal* reasons. For example, a subgroup of a large development team might choose to expose some documentation for, and specification of, their work only to specific other groups for the purposes of testing, or for political or legal reasons.

On the social front, new members of a team might not have yet learned specific tools or formalisms used in semantic properties, so using visibility to hide those properties will help avoid information-overload.

Lastly, a formal specification, especially when viewed in conjunction with standard test strategies (e.g., whitebox, greybox, blackbox, unit testing, scenario-based testing), has distinct levels of visibility. For example, testing the postcondition of a private feature is only reasonable and permissible if the testing agent is responsible for that private feature.

5.3 Inheritance

Semantic properties also have a well-defined notion of *property inheritance*. Once again, in order to avoid new and complicated extra-language semantics on the software engineer, property inheritance semantics match those of the source language in which the properties are used. Our earlier discussion of basic comments for Java methods (a *feature* property context) is an example of such property inheritance.

These kinds of inheritance come in two basic forms: *replacement* and *augmentation*.

The *replacement* form of inheritance means that the parent property is completely replaced by the child property. An example of such semantics is *feature overriding* in Java and the associated documentation semantics thereof.

Augmentation, on the other hand, means that the child's properties are actually a *composition* of all its parents' properties. These kinds of composition come in several forms. The most familiar is the standard substitution principle-based type semantics [11] in many object-oriented systems, and the Hoare logic/Dijkstra calculus-based semantics of contract refinement [12].

These formal notions are expressible using kind theory because it is embedded in a logical framework. For example, automatically reasoning about the legitimacy of specification refinement is supported, much like that seen in Findler and Felleisen work [13].

6 Tool Support

Semantic properties have been used for the past decade in academic and corporate settings. While explicit (and utilized) coding standards, positive feedback via tools and peers, course grades and monetary rewards go a long way toward raising the bar for documentation and specification quality, from our experience these social aspects are simply not enough. Process does help—regular code reviews and pair programming in particular—but tool support is critical to maintaining quality specification coverage, completeness, and consistency.

Templates were the first step taken. Raw documentation and code templates in programming environments, ranging from *vi* to *emacs* to *Eclipse*, are used. But templates only help prime the process, they do not help maintain the content.

Syntax highlighting as well as code and comment completion also helps. Both aid in programmer comprehension and efficiency. Advanced development environments such as Eclipse and Emacs support these features.

Likewise, documentation lint checkers (programs that statically check for the use of improper idioms in a language; e.g., `lint`, `javadoc`, "`gcc -Wall`," or CheckStyle), particularly those embedded in development environments and documentation generators are also useful. Source text highlighting is an extremely weak form of lint-checking. The error reports issued by Javadoc, CheckStyle, and its siblings are a stronger form of lint-checking and are quite useful for documentation coverage analysis, especially when a part of the regular build process. Finally, scripts integrated into a revision control system provide a "quality firewall" to a source code repository in much the same fashion.

But these simple tools are not enough to ensure consistency between artifacts at different refinement levels, nor do they support the automatic updating of artifacts as a system's design, specification, or implementation evolves.

6.1 Checking and Maintaining Consistency

One of the most important parts of tool support for semantic properties is the automation of checking for consistency between abstraction levels. Discrepancies between related artifacts should be detected and presented in a cohesive manner along with other standard errors.

Ongoing research here at UCD focuses on developing the detailed formal theory for automatic consistency checking over multiple refinement levels, with an implementation specifically targeting EBON, JML and Java. We target these languages for a variety of reasons, including local expertise. As seen earlier, the theory itself is applicable to any set of languages for which well-defined refinement relations are defined.

The technique under development supports the definition of all possible relations between two refinement levels, as well as an ordering on these. A consistency check uses automatic deduction of relations (choosing from the set of all possible relations), as well as user-defined relations to determine if artifacts at different refinement levels are consistent.

Part of this work has been the development of the BONc tool[2], a parser, typechecker and documentation generator for BON. It is open source, and available as a commandline tool or as an Eclipse plugin. Our consistency checker is being built on top of BONc and OpenJML, which is, in turn, built atop of the OpenJDK.

To help illustrate the aims of the tool support, consider the following example. A developer is implementing, in Java, a class for which an EBON formal model also exists. A sufficiently advanced consistency checker would detect that there is a feature in the class's formal EBON model for which there is no refinement to a Java method. The issue is flagged to the developer inside her development environment in a manner consistent with the presentation of other errors (compilation, etc.). The developer then chooses to automatically fix the issue from a set of suggested fixes—for instance, they might choose to automatically insert a method skeleton with the appropriate signature (calculated from the EBON types and the knowledge of the existing refinement relations).

6.2 Development Environments

As mentioned earlier, a verification-centric development environment, known as the Mobius Program Verification Environment (PVE), has also been built to support design, development, and formal verification using semantic properties.

The PVE is an extension to the powerful Eclipse Platform. It takes advantage of several pre-existing development tools and plugins, including PMD, FindBugs, CheckStyle, the JML tool suite, ESC/Java2, and BONc. In the near future our consistency checker will also be integrated.

7 Conclusion

Documentation reuse is most often discussed in the literate programming [14] and hypertext domains [15]. Little research exists for formalizing the semantics of semi-structured documentation. Some work in formal concept analysis and related formalisms [16,17] has started down this path, but with extremely loose semantics and little-to-no tool support.

Research by Wendorff [18,19] bears resemblance to this work both in its nature (that of concept formation and resolution) and theoretic infrastructure (that of category theory, which relates to kind theory). Development with semantic properties is differentiated by its broader scope, its more expressive formalism, and its realization in tools. Additionally, the user-centric nature of kind theory (not discussed in this article) makes for exposing the formalism to the typical software engineer a straightforward practice.

7.1 Future Work

Our work on the Mobius PVE continues. A graphical modeling environment for EBON is in the works, linking BONc to the Eclipse Graphical Modeling Framework.

Extending JML and other model-based languages like Event-B with semantic properties would follow the same course used for EBON. Because semantic properties are already integrated with Java, and given the existing tool support for JML, inter-domain interpretations will preserve a vast amount of information about JML-specified Java systems in Event-B.

[2] Available from http://kind.ucd.ie/products/opensource/BONc/

8 Acknowledgements

This work was initiated under the support of ONR grant JJH1.MURI-1-CORNELL.MURI (via Cornell University) “Digital Libraries: Building Interactive Digital Libraries of Formal Algorithmic Knowledge” and AFOSR grant JCD.61404-1-AFOSR.614040 “High-Confidence Reconfigurable Distributed Control.” Recently, the two authors have been supported by several other grants. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the author’s views and the Community is not liable for any use that may be made of the information contained therein. This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1, “LERO: the Irish Software Engineering Research Centre” and by an EMBARK Scholarship from the Irish Research Council in Science, Engineering and Technology.

References

1. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4) (1992) 40–52
2. Knuth, D.E.: Literate Programming. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information (1992)
3. Knuth, D.E., Levy, S.: The CWEB System of Structured Documentation. third edn. Addison–Wesley Publishing Company (2001)
4. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO). Volume 4111 of Lecture Notes in Computer Science., Springer-Verlag (2006) 342–363
5. Waldén, K., Nerson, J.M.: Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems. Prentice–Hall, Inc. (1995)
6. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Proceedings of the Fifteenth International Symposium on Formal Methods (FM). (2008) In press.
7. Janota, M., Kiniry, J.: Reasoning about feature models in higher-order logic. In Kellenberger, P., ed.: Proceedings of the 11th International Software Product Line Conference, SPLC ’07, IEEE Computer Society (2007)
8. Kiniry, J.R.: Kind Theory. PhD thesis, Department of Computer Science, California Institute of Technology (2002)
9. Kiniry, J., Fairmichael, F., Darulova, E.: Beetlz - a BON software model consistency checker for Eclipse. Technical report, KindSoftware Research Group, University College Dublin (2009)
10. Kiniry, J.R.: The KindSoftware coding standard. Technical report, KindSoftware Research Group, UCD (2005) Available via http://kind.ucd.ie/.
11. Liskov, B., Wing, J.M.: Specifications and their use in defining subtypes. In: Proceedings of OOPSLA’93. (1993) 16–28
12. Meyer, B.: Applying design by contract. IEEE Computer **25**(10) (1992) 40–51
13. Findler, R., Felleisen, M.: Contract soundness for object-oriented languages. In: Proceedings of Sixteenth International Conference Object-Oriented Programming, Systems, Languages, and Applications. (2001)
14. Childs, B., Sametinger, J.: Literate programming and documentation reuse. In: Fourth International Conference on Software Reuse, IEEE Computer Society (1996) 205–214
15. Fischer, G., McCall, R., Morch, A.: JANUS: Integrating hypertext with a knowledge-based design environment. SIGCHI Bulletin (1989) 105–117
16. Simos, M., Anthony, J.: Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In Devanbu, P., Poulin, J., eds.: Fifth International Conference on Software Reuse, IEEE Computer Society (1998)
17. Wille, R.: Concept lattices and conceptual knowledge systems. Computers and Mathematics with Applications **23**(6-9) (1992) 493–515
18. Wendorff, P.: Linking concepts to identifiers in information systems engineering. In Sarkar, S., Narasimhan, S., eds.: Proceedings of the Ninth Annual Workshop on Information Technologies and Systems. (1999) 51–56
19. Wendorff, P.: A formal approach to the assessment and improvement of terminological models used in information systems engineering. Software Engineering Notes **26**(5) (2001) 83–87