

Semantic Component Composition

Joseph R. Kiniry

Computing Science Department, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen
The Netherlands
kiniry@cs.kun.nl

Abstract. Building complex software systems necessitates the use of component-based architectures. In theory, of the set of components needed for a design, only some small portion of them are “custom”; the rest are reused or refactored existing pieces of software. Unfortunately, this is an idealized situation. Just because two components *should* work together does not mean that they *will* work together. The “glue” that holds components together is not just technology. The contracts that bind complex systems together *implicitly* define more than their explicit type. These “*conceptual contracts*” describe essential aspects of extra-system semantics: e.g., object models, type systems, data representation, interface action semantics, legal and contractual obligations, and more. Designers and developers spend inordinate amounts of time technologically duct-taping systems to fulfill these conceptual contracts because system-wide semantics have not been rigorously characterized or codified. This paper describes a formal characterization of the problem and discusses an initial implementation of the resulting theoretical system.

1 Introduction

Modern software systems are increasingly complicated. Combine new technologies, complex architectures, and incredible amounts of legacy code, and you have systems that no one can even partially comprehend. As complexity rises, code quality drops and system reliability suffers.

Building reliable complex systems necessitates the use of excellent software engineering practices and tools. But even with this bag of tricks, the modern software engineer is often overwhelmed with the technological problems inherent in building such complex systems. Most projects have to support multiple programming languages and models, deal with several different architectures (both hardware and software), have distributed and concurrent subsystems, *and* must be faster, cheaper, and better than their competitors.

What is worse, while these technological challenges are daunting, it is the non-technological issues that are often insurmountable. The widespread lack of system, design, and component documentation, ignorance about testing and quality assurance, and lack of communication/knowledge across teams/companies, make for a very difficult working environment. The situation is further compounded by the social problems rampant in information technology: the Not-Invented-Here syndrome; issues of trust among companies, teams, and developers; developer education and motivation; and managerial and business pressures.

1.1 Semantic Components with Conceptual Contracts

Since developers of reusable constructs (like chunks of code or paragraphs of documentation) can not say what they mean, there is a “meaning-mismatch”. This “information-impedance” muddles communication between collaborators as well as complicates composition between components.

A new semantic theory, specifically designed to address this and related problems, can help solve this “meaning-mismatch”. It is our belief that if such a product were widely available and integrated into all aspects of system development, then our systems would be easier to build, better documented, and more robust. Put more plainly, our systems would be “more correct”.

This paper aims to (1) provide a means by which components can be semantically (conceptually, formally) described, (2) present an algorithm which automatically determines when components can interoperate regardless of syntax or component model, and (3) describe the means by which the adapters necessary to compose such components can be automatically generated.

A component that is specified, manipulated, and reasoned about using this (or functionally similar) theory, tools, and techniques is what we call a *semantic component*.

The theoretical foundation for this work is a new logic for describing open collaborative reusable assets called *kind theory*, which is described in full in Kiniry’s Ph.D. dissertation [18].

1.2 Related Work

Research from several communities has similarities to this work, and this work is heavily influenced by a wide range of work in programming languages and semantics (see the extensive bibliography of the aforementioned dissertation for details).

In its most simplified form, straightforward structural subtyping can be used as a theoretical framework for semantic composition, especially in strongly typed languages. An example is found in Muckelbauer’s work in distributed object-based systems [23]. The problem with this approach is that only renaming and reordering operations are possible, and little theoretical infrastructure supports reasoning about the full semantics of components.

Stronger theoretical formalisms exist for specifying and reasoning about such compositional systems. Some of the earliest and most influential work is from Goguen [14], as it formalized and popularized the notion of parametric modules as a means of reuse. The recent work of Molina-Bravo and Pimentel [22] shows promise as well. This latter work is far from being practically applicable in the near future as there is no strong connection with real programming languages, software engineering methods, and no tool support.

Other expressive formalisms for compositionality exist, primarily in the category theoretic domain [10]. Feature logic has also been used to specify component properties and detect mismatches during composition [28, 33]. Unsurprisingly, the composition properties in Fiadeiro’s work match those of kind theory due to the latter’s categorical basis. Feature logic is a submodel of kind theory—use a first-order foundation, eliminate beliefs, and map subkinding to subsumption.

Work at CMU by Yellin and Strom, inspired by the problems with the Aesop system [13], has covered this territory before in the context of object protocols and weak, non-formal, semi-automatic adapter construction [32, 31]. The work has a very ad hoc feel, even after the development of a formal model for such architectures [2].

In a Java context, Wang et al proposed a service-event model with an ontology of ports and links, extending the JavaBeans descriptor model [30].

Other related work comes from the areas of: domain-specific languages, especially for component specification and reuse [5, 26]; the automatic programming community, e.g. early work by Barstow [3, 4] and Cleveland [8]; conceptual reuse such as Castano and De Antonellis [6]; and last but certainly not least, the software transformation systems community, including the voluminous works of Biggerstaff and Batory.

Much of this work is reviewed nicely in [9, 25, 27].

The primary difference between all of these formalisms and systems and ours is that our system has a broad, firm theoretic foundation. That foundation, kind theory, was specifically designed to reason about reusable assets in an open collaborative context. Thus, our work is not tied to a specific language or realization, integrates the domains of reuse, knowledge representation, automatic programming, and program transformation, and does so in a theoretical and systematic framework that supports global collaboration among many participants.

2 Semantic Composition

Our proposal is, on the surface, relatively simple. Instead of having only a syntactic interface to a component, we provide a higher-level semantic specification. Additionally, providing this semantic specification does not entail the need for a developer to learn any new heavyweight specification language, theory, or tools.

The key to this new solution is the notion of *semantic compatibility* [19]. Components are described explicitly and implicitly with lightweight, inline, domain-specific documentation extensions called *semantic properties*. These properties have a formal semantics that are specified in kind theory and are realized in specific programming languages and systems.

When used in tandem with a computational realization of kind theory (called a *kind system*), these *semantic components* are composed automatically through the generation of “glue” code, and such compositions are formally validated.

3 A Brief Overview of Kind Theory

Kind theory is a logic used to reason about subjective reusable assets. It is used to bridge concepts from multiple domains, and from multiple perspectives. For the purpose of this paper, the domain of interest is software components with formally specified interfaces.

Kind are classifiers used to describe reusable assets like program code, components, documentation, specifications, etc. *Instances* are realizations of kind—actual embodiments of these classifiers. For example, the paperback “The Portrait of the Artist as a Young Man” by James Joyce is an *instance of kinds* PAPERBACKBOOK and ENGLISHDOCUMENT, (and perhaps others). We write this as

Portrait : PAPERBACKBOOK \oplus ENGLISHDOCUMENT

We use kind theory to specify semantic properties because it provides us with an excellent model-independent (i.e., it is not bound to some specific programming language) reuse-centric formalism.

3.1 Structure

Kinds are described structurally using our logic in a number of ways using several core operators. Classification is covered with the inheritance operators $<$ and $<_p$; structural relationships are formalized using the inclusion operators \subset_p and \supset , equivalence has several forms, $=$ and \leq ; realization, the relationship between instances and kind, is formalized with the operators $<_r$ and $::$; composition is captured in several forms, \otimes , \oplus , and \circ ; and interpretation, the translation of kind to kind or instances to instances, is realized with the operators \leftrightarrow and \rightsquigarrow .

3.2 Operators

Inheritance *Inheritance* is a relation that holds between a *child* kind and one or more *parent* kinds. The generic notion of inheritance is that some qualities are transferred by some means from parent to child. Inheritance is the relation that is used primarily to *classify* kind.

The basic symbols that we use for inheritance are $<_p$, $<$, and $<_r$ to connote order on kind. Inheritance is a reflexive, transitive, asymmetric operation.

Inheritance in the form of classification is one of the most common structuring notions known in science. Variants of Aristotle’s *genera* and *species* and the Linnaean method has been used in various incarnations for hundreds of years to classify organisms and entities. Another example of inheritance comes from library science where researchers have developed several classification schemas for books: e.g., the Library of Congress and the Dewey Decimal Systems.

Some forms of subtyping and inheritance in modern formalisms and programming languages are sometimes also kinds of inheritance (they are not when they have nothing to do with semantic conformance).

Inclusion *Inclusion* is a relation that holds between two constructs, the *whole* and the *part*. A part is a portion of the whole. We use the term *has-a* to connote inclusion. The basic symbols that we use for inclusion are \subset_p and \supset to connote structural containment.

Consider this document. We can hierarchically decompose its entire physical structure in a variety of different ways. For example, syntactically, letters make up words, words make up sentences, sentences make up paragraphs, etc. Thus, paragraphs *have* sentences and, specifically, a given paragraph *has-a* particular sentence.

Inclusion is also a reflexive, transitive, asymmetric operation.

Equivalence *Equivalence* is a relation that holds between constructs that are the “same” in some context. Two constructs are equivalent if they are similar for one or more reasons. The basic symbols that we use for equivalence are \equiv and $=$, and the related \triangleleft .

Equivalence is a context-sensitive relation. Consider the two strings “ $Y = 2 * X$ ” and “ $Z = X * 2$ ”. Textually, they are not equivalent because they contain different characters. If interpreted in an algebraic setting, they are potentially equivalent equations, but it depends upon the specific algebras in question. If they both are interpreted in the same algebra, if that algebra is commutative, and alpha-renaming is a part of the definition of equivalence, then they are equivalent. If any of these conditions fails, or perhaps some other unusual conditions exist (i.e., one algebra is a modulus group), then the equations are not equivalent.

Equivalence is not just something that is applicable to formal mathematical statements. Consider a pin-ball machine and a Sony PlayStation. How are these two things equivalent? Obviously, both are games that people play for enjoyment, so this classification criterion is one potential equivalence class. Another equivalence is that both devices have buttons, thus some property-based criterion imply equivalence. Another more subtle equivalence class is the fact that both devices encode the group $(\mathbb{Z}, +)$ because both keep score in some fashion.

Composition *Composition* encapsulates the general notion of taking two or more constructs and putting them together in some way. Composition is a constructive operation—its result is a new thing that has some of the properties of its constituent pieces. The semantics of the specific composition operation used dictates the properties of the new construct. We define a generic semantics to which all composition operations must at least adhere. The symbols we use for composition are \circ , \otimes , and \oplus . We read $M \circ N$ as *M composed with N*.

Any constructive operation is composition. A father putting together the parts of a bicycle on Christmas Eve is performing a type of composition; the composition of bicycle parts to make a bike. The `cat` command in UNIX systems is another example of composition; this time, of two byte streams.

Realization As mentioned earlier, an instance I realizes a kind K . *Realization* is the process of stating that a specific thing is an *instance* of a specific kind. If I is an instance of a kind K , we say *I realizes K*, or *I is an instance of K*, or even *I is of kind K*. We use the symbol $:$ for realization. The choice of symbol comes from the colon character’s use in programming languages and type theory.

Realization is the kind theory peer to typing. When one states “let Z be a variable of type integer”, a type is being assigned to the symbol Z . Likewise, when we state $Z : \text{INTEGER}$ we are *kinding* the symbol Z as having the kind `INTEGER`.

Interpretation *Interpretation* is the process of evaluating a construct in some context, translating it from one form to another. The symbols that we use for interpretation are \hookrightarrow and \rightsquigarrow . We read $K \xrightarrow[A]{C} L$ as *the partial interpretation kind from kind K to kind L by agent A in context C*. Likewise, \rightsquigarrow is read *full interpretation*. When the agent and context are clear, we simply write $K \rightsquigarrow L$ and read it as *the full interpretation kind from K to L*.

An *interpretation* is a computable functional kind that preserves some type of structure. We define two sort of interpretation kind.

Full interpretations are computable functions that preserve all structure from their domain. This means that the semantics, that is, the validity, of all related constructs is maintained across interpretation. Within kind theory a full interpretation is defined as a *functor* on a specific category of kind.

A *partial interpretation* is a computable function that preserves some substructure from its domain. Partial interpretations are, categorically, *forgetful functors*.

Interpretation is a transitive operation. The identity interpretation is always defined on all kinds and instances—it is the identity function, denoted with the term *id*.

Any evaluation process is a type of interpretation. Reading this document is one kind of interpretation, evaluating a mathematical expression with Mathematica is another. In each case, data is transformed via an agent (you, the reader, in the former case, and the Mathematica process in the latter) within a specific context.

Canonicity Associated with every kind K is a full interpretation function $\llbracket \cdot \rrbracket$, read as “*canonical*”. Given a kind (instance) it returns the *canonical form* of that kind (instance). The canonical form of a canonical form is itself.

If we have a term of the form $\llbracket k \rrbracket = l$ we call the asset l the *canonical kind* of k . Likewise, for instances, we use the term *canonical instance*. When we do not distinguish kind or instance, we say *canonical realization* or *canonical asset*.

Canonicity preserves *all* structure of its domain and, since the codomain is generative (it is constructed entirely by the canonicity operation), then it can have *no* new structure. This means that it is *possible* to define a full interpretation between any two canonically equivalent assets, but not *necessary* that such interpretations exist.

Thus, canonicity induces an equivalence relation; interpretations do not.

Rewriting logic embedded in a type system are used in [18] to define and compute canonical forms of kinds and instances. All operations within kind theory but for resolution are deterministic due to the fact that their operational semantics are Church-Rosser. Resolution is defined in general rewriting logic and is both undecidable and NP-hard, but in all the common cases that we have explored in the actual use of resolution in practice are tractable and decidable¹.

3.3 Rules

Table 1. Relevant Rules.

$$\frac{\text{(Parent Interp*)} \quad \Gamma, L \rightsquigarrow K, K \hookrightarrow L \vdash \Phi \quad \Gamma \vdash K <_p L}{\Gamma \vdash L \rightsquigarrow K \cdot K \hookrightarrow L = id_L, \Phi} \quad \frac{\text{(Fully Equiv*)} \quad \Gamma \vdash U = V}{\Gamma \vdash [U] = [V]} \quad \frac{\text{(Partial Equiv*)} \quad \Gamma \vdash U \leq V}{\Gamma \vdash [V] \supset [U]}$$

The most important rules of kind theory in the context of this paper are summarized in Table 1. The gamma in these rules is an explicit *context* of sequents (kind theory sentences). Φ is a list of arbitrary sentences, id_L is the identity function on kind L , and the asterisk denotes that all of these rules are reversible (that is, they are bijections).

The rule (Parent Interp*) states that, if an inheritance relationship exists between kinds, then two interpretations must also exist: one that takes the parent to the child, preserving all structure, and a right adjoint of that map that takes the child to the parent. This rule essentially subsumes the related notions of type coercion, structural type checking, and classification.

The relationship between equivalence and canonicity is elucidated in the other two rules. First, two assets, that is, kinds or instances, are *fully equivalent* if and only if their canonical forms are identical. Second, two assets are *partially equivalent* if their canonical forms are structurally contained—that is, one of them is part of the other.

3.4 Semantics

Semantics are specified in an autoepistemic fashion using what are called *truth structures*. Truth structures come in two forms: *claims* and *beliefs*.

Claims are stronger than beliefs. A mathematically proven statement that is widely accepted is a claim. This phrasing is used because, for example, there are theorems that have a preliminary proof but are not yet widely recognized as being true.

A statement that is universally accepted, but not necessarily mathematically proven, is also a claim. Claims are not necessarily mathematical formulas. The statement “the sun will rise tomorrow” is considered

¹ This observation matches that of those working in feature logic for software configuration management as well [12, 33].

by the vast majority of listeners a true and valid statement, and thus is classified as a claim rather than as a belief.

Beliefs, on the other hand, range in surety from *completely unsure* to *absolutely convinced*. No specific metric is defined for the degree of conviction, the only requirement placed on the associated belief logic is that the belief degree form a partial order.

4 Semantic Properties

Semantic properties are domain-independent documentation constructs with intuitive formal semantics that are mapped into the semantic domain of their application. We use the term “semantic properties” because they are properties (as in property-value pairs) which have formal semantics.

Semantic properties 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 embed a tremendous amount of concise information wherever they are used without imposing the overhead inherent in the introduction of new languages and formalisms for similar purposes.

Our current set of semantic properties are listed in Table 2 in the appendix of this article, and are specified in detail in Kiniry’s dissertation [18]. To explain their use, we will focus on particular enabling aspects of kind theory and provide a few examples of their use.

When used in a particular language, semantic properties are realized using appropriate domain-specific extensions. We have integrated their use into the Java and Eiffel programming languages, as well as in the BON specification language [24, 29], through a process that we call *semantic embedding*.

4.1 Java

Semantic properties are embedded in Java code using Javadoc-style comments. This makes for a simple, parseable syntax. To give some flavor to semantic embedding, we’ll present a small example of Java code using semantic properties. Here is an example of such use, taken directly from one of our projects that uses semantic properties [15].

```
/**
 * Returns a boolean indicating whether any debugging facilities are turned off for
 * a particular thread.
 *
 * @concurrency GUARDED
 * @require (thread != null) Parameters must be valid.
 * @modifies QUERY
 * @param thread we are checking the debugging condition of this thread.
 * @return a boolean indicating whether any debugging facilities are turned off for
 * the specified thread.
 * @review kiniry - Are the isOff() methods necessary at all?
 */
public synchronized boolean isOff(Thread thread) {
    return (!isOn(thread));
}
```

The method `isOff` has a base specification, that of its Java type signature. It takes a reference to an object of type `Thread` and returns a value of base type `boolean`. But more than just its type signature is used by the Java compiler and runtime. Additionally, it is a `public` method, thus any client can invoke it, and it is `synchronized`, thus only a single thread of control can enter it, or any other synchronized method in the same containing class (called `Debug`), at once. This computational access control is managed by the Java virtual machine through the use of a monitor object attached to the `Debug` class.

Existing tools already use these properties. Some translate specifications, primarily in the form of contracts, into run-time test code. Reto Kramer’s *iContract* [21], the University of Oldenburg’s Semantic Group’s *Jass* tool, Findler and Felliason’s contract soundness checking tool [11], and Kiniry and Cheong’s *JPP* [20] are four such tools. Other tools translate specifications into structured documentation. Javadoc and its relatives are examples of such tools.

4.2 Kinding with Semantic Properties

Semantic properties are used in kinding an instance (e.g., a method, a class, a type) in the following fashion. We do not have the space to fully detail the related theory and algorithm, but a detailed example should suffice in getting the idea across². We'll focus exclusively on kinding the above Java method to this end.

It should be noted, as discussed in detail in the aforementioned thesis, that most of the below specifications are automatically generated from source assets by domain specific parsers. The basic structure of a program, its annotations, the structure and kind semantics of the programming language in which it is written, etc. are all provided for automatically. Thus, the verbosity of these specification examples should not deter the reader as nearly all of this detail is completely hidden from a typical user.

The *kind* of this method contains much more information than its type. First, the kind contains everything we have already discussed with respect to the method's signature. Additionally, a semantic interpretation of all of the documentation attached to the method is included. And, as a final element, a more complete representation of its type is also incorporated.

In more detail:

1. All the information that is inherent in the *explicit* specification of the method's signature and type are first composed.

We will call this particular instance `Debug.isOff`. With respect to classification, this construct is a Java method. This is encoded as

`Debug.isOff : JAVAMETHOD`

We interpret the method's signature as follows:

`Debug.isOff.ParameterSet \subset_p Debug.isOff`
`Debug.isOff.ReturnType \subset_p Debug.isOff`

where

`Debug.isOff.ParameterSet : JAVAPARAMETERSET`
`Debug.isOff.Parameter0 \subset_p Debug.isOff.ParameterSet`
`Debug.isOff.Parameter0 : JAVAPARAMETER`
`Debug.isOff.Parameter0Type \subset_p Debug.isOff.Parameter0`
`Debug.isOff.Parameter0Name \subset_p Debug.isOff.Parameter0`
`Debug.isOff.Parameter0Type : JAVATYPE`
`Debug.isOff.Parameter0Type \equiv java.lang.Thread`
`Debug.isOff.Parameter0Name : JAVAIDENTIFIER`
`Debug.isOff.Parameter0Name \equiv thread`
`... etc ...`

Effectively, we reflectively encode all of the type and signature information for the method in a kind theoretic context.

The structure of the related kinds like `JAVAPARAMETER` encode the necessary structure that must be provided by the interpretation, a type and a name in this case. Thus, this part of the process is no different than the parsing and typing that goes along with compiling the method.

2. All the supplementary information embedded via the use of the semantic properties and extra-type information is *also* interpreted into a kind theoretic context.

² The interested reader can read [18] at their leisure.

For example, the fact that the method is declared as having GUARDED concurrency semantics is encoded with the concurrent kind:

$$\begin{aligned} & \text{ConcurrencySemantics0 } \subset_p \text{ Debug.isOff} \\ & \text{ConcurrencySemantics0} : \text{JAVAConcurrencySemantics} \\ & \text{ConcurrencySemantics0} \equiv \text{GuardedSemantics} \end{aligned}$$

Likewise, the method’s visibility, signature concurrency (use of the `synchronized` keyword), side-effects (*modifies*), precondition, parameter and return value documentation, and meta-information (*review*) are also interpreted.

3. Finally, we interpret a more complete representation of the method’s type by taking advantage of the domain semantics of refinement. In this example, we attach stronger refinement semantics via the use of the specified, explicit, classical contract.

The existence of the precondition means that, after we interpret such, we can: (a) Check that overridden methods properly weaken the precondition for behavioral subsumption. (b) Interpret such specification into run-time test code—here that entails just a literal insertion of the code snippet into the proper place(s) in a rewritten version of the method. (c) Use this behavioral specification as a guard in semantic composition (see below).

The *kind* of this method is our formal “best-effort” at the specification of its semantics. Now, the rules of kind and instance composition come into play with respect to defining semantic compatibility.

5 Component Kind

Two instances (e.g. objects) are *compatible* if they can inter-operate in some fashion correctly and in a sound manner. For example, the two objects can perform their intended roles in an interactive manner and the composition of the two objects is as correct as the two objects when analyzed individually.

An object O is a realization of a semantic component, that is, $O : \text{SEMANTICCOMPONENT}$, if it provides sufficient information via semantic properties that a kind system can interpret its structure into a predefined kind. That is to say, this “parsing” step is an interpretation to some K which is a semantic component.

A `SEMANTICCOMPONENT` is a kind that contains both `PROVIDES` and a `REQUIRES` kinds. Each of these enclosed kinds specifies a set of kind which the component exposes, or on which it depends, respectively. This two substructures are a generalization of the common *exports* and *imports* clauses of architecture description and component specification languages.

If we wish to determine the semantic compatibility of two instances we use the following formal definition.

Definition 1 (Semantic Compatibility) *Let I and J be two semantic components. That is,*

$$\Gamma, I : \text{SEMANTICCOMPONENT}, J : \text{SEMANTICCOMPONENT} \vdash \diamond$$

Furthermore, I is part of the provisions of an enclosing semantic component C_P , and J is part of the requirements of an enclosing semantic component C_R ,

$$\begin{aligned} & \Gamma \vdash, I \subset_p P, P : \text{PROVIDES}, P \subset_p C_P, \\ & J \subset_p R, R : \text{REQUIRES}, R \subset_p C_R \vdash \diamond \end{aligned}$$

We say that P and R are semantically compatible if an interpretation exists that will “convert” R into the ontology of P , that is if there is a $R \rightsquigarrow P$ in, or derivable from, the context Γ .

I and J are semantically equivalent if $I = J$, of course.

We can test for the existence of such an interpretation by checking to see if the canonical forms of P and R structurally contain each other.

Theorem 1 (Semantic Compatibility Check)

$$\Gamma, [R] \subset_p [P] \vdash R \rightsquigarrow P$$

The proof of this theorem is straightforward. If $[R] \subset_p [P]$ then $[P] \supset [R]$ (by definition of these operators), and thus by the rule (Partial Equiv*), $P \leq R$. By the definition of partial equivalence, when $P \leq R$, a full interpretation exists that takes P to R ; that is, $P \rightsquigarrow R$. This full interpretation is exactly the “conversion” operator that we are looking for to guarantee semantic compatibility.

Finally, we say that I and J can be *semantically composed* if: (a) they are semantically compatible, and (b) their composition is realizable within their instance domain.

This realization within their instance domain is the “glue” code on which we depend for composition. We call this construct a *semantic bridge*. Put another way, a *semantic bridge* is a chain of equivalences between two instances that ensures their contextual base equivalency.

6 Examples

All the examples below are defined independently of source object language and ignore the subtle problems of class and type versioning that are solved in the full system and are not described here. These are only illustrative, not prescriptive, examples.

Finally, the “tight” coupling demonstrated below is theoretically equivalent to the more dynamic coupling found in loosely typed systems. The same rules and implications hold in such an architecture.

Some of the following examples will use the following type:

```
Type DateType
  method setDate(day: Integer;
                 month: Integer;
                 year: Integer);
  method getDate();
EndType
```

Before examining these examples, take note that if two classes are class or type compatible, they are obviously semantically compatible.

6.1 Standard Object Semantic Compatibility

```
Class Date                               Class SetDate
  method setDate(day: Integer;           callmethod writeDate(day: Integer;
                        month: Integer;   month: Integer;
                        year: Integer);   year: Integer);
  method getDate();                     callmethod readDate();
EndClass                                 EndClass
```

The methods tagged with the `callmethod` keyword are part of the REQUIRES of the class; those tagged with the `method` keyword are part of the PROVIDES part.

These classes are *not* type compatible since their outbound and inbound interfaces are of two different types (`DateType` and some new type; lets call it `AnotherDateType`).

If the only difference between the methods `setDate` and `writeDate` is *exactly* their syntax, then these classes *are* semantically compatible.

This mapping exists because both of these classes realize the same kind, that of `DATE`. We know this fact because at some point a developer defined this ontology by virtue of a *claim* or a *belief* on the classes. Such a truth structure could have been defined explicitly through the specification of a semantic property (*realizes*) on the classes, via manipulation in the development environment, or by direct input with the kind system.

Now, because `Date : DATE`, then a mapping exists from each of its parts (e.g., the `setDate` method) to each of the parts of the kind (the canonical `SETDATE` kind feature).

These maps, when used in composition, define a simple renaming (sort of an alpha-renaming for instances) for the classes. We can realize such a renaming either by directly manipulating the source text (if this is permissible in the current context), or by generating a simple wrapper class automatically. Thus, an adapter which maps calls from `writeDate` to `setDate` and from `readDate` to `getDate` will allow the composition of these two classes to perform correctly.

6.2 Extended Object Semantic Compatibility

The above example is based on a simple syntactic difference between two classes. Here is a more complex example.

Consider the following two classes.

```

Class ISODate
  -- requires: year > 1970
  method setDate(year: Integer;
                 month: Integer;
                 day: Integer);
  method getDate(): ISODate;
EndClass

Class SetDate
  -- requires: year > 0
  callmethod setDate(day: Integer;
                    month: Integer;
                    year: Integer);
EndClass

```

To compose an instance of `SetDate` with an instance of `ISODate`, we have to (a) negotiate the reordering of the parameters of the `setDate` method, and (b) check the behavioral conformance of the composition.

This reordering is just another simple form of the above alpha-renaming because the structural operators in kind theory (\subset_p and \supset) are order-independent. The behavioral conformance is verified because $year > 1970 \Rightarrow year > 0$.

Thus, we can again automatically generate the appropriate code to guarantee semantic compositionality.

6.3 Ontological Semantic Compatibility

Our final example is an example of a solution that would rely more strongly upon ontology-based semantic information encoded in kind theory.

Consider the following classes.

```

Class ISODate
  method setDate(year: Integer;
                 month: Integer;
                 day: Integer);
  method getDate(): ISODate;
EndClass

Class OffsetDate
  method setDate(days_since_jan1_1970:
                Integer);
  method getDate(): OffsetDate;
EndClass

```

Assume that the parameter `days_since_jan1_1970` was annotated with a reference to a kind that described `days_since_jan1_1970` meant. The context of such a description would necessarily have to have a *ground*— a common, base understanding that is universal.

In this case, the *ground* element is the notion of a *day*. The relationship between the parameter `days_since_jan1_1970` and the *day* ground element need be established.

This relationship might be constructed any of a number of correct, equivalent manners. For example, a direct interpretation from the triple *year/month/day* to `days_since_jan1_1970` would suffice.

Or perhaps a more complicated, multistage interpretation would be all that is available. For example, the composition of interpretations from year to month, then month to day, then day to `days_since_jan1_1970`, would provide enough information for the generation of a semantic bridge.

This composition of interpretations is automatically discovered and verified using the semantic compatibility theorem via the rewriting logic-based component search algorithm of kind theory.

7 Implementations

The earliest implementation of this research was done by a student (Roman Ginis) working with the author in 1998. We used the Boyer-Moore theorem prover (Nqthm) to hand-specify the above examples (and more) and prove semantic compositionality. Glue code was also written by hand to see what steps were necessary, in typical structured languages, for realizing the resulting interpretations.

We have now developed the theoretical infrastructure to completely describe the semantic components and reason about their composition. We also have a realization of the theory in a kind system implemented in SRI's Maude logical framework [7]. Currently, the realization of renaming, reordering, and simple interpretations is direct: it is snippets of Java program code that are explicit realizations of the corresponding interpretation.

In general,

Our next step is to “lift” these examples into a general context, automatically generating code in a variety of contexts, rather than just using pre-written, parameterized glue code.

To this end, we have developed a distributed component-based web architecture called the Jiki [16] for use as a test-bed of semantic compositionality. The Jiki's components are distributed JavaBeans, and they interact via a number of technologies including local and remote method calls, message passing via HTTP and other protocols, and a tuple-based coordination mechanism based upon Jini's JavaSpaces.

These components have already been specified with both the Extended BON [17] specification language as well as our semantic properties in their program code.

Thus, our next step is to use this example, complex application with an existing specification as a testing ground for automatic generation of glue code for a variety of interface modalities. In the end, we would like to be able to use semantic components as a kind of “Über-IDL”, and generate glue code that utilizes a large variety of equivalent communication substrates, both at compile and run-time.

Abstracting that component-based architecture, particularly with a view toward component quality-of-service, has already been finished. The result of which is what we call the *Connector Architecture*, inspired by Allen and Garlan's similar work [1], and will be described in a forthcoming paper.

8 Conclusion

We have shown that, using kind theory, we can use the same formalism to describe software components, reason about their composition, and generate verifiable “glue” code for their composition. These specifications come in the form of simple domain-independent annotations to typical program code, and such annotations are also used for documentation and testing purposes.

9 Acknowledgments

This work was performed while the student was at the California Institute of Technology in Pasadena, CA. This work was supported under ONR grant JFH1.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”.

References

1. R. Allen and D. Garlan. Beyond definition use—architectural interconnection. *ACM SIGPLAN Notices*, 29(8):35–44, 1994.
2. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
3. D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(2):73–119, 1979.
4. D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

5. Don Batory. Compositional validation and subjectivity and GenVoca generators. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 166–175, 1996. Is sometimes titled "Subjectivity and Software System Generators".
6. S. Castano and V. De Antonellis. A constructive approach to reuse of conceptual components. In *Proceedings of the Second International Workshop on Software Reusability*, pages 19–28, 1993.
7. Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. In *In Proceedings, First International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Science, Inc., 1996.
8. J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.
9. Liesbeth Dusink and Jan van Katwijk. Reuse dimensions. In *Proceedings of SSR '95*, 1995.
10. J. L. Fiadeiro and T. Maibaum. A mathematical toolbox for the software architect. In *Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD8 – '96)*, 1996.
11. R. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of Sixteenth International Conference Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
12. Bernd Fischer. Resolution for feature logics. In *Proceedings of the Workshop der GI-Fachgruppe 'Alternative Konzepte für Sprachen und Rechner'*, pages 23–34, April 1993. Also appears in *GI Softwaretechnik Trends* in August, 1993.
13. David Garlan, R. Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *International Conference on Software Engineering*. IEEE Computer Society, IEEE Computer Society, May 1995.
14. Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, February 1986.
15. Joseph R. Kiniry. IDebug: An advanced debugging framework for Java. Technical Report CS-TR-98-16, Department of Computer Science, California Institute of Technology, November 1998.
16. Joseph R. Kiniry. The Jiki: A distributed component-based Java Wiki, 1998. Available via <http://www.jiki.org/>.
17. Joseph R. Kiniry. The Extended BON tool suite, 2001. Available via <http://ebon.sourceforge.net/>.
18. Joseph R. Kiniry. *Formalizing Open, Collaborative Reuse with Kind Theory*. PhD thesis, California Institute of Technology, 2002.
19. Joseph R. Kiniry. Semantic properties for lightweight specification in knowledgeable development environments. Submitted for publication, 2002.
20. Joseph R. Kiniry and Elaine Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, Department of Computer Science, California Institute of Technology, November 1998.
21. Reto Kramer. iContract—the Java design by contract tool. In *Proceedings of the Twenty-Fourth Conference on the Technology of Object-Oriented Languages (TOOLS 24)*, volume 26 of *TOOLS Conference Series*. IEEE Computer Society, 1998.
22. J. M. Molina-Bravo and E. Pimentel. Composing programs in a rewriting logic for declarative programming. Technical Report LO/0203006v1, Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, March 2002.
23. Patrick A. Muckelbauer. *Structural Subtyping in a Distributed Object System*. PhD thesis, Purdue University, 1996.
24. Jean-Marc Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, September 1992.
25. H. Partsch and R. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.
26. Dewayne E. Perry. Software evolution and 'light' semantics. In *Proceedings of ICSE '99*, 1999.
27. C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, August 1988.
28. Ragnhild Van Der Straeten and Miro Casanova. The use of dl in component libraries - first experiences. In *CEUR Proceedings of the KI-2002 Workshop on Applications of Description Logics ADL'02*, Aachen, Germany, 2002.
29. Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, Inc., 1995.
30. Guijun Wang, Liz Ungar, and Dan Klawitter. Component assembly for OO distributed systems. *IEEE Computer*, 32(7):71–78, July 1999.
31. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.
32. Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proceedings of OOPSLA '94*, October 1994.
33. Andreas Zeller. Software configuration with feature logic. In *Proceedings of the Workshop on Knowledge Representation and Configuration Problems*, pages 79–83, Dresden, Germany, September 1996.

A Semantic Properties Summary

Table 2. The Full Set of Semantic Properties

Meta-Information:	Dependencies	generate	exception	Documentation
author	references	invariant	Pending Work	design
bon	use	modifies	idea	equivalent
bug	Inheritance	require	review	example
copyright	hides	Concurrency	todo	see
description	overrides	concurrency	Versioning	Miscellaneous
history	realizes	Usage	version	guard
license	Contracts	param	deprecated	values
title	ensure	return	since	time/space-complexity