A New Construct for Systems Modeling and Theory: The Kind

Joseph R. Kiniry
Caltech Technical Report CS-TR-98-14
Department of Computer Science
California Institute of Technology
Mailstop 256-80
Pasadena, CA 91125

October, 1998

Abstract

Our primary research goal is the development of theories and technology to facilitate the design, implementation, and management of complex systems. Complex systems, in this context, are any systems which exhibit "interesting" behavior including, but not limited to, nondeterminism, collective or emergent behavior, and adaptability.

We can claim to understand a system only when we can describe how it works (e.g. provide a specification) such that, if it is a constructive system, another can build it. This notion is our constructive peer of the traditional scientific method: repeatability of results is equivalent to repeatability of construction.

Abstraction is recognized as a key to understanding complex systems. While increasing our abstraction level results in a more **complete** metamodel (i.e. we can talk about *more* systems because we can talk about *more* complex systems), it also means a *more* **complex** metamodel.

On the other hand, we don't want to create theories and systems that only an expert can use. We need abstractions that are *useful, comprehensible,* ***and*** *manipulable* by humans (modelers, simulators, designers, developers, tool builders, etc.) **and** systems.

In our experience, the highest-level abstractions in use today (e.g. classes, objects, types, subjects) can not model the systems we are interested in exploring. A higher-level abstraction missing: an "ubertype" of sorts — *a syntactic and semantic bridge between types.*

We call this new abstraction a "*kind*". This paper will briefly describe *kinds* and provide several examples of their use.

1 Introduction

The design, implementation, and management of complex systems is not a new problem. Systems that exhibit the interesting behaviors mentioned previously have been in use for decades. Examples include everything from mainframe-based enterprise information systems to today's World Wide Web.

The tools of the trade. There are many existing tools, theoretical and practical, that are used to understand complex systems. Theories come in many forms, ranging from simple type theory[2] to the extremely complex, and some would argue, unapproachable object[1] and category[38] theories.

Practical tools, all of which are direct or indirect reifications of theoretical work, are either concrete (programmatic) or conceptual. Examples include:

Programming, specification, logic, and modeling languages. Examples include Java[24], VDM[5], HOL[23], and UML[41], respectively,

The tools that support the use and manipulation of these languages. E.g. Various IDEs like JDE[33], specification checkers like IFAD[25], theorem provers and proof assistants like Isabelle[37], and modeling tools like Together/J[34], and

Conceptual advances in systems architecture and models. E.g. metaobject protocols[29], knowledge representation[10, 47], patterns[19], compositional architectures[45], agent technology[4], and specification and proof models like UNITY[8].

Our conceptual models, languages, and tools continue to evolve, becoming more complete and capable everyday. Conversely, from my own personal experience, I postulate that the complexity of the systems that we are attempting to build and use is far out-pacing that which we can understand.

There is no magic bullet. I agree with Brooks[28] and Cox[14]; there is no magic bullet that will make all of this complexity vanish. Extra layers of abstraction (models) or systems (APIs) can help us tackle more complex problems, but usually at a loss of flexibility (we can only consider specific problems) and completeness (the tradeoff of doing more in one domain means we can do less in another).

Our perspective on the problem is different: "*Someone has to do the the hard work*". Building complex systems, even in a compositional manner, still involves understanding complicated components and relationships. Aggregations, especially ones that exhibit interesting properties, are often orders of magnitude more complex than their constituent parts. Thus, no matter what new model, language, or abstraction comes into vogue, someone still has to do the hard work.

Speculation on the state of the art. So the question arises: Why are our systems' complexity out-pacing our capability? We speculate that the answer has three facets, *none of which are technological*:

- *Isolation.* Even in this networked and ever-shrinking world, we are (relatively speaking) islands of thought in a sea of noise. Knowledge is transmitted sporadically in severely limited forms (books, papers, products, presentations) and hoarded avariciously. *How can knowledge and its associated physical and conceptual constructs be shared more efficiently?*
- *Trust.* New constructs (e.g. components, frameworks, techniques, models) are rarely reused because they are often insufficiently trusted by the consumer. Of immediate import, how is a new construct guaranteed to work in the first place? The only technologies that seem to be adopted and widely used are either those that are adopted by community choice (i.e. ISO or IEEE standards) or lack thereof (i.e. Microsoft *de facto* standards). *How can we guarantee that a new construct or model works as advertised?*
- *Economic.* Finally, rapidly becoming the most critical factor today with the widespread adoption of object-oriented languages and architectures based upon compositional principles is the following conundrum: *How can those who do the hard work reap rewards for their labor?*[1]

A new theoretic, conceptual, and practical tool: the *kind*. Because this author is, fundamentally, one half an engineer and one half a theoretician, in thinking about this problem I have come to the conclusion that *a new conceptual artifact with complementary practical tools, with a firm theoretical grounding, is necessary to help solve these problems.* My first published thoughts on the matter can be found in my second M.S. thesis[32] as well as in a recent paper[9]. The further reification and refinement of these ideas resulted in a new conceptual construct that I call a *kind*.

Why introduce *kind* now? Before going into details about what a *kind* is and how it can be used, we should consider the more relevant question: *Why can/should a new conceptual construct, such as the **kind**, be introduced now?*

In short, the answer is one of *multi-domain critical state.* I believe that we are nearing a critical point in the evolution of our *systems* (what we can and cannot accomplish), *connectivity* (information representation, sharing, and collaboration), and *collective mind-set* (commonplace virtual enterprises, code distribution, and coöpetition). Therefore, what previously might have been an unrealistic architecture and model for solving problems that didn't yet exist, now becomes an obvious and necessary additional to our set of tools.

[1] The most passionate early advocate of this problem is Cox[13], though we would argue that alternative thinkers like Stallman also fit into the picture.

Three Postulates/Axioms. Before discussing *kind*, we will present three postulates on the road to *kind*. These three statements can be viewed as the *conceptual axioms of kind* and will provide a proper frame of reference for the reader.

2 Modeling Entities, Data, and Meta

This section will briefly present the three *conceptual axioms of kind*. They can be summarized (perhaps obliquely) as follows:

Axiom 1 *Relationships are entities. There is no distinction between a thing and how things relate to one another.*

Axiom 2 *The only distinction between behavior and data is a frame of reference. Behavior must be encoded to be understood, manipulated, or executed.*

Axiom 3 *Conceptual metalevels exist independent of relations; only ground concepts are idempotent. In other words, state (and thus behavior) is potentially applicable at arbitrary metalevels; only the ground concepts, the core constructs of a given system, have no meaning outside of their reflective existence.*

Entities and Relationships.

By axiom 1, systems are composed of two types of first-class constructs: *entities*, often represented in the form of classes, objects, data structures and the like, and their *relationships*. Many existing conceptual models and systems view these two constructs as distinct. My claim is that they are not: *relationships are a specialization of entities.*

Relationships come in several forms, inheritance[12], connectors[20, 43], and aggregation[7] being the most common. All of these constructs can be described and utilized as first-class entities. They can be formally modeled, specialized, applied to other constructs, and refined. Individual relationships also have relationships to each other, thus they are a recursive structure.

Thus, relationships are simply a recursive specialization of ground entities: *relationships are entities.*

Dimensionality of Modeling: Data and Behavior.

Axiom 2 says the following: System views are composed of data and behavior, reified in the form of classes in object-oriented systems. For years, designers took a behaviorally-oriented approach to system design in the form of procedural decomposition. Then large-scale systems began to proliferate (e.g. N-tier, mainframe-based systems) and a data-oriented evolution in perspective became necessary because the application *was* the data. Finally, and most recently, a synthesis of the historical behavior- and data-oriented viewpoints dominates the market in the form of object-oriented systems.

Now, we have reached a critical juncture in the evolution of our complex systems, most clearly seen in the Web. Behavior is encoded as data (applets, Javascript, etc.) and data is used to instantiate behavior at run-time (active server pages, WebObjects, etc.). The problem is that data and behavioral encapsulation has been smashed to the wind. *There is little distinction anymore between data and behavior, but I argue that this is only true because we can no longer differentiate between the two.*

Unsurprising, we consider data and behavior to be two facets of the same construct. But, unlike most of today's systems, we believe that the sovereignty of base entities need be respected — encapsulation need be rigorously maintained. Likewise, we collapse the differentiated constructs of data and behavior: that which are descriptive (non-operational specification), that which are executable (code), and that which have both properties (executable specification). All are simply aspects of ground effects: *the only distinction between behavior and data is a frame of reference.*

Perspectives: Ground and Meta

Axiom 3 tells us that every system has many abstraction levels. The bottommost level, that which is usually the most simple, concrete, and applicable, is called *level-0* or *the ground level.* Each application of abstraction has a frame of reference. That frame of reference potentially defines a new *metalevel.* If a frame of reference F depends upon constructs in levels i, j, and k, then F's metalevel is at least $\max(i, j, k) + C$ where $C \geq 1$.

Most systems have (conceptually) arbitrarily many metalevels in their abstraction lattice. Today's systems' lattice depth is usually limited to a *level-3* or *level-4* metalevel. This limitation exists primarily because of lack of conceptualization, representation, and manipulation capabilities in today's languages and tools. Fixed frames of reference are provided by conceptual models and languages because the complexity of representational abstractions grows very quickly. Examples of such systems include metaobject protocols and meta-aware modeling languages/systems like UML, Catalysis[16], and OOCL[44].

These finite frame of reference boundaries are artificial constructs. *Conceptual metalevels exist independent of relations; only ground concepts are idempotent.* Meaning, we should be able to define as many conceptual levels as necessary to completely and accurately describe a concept or relation. In other words, the k in a *level-k* metalevel should be independent of the complexity of the level's concepts.

Absolute and Relativistic Ground

We postulate that some ground concepts are axiomatic and independent of any context. Standard examples include the integers, the notion of a set, etc. All other ground concepts are context sensitive; given a particular frame of reference, all concepts that are not defined in terms of other concepts are ground concepts, but only for that frame of reference. We are investigating this notion further.

Collapsing the Models

As one can see, the three *axioms of kind* are all about collapsing models. We are simplifying base constructions and concepts so that the resulting model will not be overburdened with core concepts and artificial structure.

Now, before briefly discussing *kind*, we will look at the current state of the art with respect to abstraction, especially with regards to the term "meta".

3 Metamodeling and Metalevels

There seems to be much confusion in the field today as to what exactly is and isn't *meta*. Meta is a term in vogue, most often applied to languages, systems, and systems that deal with data. Most disturbingly, most things designated as "meta" today do not mention a frame of reference; in other words, one is never told what construction is being subsumed by the meta-construct.

Simply put, meta means means "more comprehensive". It is a term that is normally used with the name of a discipline to designate a new but related discipline, designed to deal critically with the original one[27].

In our context, a concept is considered "meta" *only in relation to other concept(s).* A system S is meta with respect to another system S' only if S completely characterizes S'. Put another way, everything in S' can be described in S and there are concepts in S that cannot be described in S'.

Metamodeling and Metalevels.

Given our working definition of meta, let's examine metamodeling and metalevels.

Metamodeling. *Metamodeling* is the result of the process of analyzing and designing models about existing models. Architecturally, a metamodel of a modeling language describes the abstract concepts and operations that exist within the base language. Good examples of metamodels are the UML metamodel found in [11], the OPEN metamodel used in [18], and the COMMA meta-method discussed in [26].

Metalevel. A *level* is a *frame of reference*, or a *level of abstraction*, within a model. Excellent examples of metalevels are found in mathematics.

For example, consider a simple system $Z+$, defined as addition on integers. Several abstractions of this system exist: algebraic group theory[17] and analysis[39] being the obvious abstractions. These two theories can completely describe, in a succinct, complete, and accurate fashion, everything there is to know about $Z+$. They are, as universes of concepts, a metalevel above the level at which $Z+$ rests.[2]

Examples of Meta.

Examples of meta are everywhere, and are becoming more prevalent in computing every day. Appendix A contains few examples of meta; some are obvious and some obscure.

As the reader can clearly see, meta is not only everywhere around us, but is now recognized as a valuable asset and is incorporated into many modern architectures. Everyday examples include advanced Web search engines, corporate data-mining, and open architectures.

Unifying Ground and Meta

The important point to take away from this discussion of meta is this:

Theorem 1 *Entities in a universe are either* ***ground*** *concepts, (a fundamental, basic metaphysical cause, condition, or entity), or they are* ***constructive*** *concepts — they are* ***never*** *"meta", without some frame of reference.*

Therefore, when we talk of a concept[3] C, we *can not* discuss its universe U^c, its ground elements $G(C)$, or its metalevel l_c, without fixing a frame of reference $F(C)$. Thus, concepts can be completely divorced of their environment and are applicable as entities in and of themselves.

And thus, we come to understanding and appreciating *kind*.

4 A Model for Kind

Instead of providing the core mathematical axioms, theorems, and properties of *kind*[4] (which are still under development), we will motivate what *kinds* are and their uses by discussing a few examples.

More details on the publication and discovery of *kind*, and thus types, classes, interfaces, implementations, specifications, etc. can be found in [9].

A Definition of Kind.

A *kind* is a specification of a concept (in an arbitrary language) and a specification of meta-information about the concept in a formal specification language. Due to axiom 1, *kinds* can define static and dynamic n-ary relationships between

[2] There are several additional mathematical meta-theories above $G(Z+)$ and $F(Z+)$: *model theory*[22] describes how theories such as algebraic group and analysis theories relate to each other, and *category theory*[3, 38] can help describe how such characterizations of theories relate to one other.

[3] Three random examples of concepts: a class in the last model you designed, the relationship between you and your bank, and the first idea you had when you woke up this morning.

[4] Initial details are forthcoming in a second paper on semantic component composition[31].

other *kinds*. Axiom 2 implies that concepts need not have a physical manifestation (e.g. code); they are only conceptualizations which can be viewed as data or behavior, depending upon the viewer's context.

While the specification of a concept can be made in an arbitrary language, the specification of the meta-information associated with that concept must be made in a language that is usable by both humans and computer.

Our formal language of choice at this time is inspired by the Conceptual Knowledge Markup Language (CKML)[10, 35] and other knowledge representation systems[40]. CKML is a specification language for the conceptual representation and analysis of networked resources. It is fully integrated with the Web and has a formal grounding in knowledge representation and theory work of many researchers (a few references include [6, 42, 46, 48]).

Examples of Usage.

One particularly simple but compelling motivation for the use of *kind* is found in the domain of what I call *semantic component coupling*. More examples that fall in this domain can be found in [31].

Semantic Components: Problem Summary. Components communicate with messages which can be realized as method invocations.[5] Under most circumstances, objects implemented in different languages and objects written by independent developers can not communicate without significant work on the part of a developer.

Often times, the objects *should* be able to communicate, if only a little bit of "glue" existed to help them work together correctly.

Missing from all object/component systems is any notion of explicit semantics. Instead, objects communicate only by virtue of shared standards or syntax. This limitation is most evident in systems which required massive amounts of *integration*. Such systems have the properly that the total system is more complicated and fragile than the sums of the original parts.

Thus, the problem can be reduced to the following: Given the specification for N objects, which objects are semantically compatible?[6]

Definitions. Our examples will make our motivational domain clear: reuse in object-oriented systems. A few definitions are first necessary to follow the later discussion on *kind*.

Object Compatibility. Two objects are *compatible* if they can interoperate correctly and in a sound manner. Meaning, the two objects can fulfill their individuals obligations and the composition of the two objects is as correct as the two objects when analyzed individually.[7]

[5] See [30] on issues relating to this statement.

[6] A further refinement is, of course, given two components, or even two methods of two components, are they semantically compatible?

[7] The formal definition of compatibility and the other terms herein is available in [31].

Object Specification. An object specification is (minimally) a description of an object that is *complete*.

An object specification can contain extra meta-information that is *not* implicit in the object in question. Information associated can be tagged as *optional*. This information need not be considered when determining semantic compatibility.

A *core specification* is a specification that includes exactly those elements of a specification which are *implicit* and those that are *not* optional.

Complete. *Complete* means that the specification *explicitly* describes every *implicit* feature of the object in question. Features include object fields, methods, class, and type.

Specification Equivalence. Two object features are considered *equivalent* if:

- Their core specifications are exactly ground equivalent **or**,
- Their core specifications are equivalent through *semantic bridges*.

Semantic bridge. A *semantic bridge* is a chain of equivalences between two features that ensures their base equivalence. See the examples for more details on semantic bridges.

Semantic Compatibility. Two objects are semantically compatible if their core specifications are *equivalent* **and** their non-optional meta-information specifications are *equivalent*.

4.1 Examples

All the examples below are defined independently of source object language. Examples in specific relevant languages (e.g. Java, Python, Smalltalk) will be provided in the near future and as part of the implementation.

Note also, the following examples are ignoring the subtle problems of class and type versioning that are solved in the full system. These are only illustrative, not prescriptive, examples.

All the following examples will use the following classes:

```
ObjectType IllegalDateException
    var String message;
    method setMessage(message: String);
    method getMessage(): String;
end;

ObjectType DateType
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
end;
```

Note that the "tight" coupling demonstrated below is equivalent to the more dynamic coupling (with publishers and listeners) found in the Java event model (i.e. AWT, Beans, Jini). The same rules and implications hold in such an architecture.

4.1.1 Example 1: Standard Object *Class* Compatibility

Assume we have instances of the following two components. Note that the keywords in the object specifications below are adopted from [1]. `Class`, `Type`, `Fields`, and `Method` behave as expected. Imprecisely, think of classes, interfaces, attributes, and methods, respectively, in the Java language. *Dependence* methods are those methods that a component needs use to work correctly. Again, imprecisely, consider JavaBeans publishers or standard Java inline references to method invocations.

Consider the following two classes:

```
Class Date
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
end;

Class SetDate
   callmethod Date.setDate(day: Integer;
                           month: Integer;
                           year: Integer);
   callmethod Date.getDate();
end;
```

These classes *are* type compatible since their outbound and inbound type interfaces are of the same class (`Date`). Thus, `Date` and `SetDate` can be composed and the system will exhibit correct behavior, assuming that type conformance is not accidental.

4.1.2 Example 2: Standard Object *Type* Compatibility

Consider the following two of objects. Note that the dependent methods have changed slightly.

```
Class Date
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
end;
```

```
Class SetDate
  callmethod setDate(day: Integer;
                     month: Integer;
                     year: Integer);
  callmethod getDate();
end;
```

These classes *are* type compatible since their outbound and inbound type interfaces are of the same type (`DateType`). Thus, `Date` and `SetDate` can be composed and the system will exhibit correct behavior.

Both of the above examples require no additional work other than understanding the component specifications on the part of a developer, but do require considerable forethought on the part of the object designer.

4.1.3 Example 3: Standard Object *Semantic* Compatibility

```
Class Date
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
end;

Class SetDate
    callmethod writeDate(day: Integer;
                         month: Integer;
                         year: Integer);
    callmethod readDate();
end;
```

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

But, let's assume that the only different between the methods `setDate()` and `writeDate()` is *exactly* their syntax. Given this assumption, these classes *are* semantically compatible.

Thus, an adaptor which maps calls from `writeDate()` to `setDate()` and from `readDate()` to `getDate()` will allow the composition of these two classes to perform correctly.

4.1.4 Example 4: *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
    var day: Integer;
    var month: Integer;
    var year: Integer;
    method setDate(year: Integer;
                   month: Integer;
                   day: Integer);
    method getDate(): ISODate;
end;

Class SetDate
   callmethod setDate(day: Integer;
                      month: Integer;
                      year: Integer);
end;
```

To compose an instance of `SetDate` with an instance of `ISODate`, we have to negotiate the reordering of the parameters of the `setDate()` method. This reordering could be discovered at runtime via introspection on the parameters of the invoking and the receiving methods because the parameter syntax and types (luckily) match.

4.1.5 Example 5: *Ontological* Object *Semantic* Compatibility

Our final example is an example of a solution that would rely upon ontologic-based semantic information. An example of such a system is in the form of ontology markup references with the Ontology Markup Language[36] within an object description, as in CKML.

Consider the following classes.

```
Class ISODate
    var day: Integer;
    var month: Integer;
    var year: Integer;
    method setDate(year: Integer;
                   month: Integer;
                   day: Integer);
    method getDate(): ISODate;
end;

Class OffsetDate
    var days: Integer;
    method setDate(days_since_jan1_1970: Integer);
    method getDate(): OffsetDate;
end;
```

In this frame of reference, (kind theorem 1), 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. In general, the parameter `days_since_jan1_1970` need be annotated (the structured metainformation that is part of a *kind* definition) with a reference to a concept (a *kind*) that describes the semantics of `days_since_jan1_1970`.

Here are examples of two such *kinds* (motivated by the two sides of *kind* axiom 2):

By *kind* axiom 2, this concept could either be data (e.g. a lookup table) or behavioral (e.g. a component that converts `days_since_jan1_1970` to a year, month, day format.

1. The relationship between the ground concept *day* and the concept *days_since_jan1_1970* could be described in data. E.g. a lookup table might be provided that describes the static translation between instances of the two concepts.

2. Alternatively, a behavioral *kind* could be provided. This would come in the form of a piece of code (a component) that dynamically performs the transformation between instances of the two elements.

We hope that even from this simple example, the usefulness and applicability of *kind* can be understood. Of course, there is a great deal of complexity hidden under this example which we do not address in this brief document. But we hope that the reader can understand what *kind* are all about and where this work is heading.

5 Conclusion

This researcher's PhD thesis involves the exploration of the theory, use, and application of *kind*. I am working to rigorously leverage and extend existing applicable models (e.g. specification and proof models like UNITY[8]) and theories (classical type theory[2], object theory[1], category theory[3], and knowledge representation theory[48], especially in the context of software engineering[21]) in application to the problem of distributed, collaborative, chaotic, dynamic software specification, construction, and reuse. This work will result in a *theory of kind*, specifying the formal grounding of the work, as well as a simple and usable application, development model, and development process incorporating the use of *kind* in component-based software engineering.

5.1 Future Work

Work continues in the development of the *theory of kind*. A demonstration application called Jiki[8], realized as an open web architecture for component

[8]See `http://www.jiki.org/` for more information.

specification based up the Wiki web[15] has being designed and built by the Infospheres group. We will use this application as a motivating demonstration of the usefulness and applicability of *kind*.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.

[2] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[3] Andrea Asperti and Giuseppe Longo. *Categories, Types, And Structures: An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computer Science. The MIT Press, 1991.

[4] Robert Axelrod. *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton University Press, 1997.

[5] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer–Verlag, 1994.

[6] K. Biedermann. How triadic diagrams represent conceptual structures. *Conceptual Structures: Fulfilling Peirce's Dream*, 1257:304–317, 1997.

[7] C. Bock and J. Odell. A more complete model of relations and their implementation: Aggregation. *Journal of Object-Oriented Programming*, 11(5):68–70, 1998.

[8] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[9] K. Mani Chandy, Paolo Sivilotti, and Joseph R. Kiniry. A cottage industry of software publishing: Implications for theories of composition. In *Proceedings, FMPPTA '98: Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, Lecture Notes in Computer Science. Springer–Verlag, April 1998.

[10] Conceptual Knowledge Markup Language (CKML) DTD. `http://asimov.eecs.wsu.edu/WAVE/Ontologies/CKML/CKML-DTD.html`.

[11] Rational Software Corporation et al. *UML Semantics, version 1.1*. The UML 1.1 Consortium, September 1997.

[12] J.F. Costa, A. Sernadas, and C. Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31(1):5–26, 1994.

[13] Brad Cox. *Superdistribution: Objects As Property on the Electronic Frontier*. Addison-Wesley Publishing Company, 1996.

[14] Brad Cox and Andrew Novabilsky. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley Publishing Company, 1987.

[15] Ward Cunningham. The wiki wiki web. WWW, 1998. `http://c2.com/cgi/wiki?WikiWikiWeb`.

[16] Desmond D'Souza and Alan Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley Publishing Company, 1998.

[17] John R. Durbin. *Modern Algebra*. John Wiley & Sons, Inc., 1985.

[18] Donald G. Firesmith, Brian Henderson-Sellers, and Ian Graham. *OPEN Modeling Language (OML) Reference Manual*. Cambridge University Press, 1998.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[20] David Garlan. Higher-order connectors. In *Proceedings of Workshop on Compositional Software Architectures*, Monterey, California, January 1998.

[21] R. Godin, G. Mineau, R. Missaoui, M. Stgermain, and N. Faraj. Applying concept-formation methods to software reuse. *International Journal Of Software Engineering And Knowledge Engineering*, 5(1):119–142, 1995.

[22] Martin Goldstern and Haim Judah. *The Incompleteness Phenomenon: A New Course in Mathematical Logic*. A.K. Peters, 1995.

[23] M. J. C. Gordon and T. F. Melham, editors. *Introduction To HOL: A Theorem Proving Environment For Higher Order Logic*. Cambridge University Press, 1993.

[24] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996.

[25] IFAD Group. IFAD VDM tools. `http://www.ifad.dk/products/vdmtools.html`.

[26] B. Henderson-Sellers and A. Bulthuis. *Object-Oriented Metamethods*. Springer–Verlag, 1997.

[27] Mirriam-Webster Inc. *Webster's Ninth New Collegiate Dictionary*. Mirriam-Webster Inc., first digitial edition edition, 1992.

[28] Frederick P. Brooks Jr. *Mythical Man Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, 1995.

[29] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[30] Joseph R. Kiniry. On the equivalency in communication models: Messaging, rpcs, events, and tuple-space operations. Will be available as a Caltech technical report. Email the author for information., 1998.

[31] Joseph R. Kiniry. Semantic component composition. California Institute of Technology Technical Report CS-TR-98-13, California Institute of Technology, October 1998.

[32] Joseph R. Kiniry. The specification of dynamic distributed component systems. Master's thesis, California Institute of Technology, May 1998.

[33] Paul Kinnucan. *Emacs JDE (Java Development Environment)*, 1998.

[34] Object International, Inc. *Together/J 2.1 User Guide*, 2.1 edition, 1998.

[35] J.D. Olson and R.E. Kent. Conceptual knowledge markup language, an XML application. Unpublished presentation, given at the XML Developers Day, August 21, 1997, Montreal Canada, August 1997.

[36] Ontology Markup Language (OML) DTD. `http://asimov.eecs.wsu.edu/WAVE/Ontologies/OML/OML-DTD.html`.

[37] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[38] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computer Science. The MIT Press, 1991.

[39] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, Inc., 1976.

[40] R. Ruggia and A.P. Ambrosio. A toolkit for reuse in conceptual modelling. *Advanced Information Systems Engineering*, 1250:173–186, 1997.

[41] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Publishing Company, 1997.

[42] J.J. Sarbo. Building sub-knowledge bases using concept lattices. *Computer Journal*, 39(10):868–875, 1996.

[43] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[44] Edward Swanstrom. *Creating Agile Organizations with the OOCL Method*. John Wiley & Sons, Inc., 1998.

[45] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1997.

[46] P.E. van der Vet and N.J.I. Mars. Bottom-up construction of ontologies. *IEEE Transactions On Knowledge And Data Engineering*, 10(4):513–526, 1998.

[47] R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics With Applications*, 23(6-9):493–515, 1992.

[48] R. Wille. Conceptual graphs and formal concept analysis. *Conceptual Structures: Fulfilling Peirce's Dream*, 1257:290–303, 1997.

A Examples of Meta

- System run-time behavior (metaobject protocols, pragmas)
- Meta-data (databases, repositories, COM+, CORBA, XML, OMG, Coins, WWW, digital library, search)
- Reflection and introspection (Java, CLOS, OMG).
- Meta-architectures and metamodels (aspects, previously mentioned metamodels).
- Knowledge representation and multi-agent communities (KQML, KIF, agent systems)
- Specification (propagation patterns, contracts, features, views, roles, design by contract).
- Machine processable abstractions (Biggerstaff and Richter, Demeter, contracts, XML).
- Knowledge Representation (CKML, OML).
- Specification languages (VDM, Z, Larch)
- Models for specification and reasoning (UNITY, Actors)

Readers are welcome to suggest other metalevel systems to the author.