Leading to a Kind Description Language: Thoughts on Component Specification*

Joseph R. Kiniry
Caltech Technical Report CS-TR-99-04
Department of Computer Science,
California Institute of Technology,
Mailstop 256-80,
Pasadena, CA 91125

May, 1999

Abstract

The Kind Description Language (KDL[1]) is a language used for describing the interface and behavior of software components. KDL is an extension of the Object Management Group's Object Constraint Language (OCL). While OCL is only able to describe *safety* properties of a component and its features, KDL can also describe *progress* properties with temporal operators like *leads-to*. KDL also introduces several new "convenience" constructs that help simplify and clarify complex component descriptions. KDL can be used to specify a component's simple behavioral interface, as in Meyer's *Design by Contract*, the more complex temporal properties that distributed objects and components exhibit, and more.

1 Introduction

Specification languages are used to describe a variety of entities, programmatic or otherwise. Languages range in formality from the very informal, e.g. the typical natural language source code comment, to the extremely formal, e.g. specification languages like Z and VDM.

In this author's experience, a successful specification language should be simple enough for a non-expert programmer to use, but formal enough that programming tools can utilize the specifications for component testing and compo-

 A version of this paper was presented at the COOTS '99 Advanced Topics Workshop on Validating the Composition/Execution of Component-Based Systems.

[1] Our original name for this language, the Component Description Language (CDL), unfortunately had been used a number of times.

$$
\begin{array}{lll}
E & \equiv & \textbf{procedure} \\
 & & [\,\overline{in} \longrightarrow in?x;\;\; F \\
 & & \;\;\square\; \overline{out} \longrightarrow get?x;\;\; out!x;\;\; E \\
 & &]\ \textbf{end} \\
F & \equiv & \textbf{procedure} \\
 & & [\,\overline{out} \longrightarrow out!x;\;\; E \\
 & & \;\;\square\; \overline{in} \longrightarrow put!x;\;\; in?x;\;\; F \\
 & &]\ \mathit{end}
\end{array}
$$

Figure 1: A CSP specification of a stack element.

sition validation. We have designed KDL as an example of this balance between usability and formality.

What follows is a brief overview of the different types of specification languages in use today. Each example language was chosen only because it highlighted a different aspect of specification. The strengths and weaknesses of each language type are summarized, and this analysis motivates the design of KDL.

Our running example is the archetypical LIFO stack. Note that, for the most part, the specifications below describe general stacks, without commitment to a representation. The fundamental operations we will consider are `push` (to add an element to the top of the stack), `pop` (to remove the top element from the stack), `empty` (to test for an empty stack), and `clear` (to remove all elements from a stack).

2 Specification/Description Languages

In computing systems, the entities that are described are contextually dependent upon the theoretic and/or operational model of the system in which the entities exist. In other words, each specification language has a particular domain, model, and goal. We will show that existing specification languages are not applicable to the complex component architectures used today in concurrent, distributed components and applications.

2.1 Operational Specifications

Our first example is Hoare's Communicating Sequential Processes (CSP) [12]. As its name would indicate, CSP is a operational language for describing processes that are inherently sequential, communicate with messages, but collectively run concurrently. CSP is an operational language because it exactly describes the externally observable *behavior* of the processes, but not *how* they perform the operations specified (i.e. how data structure and types are encoded). The interface of the processes is *implicitly* specified via the operational description. Other operational languages include UNITY [5] and Structural Operational Semantics (SOS) [25].

See Figure 1 for an example CSP specification of a stack. Due to space issues, we only provide the specification of a single stack element in CSP and do not

```
/* An archetypical LIFO stack. */
module stack_module
{
  interface Stack: Finite_Data_Structure, Container

  /* push a new item onto the stack */
  void push(in Any item);
  /* pop the top item off the stack */
  Any pop();
  /* is the stack empty? */
  boolean empty();
  /* clear the contents of the stack */
  void clear();
}
```

Figure 2: An IDL specification of a stack.

specify the *empty* and *clear* channels. Note that we have directly mapped the standard stack interface specification to the CSP port/message communication model. Each method of our archetypical stack component is modeled as a channel in the CSP specification. This is an example of the implicit specification of a component's interface, as mentioned above.

2.2 Interface Specifications

Alternatively, a specification language might describe only the *interface* of an entity, rather than its operational behavior. CORBA's Interface Definition Language (IDL) is an example of such a specification language.

See Figure 2 for an example of a IDL specification of our stack. Note that IDL only lets us specify the *syntactic* interface of the component. In particular, only method signatures and type structure can be specified in IDL.

2.3 Expression Languages.

Some languages are considered pure *expression* specification languages. These languages are not meant to be used to denote the operational semantics of an entity. Instead, they specify predicates on the state of an entity. Typically, these predicates come in the form of pre- and post-conditions on interface features (operations like functions, procedures, and methods) and invariants on single entities or groups of entities.

Some languages provide direct support for such constructs. For example, the Eiffel programming language [23] has *requires*, *ensures*, and *invariant* keywords. These are examples of expression specification constructs. Additionally, the OMG's Object Constraint Language [28] is such an expression language.

Figure 3 is an example of a OCL specification of our stack. Note that OCL lets us specify both the interface of the component and the *externally observable* side-effects of each method. OCL does *not* let the designer specify implementation details, such as how the size attribute is manipulated. Contrast this with the Object-Z specification of the same stack in Figure 4.

```
-- An archetypical LIFO stack.
Stack
-----
-- size is an Integer attribute of the Stack component

-- push a new item onto the stack
::push(OclAny : item): Void
  pre:  -- none
  post: item = self.top()
        size = size@pre + 1

-- read the top element of the stack
::top(): OclAny
  pre:  size >= 1
  post: -- none

-- pop the top item off the stack
::pop(): void
  pre:  size >= 1
  post: size = size@pre - 1

-- is the stack empty?
::empty(): Boolean
  pre:  -- none
  post: result = (size = 0)

-- clear the contents of the stack
::clear(): Void
  pre:  -- none
  post: self.empty() = true
        size = 0
```

Figure 3: An OCL specification of a stack.

Stack

$size : \mathbb{N}$

$size \geq 0$

An archetypical LIFO stack containing integers.

$stack : \operatorname{seq} \mathbb{Z}$

INIT

$stack = \langle\, \rangle$
$size = 0$

push

$\Delta stack$
$item? : \mathbb{Z}$

$stack' = \langle item? \rangle \frown stack$
$size' = size + 1$

pop

$\Delta stack$
$result! : \mathbb{Z}$

$size >= 1$
$result! = head\ stack$
$size' = size - 1$

empty

$result! : \mathbb{B}$

$result! = (size = 0)$

clear

$\Delta stack, size$

$stack = \langle\, \rangle$
$size = 0$

size

$result! : \mathbb{Z}$

$result! = size$

Figure 4: An Object-Z specification of a stack.

2.4 Hybrid Specification Languages

Finally, some languages support the specification of both an entities interface *and* its operational behavior. Examples include the Z [4, 8], VDM [9, 13], and Larch [11] families of languages, including the recent Java-centric JML [20]. Because these languages are meant to describe fairly complex software entities, they are (perhaps sometimes overly) complex themselves. Figure 4 is the specification of our simple stack in Object-Z.

It is clear from this example that, while we can specify *everything* about the behavior and interface of a component with a language like Object-Z, it is entirely too complex for nearly all developers. Languages of this class often take *years* to master; years that no one in the competitive world of application development has to spare.

2.5 Language Analysis

To summarize strengths and weaknesses of the aforementioned language types:

- *Operational Languages (e.g. CSP).*
  - *Pro:* These languages let us specify the complete behavior and (implicitly) the interface of concurrent systems.
  - *Con:* Such specifications have too much detail, require too much expertise, and do not support object-oriented or component-based models. Proofs based upon such specifications are often very complex and subtle.
- *Interface Specification Languages (e.g. IDL).*
  - *Pro:* These languages let us specify the syntactic interface and type system of a component in a language-independent manner.
  - *Con:* These languages do not let us specify anything about the behavior or semantics of components.
- *Expression Specification Languages (e.g. OCL).*
  - *Pro:* These languages let us specify the syntactic interface and type system of a component and the externally observable behavior of its methods.
  - *Con:* These languages do not let us specify anything interesting about components that do not have method invocation-based interfaces. They also do not let us specify anything about the internal behavior of components.
- *Hybrid Specification Languages (e.g. Object-Z).*
  - *Pro:* Hybrid specification languages can specify everything about a component that the previous three languages can (interface, type, behavior, etc.).

– *Con:* These languages are extremely difficult to learn and use, few tools exist to support such languages, and most do not support object-oriented or component-based systems.

Finally, none of the above languages can capture the higher-order semantics of components or specifications nor the relationships between such constructs.

3 A Component Description Language

In this section we will briefly summarize KDL. The interested reader can find more information on KDL in [15, 16].

3.1 Theoretic and Operational Model

KDL's theoretic model is that of Abadi and Cardelli's second order object calculi as exemplified in their O-2 language [1]. This theoretic model provides classes, types, encapsulation, dynamic polymorphic method invocation, constrained parametric generic classes, and multiple inheritance.

The operational environment of KDL is a generalized component software model, capable of describing all of today's component architectures including CORBA, JavaBeans, COM, and other more esoteric variants [27]. Components are self-contained units of reuse, usually provided in the form of classes, frameworks, libraries, or patterns.

3.2 Specification Capabilities

When specifying an entity, in the case of KDL, a software component, we describe its interface, its behavior, or its implementation. KDL is a pure specification language, thus says nothing about a component's implementation. KDL only describes a component's interface, its semantic behavior, and its semantic context.

KDL supports syntactic and semantic specification via a set of constructs defined within the context of the object and component model of KDL. In other words, the core constructs of KDL have all been defined with object-oriented and component-based systems in mind. These constructs range from the standard syntactic interface description, as found in IDL and OCL, to semantic relationships between component specifications and implementations (e.g. component A is equivalent to component B under conditions C).

KDL is a synthesis of the Object Management Group's Object Constraint Language (OCL), temporal operators as found in UNITY as inspired by Pnueli [21], and conceptual knowledge representation and manipulation operators as found in [30] and applied in e.g. [10, 26].

Since KDL is an extension of OCL, we begin with all of OCL's benefits: succinct specification of syntactic component interface, type information, and externally observable behavior (pre- and post-conditions, component and class invariants, etc.).

While OCL is only able to describe *safety* properties of a component and its features, KDL, via the use of UNITY/temporal operators, can also describe *progress* properties with temporal operators like *leads-to*. For example, with the *leads-to* operator we can state that an arbitrary predicate will *eventually* hold true in a finite number of computation steps. This type of predicate cannot be specified in most of the specification languages mentioned in this article[2].

KDL also introduces several new "convenience" constructs that help simplify and clarify complex component descriptions. These constructs are motivated by many different domains:

First, relationship operators, like those specified in the UML meta-model [7], are defined. Examples include collection and relationship operations (e.g. *cdlHasA*, *cdlContainsA*, and other more complex aggregation constructs [3, 17]). These operators let us describe the (often times hidden) ways that our components relate to each other, how changes can propagate through our models, and more.

Second, we add conceptual knowledge representation and manipulation-inspired kind-theoretic [15] extensions of OCL's type-theoretic [2] operations to our language. Examples include *cdlIsTypeOf* and *cdlIsKindOf*. These operations let us describe how component designs, specifications, and implementation relate to each other, *even across object models and component architectures.* A base ontology of components is provided as part of the theory of kind. The inquisitive reader should consult the forthcoming [18] for much more information.

Additionally, some of KDL's structuring constructs are inspired by the Conceptual Knowledge Markup Language [6] and its subset, the Ontology Markup Language [24]. Our metadata core is influenced by the Dublin Core Metadata [29] as an example of a common, simple, core metadata specification.

To summarize, KDL can not only be used to specify a component's interface and externally observable behavior, (what one could accomplish through the simultaneous use of OCL, a VDM or Z variant, and Meyer's *Design by Contract* [22]), but also detail the more complex temporal properties and semantic relationships of today's concurrent and/or distributed objects and components.

3.3 An Example of KDL

To give the reader a impression of what a KDL specification looks like, we provide a small example. Figure 5 is a sample specification of our stack. Note that with KDL we can specify the interface of our stack component, its behavioral description, and its semantic context, independent of object model, component architecture, implementation language, or system architecture.

[2]Object-Z does include the standard temporal operators *always*, *atnext*, *eventually*, and *previously*.

```
-- An archetypical LIFO stack.
Stack cdlIsKindOf(Finite_Data_Structure, Container)
-----
size cdlIsKindOf(Integer) >= 0

-- push a new item onto the stack
::push(item : cdlIsKindOf(Serializable)): cdlIsKindOf(Void)
  pre:  -- none
  post: self.top() = item
        size = size@pre + 1

-- read the top element of the stack
::top(): cdlIsKindOf(Serializable)
  pre:  size >= 1
  post: -- none

-- pop the top item off the stack
::pop(): cdlIsKindOf(Void)
  pre:  size >= 1
  post: size = size@pre - 1

-- is the stack empty?
::empty(): cdlIsKindOf(Boolean)
  pre:  -- none
  post: result = (size = 0)

-- clear the contents of the stack
::clear(): cdlIsKindOf(Void)
  pre:  -- none
  post: self.empty() = true
```

Figure 5: A KDL specification of a stack.

4 KDL for Component Specification and Testing

Each construct in KDL is verifiable, either statically (during compilation or composition) or dynamically (during composition or at run-time), and/or testable at run-time. We are developing a framework that directly maps the KDL meta-model to a set of Java components and a complementary KDL parser.

We wish to use this architecture in concert with our Parsing Java Pre-Processor [19] and advanced debugging framework [14]. The synthesis of these three component frameworks, especially when used in the context of a Jiki-enabled design and development environment[3], represents the theoretic and technological edge of component specification and testing.

5 Conclusion

In designing the Kind Description Language, we have attempted to integrate the best-practices of today's specification languages. With KDL, one can specify the syntactic interface of a component, its externally observable behavior, complex temporal properties evident in concurrent and distributed components, and the subtle relationships between components and their types and specifications.

For further information. KDL is ongoing work and represents a piece of this author's Ph.D. thesis. The reader interested in keeping abreast of this work should consider joining the Distributed Coalition[4] and contributing to the development of Jiki, an open source component-based open web architecture. Additionally, one should bookmark this author's project web pages[5]. Please send suggestions and comments to `kiniry@acm.org`.

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] 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.

[4] Edmund Burke and Eric Foxley. *Logic and its Applications.* Prentice-Hall, Inc., 1996.

[3] http://www.jiki.org/
[4] http://www.distributedcoalition.org/
[5] http://www.josephkiniry.com/projects/

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

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

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

[8] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, December 1994.

[9] E.H. Dürr and J. van Katwijk. VDM++ — a formal specification language for object-oriented designs. In Bertrand Meyer, Georg Heeg, and Boris Magnusson, editors, *Proceedings of Technology of Object-oriented Languages and Systems (TOOLS Europe)*, pages 63–78. Prentice-Hall, Inc., 1992.

[10] 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.

[11] John Guttag, James J. Horning, et al., editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer–Verlag, 1993.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[13] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., 1986.

[14] Joseph R. Kiniry. IDebug: An advanced debugging framework for Java. California Institute of Technology Technical Report CS-TR-98-16, California Institute of Technology, November 1998.

[15] Joseph R. Kiniry. A new construct for systems modeling and theory: The Kind. California Institute of Technology Technical Report CS-TR-98-14, California Institute of Technology, October 1998.

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

[17] Joseph R. Kiniry. The specification of dynamic distributed component systems. Technical Report CS-TR-98-08, California Institute of Technology, May 1998.

[18] Joseph R. Kiniry. *A Theory and Architecture for Open Collaborative Reuse*. PhD thesis, California Institute of Technology, 1999.

[19] Joseph R. Kiniry and Elaine Cheong. JPP: A Java pre-processor. California Institute of Technology Technical Report CS-TR-98-15, California Institute of Technology, November 1998.

[20] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06a, Iowa State University, Department of Computer Science, July 1998.

[21] Zohar Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer–Verlag, 1992.

[22] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 2nd edition, 1988.

[23] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

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

[25] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

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

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

[28] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley Publishing Company, 1998.

[29] Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel. OCLC/NCSA metadata workshop report. `http://www.oclc.org:5046/conferences/metadata/dublin_core_report.html`, March 1995.

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