Computing in Civil Engineering: Proceedings of the First Congress
pp. 571-578, American Society of Civil Engineers. 1994

Formal Specification of AEC Product Models

Harpreet S. Chadha,[1] S.M., ASCE, John W. Baugh Jr.,[2]
A.M., ASCE, and Jeannette M. Wing[3]

Abstract: This paper illustrates the use of equational specifications in developing product data models. This approach enables a precise and abstract description of products, where both syntactic and semantic checks are used for validation. Because they are formal objects, these specifications can be validated with respect to formal requirements and combined using ordinary mathematics. In addition, the availability of mature tools from the software engineering community further supports this approach to specifying and validating product models.

1 Introduction

Product models are intended to represent information about a product in a precise and consistent form. The product may be a civil engineering structure, a personal computer, a plate element for finite element analysis, etc., and is usually modeled by integrating various submodels. The information generated during the design, manufacture, use, maintenance, and disposal of a product may be required by several design teams and computer systems in different organizations.

There is a need for languages and tools that allow the representation of product information in a computer-readable form that facilitates exchange and integration of the information models developed. Several groups are attempting to enhance integration by using languages such as Express, IDEF, and NIAM to represent product information. STEP (ISO 1992) and PDES, for example,

[1]Graduate Research Assistant, Department of Civil Engineering, North Carolina State University, Raleigh, NC 27695-7908. Internet E-mail: hschadha@eos.ncsu.edu

[2]Assistant Professor, Department of Civil Engineering, North Carolina State University, Raleigh, NC 27695-7908. Internet E-mail: jwb@eos.ncsu.edu

[3]Associate Professor, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. Internet E-mail: wing@cs.cmu.edu

use Express for developing several generic and application-specific resource constructs, which are then combined in application protocols to describe the scope and information requirements for specific application contexts. While these languages allow product information to be stated in a precise form, and tools are available to ensure proper syntax, semantic checks of models developed in these languages have received little attention. In Express, for example, integrating various models in a manner that ensures compatibility and removes repetition, and thus redundancy, is a manual process (Watson and Boyle 1993). Validating a large product model without any support for semantic analysis is a very complex task.

This paper suggests the use of equational specifications, which enhance the validation process by supporting consistency proofs of individual product models, as well as supporting compatibility of the product models that are integrated. Below, we give a brief overview of equational specifications and related software tools, including Larch (Guttag et al. 1993), a family of specification languages based on equational logic. As an illustration, a small product model from the architecture, engineering and construction (AEC) domain is specified. We then discuss related work, followed by conclusions and directions for further research.

2 Equational Specifications

An equational specification is formalized by equational theories, which are defined in a restricted form of multisorted predicate logic with equality (Ehrig and Mahr 1985). By "formalized," we mean that such specifications have precise and unambiguous semantics (Wing 1990). There are many formal specification techniques (Woodcock and Loomes 1989), each being more or less appropriate for specific applications or parts of a program. For example, finite state automata have been widely used for specifying (and verifying) network protocols (Stallings 1993), as have process algebras for specifying concurrent and real-time computations (Elseaidy et al. 1994). Formal notations enable one to state clearly and unambiguously the behavior of computation and to develop shorter, simpler proofs using deductive proof techniques. This work focuses on equational specifications, which are convenient for describing abstract data types. For brevity, only a concise and informal overview of the area is presented, but that should be enough to support the subsequent examples and discussion.

An equational specification defines a mathematical object by using equations to relate the operators defined for that object (operators simply map from a cross product of values to a single value). These equations are connected implicitly by logical conjunction, and each of the equations must be true over the domain of their variables. For example, a logical negation operator is implicitly defined by the equation $not(not(A)) = A$, where A is a logical variable. Because equations may also be interpreted as left-to-right rewrite rules (under mild constraints), we have an operational mechanism for executing, and hence testing, our specifications.

We separate an equational theory from the *interface specification*, which specifies actual software components in terms of their preconditions and postconditions, by using a *two-tiered* approach to specification (Guttag et al. 1993). Thus, equational theories remain uncluttered from error values, implementation language features, etc., since they are used only to define the assertion language of the interface specification. The separation of concerns provided by the two-tiered approach allows equational theories to be reused far more easily than other kinds of computational objects.

The Larch system is a family of formal specification languages and tools that supports the two-tiered approach to specifying software and hardware modules. One tier of specification, written in the Larch Shared Language (LSL), describes abstractions that are independent of any implementation language. Language-specific details such as exception handling are written in a Larch interface language. Below, we use Larch to construct and validate equational specifications for a subset of the General AEC Reference Model (GARM).

3 Specification of GARM

GARM (Gielingh 1988) has been proposed as a high-level abstract data model to link various modeling concepts within and outside the AEC industry. The model is based on a generic entity called a product data unit (PDU), which can be a system, a subsystem, a part, etc. GARM supports the design strategy of decomposing complex problems into smaller problems by means of functional units (FUs) and technical specifications (TSs). An FU is a *collector* of the requirements of a PDU, which may be a design problem or a product to be obtained. The solution of the problem or the product is represented by a TS. There may be any number (including zero) TSs for each FU, and likewise, each TS can be modeled as a structured set of one or more FUs. Figure 1 presents an example of various FUs and TSs connected in a tree-like fashion. A very simplified model of a roof is shown, where the desired functions can be met with a sloping or a flat roof. The sloping roof in turn has three functional components in this simplified model: the structural strength is provided by a system of trusses, the exterior consists of steel cladding, and the support system consists of steel or wood columns. Although hierarchical, GARM does support user-defined relationships among nodes in the hierarchy.

To specify GARM in Larch, we first specify the notion of a tree in LSL, and then use that to model the several GARM concepts discussed above.

The tree trait A trait is the basic unit of specification in LSL. Often, a trait, as in the specification of the *Tree* abstraction of Figure 2, defines an abstract type by introducing some operators on the type and specifying their properties. The tree trait **includes** properties of sets—the axioms for set theory are separately defined in a *Set* trait. The **generated by** clause asserts that the operators *node*

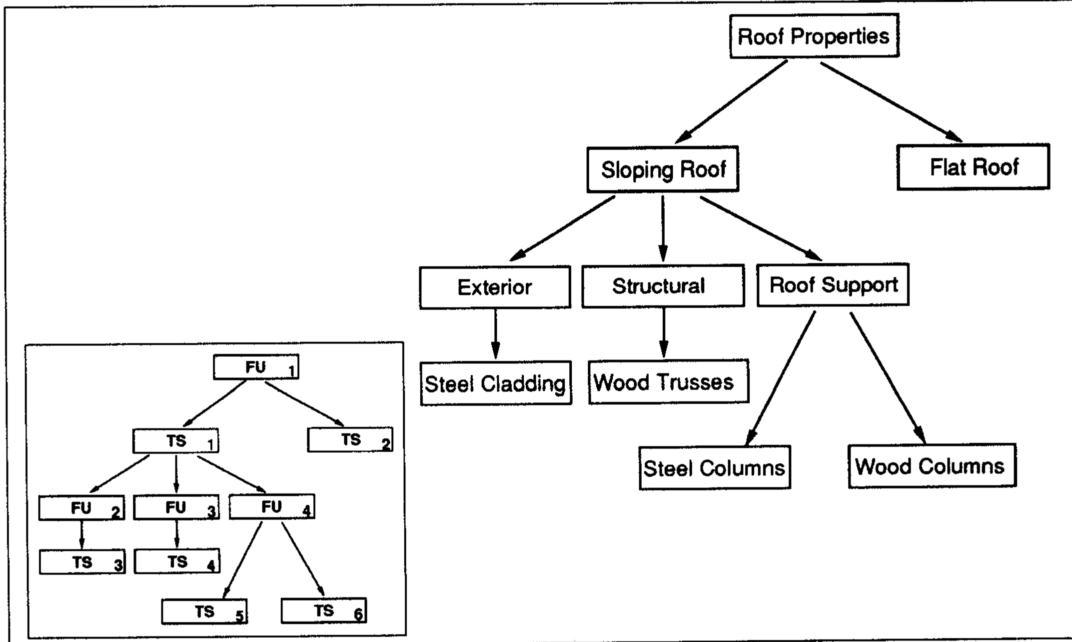

Figure 1: Example of a product represented by a hierarchy of FUs and TSs.

and *addchild* are sufficient to construct a tree. Similarly, the **partitioned by** clause asserts that all distinct instances of the *Tree* sort can be distinguished using the operators *content* and *children*. We also introduce an operator for the nearest ancestor common to two nodes in a tree, viz., least common ancestor (LCA), which is subsequently used in specifying GARM. The **implies** clause allows us to make claims that we expect to hold. For example, we expect that:

> If the LCA of two pairs of nodes in a tree are the same, and the pairs share a common node, then the LCA of the pair formed by the remaining nodes (that may not be shared) must also be in the tree.

Such claims act as semantic checks for our specifications, providing a means to ensure that the specifications written have the desired properties. To prove these properties, the LSL specifications are converted into an input suitable for the Larch Prover (LP). LP is a proof assistant for a subset of multisorted first-order logic, which is also the logic on which the Larch languages are based. The availability of tools to convert LSL specifications into input for LP, and tools to ensure correct syntax, further help in the development of clear and precise product models.

The product trait The FUs and TSs in GARM are specified in a *Product* trait as shown in Figure 3. Each node in the Product (tree) is a PDU, which is either an FU or a TS, and hence the use of *union*. A *connection* specifies

```
Tree(E, T, SetofT) : trait
    includes Set(T, SetofT)
    introduces
        node : E → T
        addChild : T, T → T
        content : T → E
        children : T → SetofT
        inTree : E, T → Bool                          % True if E is a node in T
        LCA : T, E, E → T                              % Least Common Ancestor
        childContaining : T, E → T                     % Child of T containing node E
    asserts
        T generated by node, addChild
        T partitioned by content, children
        ∀e, e1, e2 : E, t1, t2 : T
            content(node(e)) == e
            content(addChild(t1, t2)) == content(t1)
            children(node(e)) == {}
            children(addChild(t1, t2)) == insert(t2, children(t1))
            inTree(e, node(e1)) == (e = e1)
            inTree(e, addChild(t1, t2)) ==
                content(t2) = e ∨ inTree(e, t1) ∨ inTree(e, t2)
            LCA(addChild(t1, t2), e1, e2) ==
                if inTree(e1, t2) ∧ inTree(e2, t2)
                then LCA(t2, e1, e2)
                else if inTree(e1, t1) ∧ inTree(e2, t1)
                then LCA(t1, e1, e2)
                else addChild(t1, t2)
            childContaining(addChild(t1, t2), e) ==
                if inTree(e, t2) then t2
                else childContaining(t1, e)
    implies
        ∀e1, e2, e3 : E, t : T
            LCA(t, e1, e2) = LCA(t, e1, e3) ⇒
                inTree(content(LCA(t, e2, e3)), LCA(t, e2, e3))
```

Figure 2: A Tree Trait

the relations existing between pairs of functional units in a given product. An abstraction invariant defines the correct form of a product: the parent of a TS, if any, must be an FU, and vice versa; the related FUs must be *in* the tree; and the LCA of any two FUs must always be a TS. As before, the **implies** clause allows us to check whether the specifications incorporate the desired meaning. We check the following:

- A product model having an FU as the parent of another FU does not conform to the GARM model.
- The (sub)tree in Figure 1 with root TS_1 has the correct form, i.e., it satisfies the abstraction invariant.

While this example specification of a product model is not complete, it illustrates the usefulness of the modular approach used in writing and checking Larch specifications. The properties of *Sets*, which are independently validated, are used in developing the *Tree* abstraction, which is then checked for correct syntax

```
Product : trait
    includes Tree(PDU, T, SetofT), Relation(FunctionalUnit, R)
    PDU union of fu : FunctionalUnit, ts : TechnicalSolution
    P tuple of first : T, second : R
    introduces
        connector : T, FunctionalUnit, FunctionalUnit → FunctionalUnit
        connection : T, R → R
        alternating : T → Bool
        absInv : P → Bool
    asserts
        ∀p : PDU, f1, f2 : FunctionalUnit, t, t1, t2 : T, r1, r2 : R
            connector(t, f1, f2) ==
                content(childContaining(LCA(t, fu(f1), fu(f2)), fu(f1))).fu
            connection(t, [f1, f2]) == [connector(t, f1, f2), connector(t, f2, f1)]
            connection(t, r1 ∪ r2) == connection(t, r1) ∪ connection(t, r2)
            alternating(node(p))
            alternating(addChild(t1, t2)) ==
                tag(content(t1)) ≠ tag(content(t2))∧
                alternating(t1) ∧ alternating(t2)
            absInv([t, [f1, f2]]) == alternating(t) ∧ inTree(fu(f1), t)∧
                inTree(fu(f2), t) ∧ tag(content(LCA(t, fu(f1), fu(f2)))) = ts
            absInv([t, r1 ∪ r2]) == absInv([t, r1]) ∧ absInv([t, r2])
    implies
        ∀f1, f2, f3, f4 : FunctionalUnit, t1, t2, t3, t4, t5, t6 : TechnicalSolution
            not(fu(f1) = fu(f2)) ⇒
                not(absInv([addChild(node(fu(f1)), node(fu(f2))), [f1, f2]]))
            absInv([addChild(addChild(addChild(node(ts(t1)),
                addChild(node(fu(f2)), node(ts(t3)))),
                addChild(addChild(node(fu(f4)), node(ts(t5))),
                node(ts(t6)))), addChild(node(fu(f3)), node(ts(t4)))),
                ([f2, f3] ∪ [f3, f4])])
```

Figure 3: A Product Trait

and semantics. Similarly, the properties of *Tree* and *Relation* are validated and used in the *Product* trait. This methodology, of writing and checking individual models for tree and relation separately, and then checking for properties of the integrated product model, is very helpful in developing complex integrated products: the specifications of individual submodels are included in specifications of the integrated product model, which is then checked for syntax and semantics. Also, the high-level equational specifications present a more concise description of a product model compared to that given in Express (Gielingh 1988).

4 Related Work

Related work in product modeling includes the STEP and PISA (Gielingh 1993) projects that intend to address the problems of developing an infrastructure for product data technology. Checks for semantics in the models developed in these projects are, however, discussed only informally. The approach suggested in this study supports the expression of product models in a clear and precise forms that can be validated with mature tools from the software engineering community.

The use of formal specification techniques has also been suggested for graphics applications, where sophisticated mathematical models are built to describe complex objects (Dufourd 1991), and for producing reliable and reusable engineering software (Baugh 1992). These studies use equational axiomatization to provide a mathematical framework for expressing and proving properties (integrity constraints, etc.) of the described objects, as well as consistency and completeness of the specifications. A higher degree of formalism is also stressed in standardization efforts similar to STEP for product modeling—these include the CAD Framework Initiative for integrating various CAD packages (Mallis 1993), and definition of an industry standard for enterprise modeling (Kotsiopoulos 1993). Work drawing on the same divide and conquer strategy as GARM includes the use of module interconnection languages for combining software units. A related study (Rice and Seidman 1994) presents a formal model for module interconnection languages in the Z specification language (Spivey 1988).

5 Conclusions

The methodology suggested is useful for presenting product models in a clear and precise form, and for validating the properties of integrated models using deductive proof techniques. The syntax and semantic checks, as mentioned, help to ensure that the product models have the desired meaning. Further studies are needed for testing larger product models and for developing proof strategies to simplify the model validation process. While continuing to use equational specifications for more complex product models, we are also attempting to use these techniques in formalizing concepts and theories in other areas in engineering, e.g., finite element analysis.

Acknowledgments

This work is supported in part by the National Science Foundation under grant number MSS-9201697 entitled "Reusable Engineering Software Components," and by the Department of Civil Engineering at North Carolina State University.

References

Baugh, J. W., Jr. (1992). Is engineering software amenable to formal specification? In Martin, U. and Wing, J. M., editors, *First International Workshop on Larch*, 1–17. Springer-Verlag.

Dufourd, J.-F. (1991). Formal specification of topological subdivisions using hypermaps. *Computer-Aided Engineering*, 23(2).

Ehrig, H. and Mahr, B. (1985). *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin.

Elseaidy, W. M., Baugh, J. W., Jr., and Cleaveland, R. (1994). Verification of an active control system using temporal process algebra. Submitted to *Engineering with Computers.*

Gielingh, W. (1988). General AEC reference model. ISO TC184/SC4/WG1 document 3.2.2.1 [Also TNO report BI-88-150, October 1988].

Gielingh, W. (1993). Towards an infrastructure for product data technology. In Kooij, C., MacConaill, P., and Bastos, J., editors, *Realising CIM's Industrial Potential*, 70–81. IOS Press.

Guttag, J., Horning, J., Garland, S., Jones, K., Modet, A., and Wing, J. (1993). *Larch: Languages and Tools for Formal Specification.* Springer-Verlag.

ISO TC184/SC4/WG6 N29 (1992). Reference material for ISO 10303-31 — Conformance testing methodology and framework: general concepts, Version 10.

Kotsiopoulos, I. (1993). Theoretical aspects of CIM-OSA modeling. *Realising CIM' Industrial Potential*, 212–222.

Mallis, D. (1993). If it can't be specified, it can't be a spec. *Technical Brief: A Monthly Publication of CAD Framework Initiative, Inc.*

Rice, M. and Seidman, S. (1994). A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1), 88–101.

Spivey, J. (1988). *Understanding Z, A Specification Language and its Formal Semantics.* Cambridge University Press.

Stallings, W. (1993). *Networking Standards: A Guide to OSI, ISDN, LAN and MAN standards.* Addison-Wesley.

Watson, A. and Boyle, A. (1993). Product models and application protocols. In Topping, B. and Khan, A., editors, *Information Technology for Civil and Structural Engineers*, 121–129. Civil-Comp Press.

Wing, J. (1990). A specifier's introduction to formal methods. *Computer*, 23(9), 8–24.

Woodcock, J. and Loomes, M. (1989). *Software Engineering Mathematics.* SEI Series in Software Engineering. Addison-Wesley, Reading, MA.