# Toward Ontology-Based Component Composition

Kenneth Baclawski and Artan Simeqi
College of Computer Science
Northeastern University, Boston, Massachusetts
{kenb,asimeqi}@ccs.neu.edu

**Abstract**

Large-scale component-based software systems are becoming increasingly important. Building such a system requires more than just specifications for the overall system and its components. It is necessary to have a formal specification of how the components are composed to form the overall system. Furthermore, it is also necessary to have an explicit specification of the environment within which the system will be used. Finally, some mechanism must be available to make verification a tractable problem. To address all of these concerns, we suggest that component composition can be based on ontologies. An ontology is a shared understanding that allows individuals in a community to communicate. The increasing popularity of ontology-based computing has resulted in a rapid increase in the number of tools available for creating and for using ontologies. We suggest that leveraging these tools can improve the coverage and effectiveness of component composition compared with existing techniques. We also discuss a specific application domain that can be used as a testbed for the use of ontology-based techniques to achieve dynamic reconfiguration performed at run-time without direct human interaction.

Keywords: formal methods, components, component framework, component composition.

## 1 Introduction

To cope with the task of large-scale software systems, developers are increasingly using components and component frameworks. Unfortunately, it is not easy to verify the correctness of a component composition, even when the individual components have been formally specified. The problem of "feature interaction" has been recognized for some time, and entire workshops (such as [7]) have been devoted to it. In a provocative paper, Lamport gives a rather dismal assessment of the prospects for such verifications:

> Composition of open-system specifications is an attractive problem, having obvious application to reusable software and other trendy concerns. But in 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It is naive to expect them to go to the extra effort of proving properties of open-system component specifications because they might re-use those components in other systems. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years. Formal specifications of systems, with no accompanying verification, may become common sooner. [26]

One hopes that it will not be another 11 years before reasoning about compositions becomes a practical concern. One area where rigorous specifications are generating a great deal of activity is *ontologies*: declarative specifications of entities and their relationships with each other within a specialized application domain. An ontology is a shared understanding that enables a community of people to communicate. Ontologies are increasingly also being used to enable communication between autonomous computing entities (agents). To achieve effective communication between agents, ontologies must also include facts, theorems and inference rules. For more about ontologies see the Ontology FAQ [27].

In an ideal world, the components would be formally specified, the component specifications would be composed, and a rigorous proof given to verify that the composition satisfies the requirements. The problem with this idealization is that finding proofs is an intractable problem in general because theorem proving is an undecidable problem. As a result, it is necessary to find some way to make the problem tractable.

The primary technique in use today to achieve tractability is the creation of specialized languages for component composition, such as CSP [22]. Lamport [26] calls these *pseudo-programming languages*. Like all specialized languages the disadvantage of this approach is that such a language has a limited domain of applicability. Component compositions that do not fit within the constructs and assumptions of the language cannot be analyzed at all. This has resulted in a "cottage industry" of extensions to existing languages, each offering some additional constructs.

A more serious problem, stated by Lamport [26], is that these languages do not make it any easier to verify correctness of the composition. In fact, Lamport argues that pseudo-programming languages make it *harder* to verify correctness. He ends his paper with the following:

> Any proof in mathematics is compositional – a hierarchical decomposition of the desired result into simpler subgoals. A sensible method of writing proofs will make that hierarchical decomposition explicit, permitting a tradeoff between the length of the proof and its rigor. Mathematics provides more general and more powerful ways of decomposing a proof than just writing a specification as the parallel composition of separate components. That particular form of decomposition is popular only because it can be expressed in terms of the pseudo-programming languages favored by computer scientists. Mathematics has been developed over two millennia as the best approach to rigorous human reasoning. A couple of decades of pseudo-programming language design poses no threat to its pre-eminence. The best way to reason mathematically is to use mathematics, not a pseudo-programming language. [26]

Indeed, one could make a good case that the mathematical reasoning exhibited in the work of Euclid [14] in 300 B.C.E., while not completely rigorous, is more rigorous than nearly all the reasoning currently used in software development today.

In this paper, we attempt to go "back to basics" by making mathematical reasoning the primary reasoning technique for verification of component composition. We make use of the category theory technique of the *colimit* to "make that hierarchical decomposition explicit." To achieve tractability, we use ontologies that have sufficient depth and coverage.

Another point made by Lamport is that components often have implicit assumptions about the environment within which they function [26]. We advocate formalizing the specification of the environment as well as the system requirements. See Bunge [10] for a precise definition of the notion of environment. In Section 4 we discuss sensor systems as an application domain that would provide a thorough test of whether ontology-based techniques can be used in high-performance scenarios.

This particular application domain requires dynamic reconfiguration at run-time without direct human interaction. In dynamic reconfiguration, both the system requirements (over which we have some control) and the environment specification (over which one generally does not have control) can evolve in time. Such changes can trigger a request for system reconfiguration. Such a reconfiguration not only requires finding a new component composition but it also requires verifying that it satisfies the (possibly new) system requirements for the (also possibly new) environment specification at run-time under strict time constraints. Such a feat is possible only when one is dealing with a specialized domain.

## 2 Related Work

The DARPA Agent Markup Language (DAML) [13] program is developing an ontology language that is Web-enabled. The DAML program is also developing a large variety of DAML tools and ontology-based applications. DAML has many similarities to the General Relationship Model (GRM) [24] as well as the Unified Modeling Language (UML) [9]. Similarities and differences between DAML and UML are discussed in [6] along with suggestions about extending UML so that it is more compatible with DAML and other knowledge representation languages. DAML does not yet have a rules and inferencing language, but such a language is in development (see `www.daml.org`).

Knowledge-based approaches to software engineering are starting to appear within the broader area of automated software engineering. The annual Automated Software Engineering conference includes ontology-based approaches as one of its topic areas. There has been a conference [36] on knowledge-based software engineering. However, there is not a lot of activity yet. Moreover most of the work in this area uses the traditional monolithic development techniques. Object-oriented and component-based techniques are not yet common.

Component composition in general has attracted a great deal of interest, including formal approaches. Component composition is also called *architectural configuration modeling* and support for it is generally found in a class of languages called architecture definition languages (ADL). There is a good survey of most of the ADLs that support component composition in [33]. Examples include ACME [16], Aesop [15], C2 [32], Darwin [30], MetaH [8], Rapide [28], SADL [34], UniCon [37], Wright [3]. These languages vary considerably in their degree of formalization.

There are also languages that focus exclusively on component composition, such as Piccola [29]. In addition, there are many articles that introduce an *ad hoc* language within the context of the article as a means of formalizing some aspects of component composition relevant to the article. For example, [35] discusses verification techniques for object-oriented components and [12] is concerned with specification matching.

One of the few examples of the development of an ontology for a specific application domain is the treatment of astronomical software [40] by groups at SRI and NASA. They used the Snark theorem prover [41]. Another example is the work of Goguen and Tracz [21] in which the domain is avionics. They use the OBJ formal methods system [20]. Wing and Ockerbloom's treatment of type conversion [43] is another example. They use the Larch formal specification language [19]. The theorem prover for Larch is called LP [18]. Examples in business domains include the work of Kent and his colleagues [25], and Iida et al. [23].

## 3 Suggested Approach for Component Composition

In this section we give a brief sketch of how one might use ontologies as the basis for specifying and verifying component compositions. We first sketch a metamodel for component composition, i.e., the meta-ontology. We then discuss the kinds of tool that would be important for ontology-based component composition.

### 3.1 Metamodel

The primary entities are theories and proof traces as shown in Figure 1.

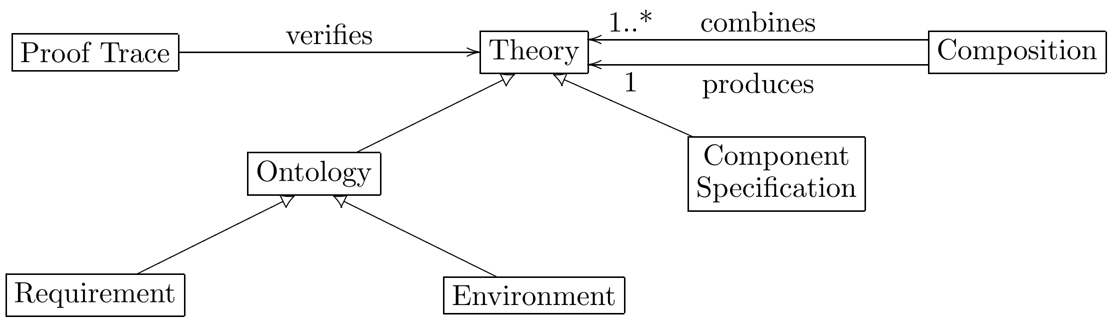

Figure 1: Component Composition Metamodel

**Theories**. A mathematical theory consists of entities, relationships, constraints, rules and theorems. A theory need not have any direct connection with any software. One important property of theories is that a theory can be reused. As a result, it is important for theories to be organized in a library from which they can be retrieved, specialized and composed with other theories. For example, the theory of real numbers can be reused in a theory of time and duration as well as in a theory of locations in space.

There are many languages for specifying mathematical theories. For our initial work we plan to use Metaslang$^{TM}$, an enhanced version of the Slang$^{TM}$ formal methods language of the Specware formal methods system [31]. Metaslang has theory management features that are especially useful, including the ability to specialize and to compose theories using the *colimit* operation from category theory. It also supports refinement of theories, including refinement to programming language code.

The simplest example of a colimit is a disjoint union of several theories, i.e., one simply forms the disjoint union of all sorts, operations, axioms and theorems. A colimit is more interesting when some of the sorts and operations are shared by the theories being composed. For example, one can import a theory into another theory. In this case, all of the sorts and operations in the imported theory are shared with the other theory. The import mechanism is commonly supported by most formal methods systems. In general, a colimit can construct theories in which sorts and operations are selectively shared.

**Ontologies**. An ontology is a theory that specifies entities and relationships for a particular domain. Ontologies are often constructed by composing smaller theories. There is now a large variety of languages for specifying ontologies. We plan to eventually use the DAML+OIL language being developed by the DAML program [13], primarily because of the many tools being developed for it and because of the many ontologies being written in DAML+OIL The one disadvantage of DAML+OIL is that it is a "work in progress." In particular, DAML+OIL does not yet have a standard logical inference language. It also does not have the theory management features supported by Metaslang. Fortunately, DAML+OIL has a formal axiomatization, and it can, in principle, be translated in an automated manner to Metaslang.

**Requirements and Environment Specifications**. Requirements that can be imposed and environments within which a system operates are also theories. We view them as special kinds of ontology. Environments are not commonly specified explicitly. An example of where they are is in Lynch's work on I/O automata [17].

**Component Specifications**. Each reusable component has a specification that is expressed in terms of the theories in the Theory Library. This expression will generally require specializing and composing theories in the Theory Library. We do require that components be coded in some programming language, but we do not require that the component be expressed in any special form (such as a class or Java Bean). In particular, a component could be a code fragment or template.

**Proof Traces**. A *proof trace* or *justification* is a formal mathematical proof represented as a data structure. Our main use of justifications is to record the proofs of the properties of component compositions. In particular, a proof trace can verify the theory produced by a component composition. It can also verify that a component composition satisfies a requirement specification in a particular environment. The latter kind of verification is not shown in Figure 1.

**Compositions**. A composition of theory components combines one or more theories to produce another theory. The actual combination need not be a simple conjunction of the theories. In particular, it is possible that the theories being composed share sorts and operations. A composition can also add requirements that must be proven. In mathematics, such a requirement is called a *conjecture*. The proof traces for the conjectures added to a composition validate the composition and furnish its justification.

It is useful to maintain a library of component compositions because one technique for constructing a component composition satisfying a requirement is to simply look it up in the library. If such a composition is not in the library, another technique is to modify a composition that is in the library.

## 3.2 Tools

The entities above are constructed, managed and used by various tools:

**Specification Composer**. This is a tool that manages and composes specifications (theories). We are currently using the Specware formal method system produced the Kestrel Institute [42]. The features of Specware were discussed above.

Other examples of composers include a composer tool in Lynch's IOA [17]. Another formal methods system that supports composition of theories is OBJ [20] which had an influence on the Specware system [38]. The idea of using the category theory notion of a colimit for composing theories was first proposed by Burstall and Goguen [11, 1, 2].

**Theorem Prover**. Many theorem provers are available, and many more are being produced by the DAML program [13]. We currently use the Snark theorem prover [41]. An important requirement of a theorem prover is that it produce a proof trace that can be queried by automated tools.

Although theorem proving is, in general, an undecidable problem, in special cases it is not only decidable but also tractable. This can be achieved by using ontological techniques such as those being developed for DAML [13]. The DAML language is a rich mathematical language, yet checking consistency of a DAML ontology is decidable. Furthermore, development tools and query engines have been developed that are fast and scalable, not only for DAML but for other knowledge representation languages. See [4, 5].

**Meta Reasoner**. This tool manages and examines the proof traces of the Component Composition Library. It is responsible for constructing and verifying component compositions. There are various strategies that it can employ. For example, when the requirements and/or the environment are changing in an incremental fashion, then it can construct new component compositions by incrementally modifying existing component compositions.

The Meta Reasoner is similar to a Planner in the AI literature. The purpose of a planner is to modify the behavior of a system in response to changes in the environment. There is a large literature on planning, and many systems have been developed for handling a great variety of scenarios. See [39] for a list of links to Web sites dealing with planning.

## 4 Dynamic Composition and Reconfiguration for Sensor Systems

The application domain to which we plan to introduce dynamic component composition is in the area of sensor systems. The sensors we consider are receivers located on a moving platform such as an aircraft. When aircraft are operating in hostile territory, attempts will be made to detect and to track their movements. This is most often done by using ground or missile based radars. Such radars emit electromagnetic radiation that illuminates the aircraft. Radiation reflected from the surfaces of the aircraft is used by the radar for detection, identification and tracking purposes. We refer to these radars as *emitters*.

Because of the tactical importance of these radars, it is important for aircraft to detect that they are being illuminated. For this reason, aircraft are equipped with receivers that attempt to sense when the aircraft is being illuminated by an emitter. The aircraft receivers are called *sensors*.

It is normally assumed that the kinds of emitter that might be encountered during a mission are known in advance. In other words, it is assumed that an *environment* is formally specified prior to the mission. In particular, the system has data on the frequencies used by the emitters and the times between successive illuminations when the emitters are operating. It is not normally assumed that one will know when a given kind of emitter will begin operating or how long it will operate.

The software system that operates the sensor will necessarily assume that the environment is known. If this assumption is invalidated during a mission, or if the requirements change during the mission, then the software system may no longer satisfy the requirements. Hence there is a need to reconfigure the system during the mission. There are several aspects of component composition in this scenario that make it especially hard:

1. Component compositions must be constructed, verified and installed while a mission is taking place. In particular, there is no possibility that the software could be taken out of service for the sake of the reconfiguration.

2. The pilot (if there is a pilot at all) is not in a position to be interacting with the system in more than a perfunctory way.

3. The reconfiguration must be accomplished in a very short period of time, typically measured in seconds, rather than minutes or hours

While there exists software that can be reconfigured at runtime as required by the first item above, there are currently no systems that can satisfy even two of the conditions above, let alone all three. Furthermore, an *ad hoc* or heuristic approach is not acceptable. Because lives can be lost due to software that does not function according to expectations, it is essential that the system provide rigorously proven guarantees of performance. Accordingly, it furnishes an extreme test of what is possible with formal methods.

## 5 Conclusion

We have discussed the possibility of basing component composition on ontologies. This suggestion is grounded in mathematics rather than in a pseudo-programming language. Rather than composing software components using software-inspired connections, it composes theories using mathematical composition techniques. It addresses the problem of tractability by relying on having an effective domain-specific ontology. We intend to use existing, established ontology-based tools, theorem provers and theory composers. Our primary contribution is to integrate these tools and to build a high-level meta reasoner that manages them. We propose to apply these ideas in the specific domain of sensor systems.

## Acknowledgements

This research was partially supported by the Defense Advanced Research Projects Agency under contract F330602-99-C-0167.

## References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. on Programming Languages and Systems*, 15,1:73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems*, 17,3:507–534, 1995.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[4] K. Baclawski. Distributed computer database system and method, December 2 1997. United States Patent No. 5,694,593. Assigned to Northeastern University, Boston.

[5] K. Baclawski. Distributed computer database system and method, February 20 2001. United States Patent No. 6,192,364. Assigned to Jarg Corporation, Waltham.

[6] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to support ontology engineering for the Semantic Web. In M. Gogolla and C. Kobryn, editors, *Fourth International Conference on the Unified Modeling Language*, volume 2185, pages 342–360. Springer-Verlag, Berlin, October 2001.

[7] D. Batory, J. Brichau, L. Carver, K. Czarnecki, E. Ernst, P. Steyaert, and S. Tyszberowicz, editors. *ECOOP 2001 Workshop on Feature Interaction in Composed Systems.* 15th European Conference on Object-Oriented Programming, Budapest, Hungary, June 18 - 22 2001.

[8] P. Binns and S. Vestal. Formal real-time architecture specification and analysis. In *10th IEEE Workshop on Real-Time Operating Systems and Software*, 1993.

[9] G. Booch, I. Jacobson, and J. Rumbaugh. *OMG Unified Modeling Language Specification*, March 2000. Available at `www.omg.org/technology/documents/formal/-unified_modeling_language.htm`.

[10] M. Bunge. *Dictionary of philosophy.* Prometheus Books, 1999.

[11] R. Burstall and J. Goguen. Introducing institutions. In E. Clarke and D. Kozen, editors, *Proceedings, Logics of Programming Workshop*, volume 164, pages 221–256. Springer-Verlag, Berlin, 1984.

[12] Y. Chen and Cheng B. H. C. A semantic foundation for specification matching. In *Foundations of Component-Based Systems*, pages 91–110, 2000.

[13] DAML. DARPA Agent Markup Language website, 2001. `www.daml.org`.

[14] Euclid. *Elements*, 300 B.C.E.

[15] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 170–185. ACM Press, 1994.

[16] D. Garlan, R. T. Monroe, and D. Wile. ACME: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–68, 2000.

[17] S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In *Foundations of Component-Based Systems*, pages 285–312, 2000.

[18] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch Prover. In *Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, 1989. Springer-Verlag Lecture Notes in Computer Science 355.

[19] Stephen J. Garland, John V. Guttag, and James J. Horning. An overview of Larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 329–348. Springer-Verlag Lecture Notes in Computer Science 693, 1993.

[20] J. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic specification in action.* Kluwer Academic, Dordrecht, Netherlands, 2000.

[21] J. A. Goguen and W. Tracz. An implementation-oriented semantics for module composition. In *Foundations of Component-Based Systems*, pages 231–264, 2000.

[22] C. Hoare. Communicating sequential processes. *Comm. Assoc. Comput. Mach.*, 21(8):666–677, August 1978.

[23] S. Iida, K. Futatsugi, and R. Diaconescu. Component-based algebraic specification. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Object-Oriented Behavioral Specifications*, pages 105–121. Kluwer Academic, Dordrecht, Netherlands, 1999.

[24] *Information Technology–Open Systems Interconnection–Management Information Services–Structure of Management Information–Part 7: General Relationship Model (ISO/IEC 10165-7.2)*, August 1993.

[25] S. Kent, K. Lano, J. Bicarregui, A. Hamie, and J. Howse. Component composition in business and system modeling. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proc. OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 91–98. Technische Universität München, September 1997.

[26] L. Lamport. Composition: A way to make proofs harder. In *Compositionality: The Significant Difference*, volume 1536, pages 402–423. Springer-Verlag, Berlin, 1998. Originally in Proc. COMPOS'97 Symposium.

[27] Ontology FAQ List. Ontology Consortium Frequently Asked Questions, 2001. `www.ontology.org/main/papers/faq.html`.

[28] D. C. Luckham, J. Vera, and S. Meldal. Key concepts in architecture definition languages. In *Foundations of Component-Based Systems*, pages 23–46, 2000.

[29] M. Lumpe, T. Achermann, and O. Nierstrasz. A formal language for composition. In *Foundations of Component-Based Systems*, pages 69–90, 2000.

[30] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, 1995.

[31] J. McDonald and J. Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3, Kestrel Institute, March 2001.

[32] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering.* ACM Press, 1996.

[33] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[34] M. Moriconi, X. Qian, and R. Riemenshneider. Correct architecture refinement. *IEEE Trans. Software Engineering*, 21(4):356–372, 1995.

[35] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, pages 137–160, 2000.

[36] H. Reubenstein and D. Setliff, editors. *Knowledge-Based Software Engineering*. Kluwer Academic Publishers, 1996. Proceedings of the Knowledge Based Software Engineering Conference held in Boston in 1995.

[37] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21,4:314–335, 1995.

[38] Y. V. Srinivas and R. Jullig. Specware$^{TM}$: Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, July 1995.

[39] R. St. Amant. AI planning resources including links to planning systems and testbeds, 1999. `www.csc.ncsu.edu/faculty/stamant/planning-resources.html`.

[40] M. Stickel, R. Waldinger, M. Lowry, Pressburger T., and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *12th International Conference on Automated Deduction*, volume 814. Springer-Verlag, Berlin, 1994.

[41] M. E. Stickel, R. J. Waldinger, and V. K. Chaudhri. A guide to SNARK, 2001.

[42] R. Waldinger et al. *Specware*$^{\mathrm{TM}}$ *Language Manual: Specware*$^{\mathrm{TM}}$ *2.0.3*, March 1998.

[43] J. M. Wing and J. Ockerbloom. Respectful type converters for mutable types. In *Foundations of Component-Based Systems*, pages 161–186, 2000.