Declarative Specification of Software Architectures

John Penix and Perry Alexander
KBSE Lab, ECECS Dept.
University of Cincinnati
Cincinnati, Ohio 45221-0030
{jpenix,alex}@ececs.uc.edu

Klaus Havelund
Recom Technologies, NASA Ames
Code IC, MS 269-2
Moffet Field, CA 94035 USA
havelund@ptolemy.arc.nasa.gov

Abstract

Scaling formal methods to large, complex systems requires methods of modeling systems at high levels of abstraction. In this paper, we describe such a method for specifying system requirements at the software architecture level. An architecture represents a way of breaking down a system into a set of interconnected components. We use architecture theories to specify the behavior of a system in terms of the behavior of its components via a collection of axioms. The axioms describe the effects and limits of component variation and the assumptions a component can make about the environment provided by the architecture. As a result of the method, the verification of the basic architecture can be separated from the verification of the individual component instantiations. We present an example of using architecture theories to model the task coordination architecture of a multi-threaded plan execution system.

1 Introduction

Large systems are specified and implemented as a collection of interconnected components. The goal is to decompose the system in such a way that the properties of the parts can be composed to create properties of the larger system. The *architecture* is the structure of the system, i.e., the assignment of functionality to components, and the interaction among components [15]. Viewing each component as an independent system with its own architecture results in an overall hierarchical system structure.

Formalisms have recently been introduced to make software architecture a more rigorous activity [1, 4, 11, 15]. Formal methods allow a designer to model aspects of a software system and apply mathematical analysis/verification techniques. In the case of software architecture, component interfaces and interconnections are defined and augmented with formal specifications and formal languages, such as process algebras, are used to formally specify component interactions.

These existing formal models of software architecture are concerned with formalizing specific architectural styles such as pipe-filter and client-server. While architectural styles abstract away many implementation details, each still represents a highly *reduced* subset of the space of possible system designs. The reduction of the design space is what makes a style usable by human designers. However, the fact that the space is reduced indicates that the choice of an architectural style is an important design decision that should not be made prior to initial requirements specification. The alternatives for decomposing the system requirements should drive the selection of a specific architectural style.

When decomposing system requirements, the goal is to capture the relationship between the behavior of the system and the behavior of its components. From the perspective of the architecture, we would like to know the effects and limits of component variation. Specifically, we want to know what component behavior is necessary to guarantee correct system-level behavior and how variation in component behavior affects the behavior of the system. For each component, we are interested in the assumptions that can be made about the environment provided by the architecture.

Effective modeling and manipulation of these relationships requires an architecture representation that abstracts out operational details and provides a declarative specification of an architecture. Declarative specifications state *what* something does without stating *how* it does it. The required functionality is separated from the non-required side effects of implementation decisions. In the case of software architecture, we must relate the behavior of the system to the behavior of the subcomponents independent of the style in which the architecture is implemented.

With this goal in mind, we have extended Smith and Lowry's methods for specifying the structure of algorithms using algorithm theories [16, 18] to specify the structure of architectures. An *architecture theory* constrains the behavior of a system in terms of the behavior of its subcomponents via a collection of axioms. The axiomatic constraints can be used to reason in both a top-down and a bottom-up manner. Given a system specification, an architecture theory, and a subset of the components in the architecture, we can determine the functionality required in the missing components. Conversely, given a collection of components and an architecture theory, we can determine the functionality of the system constructed by plugging the components into the architecture.

In this paper, we show how algebraic theories can be used to specify properties of software architectures. We begin by describing how components and interconnections can be specified axiomatically. We then describe architecture theories and their potential role in software development. Next, we give an example of using architecture theories by showing their application to modeling and verifying a plan execution system. We follow this with a discussion of related work and conclude with a summary and a statement of future work.

2 Specification Fundamentals

Specifications for both components and architectures are expressed as algebraic theories. Theories define *operations* over a collection of *sorts* and constrain the behavior of the operations via a set of *axioms*. A sort, like a type, is a set of values. Operations specify how to construct, modify and differentiate values of the sort. The axioms define equivalence sets of values in the sort.

Theory morphisms are the formal mechanism underlying two methods of composing smaller theories to form larger ones: *extension* and *parameterization* [10, 17]. A *theory morphism* maps the sorts and operators of one theory to sorts and operators of another theory such that the axioms of first theory are valid theorems in the second theory. Theory B is an *extension* of theory A if B contains all of the sorts, operators and axioms of A. An extension is represented by a theory morphism (from A to B) that maps each sort and operator to itself (the identity morphism) in the target theory.

A *parameterized theory* is a pair of theories: a *parameter theory* and a *target theory* that is an extension of the parameter theory [3]. A parameter theory is instantiated by a theory morphism that maps the parameter theory to the *actual parameter*. This activity is depicted by the parameter passing diagram in Figure 1. The resulting *instantiated theory* is constructed by computing the *pushout* of the diagram [2, 10]. This has the effect of replacing the parameter theory by the actual parameter according to the translation defined by h.

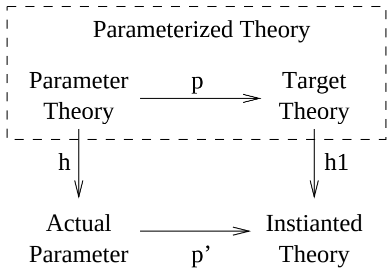

Figure 1. Parameterized Theory Instantiation

We use theories to specify components using two predicates: a *precondition* and a *postcondition* [8]. The precondition specifies the set of domain values that have a defined output, called the *legal inputs* to the problem. The postcondition specifies the relationship that must hold between a legal input and a *feasible output*. A theory based framework can also be used to specify abstract data types [5] providing a direct pathway to extend our ideas to a more complex component model.

A component is represented formally as an extension of the *problem theory* [18] shown in Figure 2(a). A specification for a specific problem is created by a specification morphism from the problem theory that provides definitions for the domain, range, precondition and postcondition. For example, a search problem is specified in Figure 2(b). A *component theory*[1] extends a problem theory by adding an axiom stating that a valid output exists for every legal input, as shown in Figure 2(c). The specification for a specific component is created by extending a component specification with definitions for the domain, range, precondition and postcondition.

3 Architecture Specification

An *architecture theory* specifies the behavior of a system in terms of the behavior of its subcomponents via a collection of axioms. Formally, an architecture theory is the target theory of a parameterized theory as shown in Figure 3. Each parameter specification of the

[1] This is a generalization of program theories [18]

(a)

```
theory ProblemTheory(D,R,I,O)
sorts D,R
operators
 I : D → Bool
 O : D, R → Bool
```

(b)

```
theory FindProblem(D,R,I,O)
includes
 List(List,Rec),
 ProblemTheory(D,R,I,O)
sorts
 D tuple of a:List, k:Key
 R tuple of z:Rec
axioms
 ∀ a:List, k:Key, z:Rec
 I(⟨a,k⟩) == true;
 O(⟨a,k⟩,z) == element(a,z) ∧ z.key = k;
```

(c)

```
theory ComponentTheory
includes
  ProblemTheory(D,R,I,O)
axioms
  ∀ x:D ∃ z:R  I(x) ⇒ O(x,z)
```

Figure 2. Problem and Component Theories

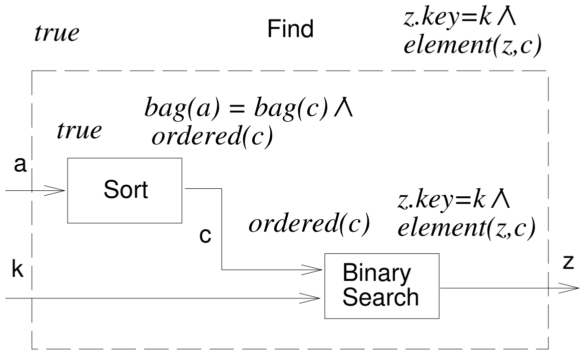

Figure 4. Example Find Architecture

```
theory FindArchitecture
includes
 FindProblem(D,R,SystemI,SystemO),
 SortComponent(List,List,SortI,SortO),
 BinSearchComponent(D,R,BinI,BinO)
axioms
 ∀ a,c:List, k:Key, z:Rec
 SystemI(⟨a,k⟩) ⇒ SortI(a);
 (SortI(a) ∧ SortO(a,c)) ⇒ BinI(⟨c,k⟩);
 (SystemI(⟨a,k⟩) ∧ SortO(a,c) ∧ BinO(⟨c,k⟩,z))
     ⇒ SystemO(⟨a,k⟩,z);
```

Figure 5. Example Find Architecture

architecture theory is a problem theory. The parameter theories are instantiated with actual system and component specifications by specification morphisms. Constructing these morphisms corresponds to specifying the system level requirements and selecting components from a library. The resulting instantiated theory is a specialized architecture theory where the definitions of the system and components are consistent with the axioms of the architecture theory. This indicates that the architecture can be used to correctly decompose the problem into the selected components.

The axioms in an architecture theory specify constraints on the component and system specifications. Constraints specify component behavior that is necessary to guarantee correct system-level behavior. They also define how variation in component behavior affects the behavior of the system. This is done by defining they system specification in terms of the component specification. Additionally, we can state assumptions that can be made by the component when it operates inside the architecture. These assumptions may be important for determining when a component can be properly plugged-in to an architecture.

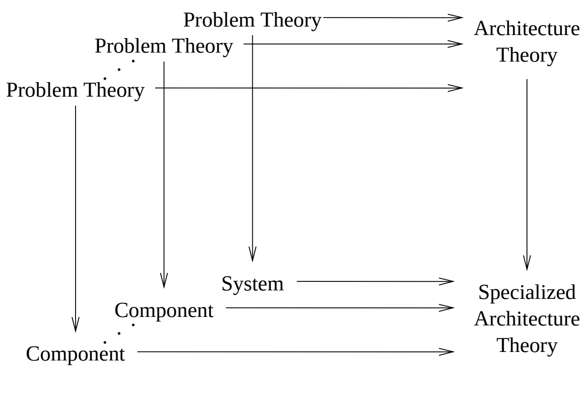

Figure 3. Architecture Theory Instantiation

For example, Figure 4 shows an architecture for the Find problem that was specified in Figure 2. The block diagram shows the component interconnections, bindings to the system level interface and the preconditions and postconditions of the subcomponents. Figure 5 shows a corresponding instantiated architecture theory. The inclusion of the problem and component theories indicate specification morphisms from those theories with the shown renamings. The first axiom states that the `Sort` component will operate over all of the legal system inputs. The second axiom specifies that the

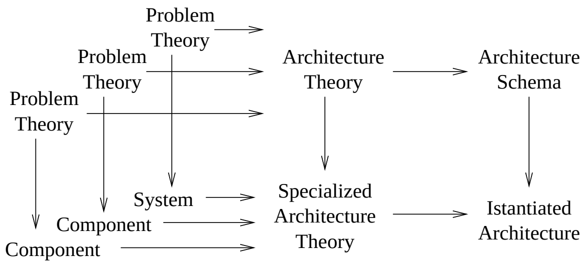

Figure 6. Overview of System Design Using Architecture Theories

combined behavior of the two components is always defined, i.e., there exists a legal output for `BinSearch`, for every legal input to `Sort`. The third axiom states that for a legal input, the behavior of the components results in the correct behavior of the system. Based on the component specifications in this example, all three axioms are valid. Therefore, the decomposition of the problem is correct.

Additional axioms can be added (by extending the theory) to describe a certain class or style of architectures. This results in a hierarchy of architecture theories that can be used to classify a problem theory and provide a control mechanism for matching an architecture theory to a problem specification [4, 17].

4 Architecture Implementation

An overview of the role of architecture theories in system design is show in Figure 6. As described in the previous section the specialized architecture theory is created by instantiating an architecture theory with a problem and component specifications. The problem decomposition described by an architecture theory is implemented via an architecture schema. An architecture theory may have several associated schemas written in a target programming language or an *architecture description language* [15]. The architecture schema is instantiated by substituting the selected components into the architecture schema.

The correctness of the implemented system requires that the constraints placed on the system by the architecture theory are guaranteed by the architecture schema. This can be verified based on a semantics of the target programming language or architecture description language [11, 15]. It is possible for the semantics of the target language to be formalized in a different logic than the specification. If this is the case, it is necessary to use *institutions* [6] to provide the formal link between the specification and implementation logics.

The verification of the architecture schema may involve a sizeable human effort. However, once complete, it holds for every instance of the architecture. The key point is that verification of the basic architecture has been separated from the verification of the individual component instantiations.

5 Example

In this section we present an example of using architecture theories to assist in the verification of a multi-threaded plan execution system. This plan execution system is one subsystem of NASA's New Millennium Remote Agent [13, 12], an artificial intelligence-based spacecraft control system architecture that is scheduled to launch in December of 1998.

5.1 Background

In the plan executive there is a collection of concurrently executing control tasks. To simplify the programming of the individual control tasks, there is a resource management layer that models the spacecraft devices in terms of various *properties* that they may have. The control tasks often require specific values of certain properties to be monitored and maintained in order to execute correctly. The resource manager must provide mechanisms for achieving, maintaining and monitoring values of properties. It must also prevent tasks with conflicting requests from executing concurrently.

Finally, when an event occurs that causes a maintained property to become violated, the resource manager must suspend the subscribed tasks and invoke a specified recovery mechanism that attempts to re-achieve the property. From the perspective of a task, the maintained properties are invariants; they are always true while the task is executing. However, from the perspective of the manager, the properties may come and go.

Some subtle complexity is added to the resource manager because it is parameterized on a collection of failure recovery mechanisms. This complicates testing because it is impossible to insert every possible failure recovery mechanism that may exist.

5.2 Architecture Specification

The architecture for the resource manager is decomposed into four components: a control task, a property locking mechanism, a property maintenance com-

```
theory LockComponent(PL,PL,I,O)
includes
 Property(P), List(P,PL),
 ComponentTheory(PL,PL,LockI,LockO)
axioms
 ∀ pl,pl':PL
 LockI(pl) == true;
 LockOpl,pl') == p ∈ pl' ⇒ isLocked(p);


theory MaintainComponent(PL,PL,MaintI,MaintO)
includes
 Property(P), List(P,PL),
 ComponentTheory(PL,PL,MaintI,MaintO)
sorts
 D is tuple of pl:PL,sc:SCState
axioms
 ∀ pl,pl':PL
 MaintI(pl) == p ∈ pl ⇒ isLocked(p);
 MaintO(pl,pl') == p ∈ pl' ⇒ ¬isLocked(p);
```

Figure 7. Lock and Maintain Components

```
theory ResourceManagerArch(D,R,I,O)
includes
 Property(P), List(P,PL),
 % System level interface:
 ProblemTheory(D,R,SystemI,SystemO)
 MaintainComponent(PL,PL,MaintI,MaintO)
 LockComponent(PL,PL,I,O)
 RecoveryComponent(D_RM,R_RM,I_RM,O_RM)
 % Task Body component
 ComponentTheory(D_body,R,TaskI,TaskO)
sorts
 D tuple of <pl:PL,a:Args,sc:SCState>
 R tuple of <pl':PL,sc':SCState>
axioms
 ∀ p:P,pl,pl':PL,a:Args,sc,sc':SCState

 TaskI(a,sc) ∧ I(<a,pl,sc>)
 ∧ (p ∈ pl ⇒ (isLocked(p) ∧ isAcheived(p)))
   ⇒ TaskO(a,db,db');

 I(<a,pl,sc>) ⇒ LockI(pl) ∧ TaskI(a);

 I(<pl,a,sc>) ∧ TaskO(<a,sc>,sc) ∧ MaintO(pl,pl')
   ⇒ O(<pl,a,sc>,<pl',sc'>);
```

Figure 8. Resource Manager Architecture

ponent and a recovery mechanism. Due to the simple component model currently supported, we pass state into components. We are currently exploring a more powerful component model based on hidden algebras [5]. However, the current method is compatible with the system implementation in LISP.

Figure 7 shows the specification of the Lock and Maintain Components. The specification of the Lock component says that all of the properties will be locked after it executes. The role of the Maintain component is to and maintain component states that all properties will be true after it executes. The abstract state of the space craft is denoted by the uninterpreted sort `SCState`. The recovery mechanism remains unconstrained in the model. Future work will characterize the general assumptions that can be made about existing and potential recovery mechanism and the implications on the system.

The architecture theory for the resource manager is shown in Figure 8. The task body is the part of the task that is executed while properties are being maintained. Because we cannot foresee what functionality the control task will perform, we leave this component uninstantiated. However, its relationship to the system level specification can still be specified. For example, the first axiom in the architecture describes the assumptions that the body can make about the architecture. To model property maintenance as an invariant, the framework must be extended by adding an invariant predicate to the generic problem theory. We are currently investigating this extension.

The system precondition must guarantee the preconditions of the components, directly or indirectly, to guarantee predictable behavior and termination of the system. Therefore, the precondition must guarantee that the lock mechanism will work and that the body will execute if the properties are achieved. The third axiom state that for legal inputs the effect of the execution of the resource manager will be be the desired effect of the task body. In effect, the property maintenance behavior is not visible externally because if does not directly effect the post condition.

5.3 Implementation

The Remote Agent Executive is implemented in LISP. An informal diagram of the implementation of the resource manager is shown in Figure 9. The control tasks are coordinated by subscribing to a lock for the desired property. There is a daemon executing concurrently with the control tasks that monitors the state of the spacecraft (via the database) and the property locks. If there is an inconsistency between the database and the locks, the daemon suspends all tasks subscribed to the property while some action is taken to re-achieve the property.

In order to verify properties of the implementation of the resource manager architecture, parts of the LISP code were hand-translating into the Promela language for the Spin model checker [9]. Promela supports modeling of multi-process systems and uses an interleaving model of concurrency. Spin does exhaustive state space

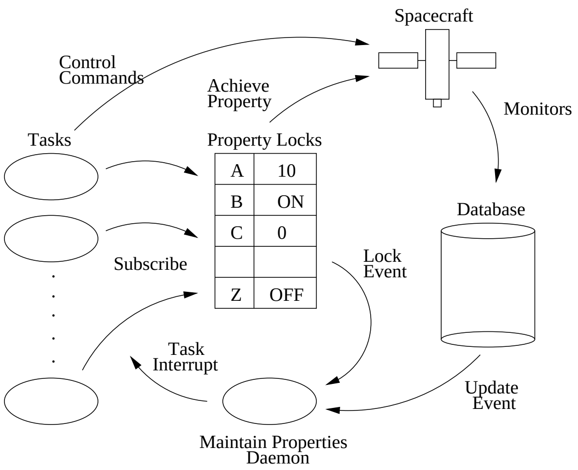

Figure 9. Remote Agent Executive Resource Manager

exploration to verify temporal properties of Promela programs.

Figure 10 shows the implementation of the Lock component. The `with-maintained-prop` function is called by the task to performs the subscription and achieving process. The task passes in a body of code to be executed while the property is maintained. Figure 11 shows the implementation of the Maintain component. This component is implemented as a daemon process that runs concurrently with the tasks. The daemon checks the consistency of the locked properties and the state of the spacecraft when ever there is an update to either.

The formal model has been successfully used to find errors in the generic architecture that were not found by testing an instantiated system. The first error was found attempting to verify that all locks are released when a task finishes executing. Spin identified a case where a task could be aborted and would fail to release its locks due to the lack of an atomic section within the release lock function call. The second bug was found when attempting to verify that a task would eventually terminate when has a property violated. In this case, Spin found a scenario where a component would not abort when it should. This was due to the lack of the atomic section within `get prop lock` that appears in the model. A complete report of this verification activity is in preparation [7].

These subtle bugs persisted in the system throughout many months of testing and simulation. This indicates the difficulty in finding such bugs in a fully

```
#define db_query(p)
  db[p.mem_prop] == p.mem_val

#define get_prop_lock(this,p,err)
  atomic{
    fail_if_incompatible_prop(p,err);
    append(this,locks[p.mem_prop].subscribers);
    if
    :: locks[p.mem_prop].mem_val == undef_value ->
        locks[p.mem_prop].mem_val = p.mem_val;
        locks[p.mem_prop].achieved = db_query(p)
    :: else
    fi;
    signal_event(LOCK_EVENT)
  }

#define achieve(p,err)
  if
  :: db_query(p)
  :: else ->
       if
       :: db[p.mem_prop] = p.mem_val
       :: err = 1
       fi
  fi

#define with_maintained_prop(this,p,task_body)
  bool err = 0;
  {
    get_prop_lock(this,p,err);
    achieve_lock_prop(this,p,err);
    task_body
  }
    unless
  {err || active_tasks[this].state == ABORTED};
  release_lock(this,p)
```

Figure 10. Model of Lock Mechanism

instantiated system. The fact the we were able to find the bugs shows the potential benefits of separating the properties of the generic architecture from the properties of the entire system. This not only allowed us to capturing important properties of the system but also reduced the size of the model so that is could be handled by exhaustive state-space exploration.

6 Related Work

Our work is an extension of the work done on Kestrel's Interactive Development System (KIDS) [16, 18]. In KIDS, the structure of specific algorithms, such as global search or divide and conquer, are represented as algorithm theories. Currently, the program schemes

```
proctype Maintain_Prop_Daemon(TaskId this){
 bit lock_violation;
 byte event_count = 0;
 bit first_time = true;
 do
 :: check_locks(lock_violation);
   if
   :: lock_violation ->
       do_automatic_recovery
   :: else
   fi;
   if
   :: (!first_time &&
      Ev[MEM_EVENT].count
      + Ev[LOCK_EVENT].count != event_count ) ->
      event_count = Ev[MEM_EVENT].count
                    + Ev[LOCK_EVENT].count
   :: else ->
       first_time = false;
       wait_for_events(this,MEM_EVENT,LOCK_EVENT)
   fi
 od
};
```

Figure 11. Model for Maintenance Daemon

that are used to implement algorithms theories result in functional style programs. Architecture theories generalize algorithm theories by specifying structure in terms of subcomponent problem theories rather than operators. This allows the construction of hierarchical systems. We are currently exploring tactics for applying architecture theories for component adaptation based on the results of specification matching [14].

Most efforts to formalize software architecture [1, 15] are targeted at formalizing specific architectural styles (pipe-filer, client-server, etc.) and not with the problem decomposition aspects of architecture. Therefore, the representations used are too operational to represent the types of relationships that we are interested in. However, formal models of architectural styles do provide an important semantic link between an architecture specification and implementation. The two following approaches are particularly well suited to fill this role due to their use of theories to describe architecture.

Gerken presents a formal foundation for software architectures that also uses theories as the main unit of specification [4]. He introduces *structure theories* that are used to interconnect components in various styles. To overcome the inherently functional (as in functional programming language) architecture of algebraic specification, he used a process logic to represent the constraints introduced by structure theories. We believe that the process logic descriptions of architectures are too operational to effectively model the relationships we are interested in. However, this approach does support modeling of invariants which is currently not supported in our framework.

Marconi et. al. [11] use theory-based architecture representations to support architecture refinement. A refinement maps an abstract architecture description in one style to a concrete architecture (an implementation) in a potentially different style. This allows program development by incremental refinements at the architectural level. Axioms are used to describe style constraints and form the basis for correctness proof of the refinement mappings. This work is concerned with architecture implementation and could be used to specify links between architecture theories and architecture schemas.

7 Conclusion

This papers describes a technique for extending declarative specification to the realm of software architecture. An *architecture theory* constrains the behavior of a system in terms of the behavior of its subcomponents via a collection of axioms. The axioms define component interconnection, interface binding and the correctness of an architecture. These relationship are specified declaratively, abstracting away implementation concerns. An example was presented using architecture theories to model the resource management system of a multi-threaded plan executive. The result was that the verification of the basic architecture was separated from the verification of the individual component instantiations. This was especially important because the instantiated system was too complex for standard verification and validation techniques.

8 Acknowledgments

The RA architecture and implementation models benefited greatly from discussions with Michael Lowry, Ron Keesing, Barney Pell, Erann Gat and Mark Gerkin. We would like to thank Michael Lowry, Stephen Seidman and many anonymous reviewers for helpful suggestions during the development of this work. Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under contract F33615-93-C-1316 and F33615-93-C-4304 and NASA Ames Research Center Contract NAS2-13605.

References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[2] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *IJCAI5*, pages 1045–58, 1977.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.

[4] Mark J. Gerkin. *Formal Foundations for the Specification of Software Architecture*. PhD thesis, Air Force Institute of Technology, March 1995.

[5] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *Proceedings of the Ninth Colloquiun on Automata, Languages and Programming*, volume 140 of *LNCS*, pages 265–281, Aarhus,Denmark, July 1982. Springer-Verlag.

[6] J. A. Goguen and R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–255, 1984.

[7] Klaus Havelund, Michael Lowry, and John Penix. Formal analysis of a space craft controller using SPIN. NASA Ames Technical Report (in preparation).

[8] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[9] Gerard J. Holzmann. *Design and Verification of Protocols*. Prentice Hall, 1990.

[10] Richard Jüllig and Yellamraju V. Srinivas. Diagrams for software synthesis. In *The Eight Knowledge-Based Software Engineering Conference*, pages 10–19. IEEE, September 1993.

[11] Mark Moriconi, Xiaolei Qian, and Bob Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[12] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams. A hybrid procedural/deductive executive for autonomous spacecraft. In P. Pandurang Nayak and B. C. Williams, editors, *Procs. of the AAAI Fall Symposium on Model-Directed Autonomous Systems*. AAAI Press, 1997.

[13] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Plan execution for autonomous spacecraft. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.

[14] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[15] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[16] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[17] Douglas R. Smith. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, LCNS. Springer Verlag, 1996.

[18] Douglas R. Smith and Micheal R. Lowry. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.