Exploiting Inheritance in Modeling Architectural Abstractions[1]

K. Hasler
R. Gamble
K. Frasier
T. Stiger
Dept. Mathematical and Computer Sciences
University of Tulsa
600 S. College Ave.
Tulsa, OK 74104 USA
phone: (918) 631-2988
fax: (918) 631-3077
email: gamble@utulsa.edu

1 INTRODUCTION

Formal (mathematical) modeling of a specification provides an unambiguous representation that allows for rigorous analysis and reasoning over properties. Architecture descriptions define an abstract representation of a system component that is amenable to formal modeling. Using formal modeling within this context provides a basis for integrated system descriptions and analysis, as well as a basis for guaranteeing properties of applications. Missing from the formal modeling of architectures, however, is the notion of inheritance.

Representations that include inheritance could provide for a taxonomy of architecture abstractions that model generic classes that can be inherited by more specific classes. Such a taxonomy would include reusable templates and abstract properties that can be inherited to relieve some of the burden of repeated specification and proof. Further benefits can be achieved when designing an integrated system comprising multiple architecture descriptions. By modeling an integrated system at a high level of abstraction, it is easier to initially derive and analyze properties. If the low level, more detailed system models can inherit the high-level specification, then the result is a savings in time and effort, along with a reduction in error (Hasler et al., 1998).

This paper describes a Distributor Controller, a type of controller component (Keshav and Gamble, 1998) that requires decisions to be made on incoming data, such as which data to pass and which known components should receive it. The Distributor Controller is an integration component that can link two heterogeneous components (Stiger and Gamble, 1997) or two complete software systems (Stiger et al., 1997).

2 DEFINING GENERIC COMPONENTS

Based on experimental analysis of integration at the architectural level (Sitaraman, 1997), Keshav and Gamble (1998) partitioned the functionality defined within an integration strategy into three basic integration elements: *Translator*, *Controller*, and *Extender*. In particular, a Controller coordinates and mediates the movement of information between components using some predefined decision-making process or strategy. Consequently, the Controller needs to know the identity of the components for which decisions are made.

Using the Controller as an example serves two purposes. First, it shows how components can be expressed generically and then specialized via inheritance. Second, it shows that integration elements can be modeled at the same level of abstraction as architecture descriptions. Thus, it is possible to achieve consistent representation and analysis across the spectrum of architecture descriptions.

The three models of a Controller component are based on whether or not explicit port names are required. These models are the Distributor, the Composer, and the Coordinator (see Figure 1). This section models an abstract Distributor Controller (DC) component as a class. DC collect inputs in a single source with no knowledge of who is sending the information and then distributes the information to known components according to predefined strategies. Thus, it requires *a priori* knowledge of the components to which its output connects.

[1] This research is sponsored in part by the AFOSR (F49620-98-1-0217). The first author, on leave from Univ. of Wisconsin, La Crosse, was sponsored by the CRA Distributed Mentor Project.

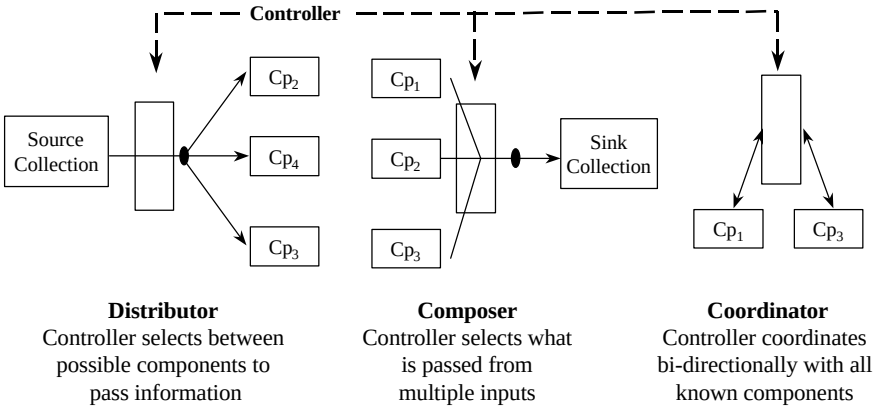

Figure 1: Functional Models of a Controller Integration Element

3.1 Properties of a Distributor Controller

Our formal modeling approach extends the approach first presented by Abowd, et al. (1995), in which an architecture description (component, connector, configuration) is modeled by an *object*, a *state* description and a *step* description to transition the state. Our extensions to OSS include rules to identify properties of the components that guide the designer to choose whether a port or collection variable is appropriate (Stiger, 1997). These rules also identify the appropriate connector models depending on the choice of port or collection variable. Our extension rules, naming conventions, and formats for specifying these variables facilitate a consistent representation across heterogeneous components.

Due to space constraints, we will textually define those properties that comprise the model of DC as shown in Figure 1. The formal model using Object-Z (Duke et al., 1991; 1994) can be found in (Hasler et al., 1998). The properties are partitioned into:

(1) Inherited information,
(2) User-defined types,
(3) Object constants and their inter-relationships,
(4) Private functions of the object,
(5) State variables and their relationships to the object constants, and
(6) State transition constraints.

The benefit of this description format is that it is applicable to most computation components expressed at the architectural level. Thus, a consistent representation and template for description can be used for architectural patterns, as well as integration architectures. This consistency allows for easier analysis across an integrated system to (a) determine interoperability problems, (b) to guarantee solutions given a particular integration architecture, and (c) to express and maintain properties over the integrated system. The benefit of incorporating an object model description is the ability to inherit properties within specialized or application dependent components.

Inherited information. Since DC is a high-level generic component, it does not inherit any information.

User-defined types. One of the keys to successful inheritance is the allowance of variable or generic types in the definition of the model. (Whether they are variable or generic depends on the modeling language.) For example, we can assign the variable types DCSTATE, **set** DATA, and CID to DC. Semantically DCSTATE can be instantiated to any component state type, **set** DATA can be instantiated to any complex data type, and CID can be instantiated to any component identifier. The component identifier is needed because DC must know where to send its output. The variable types are instantiated by name replacement and by using them consistently throughout a new, specialized schema.

Object constants and their inter-relationships. The constants set up the object with all possible allowable information. For DC, we are concerned with the allowable input data, the output component identifiers and the allowable data for each, the states the controller can be in, its start state, and a function that describes the allowable model of the state transition. Basic constraints on the object constants are that each component identifier has designated allowable data called its alphabet and

that the start state is an allowable state. The transition function requires that the data it uses to compute a state change is related to the allowable input data in some way (using a private function **ispartof**) and that the output resulting from the state transition can go on some allowable output port representative of a component identifier.

Private functions. These functions operate internally on the object data. They are defined such that they can be inherited and changed to suit specialized objects. For example, **ispartof** might be defined as "element of" or "subset of" depending on the data types used by a subclass of Distributor Controller. Other internal functions are **related**, **determine**, **add**, and **remove**. These functions are associated with state changes. For example, **determine** employs decision-based strategies to produce the desired output for the appropriate components. This determination is application dependent, allowing only a type definition to be inherited from DC.

State variables and their relationships. The state information represents a snapshot of DC during the course of its execution. This information comprises variables for the current state, which in this case is a combination of local variables and decision-making strategies. The snapshot includes the value of current input data on which computation will be performed, as well as the state of the output ports, represented by component identifiers. Constraints are placed on the state variables to conform to the object constants previously expressed. For example, the data on the output ports at any given time must be allowed by the constant alphabet for each port.

State transition constraints. This is the definition of the actual state transition given the state variable descriptions and their relationship constraints. The private functions **related, determine, add,** and **remove** are used to manipulate allowable input data, remove it as input, and add the decided upon output data to the appropriate component identifier ports.

3 DEFINING SPECIALIZED COMPONENTS

In this section we show how a Controller component for two architectures can be more easily expressed by inheriting from the class of Distributor Controller. Specifically, we discuss the formal models of the Blackboard and Rule-based System (RBS) architectures. The diagrammatic configurations of these architectures from (Stiger & Gamble 1997, Gamble et al 1998) are shown in Figure 2.

In abstract terms, one could think of the Blackboard architecture as a representation of experts working on the blackboard, making changes and seeing others' changes as the events produced. The blackboard component acts as a repository of hypotheses, intermediate deductions, and goals. The experts are represented as Knowledge Sources (KSs) that may access the blackboard concurrently and even compete for its resources. As changes are made to the blackboard, it outputs those changes as a sequence of events. All communication between KSs goes through the blackboard.

To encode the Blackboard architecture, a Distributor Controller is needed to coordinate among KSs, determining their non-interfering interaction upon the current events posted by the blackboard. The decision strategies may be based on conflict resolution to choose the right KS. The Blackboard Distributor Controller must know the preconditions that invoke each KS to determine if a KS can act on the current events. Passed events are FIFO queued to allow changes to the blackboard to occur concurrently with the controller's decisions. There is no way to force a KS to act, once it is invoked. Also, it is possible for an event or set of events not to invoke any knowledge sources. A Composer Controller (see Figure 1) is used to collect the KS changes in order to produce a consolidated sequence of changes to the blackboard for incorporation.

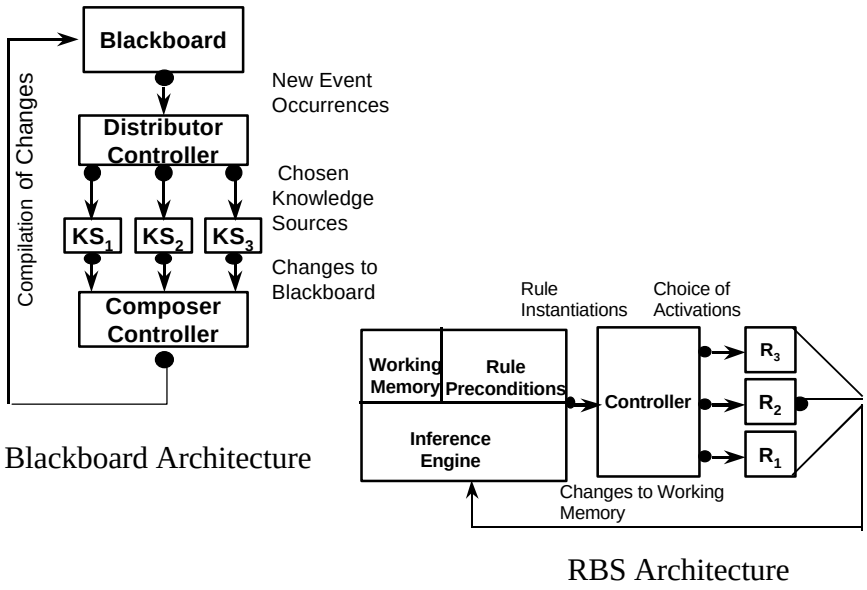

Figure 2: The Blackboard & Rule-based System Architectures

RBS Architecture. The RBS architecture requires three component types: (1) an Inference Engine that holds working memory, (2) Rules that embody the knowledge of the system, and (3) a specialized Distributor Controller that uses conflict resolution to determine which rule can update working memory. In general, RBSs operate on a discrete cyclic of match-select-execute. The match cycle matches the rules from a knowledge base with the facts, premises, and deductions that form working memory. The inference engine determines, using the match, which rules can execute to modify working memory. The selection stage of the cycle determines which rule (in a typical serial system) should update working memory. One such conflict resolution strategy is to focus on the changes most recent to working memory. Once the chosen rule executes, the cycle continues until there are no more matches. As shown in Figure 2, the Inference Engine provides rule instantiations to DC for its conflict resolution process. Since the system as modeled is constrained to allow only a single rule to act at each cycle, no Composer Controller is needed to collect multi-rule actions.

Inherited information. In both the Blackboard and RBS architectures, the properties of the generic Distributor Controller are inherited for their respective Distributor Controllers, called BB-DC and RBS-DC. A subclass is used instead of an instance because there may be many versions of a Distributor Controller that are dependent on particular application domains. Thus, more flexibility is achieved in specifying later components that may be a subclass of BB-DC and RBS-DC.

User-defined types. In BB-DC, the variable types of DC are replaced, in order, by BCSTATE × **set** PRECONDITION, **seq** EVENT, and KSID. With this binding, the state type includes the KS preconditions as expressed by ordered pairs of the form BCSTATE × **set** PRECONDITION above, the data type for input and output is a sequence of events, and the components coordinated by BB-DC are represented in the controller's ports as KS identifiers. For example, where states was of type DCSTATE ↔ **set** STRATEGY in DC class, it is defined in the BB-DC subclass as type (BCSTATE × **set** PRECONDITION) ↔ **set** STRATEGY. In RBS-DC, the variable types of DC are respectively replaced by RCSTATE, INSTANTIATION, and RID, binding the state type to RCSTATE, the data type to INSTANTIATION, and the component identifier to RID for rule identifier.

Object constants and their inter-relationships. The type changes are automatic with the above bindings, thus the constants are inherited directly from DC for both architectures. No new constants are added to either architecture definition. However, in the BB-DC subclass, there is the additional constraint that the controller processes only one event at a time. This constraint is explicitly stated because BB-DC can collect a sequence of events from the blackboard component, but can only act upon one at a time.

Private functions. The class functions, **ispartof, related, remove**, and **add** can be axiomatically defined for both BB-DC and RBS-DC subclasses. Each subclass can define them differently. As with the constants, the type changes for the functions are automatic with the initial bindings. The distinct specialized definitions are an essential part of the modeling approach, in that they can be tailored to particular architecture descriptions or application domains. The function **determine** is application dependent so it retains its generic definition.

State variables and their relationships. By inheritance, the state variables and their relationships in DC are implicitly included in BB-DC and RBS-DC. A new state variable is needed in BB-DC to represent the preconditions of the KSs. In RBS-DC, a constraint is added that requires the controller to process all input at once. The addition of variables and constraints in the formal model is acceptable as long as previous type and constraint definitions are not violated.

State transition constraints. The state transition assertions of DC are directly inherited with the type bindings defined above. No changes are necessary.

4 DISCUSSION

Given that any of the component, connector, and configuration models of architectural descriptions can be placed in a consistent format that allows inheritance, several design and analysis simplifications result. First, integration components, such as DC, can be predefined for inclusion or specialization into an integrated system. Second, by employing inheritance, we can develop a critical mass of formal models in a structured hierarchy of architecture descriptions. Using these models we can analyze heterogeneous architectures without having to prove inherited properties.

REFERENCES

Abowd, G., Allen, R., Garlan, D. (1995) Formalizing style to understand descriptions of software architecture. *ACM TOSEM*.

Duke, R., King, P., Rose, G., and Smith, G. (1991) The Object-Z specification language: Version 1. Software Verification Research Centre, Department of Computer Science, The University of Queensland, Technical Report 91-1.

Duke, R., Rose, G., and Smith, G. (1994) Object-Z: A specification language advocated for the description of standards. Software Verification Research Centre, Department of Computer Science, The University of Queensland, Technical Report 94-45.

Gamble, R., Stiger, P.R., Plant, R.T. (1998) The rule-based architecture style and its application to contemporary knowledge-based systems, with P.R. Stiger, and R.T. Plant. TR UTULSA-MCS-98-2, Dept. of Mathematical and Computer Sciences, University of Tulsa.

Hasler, K.M., Gamble, R.F., Frasier, K.A., and Stiger, P.R. (1998) Architectural Style Modeling. TR UTULSA-MCS-98-11, Dept. Mathematical and Computer Sciences, University of Tulsa.

Keshav, R. and Gamble, R. (1998) Towards a taxonomy of architecture integration strategies. TR UTULSA-MCS-98-7, Dept. MCS, University of Tulsa.

Sitaraman, R. (1997) Integration of software systems at an abstract architectural level. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa.

Stiger, P. (1997) An assessment of architectural styles and integration components. M.S. Thesis, Department of Mathematical Computer Sciences, University of Tulsa.

Stiger, P.R., and Gamble, R.F. (1997) Blackboard systems formalized within a software architectural style. Int'l conference on Systems, Man, Cybernetics.

Stiger, P.R., Gamble, R.F., Bauer, S., and Smith, S. (1997) Fitting an application specification to an architectural style. TR UTULSA-MCS-1997-25, Dept. Mathematical and Computer Sciences, University of Tulsa.