Architecture Abstraction Tower

Zhenyu Wang
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
(01) 412-2682582
zwang@cs.cmu.edu

1. ABSTRACT

Bridging architectural models and implementations remains a research challenge to be addressed. In this position paper, I propose an approach based on the construction of an *architecture abstraction tower* in which layers of system models are *causally* connected. I describe how abstraction towers can be used for architectural conformance analysis, debugging, reflection and interpreting architecture analysis results.

1.1 Keywords

Causal Connection, Architectural Layer

2. INTRODUCTION

Architectural models are meaningless unless they can be associated with concrete systems. Architecture analysis does not help much if there is no accurate mapping from architectures to implementations. Further, as Murphy argued in [1], we need to be able to deduce the impact of system evolution on architectures. Other architecture related engineering activities (such as architecture instrumentation [2] and reflection) also necessitate the capability to bridge software architectures and implementations.

I believe that the *architecture abstraction tower* is a promising approach to achieve this goal. Intuitively, an abstraction tower is a stack of abstract models that describe a software system. Usually these models differ in abstraction levels. The unique feature of an abstraction tower is that modifications made to any model in the tower can be propagated across the whole stack. Through architecture abstraction towers, any modification made to the implementation can be reflected to the architecture level (when applicable) and vice versa.

3. ARCHITECTURE ABSTRACTION TOWER

An architecture abstraction tower is a stack of abstract models for a system. At the bottom of the tower is an implementation model and the top of the tower is a high level architecture model specified in some ADL (figure 1). A model is *causally* connected to models at adjacent layers, which means that changes in one model will lead to changes in the two models when applicable[1]. Through layers of causally connected models, we can connect an implementation with an architecture specification so changes in implementations will lead to changes in architecture models and vice versa.

A meta-model is associated with a layer (or several layers). The meta-model describes a set of fundamental structures, properties associated with these structures and basic operations on the structures. The operations can be used to modify properties and structures (by creating or destroying them, creating connections between them, etc.). A model is implicitly specified through sequences of operations. We do not differentiate between a static model and modifications to the model; instead, a model is thought to be created from an empty spec through a set of operations.

[1] Since models differ in abstraction levels, certain changes in one model may be of no interest to adjacent models.

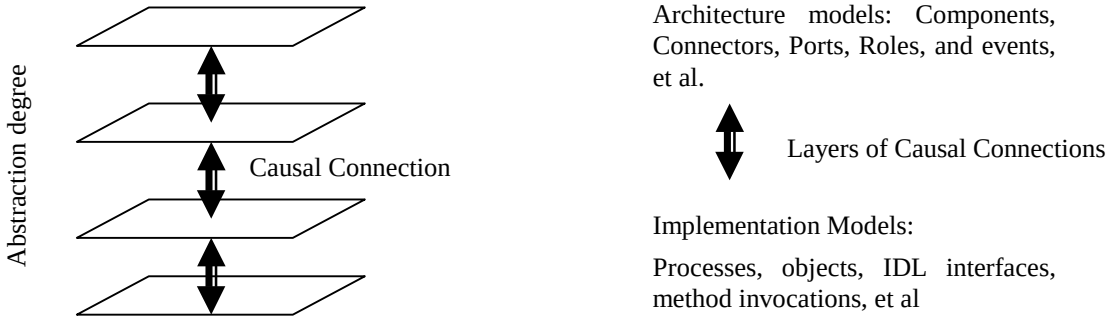

Figure 1: Architecture Abstraction Tower

For example a model with two structures and one connection between them is treated as creating structure one, creating structure two, creating connection between them (the timing and order of the operations do not matter).

A causal connection connects two models at different abstraction levels through a set of causal rules. A causal rule specifies how a model changes if modifications (operations) are made to another model. For example, a sequence of relevant system calls that create a shared memory connection (at layer n) may result in the creation of a data flow (at layer n+1). As another example, creating an interface in layer n+1 may lead to the creation of multiple interfaces in layer n.

The meta-model associated with the bottom layer of the tower captures fundamental structures that appear in an implementation. These structures include both static structures such as classes and dynamic structures such as processes. I am working on such a meta-model for C++ based applications. Table 1 shows some of the structures and operations I am considering.

Properties such as class name and object types are associated with relevant structures. A model can be created using static code inspection tools such as Refine™ and dynamic monitoring tools such as FLEA[5].

In my current work, the meta-model for the top layer is adopted from Wright[6]. The fundamental structures are components (and types), connectors (and types), ports, roles, processes and events. Basic operations include creating and destroying these structures, firing events and modifying properties.

I am also working on a formal language that can be used to specify causal rules. A rule is a trigger-action pair where both trigger and action are pattern of operations (at different layers). Triggers and actions can be attached with predicates that govern when a rule is applicable.

For example, a rule may specify that if a structure X (e.g. an object or a process) at layer N creates a shared memory region and then structure X and Y both get a handle on this region, then a dataflow connection can be created in layer

Table 1

Fundamental Structures	Basic Operations
Data	Declare, destroy, access, copy
Object	Declare, create, destroy, reference, copy, method invocation
Interface (for CORBA/COM)	Create, associate
Class	Declare, inherit, associate
Procedure	Declare, enter, exit
File	Create, reference
Thread	Create, destroy, system call
Process	Create, destroy, system call
Stack Frame	Create, destroy
Function call	Invoke

N+1 (which in turn might trigger some higher level rules).

Another rule might specify that if two relevant interfaces are attached to a structure, then a single interface can be created at a high level and then every action issued from the low level interfaces maps into actions issued from the high level interface.

As a further example, suppose we have a simple application[2] (shown below in pseudo code) that transforms strings:

```
…
void main()
{
        String buffer;
                // data declared
        cin >> buffer;
                // get string from input
        String_reverse(buffer);
                // reverse the string
        String_Capitalize(buffer);
                // capitalize the string
        String_deRepetition(buffer);
                // remove repetitive characters
        cout << buffer;
                // output the string
}
```

String_reverse, string_capitalize and string_deRepetition are predeclared procedures.

Following rules[3] set up connections between the application and a high level object-oriented architecture.

```
OnClassDeclared(?class) →
        CreateComponentType(?class)
        CreatePort("methods",
                GetComponentType(?class))

OnObjectDeclared(?Instance, ?class) →
        CreateComponentFromType(?Instance, ?class)

OnProcedureDeclared(?p|?p = String_*) →
        AddEventToPort(?p,
                GetComponentType("String").methods)

OnProcedureExit(?p, ?param) →
        GetComponent(?param).FireEvent(GetEvent(?p
))
```

The first rule creates a component type String with a port "methods" when it detects that class String is declared. The second rule creates a component buffer. The third rule adds an event to the port whenever a procedure whose name begins with String is declared. The fourth rule fires an event if a procedure gets invoked.

[2] This small application was originally described by Bob Monroe

[3] These rules only show possible causal connections to convey the idea. They are not written in the formal notation I am working on.

Following rules[3] set up the connection between the application and a high level pipe-filer architecture.

```
OnDataDeclared(?d) || OnObjectDeclared (?d) →
                CreateDataComponent(?d)

OnProcedureEnter(?p, ?param) && OnDataAccess(?d)→
        CreateDataAccessComponent(?p)
        CreateDataflowConnection(GetComponent(?p)
                ,GetComponent(?d))
        GetComponent(?p).FireEvent(ReadorWrite ?d)

OnDataAccessComponentFireEvent(?p, read ?d)
+ OnDataAccessComponentFireEvent(?p, write ?d) →
        CreateFilterComponent(?p)

OnDataAccessComponentFireEvent(?p, write ?d)
+ OnDataAccessComponentFireEvent(?q, read ?d)
        →CreatePipeConnector(GetComponent(?p),
                                GetComponent(?q))
```

The first rule creates a data component buffer in the second layer (the rule is fired on implementation layer). The second rule creates three data access components when the three procedures modify the content of the string buffer. The third rule creates filter components in the third layer when it detects that the data access components read the data and then write it back. The fourth rule creates two pipe connectors in the third layer when it detects that one data access component reads the data after another component writes it.

Besides formalisms for specifying causal rules, I am working on a *guardian environment* that keeps the integrity of an abstraction tower by managing and firing eligible causal rules. (As I mentioned before, the initial construction of any model is treated as a sequence of modifications to an empty specification)

4. APPLICATIONS

An abstraction tower can be used for:

consistency maintenance between architectures and implementations during evolution. Since changes made to the implementation (for example, adding news classes or communication channels) will be propagated into the architecture level by the abstraction tower, architects can predict how system architectures might change during evolution.

high level debugging and architecture level instrumentation. Layered causal connections provide mapping between low level activities and high level activities. It can be easily used to support architectural debugging and instrumentation [2]. By blocking the execution of an implementation when the environment fires certain causal rules (for example when the creation a

process leads to the creation of a architectural component), users can set breakpoints at the architecture level and single step trace through connector protocols. External tools can be integrated with the guardian environment so users can add extra processing when rules are fired, for example, an external tool may log all data sent along a pipe connector.

conformance analysis [3][4]. The abstraction tower maintains the current snapshots of system structural models. These models can be checked against an ideal model. The conformance analysis can also be made online. An *architecture monitor* (built upon a guardian environment) can be built to run parallel with an application to make sure that the application architectural model is always preserved (and send out warning messages when it is not).

architectural reflection. Architectural reflection lets an application reason about its own architecture. The environment can be integrated with an application so the application can be informed of its own architecture (for example, get the internal dataflow) when necessary and take suitable actions, for example, to perform dynamic configuration.

bridging analysis results. The mapping set up by the abstraction tower will provide meanings to analysis results. For example when we need to increase the throughput property of an architectural component for performance reasons, the abstraction tower can map this into concrete implementation changes (increase thread priority, bandwidth, etc.).

5. CONCLUSION

This paper briefly describes the notion of an architecture abstraction tower and how it can help bridge the gap between implementations and architecture models. Causally maintaining layers of model at different abstraction level, abstraction towers can be used in several architecture-related engineering activities.

I am working on enabling notations and tools to bridge implementations and architecture models in Wright based on the idea of abstraction towers. Specifically, I am developing a meta-model for C++ based implementations and formalism for capturing causal rules between models. I am also developing a guardian environment that keeps the integrity of an abstraction tower.

6. ACKNOWLEDGEMENT

The idea described in this paper originated from several discussions with Robert Balzer in the summer of 1997. I also want to thank Bob Monroe for his comments on the draft.

7. REFERENCES

[1] G.Murphy, Architecture for Evolution, *Joint Proceedings of the ACM SIGSOFT 96 Workshops ISAW-2 and Viewpoints 96*

[2] R. Balzer, Instrumenting, Monitoring and Debugging Software Architectures, *http://www.isi.edu/software-sciences/papers/instrumenting-software-architectures.doc*

[3] M. Sefika, Monitoring Compliance of a Software System with Its High-Level Design Models, *Proceedings of 18th International Conference of Software Engineering*

[4] D.Harris, Reverse Engineering to the Architectural Level, *Proceedings of 17th International Conference of Software Engineering*

[5] FLEA Manual, http://www.compsvcs.com/flea-manual.html

[6] R.Allen, A Formal Approach to Software Architecture, PhD Thesis, Carnegie Mellon University, CMU-CS-97-144