COM Revisited: Tool-Assisted Modelling of an Architectural Framework

Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, MA 02139, USA
dnj@lcs.mit.edu

Kevin Sullivan
Dept of Computer Science
University of Virginia
151 Engineer's Way
Charlottesville, VA 22903, USA
sullivan@virginia.edu

ABSTRACT

Designing architectural frameworks without the aid of formal modeling is error prone. But, unless supported by analysis, formal modeling is prone to its own class of errors, in which formal statements fail to match the designer's intent. A fully automatic analysis tool can rapidly expose such errors, and can make the process of constructing and refining a formal model more effective.

This paper describes a case study in which we recast a model of Microsoft COM's query interface and aggregation mechanism into Alloy, a lightweight notation for describing structures. We used Alloy's analyzer to simulate the specification, to check properties and to evaluate changes. This allowed us to manipulate our model more quickly and with far greater confidence than would otherwise have been possible, resulting in a much simpler model and a better understanding of its key properties.

KEYWORDS

Architectural style; integration frameworks; Microsoft COM; Alloy; constraint solver; formal specification; formal methods.

1 INTRODUCTION

Recent research has shown that some widely used architectural standards have serious design flaws. Garlan et al, for example, found a variety of problems in the DoD's High Level Architecture (HLA) for distributed, component-based simulation [2]. In this paper, we follow up on study [13] in which Sullivan et al captured, in a formal model, a defect of Microsoft's Component Object Model (COM) [3].

Because such standards are the infrastructure for much industrial development, their design and validation is a major concern. Moreover, articulating their key properties is vital for their effective and safe use.

Formal modeling and analysis are often advocated as aids to design. For control-oriented properties, there are tools that allow formal models to be simulated and checked automatically. For structural properties, however, tools have not generally been available, and the construction of formal models has been more manual, and thus less appealing to practitioners.

SIGSOFT 2000 (FSE-8) 11/00 San Diego, CA, USA

The study described here is unusual in two respects. First, it involves automatic analysis of structural properties, expressed in a logical language. Most studies of automated formal analysis have either involved control-oriented properties, or have restricted the language to an operational subset. This study supports our contention that executability and abstraction (ie, implicit specification) are not incompatible, as often thought. Second, our use of the analysis focuses on the refinement of the formal model. Most previous studies have emphasized the use of analysis for ex post facto checking of properties; ours shows how analysis contributed to the form of the model itself.

In this paper, we explain how we used the Alloy Analyzer (AA) [9] in the development of a model of Microsoft COM's query interface and aggregation mechanism. We started by translating an existing model from the Z specification language to Alloy [8], in order to exploit AA's analysis. Although there are many tools available for Z, none offer the kind of automatic, deep semantic analysis that AA provides. Using AA, we were able to simplify the model dramatically, and in so doing to sharpen our understanding of the key properties the model expresses.

It is hard to convey in our paper how much fun this experience was, and how humbling, as we discovered again and again how subtle even a small model can be, and how many errors we made as we refined it. We have come to think that one should have very little confidence in a model that has not been subjected to some form of deep semantic analysis. Although theorem proving might also have exposed our errors, it would have been far more labor intensive. AA is fully automatic and requires no manual assistance (beyond the selection of a scope, explained below). In our sessions using the tool to refine our model, almost all the time was spent formulating queries for analysis and reacting to them.

By increasing our confidence in changes we made to the model, the tool allowed us to explore alternative formulations far more rapidly, and to spend more time understanding the model and refining it, and less time checking that it said what we intended. The tool helps to mitigate one of the major risks of formalization: that the model does not reflect the designer's intent. It makes modeling more like programming, and thus less daunting to non-mathematicians. Like any tool, it can of course be abused, but, unlike a debugger, it seems to encourage a more systematic approach to modelling, by giving the designer an incentive to formulate properties that are expected to hold, and by reducing the cost of reworking.

This paper is a case study intended to illustrate the kind of analysis that is possible in the context of a realistic problem. It does not present general guidelines, let alone a comprehensive method. Nevertheless, the approach we took is likely to be applicable to many problems of this sort. Alloy, our modeling language, is well suited to describing architectural structure, and seems to be compatible with several recently developed architectural modeling languages [5], such as AML [18], Darwin [12], and Armani [14], whose tools do not currently provide the kind of deep semantic analysis offered by AA.

The rest of this paper is organized as follows. Section 2 presents an overview of COM focusing on its design rationale. Section 3 presents a brief informal specification of the central elements of COM that we then discuss in formal terms for the rest of the paper. Section 4 presents a new formal model of COM, in Alloy, which we derived by simplifying an initial Alloy model (given in the appendix) obtained by direct translation of our Z model.

Sections 5 and 6 describe the analysis. Section 5 presents the results of checking a variety of properties, and illustrates the use of the Alloy Analyzer to generate sample configurations. Section 6 explains how we used the analyzer to simplify and refine the model. Section 7 addresses the runtime performance of the Alloy Analyzer as we applied it in these tasks. Section 8 discusses related work. Section 9 summarizes and concludes.

2 OVERVIEW OF COM

The COM specification [13] defines an infrastructure for the creation, operation, and management of components. COM is intended to overcome shortcomings in earlier models for component-based software. In particular, language-based models, such as C++, did not foster rich component markets. In retrospect, such models had numerous problems.

First, binary level procedure call interoperability was not specified by C++, so incompatibilities can arise between object modules generated by different compilers. COM resolves this problem by defining a standard binary format supporting procedure call among components, irrespective of source language or compiler technology [3].

Second is the problem of sharing of components. When components are distributed as object libraries, changing the version of an installed component can break other applications that depend on it, since two versions of the same component can export interfaces that differ in syntax and semantics.

COM resolves this problem by introducing a language-independent, binary version of dynamic type coercion, called *interface negotiation*. Every interface of a component is required to provide a method for obtaining interfaces of other types on the same component. Clients are required always to 'query' a component for interfaces of desired types. If a component does not support the requested type, the client has an opportunity to respond gracefully, rather than failing catastrophically [3].

COM also stipulates that the syntax and the semantics of all interface types are fixed: an interface type always means the same thing. Changing the syntax or semantics of an interface requires the introduction of a new type. For this reason, it is common in COM systems to find multiple versions of an interface, such as *IPersist* and *IPersist2*.

A third issue is runtime performance. When components are hierarchically nested, an outer object may sometimes need to export an interface that is already implemented by an inner component. Traditional object-oriented design would require that the outer component export an interface that simply forwards calls to the corresponding inner interface. This incurs an unnecessary forwarding cost.

COM therefore introduced an aggregation mechanism in which an outer object can pass off an interface of inner object as its own. When a client of the outer component queries one of the component interfaces for an interface that is implemented by an inner component, the outer component queries the inner for the requested interface and passes the result to the client. Calls made through such an interface are implemented directly by the inner component. In high performance applications with deep nesting, the savings can be significant [13].

It is clear even from the published COM specification [13] that interface negotiation and aggregation interact in complex ways. When an interface that is exposed by an outer component but implemented by an inner component is queried, the query must be treated as pertaining to the outer component. COM thus requires aggregated components to delegate queries made to inner interfaces to interfaces implemented by the outer component.

Another problem arises. If an inner component delegates *all* queries, then not even the outer can obtain inner interfaces. COM thus requires that every inner component provide a special non-delegating interface. When an outer component is queried for an interface that is implemented by an inner component, the outer queries this non-delegating interface.

The basic problem that we found our earlier analysis is that inner components cannot generally be first-class COM objects. All of their interfaces but for non-delegating interfaces must delegate to the outer component. Consequently, inner interfaces do not follow the standard rules of COM, and one cannot build a system in which multiple aggregated components treat each other as ordinary COM components.

3 ESSENTIAL FEATURES OF COM

In this section we outline with more precision the core object model of COM. Our description corresponds closely to the formal model presented later.

There are two kinds of COM *components*. First there are those that follow all of the normal rules of interface negotiation, which we call *legal*, or *outer*, components. Then there are those that are aggregated by other components. We call them *inner* components. Since inner components do not obey standard COM rules, they are not legal in our technical sense, although our specification admits them, in order to model realistic systems.

An outer component has one or more interfaces. As we will see, an inner component has two or more interfaces. An interface is said to satisfy one or more interface *specifications*, each of which has a corresponding unique interface identifier, or *IID*. In this paper, we abstract from most details of interfaces, themselves, and so we model interfaces as simply having IIDs. A component thus also has a set of IIDs, namely those of all its interfaces.

Every interface must satisfy a special interface specification called *Unknown* that defines an operation called *QueryInterface*, or *QI*. QI takes an IID as a parameter. A call to QI must succeed or

```
model COM {
 domain {Component, Interface, IID}
 state {
  interfaces : Component+ -> Interface
  ciids : Component -> IID
  first, identity : Component -> Interface!
  eq : Component -> Component
  iids, iids_known : Interface -> IID+
  qi [Interface] : IID -> Interface?
  reaches: Interface -> Interface
  LegalInterface : Interface
  LegalComponent : Component
  aggregates : Component -> Component
  }

 inv ComponentProps {
  all c | (c.first + c.identity) in c.interfaces
  all c | all i : c.interfaces | all x: IID | x.qi[i] in c.interfaces
  }

 def ciids {all c | c.ciids = c.interfaces.iids}
 def eq {all c | c.eq = {d | d.identity = c.identity}}
 def iids_known {all i | i.iids_known = {x | some x.qi[i]}}
 def reaches {all i | i.reaches = IID.qi[i]}

 inv Identity {some Unknown |
    all c | all i: c.interfaces | Unknown.qi[i] = c.identity}
 inv InterfaceLegality {all i: LegalInterface, x: i.iids_known | x in x.qi[i].iids}
 inv ComponentLegality {LegalComponent.interfaces in LegalInterface}
 inv Reflexivity {all i: LegalInterface | i.iids in i.iids_known}
 inv Symmetry {all i, j : LegalInterface | j in i.reaches -> i.iids in j.iids_known}
 inv Transitivity {all i, j : LegalInterface |
     j in i.reaches -> j.iids_known in i.iids_known}

 inv Aggregation {
  no c | c in c.+aggregates
  all outer | all inner : outer.aggregates |
   (some inner.interfaces & outer.interfaces)
   && some o: outer.interfaces |
       all i: inner.interfaces - inner.first | all x | x.qi[i] = x.qi[o]
  }
}
```

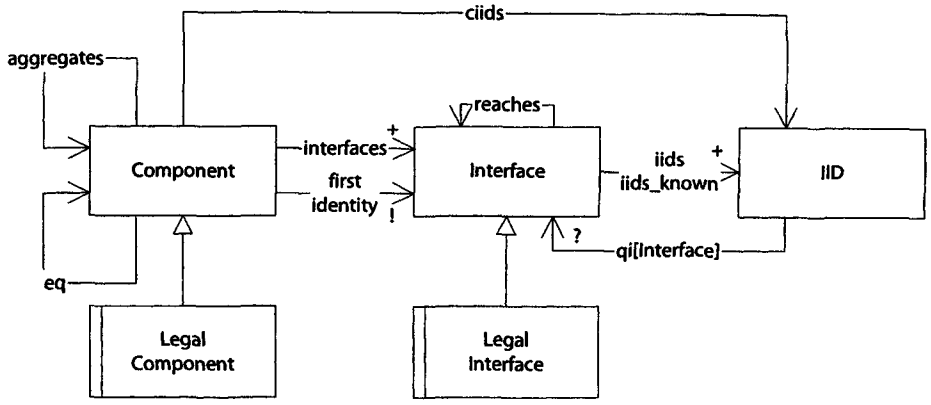

Figure 1: The new model of COM

fail. When a call succeeds it must return an interface; otherwise it must not return an interface. Calling QI through an interface of an outer component with a given IID succeeds if and only if the component exports an interface satisfying that IID; otherwise the call fails. If such a call succeeds, the returned interface must be an interface of the outer component and the interface must satisfy the given IID. We will discuss the rules for QI for interfaces of inner components in a moment.

A component, inner or outer, has a fixed identity, defined to be the special interface returned when QI is called through any interface of the component with the IID *Unknown*. Two components are COM-equal if and only if they have the same identity.

A component can aggregate any number of other components, to arbitrary depth. By definition, a component aggregates another if and only if there is some interface of the second component that is also an interface of the first component. No component can aggregate itself, either directly or indirectly. A COM component that no other component aggregates is an *outer* component. A COM component that is aggregated by some other component is an *inner* component.

An inner component must have one distinguished interface, *first*. This is the non-delegating interface of the inner component. Querying the first interface of an inner object is required to return a corresponding interface on the inner component, if there is one. But querying an interface of the aggregated component other than first for any IID must produce the same result as if QI had been called with that IID on some interface of the directly or indirectly enclosing outer object. It is in this sense that an inner component does not follow the standard rules of COM.

4 A NEW MODEL

Our new formalization of the core object model of COM (Figure 1) is written in Alloy [8], a first-order notation that attempts to combine the best features of Z [17] and UML [15]. From UML and its predecessors, it takes various declaration shorthands, navigations, and a focus on set-valued rather than relation-valued expressions; from Z, it takes schema structuring and a simple set-theoretic semantics. A detailed rationale for Alloy's design is given elsewhere [8]. The diagram on the right hand side of Figure 1 corresponds exactly to the state declarations of the *domain* and *state* paragraphs of the textual specification.

4.1 Alloy Basic Concepts (ABC)

Domains. The *domain* paragraph introduces basic sets that partition the universe of atoms. Alloy is strongly but implicitly typed; there is a basic type associated with each domain (which in Z would be declared explicitly as a 'given type'). *Component*, *Interface* and *IID* model respectively the components, interfaces and interface identifiers in a COM system. Unlike a given type, a domain is a set of atoms that exist in a particular state and not a platonic set of possible atoms. So *Component* represents the set of components in a particular configuration, not the set of all imaginable components.

Multiplicities. The symbols + (one or more), ! (exactly one) and ? (zero or one) are used in declarations to constrain sets and relations. The declaration

$$r: S\,m \text{ -> } T\,n$$

where *m* and *n* are multiplicity symbols, makes *r* a relation from *S* to *T* that maps each *S* to *n* atoms of *T*, and maps *m* atoms of *S* to each *T*. So *first*, for example, maps each component to exactly one interface, and is thus a total function; and *interfaces* maps at least one component to each interface. Similarly, the declaration

```
S:T m
```

makes *S* a set of *m* atoms drawn from the set *T*. Omission of a multiplicity symbol implies no constraint, so *LegalInterface* is a set of interfaces that may have any number of elements. The declaration

```
qi [Interface] : IID -> Interface?
```

makes *qi* an 'indexed relation'; for each atom *i* in the set *Interface*, *qi[i]* is a relation from *IID* to *Interface* (which, as the multiplicities indicate, is a partial function).

Expressions. All expressions in Alloy denote sets of atoms. The conventional set operators are written in ASCII form: + (union), & (intersection), - (difference). The *navigation* expression *e.r* denotes the image of the set *e* under the relation *r*: that is, the set of atoms obtained by 'navigating' along *r* from atoms in *e*. In *e.+r*, the image under the transitive closure of *r* is taken instead: that is, navigating one or more steps of *r*. Scalars are treated as singleton sets. This allows us to write navigations more uniformly, without converting between sets and scalars or worrying about the difference between functions and more general relations. So the expression

```
inner.interfaces - inner.first
```

for example, denotes the set of interfaces of *inner*, with *inner's first* interface taken away. Finally, the keyword *in* denotes subset, and because scalars are sets, doubles as set membership.

Quantifiers. The existential and universal quantifiers are written *some* and *all*. Less conventionally, *no x | F* and *sole x | F* mean that there is no *x* and at most one *x* that satisfy *F* respectively. Quantifiers are used in place of set constants, so

```
some inner.interfaces & outer.interfaces
```

for example, says that there is some interface in the intersection of *inner.interfaces* and *outer.interfaces*. Bounds of quantified variables may optionally be omitted; in

```
all c | c.ciids = c.interfaces.iids
```

the variable *c* is inferred to belong to domain *Component*, and could have been written equivalently as

```
all c: Component | c.ciids = c.interfaces.iids
```

Here, we have omitted most bounds for brevity's sake, but used variable names consistently to avoid confusion: *c* and *d* for components; *i* and *j* for interfaces; and *x* for interface identifiers.

Paragraphs. An Alloy model is divided into paragraphs much like Z schemas, but Alloy distinguishes different kinds of constraints. An invariant (introduced by the keyword *inv*) models a constraint in the world being modelled; a definition (*def*) defines one variable in terms of others, and can in principle always be eliminated along with the variable being defined. An assertion (*assert*) is a putative theorem to be checked. A condition (*cond*) is a constraint whose consistency is to be checked, but unlike an invariant is not required always to hold.

4.2 An Alloy Model of COM

The essential variables that model a COM configuration are:

- the domains *Component*, *Interface* and *IID*, which model the sets of COM component objects, interfaces and interface identifiers respectively;
- the relation *interfaces* that maps a component to the set of interfaces it provides;
- the functions *first* and *identity* that map each component to two special interfaces: the interface that is first obtained when the component is created, and the interface that determines the component's identity;
- the relation *iids* that maps an interface to the set of interface identifiers that describe it.
- the indexed relation *qi*, which models the query interface method, and for each interface *i*, gives a partial function *qi[i]* from identifiers to interfaces.

It is clear from the published descriptions of COM that aggregated components must obey additional rules. What is less clear, and is the key lesson of our previous study [16], is that if hiding is to be supported – as intended by COM's designers and reflected in common practice – the standard rules must be *weakened* for aggregated components. In our model, we therefore represent those components that obey the standard rules as a subset *LegalComponent*, to which inner components will not generally belong, and we impose additional rules on inner components in a separate invariant.

The invariant *ComponentProps* gives the basic properties of all components: that the special *first* and *identity* interfaces must be interfaces of the component, and that, for every interface *i* of a component *c*, the result of a query on any identifier *x* must (if it exists) be an interface of *c*.

A sequence of definitions is then given for the relations we have not mentioned so far. The first introduces a shorthand, *ciids*, that maps a component to the set of identifiers of all its interfaces. The second specifies the standard COM notion of equality for components: the relation *eq* maps a component to those equal to it, defined as the components with which it shares an identity interface. The third makes explicit a notion mentioned informally in our previous model: *reaches* maps an interface to the interfaces that might be reached from it by a single query. The fourth defines *iids_known* to map an interface to the identifiers that are known to it, in the sense that queries on those identifiers are successful.

The invariant *Identity* states a crucial property of the identity relation: that a query with the identifier *Unknown* on any interface of a component yields the component's identity interface. The invariant actually says not only that such an identifier *exists*, but that it is the same one for all components.

The invariant *Legality* gives the condition under which an interface is legal: that a query for any known identifier *x* must yield an interface whose identifiers include *x*.

The invariant *LegalComponents* says that a legal component has only legal interfaces.

The following three invariants capture widely cited rules of COM. By quantifying over only legal interfaces, we allow interfaces to exist that do not obey them.

- *Reflexivity* says that all the identifiers of an interface are known to it: that is, if an interface has an identifier itself, then a query on that interface for that identifier must succeed.

- *Symmetry* says that if an interface *j* can be reached from an interface *i* by a query, then the identifiers of *i* must in turn be known to *j*.
- *Transitivity* says that says that if an interface *j* can be reached from an interface *i* by a query, then the identifiers known to *j* must also be known to *i*: in other words, any IID that takes you somewhere from *j* also takes you somewhere from *i*.

```
// every iid a component exports is known to all its interfaces
assert Theorem1 {
  all c: LegalComponent, i: c.interfaces | c.ciids = i.iids_known
  }

// can't hide iids of legal inner components
assert Theorem2 {
  all outer | all inner : outer.aggregates |
    inner in LegalComponent -> inner.ciids in outer.ciids
  }

 // aggregation merges identities
assert Theorem3 {
  all outer | all inner : outer.aggregates | inner in outer.eq
  }

// iids of a shared interface cannot be hidden
 assert Theorem4a {
    all c1: Component, c2: LegalComponent |
      some (c1.interfaces & c2.interfaces) -> c2.ciids in c1.ciids
  }

// any sharing merges identities
 assert Theorem4b {
    all c1, c2 | some (c1.interfaces & c2.interfaces) ->c1 in c2.eq
    }
```

Figure 2: Theorems from [12]

Finally, the invariant *Aggregation* says what it means to be an aggregate, and imposes additional rules on aggregated components. First, no component can be an aggregate of itself, directly or indirectly. Second, if *outer* aggregates *inner*, then they share an interface, and *outer* has an interface that gives the same query results, for any identifier, as every interface of *inner* bar its special *first* interface. Concretely, this holds because the inner component delegates all of its queries to the outer component, except for those on its *first* interface.

5 ANALYZING THE MODEL

Given our formal model, we can pose questions about it to the Alloy Analyzer. The model can act as an oracle, answering questions about COM (only correctly, of course, if the model is faithful).

We discuss two kinds of question. First, we check the theorems given in our earlier study. These expose some quite subtle consequences of the design of COM. Second, we pose a variety of questions that would more likely be asked by someone unfamiliar with COM.

The Alloy Analyzer is not a decision procedure. Every analysis is performed within a *scope* that limits the number of atoms in each domain. All the counterexamples that we found during our investigation required no more than a scope of 3 – that is, 3 components, 3 interfaces and 3 identifiers – and most in fact required only a scope of 2. In theory, we are not entitled to draw any conclusion when the analyzer fails to find a counterexample to a theorem. In practice, however, as the scope increases, the existence of a counterexample becomes increasingly unlikely. So we usually interpret the analyzer's responses to the question 'does this theorem hold?' as *no* and *probably*, rather than *no* and *maybe*.

```
Analyzing NewRuleImpliesReflexivity ...
Scopes: Interface(2), IID(2), Component(2)
Conversion time: 0 seconds
Solver time: 0 seconds
Counterexample found:
Domains:
 Component = {C1}
 IID = {II0,II1}
 Interface = {In0,In1}
Sets:
 LegalComponent = {}
 LegalInterface = {In1}
Relations:
 aggregates = {}
 ciids = {C1 -> {II0,II1}}
 eq = {C1 -> {C1}}
 first = {C1 -> In1}
 identity = {C1 -> In1}
 iids = {In0 -> {II0,II1}, In1 -> {II0,II1}}
 iids_known = {In0 -> {II0,II1}, In1 -> {II1}}
 interfaces = {C1 -> {In0,In1}}
 qi = {
  In0: {II0 -> In0, II1 -> In1}
  In1: {II1 -> In1}}
 reaches = {In0 -> {In0,In1}, In1 -> {In1}}
Skolem constants:
 Unknown = II1
 i = In1
```

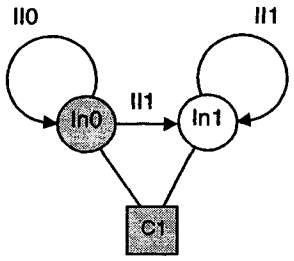

Figure 3: Counterexample to assertion of Section 5.2
The Alloy Analyzer's textual output lists each domain, and its value as a set of atoms (whose names are generated from the domain names), and then the set and relation variables with their values. The Skolem constants give witnesses for quantified variables: for example, i is the variable in the Reflexivity invariant and its value shows which interface violates the Reflexivity rule. We show some of the instance graphically, with components as boxes, interfaces as circles, and the results of queries as arrows. Shaded elements are not legal.

5.1 Some Valid Theorems

The theorems of Figure 2 are taken from our earlier paper [16]. The Alloy Analyzer finds no counterexamples to these in a scope of 4. Ignoring all the constraints, this corresponds to a space of about 10^{60} configurations. It is hard to estimate how many legal configurations there are in that scope.

```
//show me doubly nested aggregation
cond DoubleAggregation {some c | some c.aggregates.aggregates}
// show me sharing without any aggregation
cond SharingWithoutAggregation {
some c, d | (some c.interfaces & d.interfaces) && no
Component.aggregates
  }
// show me a legal component that aggregates two others
cond AggregateTwo {some c: LegalComponent | not sole c.aggregates}
// show me an interface that knows more than its own identifiers
cond KnowsMore {some i | some i.iids_known - i.iids}
// show me a component whose identity interface is not its first
cond FirstNotIdentity {some c | c.identity != c.first}
// are all interfaces of a legal component reachable from one another?
assert ComponentKnows {
  all c: LegalComponent | all i, j: c.interfaces | i in j.reaches
  }
// are all interfaces reachable from a legal interface legal?
assert ReachesLegal {all i: LegalInterface | i.reaches in LegalInterface}
// if interface i reaches interface j, does j reach i?
assert ReachesSym {all i, j: LegalInterface | i in j.reaches -> j in i.reaches}
```

Figure 4: A variety of simple queries
All conditions are consistent, all assertions are invalid

- *Theorem 1* says that for any legal component, the identifiers known to any interface of that component are all the identifiers of the component. In other words, every interface identifier of a component is accessible from every interface of that component.
- *Theorem 2* says that if a component *outer* aggregates a legal component *inner*, then *inner*'s identifiers belong also to *outer*. In other words, hiding legal components is generally not permitted: the standard rules must therefore be weakened for inner components.
- *Theorem 3* says that if *outer* aggregates *inner*, the two components are equal and cannot be distinguished. It could have been written more succinctly as

  assert Theorem 3 {all i | i.aggregates in i.eq}

- *Theorem 4a* extends *Theorem 2* to any case in which two components share an interface and one is legal. *Theorem 4b* extends *Theorem 3* similarly, but does not require legality. These two theorems demonstrate that the underlying problem with COM is deeper than it might seem, and that a different notion of aggregation may not help.

5.2 An Invalid Theorem

In our earlier work [16], we suggested that the traditional COM rules, captured in our model by the *Reflexivity*, *Symmetry* and *Transitivity* invariants, along with other axioms of COM, could be replaced by a single rule:

```
inv NewRule {
  all c: LegalComponent | all i: c.interfaces | c.iids in
i.iids_known
  }
```

```
Analyzing DoubleAggregation ...
Scopes: Interface(3), IID(3), Component(3)
Conversion time: 0 seconds
Solver time: 0 seconds
Instance found:
Domains:
 Component = {C0,C1,C2}
 IID = {II0,II1,II2}
 Interface = {In2}
Sets:
 LegalComponent = {C2}
 LegalInterface = {In2}
Relations:
 aggregates = {C0 -> {C1}, C2 -> {C0,C1}}
 ciids = {C0 -> {II2}, C1 -> {II2}, C2 -> {II2}}
 eq = {C0 -> {C0,C1,C2}, C1 -> {C0,C1,C2}, C2 -> {C0,C1,C2}}
 first = {C0 -> In2, C1 -> In2, C2 -> In2}
 identity = {C0 -> In2, C1 -> In2, C2 -> In2}
 iids = {In2 -> {II2}}
 iids_known = {In2 -> {II2}}
 interfaces = {C0 -> {In2}, C1 -> {In2}, C2 -> {In2}}
 qi = {
  In2: {II2 -> In2}}
 reaches = {In2 -> {In2}}
Skolem constants:
 $54 = C1
 Unknown = II2
 c = C2
```

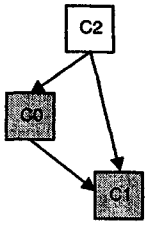

Figure 5: Instance generated for DoubleAggregation (Fig. 4)

This rule states that every interface of a legal component knows all the identifiers the component exports.

This rule follows trivially from Theorem 1, so we know that it follows from the other rules. We wondered whether it might in fact be equivalent to them. To check this, we turned all the rules – the three old ones and this new one – into conditions, and made assertions of the form

```
assert NewRuleImpliesReflexivity {
  NewRule -> Reflexivity
  }
```

To our surprise, this turned out to be false: the new rule is weaker than the old rules. The Alloy Analyzer generated the counterexample shown in Figure 3. The difference hinges on the tricky relationship between legal components and legal interfaces. The counterexample involves a component that is not legal, so our new rule does not apply. The old rules, however, apply to all legal interfaces, whether or not they belong to legal components.

In fact, one might have suspected a more trivial problem. The invariant *LegalComponents* does not define components to be legal if their interfaces are legal; it only says that a legal component must have legal interfaces. Setting *LegalComponent* to the empty set cannot rule out a configuration, since it makes the invariant vacuously true.

We might therefore change our model to say that a component all of whose interfaces are legal must be deemed legal itself. Our counterexample, however, shows that this will not solve this par-

```
Analyzing Transitivity-Same ...
Scopes: Interface(3), IID(3), Component(3)
Conversion time: 1 seconds
Solver time: 2 seconds
Counterexample found:
Domains:
 Component = {C1}
 IID = {II0,II1,II2}
 Interface = {In0,In1,In2}
Sets:
 LegalComponent = {}
 LegalInterface = {In0,In1}
Relations:
 aggregates = {}
 ciids = {C1 -> {II0,II1,II2}}
 eq = {C1 -> {C1}}
 first = {C1 -> In1}
 identity = {C1 -> In1}
 iids = {In0 -> {II2}, In1 -> {II1,II2}, In2 -> {II0,II1,II2}}
 iids_known = {In0 -> {II0,II1,II2}, In1 -> {II1,II2}, In2 -> {II0,II1,II2}}
 interfaces = {C1 -> {In0,In1,In2}}
 qi = {
  In0: {II0 -> In2, II1 -> In1, II2 -> In2}
  In1: {II1 -> In1, II2 -> In0}
  In2: {II0 -> In2, II1 -> In1, II2 -> In0}}
 reaches = {In0 -> {In1,In2}, In1 -> {In0,In1}, In2 -> {In0,In1,In2}}
Skolem constants:
 Unknown = II1
 i = In1
 j = In0
```

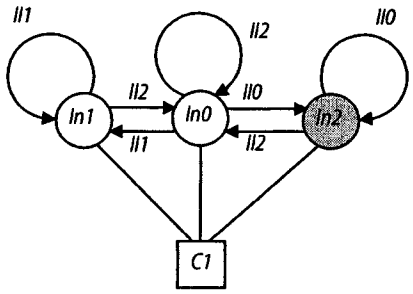

Figure 6: Counterexample to assertion of Section 6

ticular problem, since one of the interfaces of the component is legal and the other is not.

5.3 Some Simpler Questions

The analyses we have discussed so far are the kind that we typically perform after a good deal of work on a model. In the early stages of development, or when facing the model for the first time, it is very helpful to pose a variety of simpler queries.

Some such queries are shown in Figure 4. They fall into two categories. *Assertions*, as we have seen already, are putative theorems that are analyzed for counterexamples: that is, the analyzer searches for an instance of the formula's negation. All three theorems given are invalid, the last for a subtle reason discussed below.

Conditions are analyzed for consistency: the analyzer searches for an instance of the formula itself. So executing the condition *DoubleAggregation* for example asks the analyzer to find a configuration in which there are at least two levels of aggregation. As often happens, the instance produced (Figure 5) is a slight surprise: it shows that it is possible for the outermost component and its aggregated component to aggregate a third component together.

6 SIMPLIFICATION BY ANALYSIS

The primary contribution of analysis to our model was not so much in what was included but rather what was omitted. We started this case study by transcribing the second author's earlier Z model into Alloy; the result appears in the Appendix and is almost identical to the original, bar a few simplifications Alloy syntax allows. We then simplified the model drastically, using the Alloy Analyzer to check the soundness of every step. The final model is roughly half the size of the original Z model.

Incidentally, the analyzer caught two typographical errors that slipped through copy editing into the journal version of the Z model [16], eg, the substitution of $=$ for $\neq$. These errors were not present in an earlier conference version of the paper. Perhaps the typographic nature of Z exacerbates such problems, and suggests another reason to prefer ASCII notations.

It is hard in a paper to communicate the feel of using the Alloy Analyzer in fast-cycle interactive design. As our performance figures indicate, the analyzer finds instances in seconds, so it can be used interactively, adjusting the model and analyzing it in small, incremental steps. Moreover, with an automatic checker at hand, one tends to be more daring, experimenting with more radical changes, in the knowledge that their consequences can be immediately checked. In this section, we explain informally what kinds of changes we made and how the analyzer supported them.

We started by noticing that interface types, although discussed in the earlier paper and present in the model, appeared to play no role in the theorems of interest. So we eliminated them from the model and ran the theorems again. The theorems failed. By studying the counterexamples, we realized that the invariants about types had one important implication – that every interface has at least one identifier – that was now missing. So we added this constraint (in the declaration of *iids*), and moved on.

We then turned to the special interface identifier called *IID_Unknown* in the original model. We wondered whether this notion was really necessary, since it didn't appear explicitly in any theorem. In the discussion of the original Z model, the informal text referred to the existentially quantified variable *o* in the *Aggregation* invariant as being the *IUnknown* interface. Was this a formal claim, or an assertion about how aggregation is implemented in practice?

We formulated and checked a theorem to find out, and discovered that such a relationship is not implied. Emboldened by this, we took the unknown interface and its special identifier out of the state declarations, and made the identifier local to the *Identity* invariant. We tried a different version of the invariant in which the identifier *Unknown* can be different for each component – that is, we swapped the two outermost quantifiers. The theorems involving equality broke; a global identifier for the identity interface is necessary.

The most substantial simplification was to the notion of transitivity. The original model gave the following invariant

```
inv Transitivity {
 all a, b, c : LegalInterfaces, iidA, iidB, iidC |
  iidA in a.IIDsOfInterface &&
  iidB.QI[a] = b &&
  iidC.QI[b] = c
  -> some iidC.QI[a]
 }
```

and its accompanying text explained that '...informally, if *QueryInterface* can get you from here to there and there to somewhere else, it can get you from here to somewhere else'.

It seemed odd that such a simple intuition should require such a complex invariant. So we postulated, in a rather cavalier fashion that we could write instead

```
inv Transitivity {
  all i, j, k | j in i.reaches && k in j.reaches -> k in i.reaches
  }
```

and define *reaches* (as in Figure 1) by saying that an interface *i* reaches an interface *j* if there is a query of *i* with some identifier that yields *j*. To check this, we turned both old and new versions of the invariant into conditions, and checked the assertion

```
assert {Transitivity-Old <-> Transitivity-New}
```

It turns out that both directions of the implication are false. The informal explanation of the original invariant did not explain that the notion of getting from here to there is rather subtle, and depends on what you mean by 'getting', by 'here' and by 'there'! The original invariant does not in fact require that *c* be reachable from *a*, only that some interface be reachable from *a* with the identifier that *b* used to obtain *c*.

This motivated the definition of the relation *iids_known*, and the new formulation of transitivity:

```
inv Transitivity {
  all i, j : LegalInterface |
    j in i.reaches -> j.iids_known in i.iids_known
  }
```

which says that if you can reach *j* from *i*, then *i* knows about (that is, will yield an interface for) any identifier that *j* knows about. We performed a similar comparison of this new version and the original one, and discovered that it is in fact stronger.

The counterexample in Figure 6 shows why. The new invariant does not require the identifiers of *j* to yield legal interfaces; but the original does by the quantification bound of *c*.

We made a number of other uninteresting changes, including renamings. The Alloy Analyzer's type checker caught many slips on the way.

7 PERFORMANCE

For most runs of the Alloy Analyzer in the course of this study (perhaps a hundred or so in total), an instance was found. This never took more than about 10 seconds. For the problems of Figure 1, all take less than 4 seconds in a scope of 3. The more complicated transitivity check of section 6 took 7 seconds. Moreover, in a scope of 2, the analyzer found an instance for most of these, and if it did not, completed its search in a couple of seconds. So we would usually set the scope to 2, and increase it to 3 if no instance was found.

Checking valid theorems is computationally harder, since the analyzer must exhaust the space within the specified scope. Nevertheless, the performance is still good. Table 1 shows the timings for the theorems of Figure 2 for various scopes. The top line shows the number of bits that would be required to encode a single configuration, so the last entry says that there are roughly 2^{195}, or 10^{60}, cases to consider in a scope of 5. Because the Alloy Analyzer is based on a SAT solver that uses heuristics, its performance is hard to predict, so some theorems take much longer than others. There seems to be a scope for each problem at which the analysis falls into the intractability tar-pit, but fortunately this scope is large enough to give us some confidence that the theorem at stake is valid.

	scope of 2	scope of 3	scope of 4	scope of 5
Space (bits)	36	69	120	195
Theorem 1	2s	3s	20s	150s
Theorem 2	2s	3s	40s	609s
Theorem 3	2s	2s	5s	11s
Theorem 4a	2s	2s	21s	97s
Theorem 4b	2s	2s	3s	5s

Table 1: Analysis times for checking theorems

All analyses were performed on a Pentium II with a 233MHz processor and 192MB of memory. Version 1.1 of the Alloy Analyzer was used, with symmetry reduction turned on.

8 RELATED WORK

Formalizing software architectures is not a new idea. Abowd et al. [1] gave a formal semantics to informal architectural diagram using Z, and Garlan et al. [6] showed how such models enabled efficient development of tools for architectural design. Recently, a number of 'architectural description languages' have been developed, some of which (such as AML [18] and Darwin [12]) are very similar to Alloy and could be translated into it straightforwardly.

Although there are now many formalizations of architecture, very few have exploited mechanical analysis. Luckham and Vera use posets [11] to describe event orderings, and thus address an aspect of architecture that is complementary to the structural aspect we address here. More closely related is Le Métayer's analysis of evolving configurations graph grammars [10]. This framework is more specialized than ours, and while admitting proof, does not allow a full range of structural properties to be analyzed. Inverardi and Wolf [7] formalized architectures with chemical abstract machines, which, like algebras for specifying datatypes, are very expressive but likely to be intractable and hard to automate.

An analysis of the High Level Architecture (HLA) framework was performed using Damon's Ladybug tool [4]. The analysis was used to find bugs in HLA itself; in this case study we focused instead on the refinement of the model. Ladybug, like its predecessor Nitpick, has an input language based on Tarski's relational calculus, which does not include quantifiers, indexed relations or object-model-style declarations. An invariant such as *Identity* would have been quite tricky to express.

There are many general purpose specification languages, such as Z and VDM, that are suitable for describing structural properties of software. These are more powerful than Alloy, but are not currently supported by automatic semantic analyses. What distinguishes Alloy's analysis is the combination of concreteness (in the instances generated) and abstraction (in the implicit nature of the specification); existing tools tend to provide animation only by requiring that the specification be more program-like.

Model checkers provide the kind of deep semantic analysis that our analyzer provides, but they are designed for state transition systems, and not for analyzing logical consequence. Their input languages, moreover, tend not to support the description of structural properties, usually providing only low-level datatypes.

9 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a case study using Alloy and the Alloy Analyzer to refine a model of interface negotation in COM.

The use of Alloy and its analyzer allowed us to experiment with a confidence that manual manipulation would not have engendered, but with a nimbleness incompatible with more heavyweight tools, such as theorem provers. We were continually astonished at how hard it turned out to be to preserve the precise meaning of the COM specification as we restructured it, and at how effective the tool was in revealing subtle changes in meaning.

Our model illustrates the power of simple first-order logic. Conjunction allows properties to be developed and expressed incrementally, but can result in unexpected non-local interactions. A tool such as the Alloy Analyzer helps mitigate the risks of logical specification, while retaining its benefits. The continual, interactive, fully automated feedback produces an emotional reward – similar to that provided by running a program – which, while easy to dismiss, may be key to making formal modeling and analysis attractive to practitioners.

Our experience with the Alloy Analyzer suggests a number of ways in which it might be improved. Graphical display of instances, implemented since our case study, would have been a major help. It might also be nice to have a configuration management system that retains previous versions of models with their analysis results, and allows the user to walk around a tree of modified models. The deficiences of the Alloy language – principally lack of arithmetic, a flexible structuring mechanism, and a sequence datatype – were not impediments in this particular study.

Since Alloy is a rather pure language with very little domain-specific baggage, it might make a good intermediate language into which architectural description languages are translated. This is perhaps one route by which our tool and approach could be integrated into standard architectural practice.

ACKNOWLEDGMENTS

We dedicate this paper to John Gannon, who suggested to us that we collaborate to explore the use of Alloy to analyze our formal model of COM. We miss John for his extraordinary kindness, encouragement and insight.

This research was funded by the National Science Foundation (under grants CCR-9523972 and CCR-9804078), by the MIT Center for Innovation in Product Development (under NSF Cooperative Agreement Number EEC-9529140), and by an endowment from Douglas T. Ross.

The Alloy Analyzer is freely available for download at http://sdg.lcs.mit.edu/alloy. The Alloy model, along with a version of the paper in a more readable format, are available at http://sdg.lcs.mit.edu/~dnj/publications.

APPENDIX: ORIGINAL MODEL

The following model is the original translation of the Z model of [16] into Alloy.

```
model COM {
 domain {Interface, InterfaceType, IID, Component}
 state {
  IUnknown : fixed InterfaceType!
  IID_IUnknown : fixed IID!
  IIDOfInterfaceType : InterfaceType! -> IID!
  InterfaceTypesOf : Interface -> InterfaceType
  IIDsOfInterface : Interface -> IID
  QI [Interface] : IID -> Interface?

  interfaces : Component -> Interface
  iids : Component -> IID
  firstInterface, identity : Component -> Interface!

  eq : Component -> Component
  LegalInterfaces : Interface
  LegalComponents : Component

  Aggregates : Component -> Component
 }

 inv BasicRels {
  IUnknown.IIDOfInterfaceType = IID_IUnknown
  all i | IUnknown in i.InterfaceTypesOf
  all i | i.IIDsOfInterface =
      i.InterfaceTypesOf.IIDOfInterfaceType
 }

 inv ComponentProps {
  all c | c.firstInterface in c.interfaces
  all c, d | all i : c.interfaces | d.QI[i] in c.interfaces
  all c | c.iids = c.interfaces.IIDsOfInterface
 }

 inv IdentityAxiom {
  all x, i | i in x.interfaces -> IID_IUnknown.QI[i] = x.identity
 }

assert InterfaceEquality {
  all x | x.identity in x.interfaces
 }

 inv ComponentEquality {
  all x, y | y in x.eq <-> x.identity = y.identity
 }

 inv Legality {
  all i: LegalInterfaces, d: IID |
      some d.QI[i] -> d in d.QI[i].IIDsOfInterface
 }

 inv Reflexivity {
  all a: LegalInterfaces, iidA | iidA in a.IIDsOfInterface ->
       some iidA.QI[a]
 }

 inv Symmetry {
  all a, b : LegalInterfaces , iidA, iidB |
   iidA in a.IIDsOfInterface && iidB.QI[a] = b ->
      some iidA.QI[b]
 }

inv Transitivity {
 all a, b, c : LegalInterfaces, iidA, iidB, iidC |
   iidA in a.IIDsOfInterface &&
   iidB.QI[a] = b &&
```

```
      iidC.QI[b] = c
      -> some iidC.QI[a]
    }

    inv LegalComponent {
     all c: LegalComponents | c.interfaces in LegalInterfaces
     }

    inv Aggregation {
     all outer | all inner : outer.Aggregates |
      some (inner.interfaces & outer.interfaces)
      && (some o: outer.interfaces |
            all i: inner.interfaces - inner.firstInterface |
               all d | d.QI[i] = d.QI[o] ))
    }

  assert Theorem1 {
    all c: LegalComponents, iidA | all i: c.interfaces |
      some iidA.QI[i] <-> iidA in c.iids
    }

  assert Theorem2 {
    all outer | all inner : outer.Aggregates |
      inner in LegalComponents -> inner.iids in outer.iids
    }

  assert Theorem3 {
    all outer | all inner : outer.Aggregates |
      inner.identity = outer.identity
    }

  assert Theorem4 {
    all inner, outer |
      some (inner.interfaces & outer.interfaces) ->
       ((inner in LegalComponents -> inner.iids in outer.iids)
       && inner.identity = outer.identity)
    }
}
```

REFERENCES

1. Abowd, G.D., R. Allen and D. Garlan, "Formalizing style to understand descriptions of software architecture," *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, Oct. 1995, pp. 319–364.
2. Allan, R.J., D. Garlan and J. Ivers, "Formal modeling and analysis of the HLA component integration standard," *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 1998).*
3. Box, D., *Essential COM*, Addison-Wesley, 1998.
4. Damon, C.A., R. Melton, R.J. Allen, E. Bigelow, J. M. Ivers and D. Garlan. *Formalizing a Specification for Analysis: The HLA Ownership Properties.* Technical Report CMU-CS-99-126, School of Computer Science, Carnegie Mellon University, 1999.
5. Di Nitto, E., and D. Rosenblum, "Exploiting Architecture Description Languages to Specify Architectural Styles Induced by Middleware Infrastructures," *Proc. 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 13-22.
6. Garlan, D., R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.
7. Inverardi, P. and A.L. Wolf, "Formal specification and analysis of software architectures using the chemical abstract machine model," *IEEE Transactions on Software Engineering, SE-21,4*, April 1995, pp. 373–386.
8. Jackson, D., *Alloy: A lightweight object modelling notation*, Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.
9. Jackson, D., I. Schechter and I. Shlyakhter, "Alcoa: the Alloy Constraint Analyzer," *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000.
10. Le Métayer, D., "Software architecture styles as graph grammars," *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 15-23, 1996.
11. Luckham, D.C. and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering, Vol 21*, No 9, pp.717-734. Sep. 1995.
12. Magee, J., and J. Kramer, "Dynamic Structure in Software Architectures," *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes, Vol. 21, No. 6, ACM Press, October 1996, pp. 3–14.
13. Microsoft Corporation, *The Component Object Model Specification*, version 0.9, October 24, 1995, available at: www.microsoft.com/com/resources/comdocs.asp.
14. Monroe, R.T. *Capturing Software Architecture Design Expertise With Armani.* Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, October 1998.
15. Rumbaugh, J., I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley Object Technology Series, 1998.
16. Sullivan, K.J., M. Marchukov and D. Socha, "Analysis of a conflict between interface negotiation and aggregation in Microsoft's component object model," *IEEE Transactions on Software Engineering*, July/August, 1999.
17. Spivey, J.M., *Understanding Z: A Specification Notation and its Formal Semantics*, Cambridge Tracts in Theoretical Computer Science, 1988.
18. Wile, D., "AML: An Architecture Meta Language," *Proceedings 14th International Conference on Automated Software Engineering*, Cocoa Beach, FL, October 1999, pp. 183–190.