Analysis of Architectures using Constraint–Based Types

John Peterson and Martin Sulzmann
Department of Computer Science
Yale University
New Haven, CT 06520

October 30, 1998

Abstract

Constraint–based types provide a formal foundation for many varieties of *architectural analysis*. A software architecture, typically defined using an Architecture Definition Language (ADL), exposes the overall structure of a complex system in a way that allows validation and computation of overall system properties. We argue that the *HM(X)* type system can be used to implement many analyses in a simple and elegant way. HM(X) uses an extensible *constraint subsystem*: given a user-defined syntax and semantics (via a constraint solver) for an analysis domain, the HM(X) type system infers system-wide properties from the architectural definition. The core inference engine is thus extended to different forms of analysis by using different constraint solvers. We demonstrate our ideas by constructing a unit analysis built on top of HM(X). We also show how HM(X) provides a basis for the analysis of dynamic or higher-order architectural definitions.

1 Introduction

A software architecture exposes the basic structure of a system for analysis. The architectural description defines the system components and their properties as well as the interconnections between the components. Analysis infers overall system properties from component properties and their interconnections; validation is used to ensure that the overall system meets criteria for safety or correctness. We assume that the properties of individual component are known, either declared as part of the component's specification or, perhaps, inferred from the definition of a component. But what sort of analysis should be incorporated into an Architecture Definition Language (ADL)? What properties should be computed or verified? We contend that the role of an ADL is not to provide a fixed set of analyses but rather to enable the construction of analysis tools specific to the domain of the system. That is, rather than constructing highly specialized, domain specific ADLs, we should instead make ADLs adaptable to a wide variety of domains by providing tools to easily define new architectural analysis techniques.

Our constraint-based analysis system provides a framework in which many different kinds of analysis can be expressed naturally and succinctly. This enables the addition of new analysis domains to the architectural analyzer with a minimum of effort. We also show that these same techniques may be applied to *dynamic architectures*, allowing analysis to be applied to families of architectures as well as a specific architecture.

In this paper we demonstrate a *unit analyzer* built using our methodology. We have studied other analysis domains, including security analysis and propagation delay analysis; these domains are no more difficult to construct than unit analysis so we focus on this one domain. Due to space limitations we refer to [PS98] for the discussion of these other

domains. Our analysis is based on the HM(X) type system [OSW98], a polymorphic type checker based on the Hindley-Milner type checker. HM(X) extends this widely-used type system using user-defined constraint subsystems. Our approach has a number of significant advantages:

- Polymorphic types describe both values within an analysis domain and the relationships between such values. This is a simple and natural way of expressing the propagation of information during analysis.
- We reuse a single inference engine for many different forms of analysis.
- Type systems/inference have a strong mathematical foundation, allowing formal proofs of analysis properties.
- Our type system can handle both static and dynamic architectures. That is, some system components may remain unspecified during analysis.

2 Analyzing a Software Architecture

For concreteness we use the ACME architecture definition language; other languages capable of describing the properties and interconnection of components would be equally suitable. In particular, UML could also be extended to include analyzers build within our framework. In the interest of simplicity, we use only the following ACME features:

- Components: the basic unit of software described by the architecture.
- Ports: specific interfaces to a component.
- Properties: attributes attached to either ports or components.
- Attachments: connections between ports.

Each analysis is defined by a domain-specific constraint solver, defining an inference domain to the HM(X) type checker, and component property declarations, describing each component from the perspective of the chosen analysis domain. Constraint solvers for the HM(X) type checker are not difficult to write but, however, it is beyond the scope of this paper to describe the programming details involved in this task. There is much literature regarding constraint solving; we only mention [BS94]. The HM(X) constraint solver is generally a straightforward translation of existing constraint solvers. The primary difference between HM(X) constraint solvers and more conventional ones is the presence of type variables within HM(X) constraints.

Each domain is assigned a unique property name; this property associates a component port with its properties in that domain. The syntax of the property is recognized by a domain-specific parser; this allows properties to be expressed in a way natural to the underlying domain. These properties may contain type variables which are scoped over an entire component. This allows type dependencies between various ports in a component to be expressed using a common type variable name.

Ports need not have type definition for every form of analysis; missing properties are defaulted to the most general type: a fresh type variable. An analysis that does not apply to some ports of a component thus will not generate type mismatch errors for those ports omitting the property declaration for that analysis.

For example, this ADL declaration (using ACME[GMW97] syntax)

```
Component RemoteCopy = {
  Port input = {
    property valueType = a};
  Port output = {
    property valueType = a};}
```

defines a *polymorphic* connector that copies a value of any type from input to output. The type `a` is a *type variable* representing an unknown type. The use of the same type variable in two different ports constrains each use of `RemoteCopy` to have the same type at the `input` and `output` ports. We (arbitrarily) choose the property `valueType` to denote the type of the copied value. Different instances of this component may be used to copy different types of values.

Type variables may be subject to constraints. Constraints arise naturally in many real-world problem domains. For example, performance constraints (time, space, number of messages), numeric range checking (array bounds), This kind of polymorphism is often referred to as *constrained* polymorphism.

We now turn to *unit analysis*. Units such as meters, seconds, grams, or meters/second give meaning to numeric quantities. In a well-formed program, these units must be used consistently: passing a value measured in inches to a component that expects meters is a error that we detect via unit analysis over the ADL. Unit analysis is of particular interest to us since many operations are naturally polymorphic with respect to units. For example, the addition operation can be used for values of any unit but the units of the input values must be consistent. The following component exploits unit polymorphism:

```
Component RateMonitor =
  { Port monitoredValue = { Property Unit = a};
    Port timeClock = { Property Unit = s};
    Port rate = { Property Unit = a/s}}
```

The component RateMonitor monitors a value and, using an external clock, and calculates the rate of change in the value being monitored. The unit domain is declared by the property name Unit. We use *constrained* types to express the relationship between the units of the `rate` output and the inputs, with the output having units that are the ratio of the input units. Note the unit of the `rate` port is an expression over multiple type variables. Our system allows units to be expressed by an arbitrary set of base units and expressions the operators `*` and `/`. A larger example is found in the next section in which we additionally introduce the concept of a dynamic architecture.

A description of constrained polymorphic type inference is beyond the scope of this paper. The interested reader is referred to [OSW98]. Users, however, need understand only how to express dependencies using type variables, not how these dependencies are exploited during the inference process.

3 Dynamic and Higher-Order Architectures

Architectural descriptions are not limited to complete, static systems in which all components and connections are fully specified. We also wish to capture *design patterns* at the architectural level. For example, we wish to express patterns such as the attachment of a temperature correction table to an arbitrary sensor. We thus turn to *higher-order* architectures: ones capable of defining a pattern for connecting a set of components, some of which are as yet unspecified.

What role can architectural analysis play when the underlying system is not fully specified? Instead of waiting for the architecture to be fully instantiated for analysis, we instead apply analysis directly to the parameterized architecture. The analysis characterizes the valid configurations of parameters to the architecture or defines a simplified relationship between attributes of the parameters and overall system attributes.

We introduce higher-order architectures by adding a simple parameterization construct to ACME, allowing the definition of *architectural abstractions*. The following defines an architecture which is parameterized over a set of components:

```
Architecture name (c1, ... , cn) =
  { body }
```

This construct is somewhat similar to architectural templates being developed for ACME. Within the scope of the named architecture, parameter components c1 through cn are unknown entities. This construct defines patterns to be elaborated by the ADL, yielding a fully instantiated architecture, or, by postponing elaboration until execution time, it implements dynamic architectures. For the purpose of our analysis it does not matter when the parameters are instantiated.

Abstractions such as the `Architecture` construct may be defined anywhere in the architectural system definition and may passed as arguments to other abstractions. The HM(X) type inferred for an abstraction defines a relationship between the types of the unknown arguments and the type of the resulting system. The inferred type of an abstraction expresses the relationship between the argument types the type of the resulting system.

Consider the following architectural definition, expressed in a slightly extended version of ACME:

```
// A 2-way broadcast preserves units from input to outputs
Component Broadcaster = {
  Port input   = {Property Unit = a};
  Port output1 = {Property Unit = a};
  Port output2 = {property Unit = a}}

// A higher order architecture to connect two components in parallel,
// using an arbiter to chose which output to use
Architecture parallelCompose(component1, component2, arbiter) = {
  Component broadcast = new Broadcaster;
  Attachments {
    parallelCompose.input to broadcast.input;
    parallelCompose.output to arbiter.output;
    broadcast.output1 to component1.input;
    broadcast.output2 to component2.input;
    arbiter.input1 to component1.output;
    arbiter.input2 to component2.output; }
```

A graphical representation of the component dependencies can be found in Figure 1. Parameter components component1, component2 and arbiter are emphasized. In the same figure, we visualize the data flow of the parameterized architecture parallleCompose. The three parameter components, component1, component2 and arbiter are passed as parameters to the parameterized architecture parallelCompose. The overall result is attached to the output port. What kind of properties do the three variable components have to fulfill? This can be observed by the type description inferred for parallelCompose:

```
parallelCompose =
 { Property Component = { component1 { Port input = { Property Unit = a};
                                       Port output = {Property Unit = b}};
                          component2 { Port input = { Property Unit = a};
                                       Port output = {Property Unit = c}};
                          arbiter { Port input1 = { Property Unit = b};
                                    Port input2 = { Property Unit = c};
                                    Port output = { Property Unit = d}}}}
```

Components component1 and component2 need to consist of at least an input and an output port. The arbiter component must have two input port s, named input1 and input2, and an output port. The inferred type shows that the two components must share a common unit for their inputs but that they may return different output units, so long as the arbiter component accepts as input these same types.

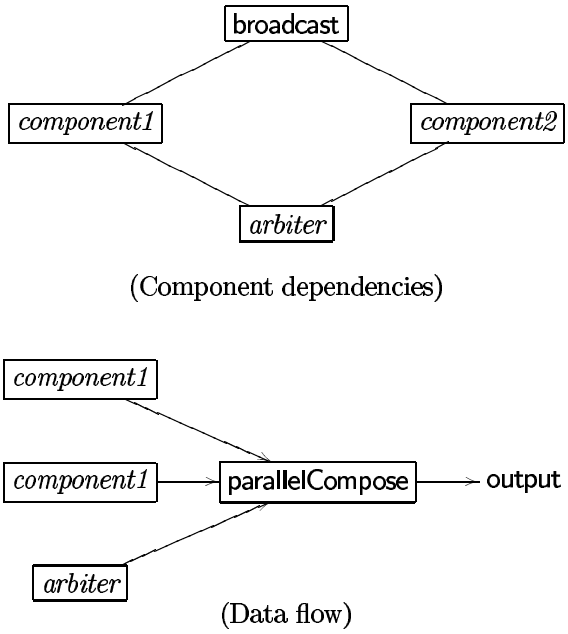

Figure 1: Dynammic architecture

4 Conclusions

We have demonstrated a general framework supporting many varieties of architectural analysis. Using the HM(X) polymorphic type system, as specialized by a simple domain-specific constraint solver, complex analysis or validation tasks can be applied to an architecture with relatively little effort. Since we do not rely on any assumptions about any underlying type system, our system operates solely at the architectural level and does not restrict the implementation of the architecture in any way. Novel aspects of our approach include:

- We are not committed to a specific domain or analysis. Our system is similar to other general analysis frameworks such as attribute grammars or predicate calculus.
- Our analysis works nicely with architectural abstractions, making it useful for higher-order or dynamic architectural specifications.
- The engine which drives the analysis, polymorphic typing with constraints, has been studied extensively and has many useful formal properties.
- Type signatures assigned to components by our system are usually quite descriptive and understandable to ordinary programmers, providing an effective way to describe and document component properties.

The unit analysis presented here is not trivial. There are already a couple of systems that deal solely with unit analysis [Hou83, WO91, Ken94] but our framework handles this analysis in a more general way, needing less than a page of specialized code to define unit analysis.

How difficult is it to add new types of analysis within our system? This depends on the complexity of the constraint system needed for the underlying domain. So far, each analysis we have implemented has used a very simple constraint solvers. For example, the constraint solver for unit analysis consists of a single function to convert unit types into a canonical form, about 15 lines of code written in Haskell. The surrounding HM(X) engine is much

larger but this code is shared by every analysis. As we implement other forms of analysis within this framework, we expect to gain further evidence of the power of our techniques and get a better feeling for the types of analysis naturally supported by the HM(X) type system.

Not all forms of analysis are well matched to the HM(X) type inference system. Algorithms that examine the system all at once rather than incrementally over subregions of the architecture are not as well suited to our approach since constraints cannot be simplified for subregions. For example, some graph problems such as network flow do not encode nicely as a constraint solver. HM(X) should not be considered as a replacement for existing analysis systems but rather as one tool among many.

Acknowledgments

This work is supported by NSF Grant CCR-9633390. We thank Paul Hudak for helpful discussions about the paper.

References

[BS94] F. Baader and J.H. Siekmann. Unification theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 41–125. Oxford University Press, Oxford, UK, 1994.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, November 1983.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.

[OSW98] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 1998. to appear.

[PS98] John Peterson and Martin Sulzmann. Analysis of Architectures using Constraint–Based Types. Research Report YALEU/DCS/RR-1157, Yale University, Department of Computer Science, 1998.

[WO91] Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In J. L. Lassez and G. D. Plotkin, editors, *Computational Logic: in honor of J. Alan Robinson*, pages 479–486. MIT Press, 1991.