Formal Approaches to Software Architecture

Jaroslav Keznikl
keznikl@d3s.mff.cuni.cz
Modeling and Analyzing Architectural Change with Alloy

Published at: SAC’10

Klaus Marius Hansen
(University of Copenhagen)

Mads Ingstrup
(Aarhus University)
Why should we be interested?

- Formalization of research ideas
- Position paper about modeling in Alloy
- My personal interest in architecture models
Goals

- Formal architectural model
- Model of architectural changes
- Verification of architectural properties after sequence of changes
- Searching for operations that produce an architecture with certain properties
- Automated, usable at runtime
Recall: Alloy

- Modeling language
- Sets, elements, relations, predicates in first-order logic on the sets and relations
- Alloy Analyzer tool
  - finding an instance conforming to a given specification and a predicate (run command)
  - finding a counterexample that satisfies a given specification and violate an assertion
  - bounded scope
Recall: Alloy Syntax Basics

sig A {}                          // A is a set
sig C extends A {}               // C ⊆ A
sig B { p: set A }               // p ⊆ B × P(A)
sig D { q: A->B }                // q ⊆ D × A × B

// fact { q = D->A->B }

fact { all a: A | some b: B | a in b.p }
pred test[a: A] { some b: B | a in b.p }
fact { all a: A | test[a] | ... }
Recall: Alloy Syntax Basics

```alloy
sig A {}
// A is a set
sig C extends A {}
// C ⊆ A
sig B { p: set A }
// p ⊆ B × P(A)
sig D { q: A->B }

fact { all a: A | some b: B | a in b.p }

pred test[a: A] { some b: B | a in b.p }

fact { all a: A | test[a] | ... }
```

- All elements of A are in the p relation with some element of B.
- The same as b in p.a.
Architectural configuration

Closely maps to an implementation in OSGi

```
Component
  \-- imports/exports
      \-- *

Module

Device
  \-- provides
      \-- *

Service
  \-- requires/provides
      \-- *

interface
  \-- requires/provides
      \-- *

components

binding
  \-- provided
      \-- required

Formal Approaches to Software Architecture
```
Modeling architectural configurations

- Structural model = sets and relations

```plaintext
sig Configuration {
  elements: set Element
}

abstract sig Element {}

sig Service extends Element {
  bindings: Service -> Interface,
  provides: set Interface,
  requires: set Interface,
  started: Bool
}
```

(bindings \subseteq Service \times Interface)
• Consistency invariants = facts on sets and relations

```
fact {
    all s : Service | some s.bindings implies
    s.started = True

    all s : Service | all s' : Service, i : Interface {
        s'->i in s.bindings implies ( 
        i in s'.provides and
        i in s.requires)
    }
}
```
Architectural style = predicate on a configuration

\[\text{sig}\ \text{RingService}\ \text{extends}\ \text{Service} \{\]
\[\text{next: lone RingService}\]
\[\}\{\]
\[\text{all } s : \text{Service} | s \text{ in next iff some } i : \text{Interface} |\]
\[s \rightarrow i \text{ in bindings}\]
\[\}\]

\[\text{pred}\ \text{ring_style}[c : \text{Configuration}] \{\]
\[\text{let}\ \text{ring}\_\text{services} = \text{RingService} \& c.\text{elements} \&\]
\[\text{started.}\text{True} \{\]
\[\text{all } rs : \text{ring}\_\text{services} | \text{ring}\_\text{services} \text{ in } rs.^{\text{next}}\]
\[\}\]

\]
Architectural change

- Architectural operations
  - starting/stopping of a device/service
  - (un)deployment of a component
  - (un)binding of an interface

- Architectural trace
  - Sequence of configurations where each step is caused by an architectural operation
  - Architectural script
Modeling architectural change

- Change = transition from one configuration to another
  - Configurations become **totally ordered**
- Mutable state is related to configurations
- Service example:
  - Mutable state: bindings, started

```csharp
sig Service extends Element {
    [...] bindings: Service->Interface->Configuration
    started: Bool->Configuration
}
```
Modeling architectural change

- Change = transition from one configuration to another
  - Configurations become **totally ordered**
- Mutable state is related to configurations
- Service example:
  - Mutable state: bindings, started

```plaintext
sig Service extends Element {
  [...]
  bindings: Service -> Interface -> Configuration
  started: Bool -> Configuration
}
```

bindings ⊆ Service × Service × Interface × Configuration
Modeling architectural operations

- Architectural operation = a predicate on configurations

```plaintext
pred undeploy_component[
    c,c’: Configuration, d: Device, cmp: Component
] {
    // Preconditions
    d.started.c = True
    cmp in d.components.c
    // Postconditions
    d.components.c’ = d.components.c – cmp
    // Invariant: all other local state remains unchanged
    deploy_undeploy_invariant[c, c’, d, cmp]
}
```
• All configuration changes allowed only through one of the operations

```java
fact {
    all c: Configuration, c': c.next {
        some d: Device | start_device[c, c', d] or
        ...
    }
}
```
Modeling architectural traces

- Architectural trace = sequence of operations

```plaintext
run example {
  some dev: Device, cmp: Component {
    let c0 = first, c1 = c0.next, c2 = c1.next {
      init_configuration[c0]
      start_device[c0, c1, dev]
      deploy_component[c1, c2, dev, cmp]
    }
  }
}
```
Analyzing Architectural Change
Architectural change verification

- **Input**: a configuration $C_{\text{initial}}$, an architectural script $O_1, \ldots, O_n$, a predicate $P_{\text{goal}}$

- **Output**: does the architectural script preserve the $P_{\text{goal}}$? (if not, find a counterexample)

  i.e., having $C_0 = C_{\text{initial}}$, $C_1 = C_0$.next, ..., $C_n = C_{n-1}$.next, does $O_1[C_0, C_1]$ and ... and $O_n[C_{n-1}, C_n]$ implies $P[C_n]$ hold?

- **Idea**: Employ the Alloy Analyzer, able to find a counterexample that satisfies a given specification and violates an assertion (the check command)
Style-related properties

```plaintext
assert script_preserves_ring_style {
  all d: Device, cmp: Component, rs: RingService {
    let c0 = first, c1 = c0.next, c2 = c1.next, c3 = c2.next { (
      init_configuration[c0] and
      start_device[c0, c1, d] and
      deploy_component[c1, c2, d, cmp] and
      start_service[c2, c3, d, cmp, rs]
    ) implies
      ring_style[c0] and ring_style[c3]
  }
}
```

General properties

```plaintext
assert no_device_started_and_stopped {
  init_configuration[first] implies
  all d: Device, c: first.nexts | not True + False in d.started.c
}
```

True, False are regular set elements
Architectural change planning

- **Input**: a configuration $C_{\text{initial}}$, a predicate $P_{\text{goal}}$
- **Output**: an architectural script $O_1, \ldots, O_n$, that leads from $C_{\text{initial}}$ into a configuration $C_{\text{goal}}$, where $P_{\text{goal}}[C_{\text{goal}}]$ is satisfied

**Idea**: Employ the Alloy Analyzer, able to find an instance conforming to a given specification and predicate (the $\text{run}$ command)

i.e., find $C_0 = C_{\text{initial}}$, $C_1$, $\ldots$, $C_{n-1}, C_n = C_{\text{goal}}$, such that $O_1[C_0; C_1]$ and $\ldots$ and $O_n[C_{n-1}; C_n]$ and $P[C_n]$ holds.
The operations have to be associated with configurations

abstract sig Operation {}

one sig None, Start_Device, ... extends Operation {}

sig Configuration {
    operation: Operation
}

pred start_device[c, c': Configuration, d: Device] {
    [...]  
    c'.operation = Start_Device
}
Architectural change planning example

- Alloy Analyzer finds a sequence of configurations satisfying the planning predicate

```alloy
pred planning_example[d: Device, cmp: Component] { 
  // Preconditions on first configuration
  d.started.first = False
  // Postconditions on last configuration
  d.components.last = cmp
  d.started.last = True
}
run planning_example
```
Alloy-based planning vs. AI (IPP) planning

- IPP: still much faster
- Alloy: automated, analyzable output, suitable syntax
Conclusion

- Alloy-based planning vs. AI (IPP) planning
  - IPP: still much faster
  - Alloy: automated, analyzable output, suitable syntax

- OSGi-based implementation
  - Alloy used at runtime for correctness checking of architectural changes
Thank You

Resources:
Flickr/André Mouraux/Amiens cathedral
Dynamic Architecture/David Fisher
Flickr/Rian Castillo/Hearst tower