An Architectural Pattern for Non-functional Dependability Requirements

Lihua Xu
Hadar Ziv
Debra Richardson
Thomas Alspaugh

Donald Bren School of Information and Computer Sciences,
University of California, Irvine

Outline

- Research Agenda
- Our approach
  - Extend the distinction between functional versus nonfunctional requirements
  - Propose an architectural pattern, model dependability requirements in software architectures directly and explicitly
- Example
- Conclusions

Research Agenda

Motivation:

- The intersection of three areas of research:
- Requirements Engineering:
  - Goal Refinement [Lamsweerde et al]
  - The NFR Framework [Mylopoulos et al]
    - *NFRs specified during Requirements Engineering are often verified after implementation*
- Software Architecture:
  - Original requirements not always visible, traceable
    - *Non Functional Requirements (NFRs) are especially underrepresented*
- Aspects:
  - Aspects have the potential to seamlessly model and integrate NFRs through architectures to implementations
    - *Need development methodology from NFRs through architectures to AOP solutions*
    - *Need corresponding analysis and testing of the artifacts of such methodology*

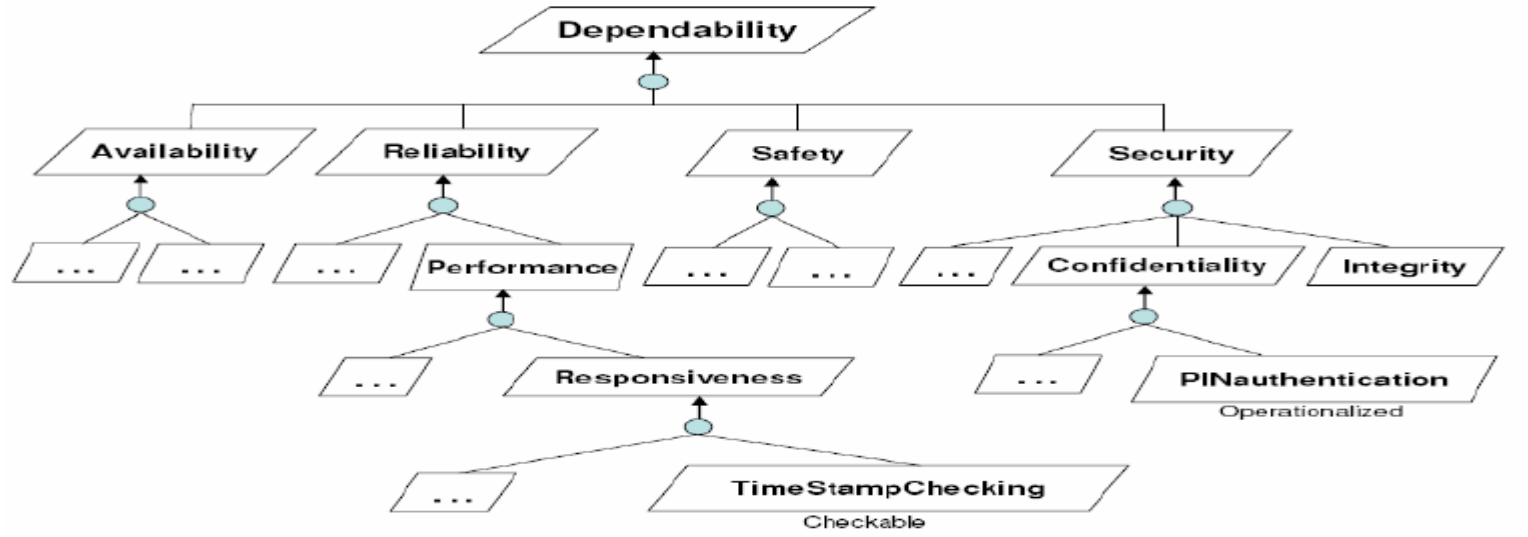
Additional Objectives:

- separation of cross-cutting functional and nonfunctional concerns at the architecture level
- architectural analysis against NFRs early in the software lifecycle
- establishing confidence of properly chosen architecture style and designed architecture before the architecture is implemented.

Our Approach

- Model NFRs in software architectures directly and explicitly
  - Rely on the “design decision” made for each NFR
- Three types of Requirements
  - Functional
  - Operationalizable Nonfunctional
  - Checkable Nonfunctional
- Types of Architectural Components
  - Core Components
  - Aspectual Components
  - Monitoring Components
- Connectors
  - XML Binder

Requirements Classification



NFRs:

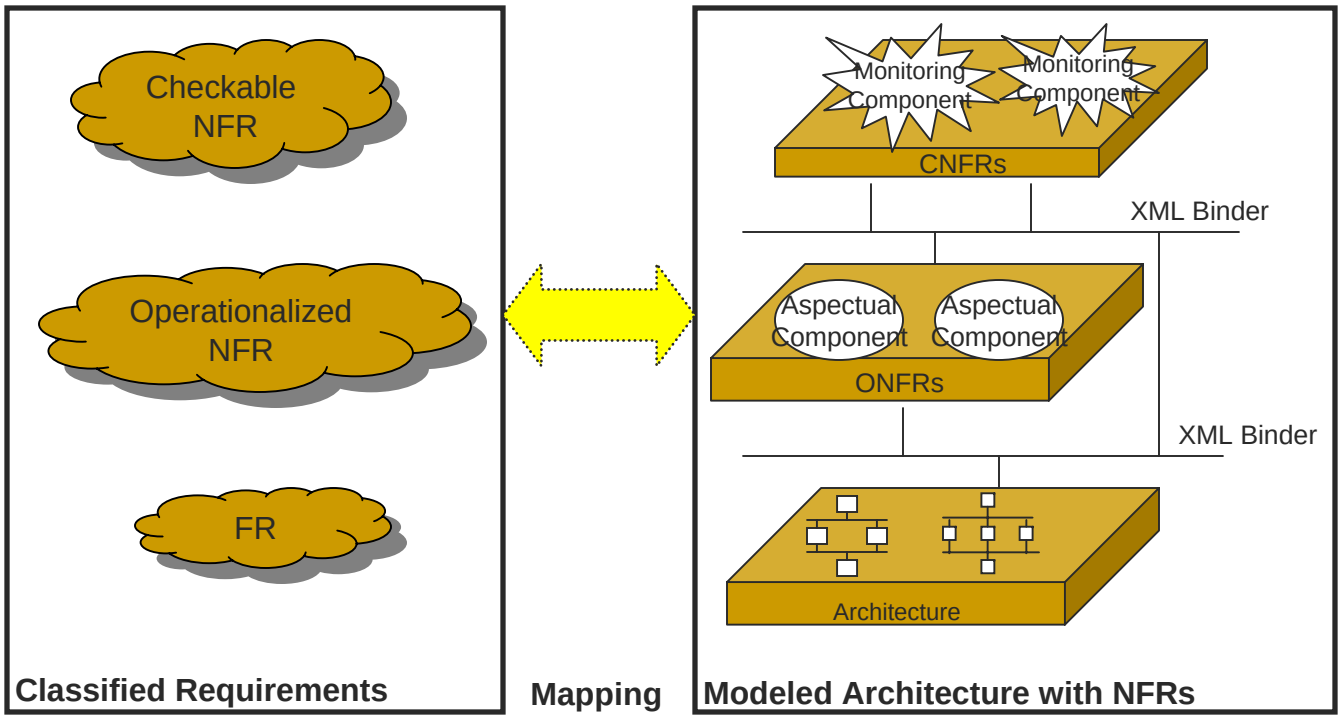
- *Operationalizable*:

  Upon decomposition to “design decision”, the chosen strategy can be realized by functional components in the software architecture

- *Checkable*:

  The chosen strategy is to monitor functional behavior to check and verify that desirable quality properties are met

Requirements & Architectures



XML Binder

```
Component aspect ConfidentialityInterceptor {
        PlayerIDProtection () {
            // the code for checking PlayerID goes here
        }
        …
}
```

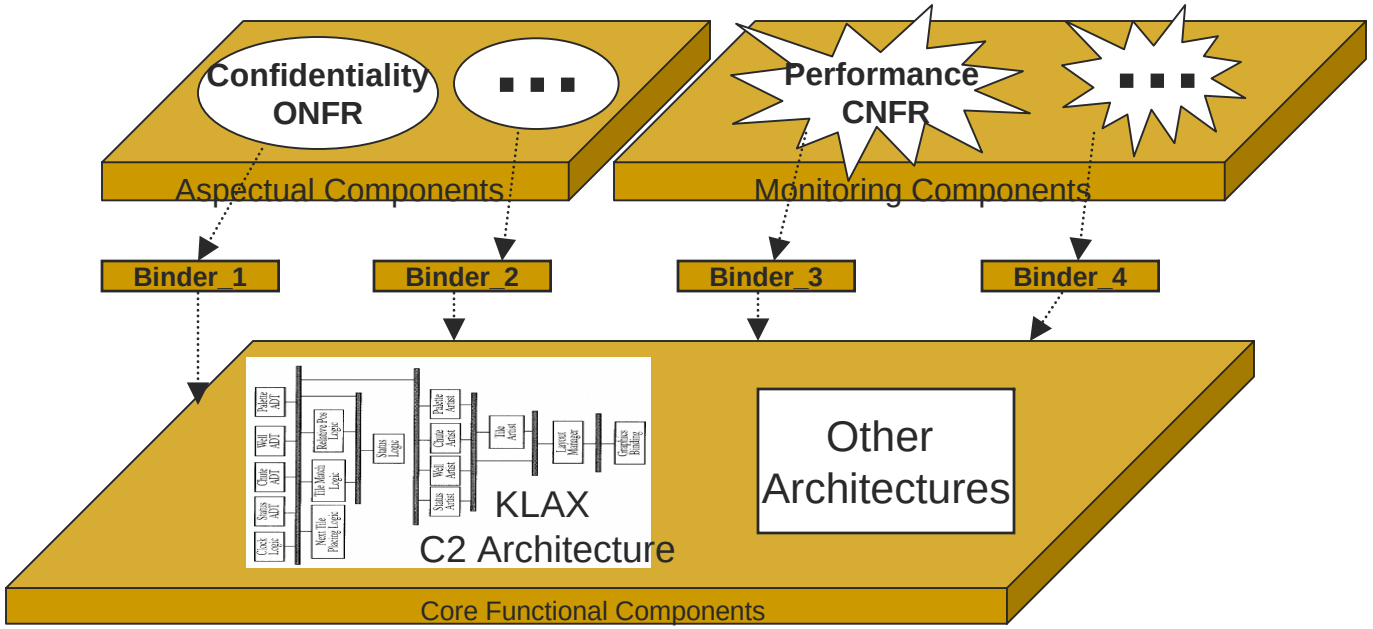
```
<xml>

   <Binder id = "confidentiality">

      <pointcut  id="Update">
         <or>
            <pointcut type="component"
      pattern="Received(event)&&Preparing(action)"  />
         </or>
      </pointcut>

      <interceptor
            Component = "ConfidentialityInterceptor">
         <advice
            type = "before"
            pointcut-refid = "update"
            interceptor-method  =  "PlayerIDProtection()"
         />
      </interceptor>

   </Binder>

</xml>
```

XML Binder II

```
Component monitor ResponsivenessInterceptor {
        TimeStampCheckingBefore () {
        // the code for gathering starting time data goes here
            // add one timestamp to the start point of
            // each function where the request has been made
            // and sent out
        }
        TimeStampCheckingAfter () {
        // the code for gathering stopping time data goes here
            // add one timestamp to the end point of
            // each function where the request has been received
            // and updates have been made accordingly
         }
        TimeStampChecking () {
        // the code for verifying the timeslot used for
        // performing the request goes here
            // record the time difference between timestamps
            // resulting from the previous two methods,
            // and verify against the requirement
        }
        …
}
```

```
<xml>

    <Binder id = "responsiveness">

     <and>
       <pointcut  id="Starting">
           <or>
               <pointcut
       type="component"
       pattern="Preparing(event)&&Sending(event)"  />
           </or>
        </pointcut>

        <pointcut  id="Update">
           <or>
               <pointcut
       type="component"
       pattern="Received(event)&&Updated()"  />
           </or>
         </pointcut>
       </and>

       <interceptor
               Component = "ResponsivenessInterceptor">
           <advice
               type = "before"
               pointcut-refid = "Starting"
interceptor-method=TimeStampCheckingBefore()"/>
           <advice
               type = "after"
               pointcut-refid = "Update"
interceptor-method=TimeStampCheckingAfter()"/>
       </interceptor>

    </Binder>

</xml>
```

Architectural Pattern



Differences from Previous Work

- NFRs as first class requirements elements that will be mapped into architectural design elements
- Provide clear means and guidance to identify the related core components for each NFR, and to integrate the several types of components
- Generality: Can be used in conjunction with existing architectural styles or other approaches to modeling and mapping of NFRs

Conclusions

An architectural pattern to support multiple views of software architecture design:

- Traditional architectural design
- Impose constraints for making the architecture designed correspond and “implement” those NFRs
- Many to many relationships

A step toward a broader set of objectives:

- “Seamless” synthesis from NFRs through architectures to aspect-oriented solutions
- Analysis and testing of development artifacts
- Traceability of development artifacts