Model Checking Requirements at Run-time in Adaptive Systems

Prof: Paola Inverardi
Università dell'Aquila – Dip. di Informatica

Phd student: Marco Mori
IMT Institute for Advanced Studies Lucca

ASAS 2011

Workshop on Assurance for Self-Adaptive Systems

Outline

- The Approach
- Motivating Scenario
- Assurance Framework
  - Evolution and Execution Formalization
- Assurance Process
  - Example: Assurance Process
- Conclusion

Introduction

- In the ubiquitous environment applications are sensitive to the external conditions
- Self-adaptive systems aim at adjust various artifacts or attributes in response to changes in the self and in the context of a software system
  - Self is the whole body of software, mostly implemented in several layers (e.g. new requirements)
  - Context is everything in the operating environment that affects the system properties and its behavior

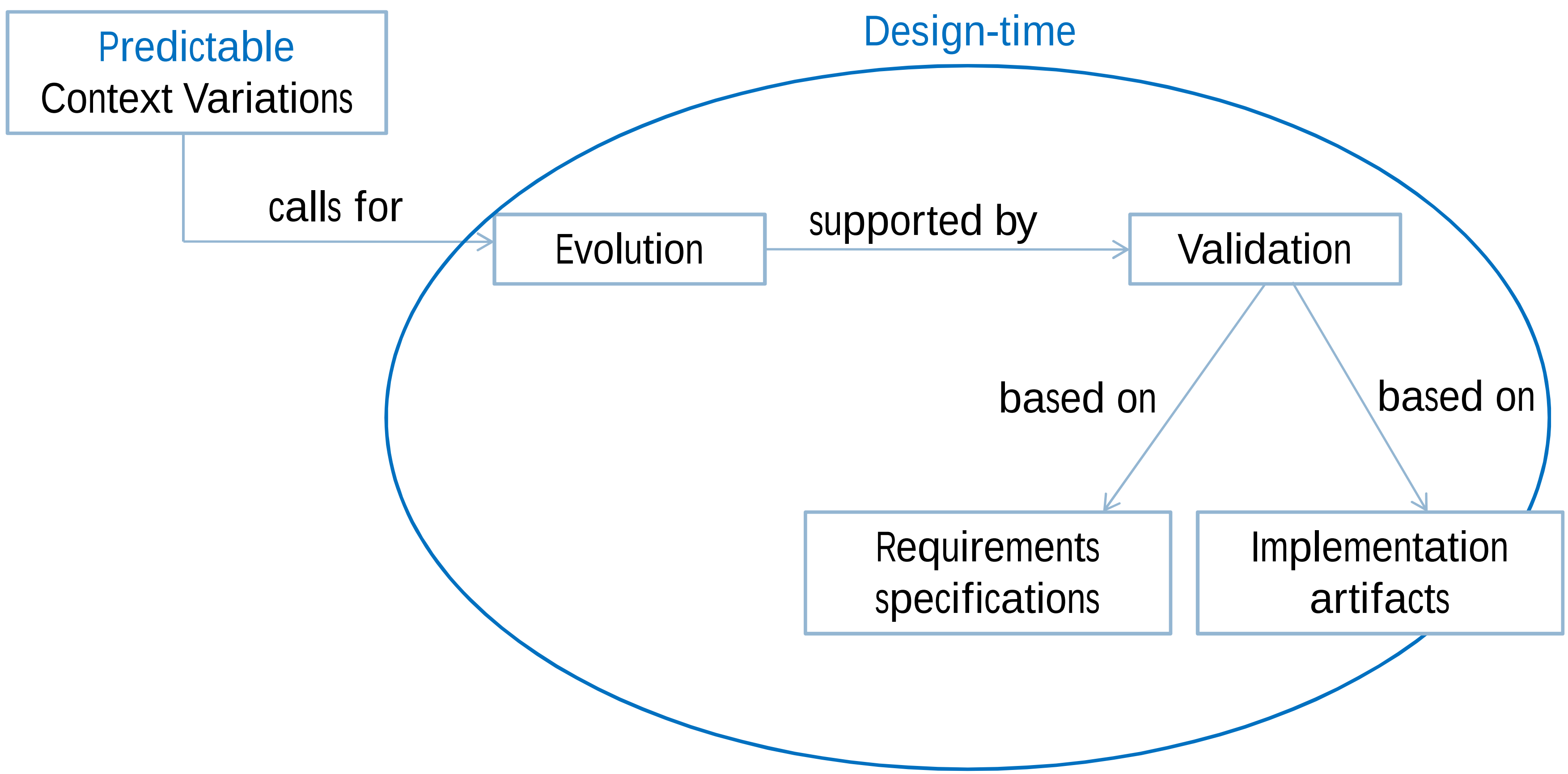
System Evolution

- Software engineer defines a set of system alternatives at design-time (having in mind the possible contexts)….
- … But new unforeseen contexts may appear at run-time (New resources, new values for old resources, etc…)
- The user may specify a new requirements which represents his new need in the unforeseen context
- At run-time the set of system alternatives may have to be augmented to satisfy the new requirement

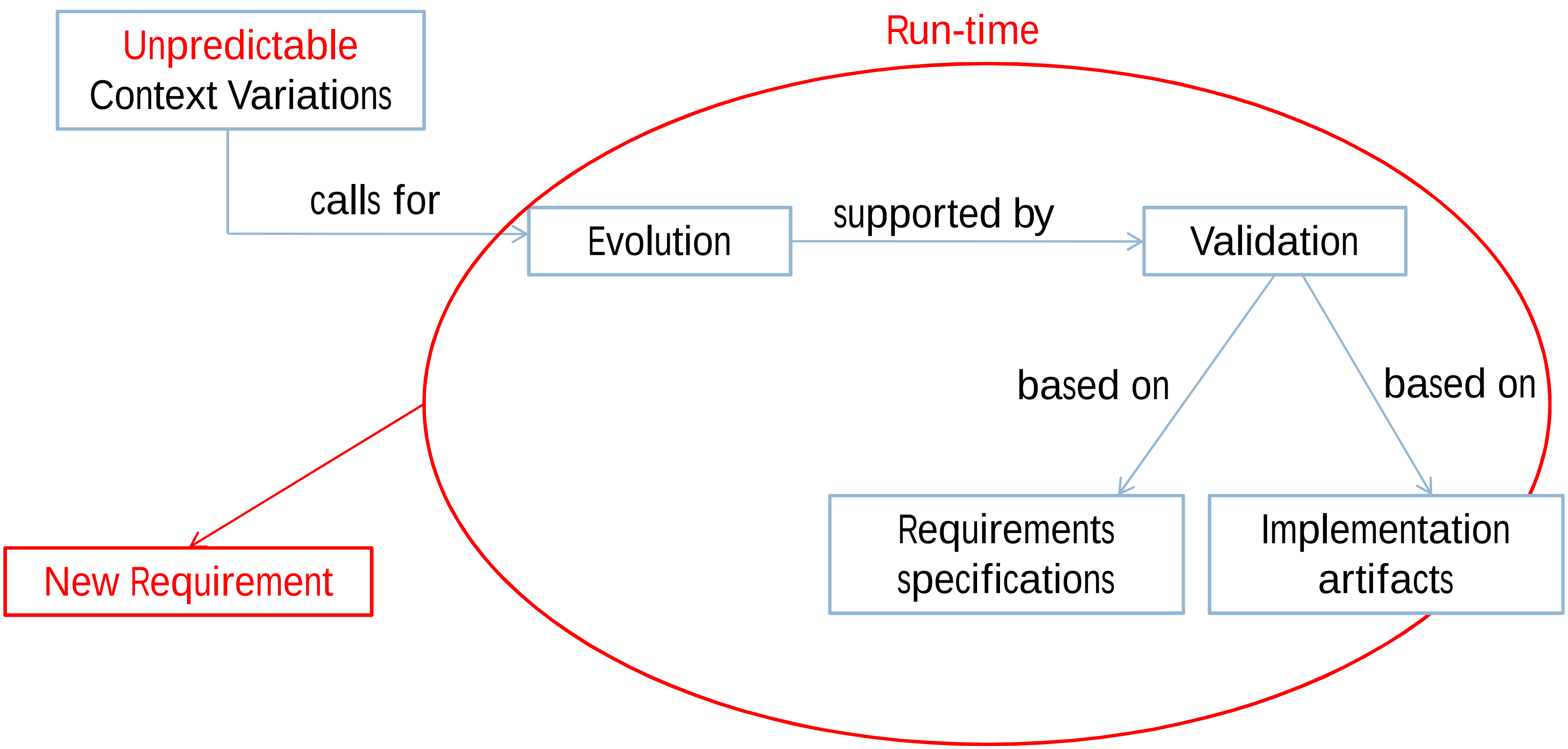
High-assurance

- To prevent the system incorrect behavior the evolution has to be supported by validation mechanisms
  - At design time: we have to validate the known system alternatives
  - At run-time: we have to validate new system alternatives
- Considering actual system model (code) can better prevent the system incorrect behavior than considering high-level models

The Approach



The Approach



Run-time High-Assurance

- A new definition:
  - "High-assurance provides evidence that the system satisfies continuously its functional or non-functional requirements thus maintaining the user's expectations despite predictable and unpredictable context variations"
- Unpredictable context variation

↓

- New requirements at run-time

↓

- Run-time assurance techniques for a perpetual assessment of un-anticipated evolutions [ChLeGi09]

[ChLeGi09] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. Software Engineering for Self-Adaptive Systems, volume 5525 of LNCS, 2009

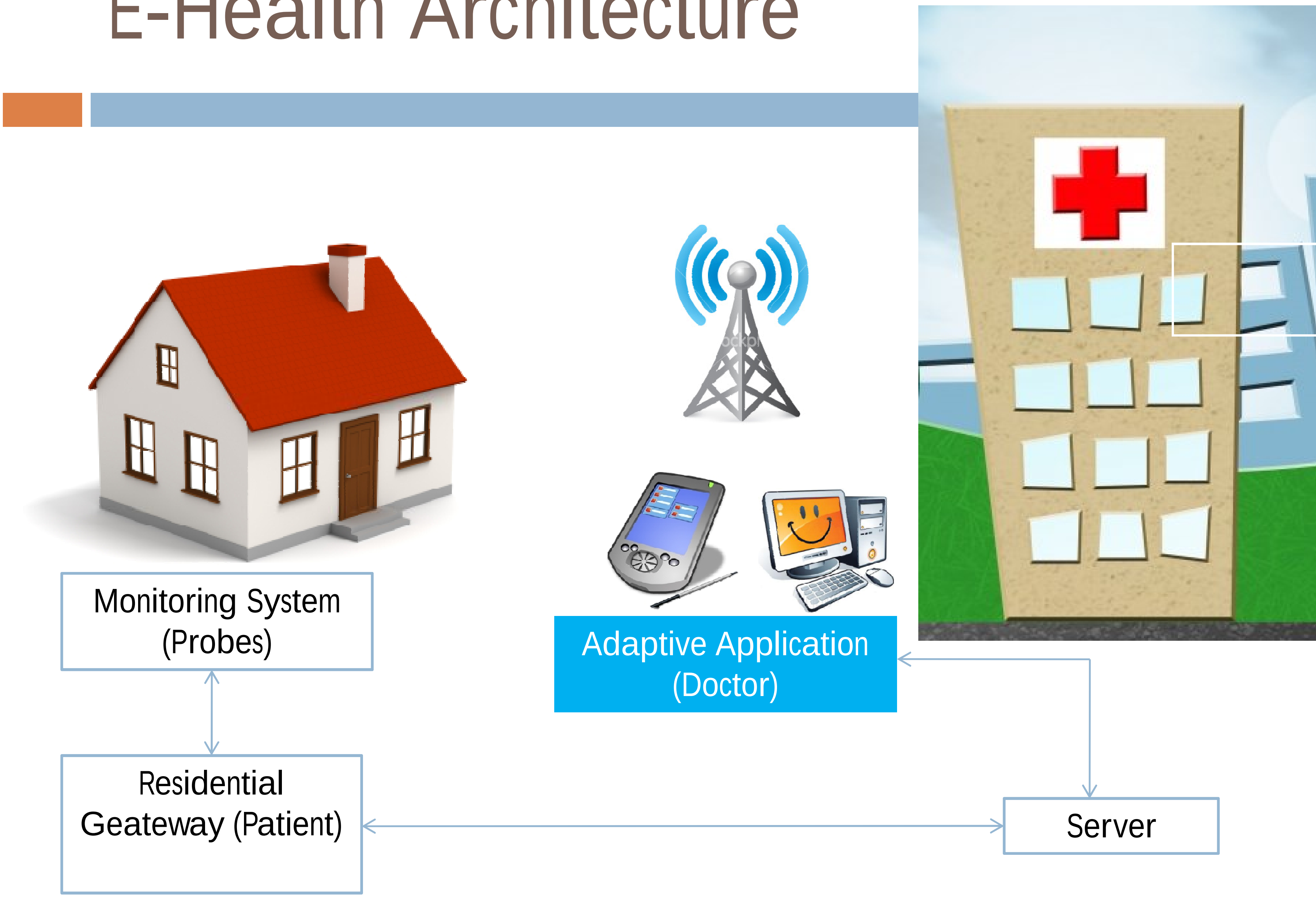
Run-time High-Assurance

- In the literature there are many attempts of evaluating high-assurance at design-time for adaptive systems
  - Discovering miss-behaving requirements [AlMoK09]
  - Model checking alternative behaviors [ClHe11]
- Almost, no support for run-time high-assurance activities over run-time evolved requirements
  - Run-time model checker for evolving probabilistic models [FiGh11]
- No support for run-time high-assurance of actual (code) system models

Motivating Scenario

- E-Health distributed application to monitor vital parameters belonging to elderly people
- Probes sense patient information whereas the home gateway transmit them to the hospital
- Doctors visualize the trends of pulse oximetry and heart rate through PDA and desktop devices
- Adaptive behavior:
  - Set of system alternatives to visualize the vital parameters at the doctor's device as textual or graphical representation (possibly real-time)
  - Each alternative
    - has a different requirements specification
    - consume a certain amount of resources to be provided by the environment (e.g. memory, CPU, etc…)

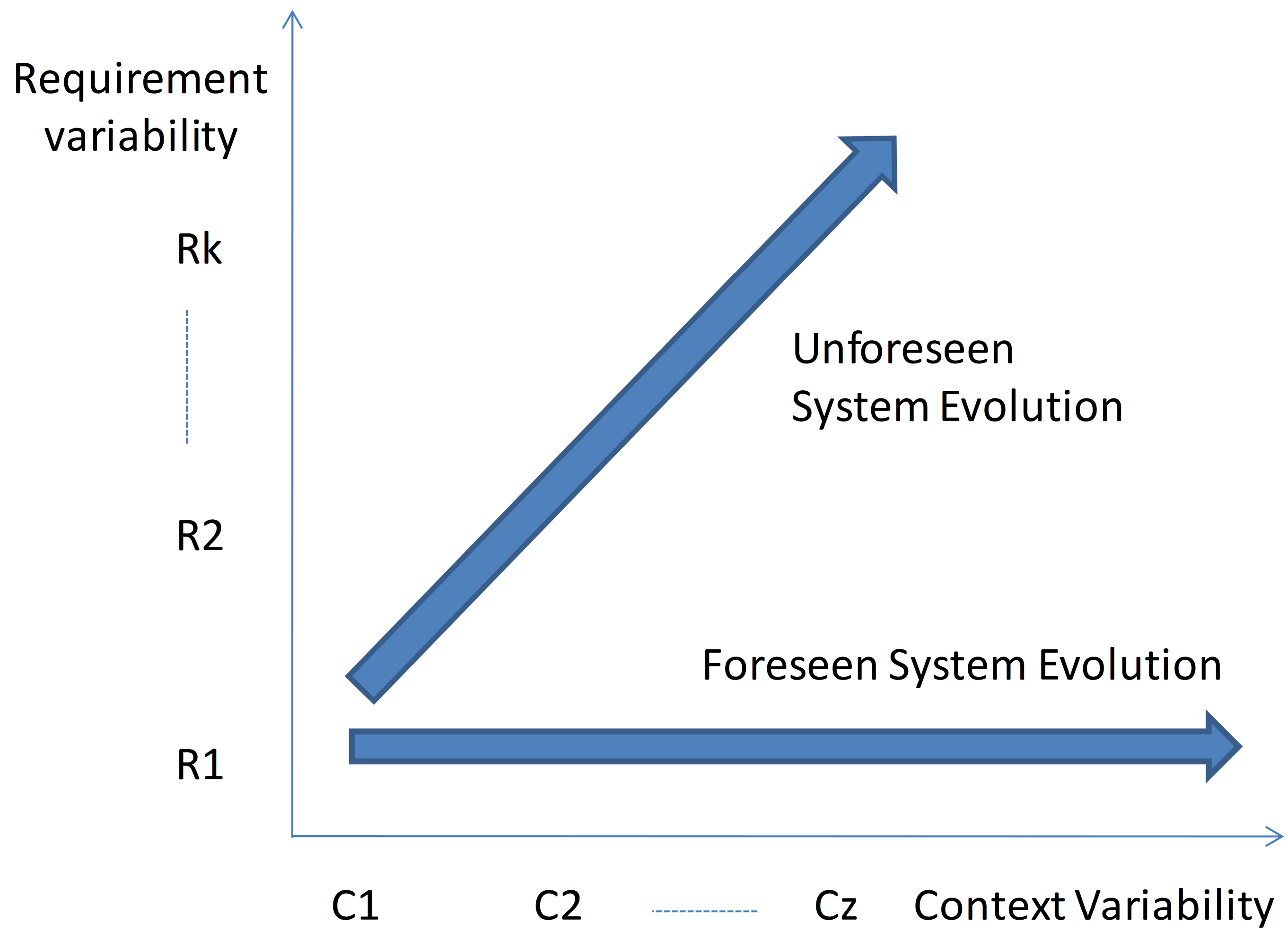
E-Health Architecture



Assurance Framework

- Supports the consistent evolution of adaptive applications starting from the requirement level
- Supports design-time and run-time evolutions
- System variability can be expressed following the Software Product Line Engineering perspective (SPLE)
- Supports a formal definition of high-assurance

Evolution Taxonomy (1/2)



Evolution Taxonomy (2/2)

- Foreseen Evolution:

  foreseen context variations → among statically analyzed system alternatives the framework choose the most suitable [MoLi11]

- Unforeseen Evolution:

  unforeseen context variation → switching towards an un-anticipated system alternative which satisfies a new requirement (@ run-time)

[MoLi11] M. Mori, F. Li, C. Dorn , P. Inverardi, S. Dustdar. "Leveraging State-based User Preferences in Context-aware Reconfigurations for Self-adaptive Systems". International Conference in Software Engineering and Formal Methods (SEFM). Montevideo, 2011

Requirements Taxonomy

- A concern is a matter of interest in a system
- The requirement taxonomy is created by the taxonomy of concerns:
  - (i) Functional requirements ⇨ functional concerns
  - (ii) Performance requirements ⇨ performance concerns
  - (iii) Quality requirements ⇨ quality concerns
- Constraint requirements restrict the solution space of meeting (i), (ii), (iii) [GL07]

[GL07] M. Glinz. On non-functional requirements. In Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, pages 21–26, 2007

System Notation (1/2)

- System variability can be expressed following the Software Product Line Engineering perspective (SPLE) [KeKu98]
- The single unit, the so called feature, represents the smaller part of a service that can be perceived by a user
- Features are combined into configurations in order to produce the space of system alternatives
- Inspired by the SPLE we adopt the notion of *feature interaction phenomenon* as our notion of high-assurance
- A system configuration shows a *feature interaction phenomena* if its features run correctly in isolation but they give rise to undesired behavior when jointly executed

[KeKu98] D. O. Keck and P. J. Kuhn. The feature and service interaction problem in telecommunications systems. a survey. IEEE TSE, 24(10):779–796, 1998

System Notation (2/2)

- System is a set of unit of behavior defined as triple (R,I,C) [ClHe08] where:
  - R is a functional, performance or quality requirement (context independent)
  - I is the code implementation (e.g. Java)
  - C: constraint requirement (context dependent)
- A configuration $G_F = (R_F, I_F, C_F)$ is obtained by combining a subset of features F
- We assume to have an abstract union operator to combine features, which is expressed in terms of union operator for R, I and C
  - Given two features $f_1 = (R_1, I_1, C_1)$ and $f_2 = (R_2, I_2, C_2)$ their union is defined as:

$$f_1 \cup_f f_2 = (R_1 \cup_R R_1, I_1 \cup_I I_2, C_1 \cup_C C_2)$$

[ClHe08] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In FASE, pages 16–30, 2008

Example: Feature

$$R_{graphOxygen} = [\ ](GraphOxViewer.ViewGraphOx(Graph)) \rightarrow (<> GraphOxViewer.outcome)$$

$I_{graphOxygen}$

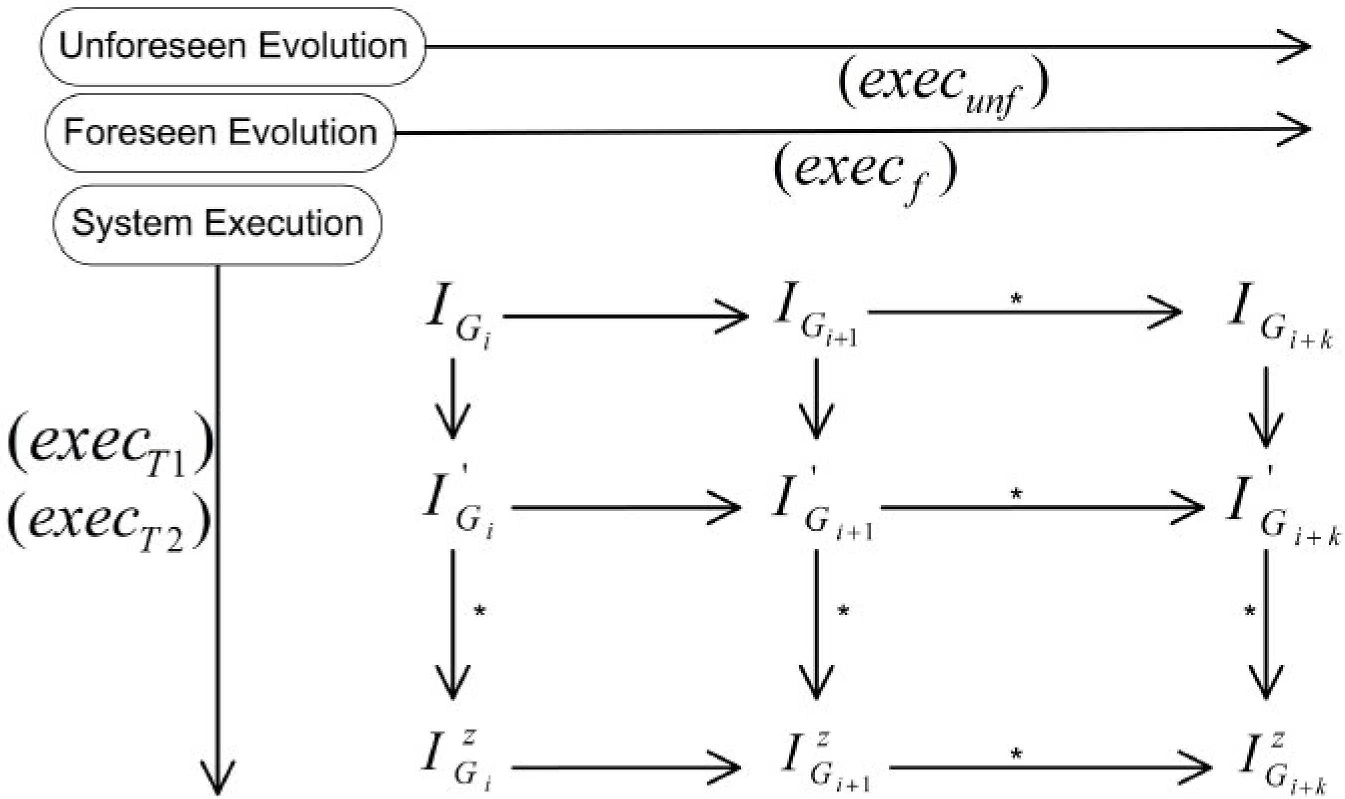
```
public  class GraphOxViewer{
. . .
public void viewGraphOx (Graph g) throws Except ion {
      Annotation.resources ("mem(50) , CPUClockRate (1000)");
      for (int i = 0; i<10; i++){
                  XYDataItem dataOx = OximetryRetr.getOximetryData();
                  dataVectOx .add (dataOx);
      }
      g.DisplayGraph (dataVectOx);
      outcome = Checker.Check(g.currData, dataVectOx);
      if (!outcome ) { throw propertViolation;}
}. . . }
```

$$C_{graphOxygen} = mem \geq 50 \wedge CPUClockRate \geq 1000Khz \wedge oxygenationProbe = true$$

Evolution and Execution

- The systems move state by state $\sigma = < \sigma_s, \sigma_c, \sigma_e >$
  - σ_s is the internal state portion managed by I which does not affect any of the evolution scenarios
  - σ_c is the portion of external state which addresses the foreseen evolution. It represents the current context state
  - σ_e is the portion of external state which addresses the unforeseen evolution. It may contains either a new requirement $< R_{New}, + >$ arising from the user or a requirement to delete $< R_{Del}, - >$.
- Whenever no unforeseen evolution is required this portion of state is empty $\sigma_e = 0$
- We assume that a monitor exists that runs in parallel with the system $monitor(\sigma_c)$

Evolution and Execution



Execution

$$exec_{T_1} \quad \frac{monitor(\sigma_c) = false \qquad I_{G_i}^{<\sigma_s,\sigma_c,\sigma_e>} \rightarrow_{exec_{T1}} I_{G_i}'^{<\sigma_s',\sigma_c,\sigma_e>} \qquad \sigma_e = \varnothing}{\langle I_{G_i}, < \sigma_s, \sigma_c, \sigma_e > \rangle \rightarrow_{exec_{T1}} \langle I_{G_i}', < \sigma_s', \sigma_c, \sigma_e > \rangle}$$

$$exec_{T_2} \quad \frac{monitor(\sigma_c) = false \qquad I_{G_i}^{<\sigma_s,\sigma_c,\sigma_e>} \rightarrow_{exec_{T2}} I_{G_i}'^{<\sigma_s,\sigma_c',\sigma_e>} \qquad \sigma_e = \varnothing}{\langle I_{G_i}, < \sigma_s, \sigma_c, \sigma_e > \rangle \rightarrow_{exec_{T2}} \langle I_{G_i}', < \sigma_s, \sigma_c', \sigma_e > \rangle}$$

Foreseen Evolution

$$monitor(\sigma_c) =< true, \sigma_{c'} >$$

$$exec_f \quad \frac{I_{G_i}^{<\sigma_s,\sigma_c,\sigma_e>} \rightarrow_{exec_f} I_{G_j}^{<\sigma_s,\sigma_c',\sigma_e>} \qquad BestRanked(\sigma_c') = G_j}{\langle I_{G_i}, < \sigma_s, \sigma_c, \sigma_e > \rangle \rightarrow_{exec_f} \langle I_{G_j}, < \sigma_s, \sigma_c', \sigma_e > \rangle}$$

Unforeseen Evolution

$$exec_{unf} \frac{monitor(\sigma_c) = false \quad < R_{New}, + > \in \sigma_e \quad SearchEngine(R_{New}) = f \quad Verify(G_j) = true \quad I_{G_i}^{<\sigma_s, \sigma_c, \sigma_e>} \rightarrow_{exec_{unf}} I_{G_j}^{<\sigma_s, \sigma_c, \varnothing>} \quad G_j = G_i \cup_f f}{\langle I_{G_i}, < \sigma_s, \sigma_c, \sigma_e > \rangle \rightarrow_{exec_{unf}} \langle I_{G_j}, < \sigma_s, \sigma_c, \varnothing > \rangle}$$

Assurance Process (1/3)

- Given a running configuration $G_F = (R_F, I_F, C_F)$ and a new feature $f_{New} = (R_{New}, I_{New}, C_{New})$ implementing the new requirement, we have identified three notions of correctness:
  - (i) $R_F \cup_R R_{New}$: joint requirement satisfiability
  - (ii) $(C_F \cup_C C_{New})[c_s / x]$: joint context requirement validity in the current context state [InMo11]
  - (iii) $I_F \cup_I I_{New} \vdash R_F \cup_R R_{New}$: joint implementation satisfies the joint requirement
- We focus on check (iii) which checks the inconsistency at implementation level

[InMo11] P. Inverardi and M. Mori. Requirements Models at Run-time to Support Consistent System Evolutions. In Requirements@Run-time. 2011

Assurance Process (2/3)

- Given a running configuration $G_F = (R_F, I_F, C_F)$ and a new feature $f_{New} = (R_{New}, I_{New}, C_{New})$ implementing the new requirement, we have identified three notions of correctness:
  - (i) $R_F \cup_R R_{New}$: joint requirement satisfiability
  - (ii) $(C_F \cup_C C_{New})[c_s / x]$: joint context requirement validity in the current context state [InMo11]
  - (iii) $I_F \cup_I I_{New} \vdash R_F \cup_R R_{New}$: joint implementation satisfies the joint requirement
- We focus on check (iii) which checks the inconsistency at implementation level
- LTL requirements as R and Java code as I

[InMo11] P. Inverardi and M. Mori. Requirements Models at Run-time to Support Consistent System Evolutions. In Requirements@Run-time. 2011

Assurance Process (3/3)

- We exploit the Java Path Finder (JPF) tool [JpfCore] in order to validates requirements R with respect to Java classes I:
  - We have implemented a procedure to check the satisfaction of R
  - If the result of this check is negative an exception is thrown
  - JPF checks if at least a path of execution generates un-handled exceptions
  - If the exception is not thrown in any of the execution paths the property is satisfied

[JpfCore] http://babelsh.arc.nasa.gov/hg/jpf/jpf-core

Example: Assurance Process

- A certain configuration G is running at the doctor device to visualize the oxygenation data graphically
- A new sensor to detect the respiratory rate is added to the system as a new UPnP device
- The doctor is notified of the new probe, as a consequence he specifies a new requirement:
  - R= "*Receive and view the respiratory rate data*"

Example: Assurance Process

- A certain configuration G is running at the doctor device to visualize the oxygenation data graphically
- A new sensor to detect the respiratory rate is added to the system as a new UPnP device
- The doctor is notified of the new probe, as a consequence he specifies a new requirement:
  - R= "*Receive and view the respiratory rate data*"
- Two different features are proposed each one implementing R with a different visualization modality:
  - []GraphRespRViewer.viewGraphRespR(Graph)→<> GraphRespRViewer.outcome
  - []GraphRespRViewer.viewTextRespRate(Text)→ <> TextRespRViewer.outcome

Example: New Feature

$R_{graphRespRate} =$
$= [](GraphRespRViewer.viewGraphRespR(Graph) \rightarrow$
$(<> GraphRespRViewer.outcome))$

$I_{graphRespRate}$:

```
public class GraphRespRViewer {
boolean outcome=false;
private static Exception propertyViolation;
...
public void viewGraphRespR(Graph g) throws Exception{
...
for(int i = 0;i<10;i++){
XYDataItem dataRespR = RespRRetr.getRespRData();
dataVectRespR.add(dataRespR);}
g.displayGraph(dataVectRespR);
outcome = Checker.Check(g.currData, dataVectRespR);
if (!outcome){throw propertyViolation;}}...}
```

- After the invocation of the method “viewGraphRespR” the function “Check” attests that the graphical widget contain exactly the retrieved data
- Exploiting Java Path Finder we check if at least a path of execution leads to the un-handled exception “propertyViolation”

Example: Consistency Check

- Model checking the augmented requirement w.r.t. the augmented implementation

$$I_G \bigcup_I I_{graphRespRate}$$

$$\vdash$$

$$R_G \bigcup_R R_{graphRespRate}$$

$$R_{GNew} = R_{graphOxygen} \cup_R R_{graphRespRate} \cup_R \ldots =$$
$$[]((GraphOxViewer.viewGraphOx(Graph) \rightarrow$$
$$(<> GraphOxViewer.outcome)) \wedge$$
$$(GraphRespRViewer.viewGraphRespR(Graph) \rightarrow$$
$$(<> GraphRespRViewer.outcome))) \cup_R \ldots$$

$$I_{GNew} = I_{graphOxygen} \cup_I I_{graphRespRate} \cup_I \ldots =$$

```
public class VariantGNew{
static Graph myGraphViewer;
public static void Execute() throws Exception{
myGraphViewer =  new Graph();
GraphOxViewer graphOx =new GraphOxViewer();
GraphRespRViewer graphRr = new GraphRespRViewer();
graphOx.viewGraphOx(myGraphViewer);
graphRr.viewGraphRespR(myGraphViewer);
}...}
public class GraphOxViewer{
boolean outcome=false;
private static Exception propertyViolation;
...
public void viewGraphOx(Graph g) throws Exception{
...
for(int i = 0;i<10;i++){
XYDataItem dataOx = OximetryRetr.getOximetryData();
dataVectOx.add(dataOx);}
g.displayGraph(dataVectOx);
outcome = Checker.Check(g.currData, dataVectOx);
if (!outcome){throw propertyViolation;}}...}
public class GraphRespRViewer {
boolean outcome=false;
private static Exception propertyViolation;
...
public void viewGraphRespR(Graph g) throws Exception{
...
for(int i = 0;i<10;i++){
XYDataItem dataRespR = RespRRetr.getRespRData();
dataVectRespR.add(dataRespR);}
g.displayGraph(dataVectRespR);
outcome = Checker.Check(g.currData, dataVectRespR);
if (!outcome){throw propertyViolation;}}...}}
```

Example: Consistency Check

- Model checking the augmented requirement w.r.t. the augmented implementation

$$I_G \cup_I I_{graphRespRate}$$

$$\vdash$$

$$R_G \cup_R R_{graphRespRate}$$

$R_{GNew} = R_{graphOxygen} \cup_R R_{graphRespRate} \cup_R \ldots =$
$[]((GraphOxViewer.viewGraphOx(Graph) \rightarrow$
$(<> GraphOxViewer.outcome)) \wedge$
$(GraphRespRViewer.viewGraphRespR(Graph) \rightarrow$
$(<> GraphRespRViewer.outcome))) \cup_R \ldots$

$I_{GNew} = I_{graphOxygen} \cup_I I_{graphRespRate} \cup_I \ldots =$

```
public class VariantGNew{
static Graph myGraphViewer;
public static void Execute() throws Exception{
myGraphViewer =    new Graph();
GraphOxViewer graphOx =new GraphOxViewer();
GraphRespRViewer graphRr = new GraphRespRViewer();
graphOx.viewGraphOx(myGraphViewer);
graphRr.viewGraphRespR(myGraphViewer);
}...}
public class GraphOxViewer{
boolean outcome=false;
private static Exception propertyViolation;
...
public void viewGraphOx(Graph g) throws Exception{
...
for(int i = 0;i<10;i++){
XYDataItem dataOx = OximetryRetr.getOximetryData();
dataVectOx.add(dataOx);}
g.displayGraph(dataVectOx);
outcome = Checker.Check(g.currData, dataVectOx);
if (!outcome){throw propertyViolation;}}...}
public class GraphRespRViewer {
boolean outcome=false;
private static Exception propertyViolation;
...
public void viewGraphRespR(Graph g) throws Exception{
...
for(int i = 0;i<10;i++){
XYDataItem dataRespR = RespRRetr.getRespRData();
dataVectRespR.add(dataRespR);}
g.displayGraph(dataVectRespR);
outcome = Checker.Check(g.currData, dataVectRespR);
if (!outcome){throw propertyViolation;}}...}}
```

Example: Consistency check

- Java Path Finder finds out a un-handled exception which is thrown by the “viewGraphRespR” method
- The graph does not contain exactly the data belonging to the respiratory rate but also the data belonging to the oxygenation

Conclusion

- We have devised an automatic procedure to check high-assurance at run-time with JPF
- Pros
  - Automatic check to prevent the system from adopting incorrect (in-consistent) behavior
  - Consistency checks performed over actual system model (Java code)
- Cons
  - To check: scalability and performances of the run-time model checking
- As for future work
  - Applying our methodology to a comprehensive set of case studies

References

[AlMoK09] M. Alferez, A. Moreira, U. Kulesza, J. Araujo, R. Mateus, and V. Amaral. Detecting feature interactions in spl requirements analysis models. In FOSD, pages 117-123, 2009

[ClHe11] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In ICSE, pages 321-330, 2011

[FiGh11] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In ICSE, pages 341-350, 2011

Thanks!

- Questions?