A Software Lifecycle Process to Support Consistent Evolutions

Paola Inverardi[1] and Marco Mori[2]

[1] Dip. di Informatica, Università dell'Aquila
paola.inverardi@di.univaq.it
[2] IMT Institute for Advanced Studies Lucca
marco.mori@imtlucca.it

Abstract. Ubiquitous software systems evolve their behavior at run-time because of uncertain environmental conditions and changing user needs. This paper describes our approach for a model-centric software evolution process of context-aware adaptive systems. Systems are represented following the feature engineering perspective and this modeling supports foreseen and unforeseen evolution. The first one deals with foreseen contexts while unforeseen evolutions address new user needs arising at run-time possibly in response to unforeseen context changes. The main contribution of this paper is a generic software lifecycle process for context-aware adaptive systems that allows systems to be managed both at design time and at execution time by exploiting suitable models. The approach supports both static and dynamic decision-making mechanisms to enact evolutions and to check the evolution consistency.

Keywords: Context-aware adaptive systems, software lifecycle process, variability model, consistent evolution.

1 Introduction

In the era of ubiquitous computing, software systems have to be designed and developed taking into account the information coming from the surrounding environment. This new dimension, called *context*, has to be exploited to make systems flexible and adaptive. Context is not completely known at design time thus making the process of designing and developing ubiquitous applications continue at execution time [13,18,25]. Software engineers define software alternatives having in mind a partial representation of the context in which the system is going to operate. Since it is not always possible to have a complete representation of the environment, the software engineer cannot provide all the software alternatives at design time. In addition resource-constrained devices limit the number of admissible alternatives. Thus the set of software alternatives provided at design time may have to be augmented in order to face new unforeseen environmental conditions.

We consider two systems-related characteristics: *context-awareness* expresses the ability of accessing and exploiting environmental information [7,21,28], and *adaptivity* which makes a system flexible by supporting behavioral variations [47,3,43]. However, systems should evolve in a consistent way with respect to the context in which they operate. Models can play a key role for developing and evolving context-aware adaptive applications since they support the consistent evolution required by context variations. Different models are required to achieve a consistent evolution:

- a model for representing the system and its variability;
- models to represent the context surrounding the system;
- requirement engineering models;
- models representing executable artifacts;
- a software process model for the adaptive application.

All mentioned models should be exploited and managed at run-time when the development of the system is still required [10,44]. Therefore, on one hand models should provide the right level of adaptivity for the system while on the other hand they should be suitable in terms of required computational effort. This means that the time required to accomplish the model-based consistency check should be negligible with respect to the interval between consecutive evolution requests.

Models should be exploited by an integrated support in order to automate, as much as possible, the process of developing and evolving adaptive applications. This would enable the software engineer to reuse a set of "good practice" and tools for building and maintaining such applications [37].

This paper defines a generic software lifecycle process for context-aware adaptive systems. The process we propose supports two kinds of evolution while keeping the system consistent with the context. *Foreseen evolution* addresses foreseen context variations whereas *unforeseen evolution* deals with unforeseen context variations. Figure 1 shows how context affects both system evolutions. In the foreseen evolution the system evolves in order to keep satisfied a fixed set of requirements by switching among different software alternatives provided at design

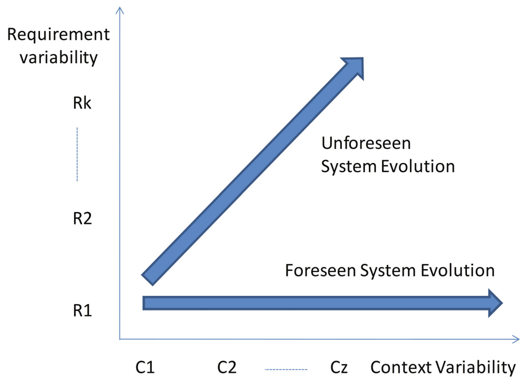

Fig. 1. System evolutions

time for different known contexts. In the unforeseen evolution the system evolves to satisfy changing user needs (requirements) that may emerge as a consequence of an unforeseen context variation. The system evolves by switching to a new un-anticipated software alternative whose behavior includes new functionality necessary to satisfy the emerging requirement. We represent the system following the Software Product Line Engineering (SPLE) perspective since it breaks the system complexity into feature components thus reducing the impact that any change may have on the system [29]. In addition the SPLE perspective already provides models to manage the system and to support consistent evolution.

This paper extends preliminary work [22] on the use of feature engineering for modeling the evolutions of context-aware adaptive systems. Our process is amenable to develop and evolve highly-configurable systems with interfering features. It exploits the SPLE perspective in order to provide a uniform abstraction to all the development approaches that consider a system to be made out of a combination of basic software entities, such as the Component Off The Shelf (COTS) approach or the service-oriented one. We assume that we have a set of basic behavioral elements as input to our software process. These basic elements will contain implementation artifacts with corresponding requirements specifications. It is worth stressing that while we do address the problem of managing model-centric evolution we will only briefly comment on the mechanisms to enact the evolution. We will assume that the system may be reconfigured when it is in a state in which the evolution is allowed (e.g. quiescent state or weaker notions [30,46]).

In this paper we also define a generic evolution framework to support our software process in performing its run-time activities. To this end, the framework implements a control loop to monitor and to evolve the adaptive application.

The contributions of this paper are:

- a process methodology to design and develop context-aware adaptive applications;
- a set of models to represent the system and the context along with their evolutions;
- a methodology to check the consistency based on the context;
- an architecture implementing the support for the evolutions.

We will explain our approach by means of an adaptive application which elaborates a Mandelbrot fractal [33] that better fits the characteristics of the mobile device (CPU, memory, number of display colors,...). The application requirements consist in visualizing a fractal image to the user through the device screen. The higher the level of context resources available, the more beautiful will be the fractal image shown to the user. The fractal context-aware adaptive system will be modeled through a set of features which represent the basic alternative behaviors to color, build and view the fractal image. At run time features may need to be activated or de-activated based on the context-resource availability changes. In addition because of environment unpredictability, the user may want to introduce new unforeseen behavior as the system operates in an unforeseen context. For example whenever the unforeseen device characteristics makes the visualization of the fractal image format impossible, the user may guide the introduction of a new software plug-in to decode that specific format.

The remainder of this paper is structured as follows. Section 2 describes related work to address system evolutions while Section 3 introduces the basic models of our generic evolution framework. In Section 4 we define the software lifecycle process to design and develop context-aware adaptive applications. Section 5 describes how our process supports foreseen and unforeseen evolutions. Section 6 proposes the interface architecture which implements the generic evolution framework along with a possible instance with current practice technologies. Section 7 provides a summary of our contribution and a discussion of future work.

2 Related Work

In the literature several frameworks address system evolution. They exploit models with different granularity such as context-aware requirements models, context-aware architectural models and context-aware implementation models.

The Rainbow framework [17] enables architectural self-adaptation by exploiting predefined adaptation rules. System components are reconfigured based on decisions taken at design time while no un-anticipated adaptations can be achieved. The framework supports non-functional reconfigurations while it does not consider the consistency checking of the evolution. The context is not explicitly modeled but simple variables are considered in the framework.

The PLASTIC approach [4] applies reconfigurations at the implementation level by exploiting an explicit definition of context model. The approach supports non-functional reconfigurations of statically defined Java artifacts driven by context variations. The framework only deals with foreseen evolution while run-time evolution is not allowed. The Javeleon framework [20] as well as the JavAdaptor framework [41] aims to support the run-time evolution by means of transparent dynamic updates of running Java applications. Developers can simply evolve their applications at run-time and they can trigger an on-line update without stopping the running application. Javeleon and JavAdaptor do not support a definition of context for the evolution but the developers is directly in charge of injecting new behaviors in the application at run-time. These approaches as well as the PLASTIC framework do not provide a process to assess the consistency of foreseen and unforeseen evolution. Ali et al. [2] propose a goal-based framework to enact the evolution among system variants at requirement level. This approach provides a context analysis phase to discard variants that are inconsistent based on the context predicates. Nevertheless, it only supports the design-time analysis on the contextual goal model. Qureshi and Perini [42] have defined a framework for requirement engineering to distinguish activities at design-time from activities at run-time. They have provided a mechanism to evolve the requirement specification at run-time driven by the user thus supporting a notion of unforeseen evolution. Nevertheless the proposed method is not applied to a real case study and no definition of consistency is considered in the framework. Kramer and Magee [31] have presented a three-layered conceptual model to support the architectural reconfiguration of self-managing systems. They consider a Component layer, a Change management layer and a Goal management layer. The Goal layer identifies the plan to execute while the Change layer enacts the plan execution by interacting with the Component layer. This feature supports reconfigurations required by new requirements arising at run-time, i.e. unforeseen evolutions. The framework provides functional and non-functional evolutions but it lacks a definition of consistency checking for the composition of components.

To the best of our knowledge the frameworks presented in the literature only apply reconfigurations at specific granularity levels, either at requirements models, or at architectural models or at implementation models. Only a few of them support evolution at run-time while there is almost no support to check the consistency of the evolution. In order to provide high-assurance for context-aware adaptive applications it is necessary to support a definition of consistency as proposed by Zowghi and Gervasi [48]. They suggest that an effective support to consistency is based on system models at the different granularity levels, ranging from the problem space models to the solution space models. We claim that adaptive applications are not developed and evolved following a software process which considers all these models together thus making it difficult to effectively support the consistency of the evolution.

3 Evolution Framework

The evolution framework we propose is characterized by different building blocks to represent the system along with its variability. The system is represented by *units of behavior* which are composed through a *feature diagram* into different *system configurations*. The *context model* enacts the switching process among configurations and it supports the *consistency checking* process for the evolution.

Our evolution framework implements a MAPE (Monitoring Analyze Plan and Execute) cycle in order to support the supervision, execution and evolution of adaptive applications [12]. In Figure 2 we show how the framework implements each of the four phases.

A *monitoring* phase activity collects information from the environment and from the user in order to establish if evolution is required or not. On the one hand, foreseen context variations and user preferences variations may both enact foreseen evolutions. The first influences the admissibility of the system variants, whereas the second influences the fitness of the system variants. On the other hand, unforeseen context variations may force the user to introduce a new requirement into the running variant.

The *analyze* phase determines if the variant to adopt is consistent or not. In case of foreseen evolution we consider the set of system variants that are consistent at the current context state. The consistency at each different context state is proven at design time. In case of unforeseen evolution the analysis is

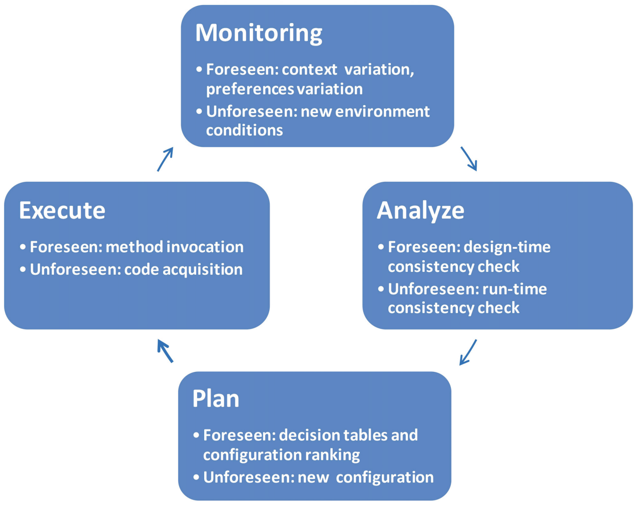
Monitoring
• Foreseen: context variation, preferences variation
• Unforeseen: new environment conditions
Analyze
• Foreseen: design-time consistency check
• Unforeseen: run-time consistency check
Plan
• Foreseen: decision tables and configuration ranking
• Unforeseen: new configuration
Execute
• Foreseen: method invocation
• Unforeseen: code acquisition

Fig. 2. MAPE cycle

performed at run-time by checking the consistency for the un-anticipated variant. This variant will contain the same set of features as the current variant plus a new feature that implements the new requirement specified by the user.

After the analyze phase the *planning* phase supports the decision-making process for the variant to adopt. For the foreseen evolution, a ranking mechanism establishes the most suitable variant based on context and user preferences. For the unforeseen evolution the new variant which has been proven consistent at the analyze phase is put forward to the execution phase.

Finally at the *execution* phase the system switches from the current to the target variant. The target variant is enacted through its entry point method. For the unforeseen evolution it is also necessary to incorporate a new code artifact into the target variant before enacting its execution.

In the following we define the elements of our approach before describing the software development process for context-aware adaptive systems in Section 4.

3.1 Context Model

In our approach the context model expresses the set of external entities that are beyond the system's control but which may influence the system execution. Our context model consists of key-value pairs and it is defined using two perspectives: the *context structure* and the *context space*. The *context structure* expresses resources in term of types and categories. We adopt the resource taxonomy where each context element belongs either to the system, to the user or to the physical environment. In addition we consider the resource types enumerate, boolean and natural [32]. In Figure 3 is depicted a context structure which conforms to the meta-model shown at the left side of the figure.

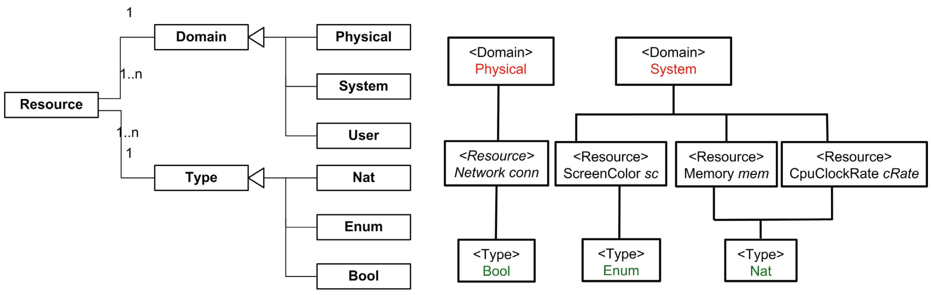

Fig. 3. Context structure: meta-model and model

The *context space* expresses the variability for the resource assignment. Each resource is identified through a tag $ResId$ and it can assume one among its admissible values contained in $dom(ResId)$. The context space for the resources $ResId_1, ..., ResId_n$ is defined as the Cartesian product:

$$S = \bigotimes dom(ResId_i) \quad s.t. \quad i = 1, ..., n \tag{1}$$

Each valid assignment of resources $\overrightarrow{c} \in S$ will be considered as a different context state. Let us consider four different resources respectively expressing the free memory, the CPU clock rate, the number of screen colors and the network availability (0 if false, 1 if true): $dom(mem) = \{100, 150, 250, 350\}$, $dom(cRate) = \{200, 400, 600\}$, $dom(sc) = \{256, 4096\}$, $dom(conn) = \{0, 1\}$. The context model space will be composed of $4 * 3 * 2 * 2 = 48$ states.

3.2 Unit of Behavior

In our vision we represent context-aware adaptive applications in terms of sets of dynamic units called features. A feature is the smaller part of a service that can be perceived by the user. We define a feature by a context-independent requirement, a context-dependent constraint requirement, and an implementation part. The notion of requirements we adopt follows the taxonomy proposed by Glinz [19]. The definition of requirements is based on the *concern* to which a requirement pertains. Given that a concern is a matter of interest in a system, the taxonomy considers functional requirements which pertain to functional concerns, performance requirements which pertain to performance concerns and specific quality requirements which pertain to quality concerns. In addition constraint requirements limit the solution space of functional, performance and specific quality requirements. We adopt this taxonomy and we exploit the feature definition in [14] in order to propose our definition of a feature as a triple $f_i = (R_i, I_i, C_i)$, where:

- R_i is a conjunction of functional, performance and specific quality requirements (context-independent); an example of a functional requirement is:

Compute and visualize each fractal pixel. A quality requirement is: *The image is visualized a pixel at a time* which in terms of implementation consists in assigning the value *Immediate* to the quality property *DisplayModel.*
- I_i is the the component/service implementing the feature. It is expressed as Java code, e.g. see Figure 4.

```
public class MandelCanvas{
 ...
 public void generateImmediateFractal(){
  image = Image.createImage(width, height);
  Graphics imageGraphics_DirectCanvas = image.getGraphics();
  for (int x = 0; x < width; x++) {
   for (int y = 0; y < height; y++){
    FractalPixel pixel_DirectCanvas = drawFractalPixel(x,y);
    repaint();
   }}
}...}
```

Fig. 4. Example: feature implementation

- C_i is a context-dependent constraint requirement defined as a predicate over the context entities, e.g. *mem* $\geq$ *120kb.*

3.3 System Configuration

A *system configuration* is obtained assembling a set of features. Each configuration expresses the set of functionalities that a system shows to a user at a certain step of the evolution. Given the set of features F, a system configuration is a triple obtained combining each feature in F as $G_F = (R_F, I_F, C_F)$. At this level of description we do not explain how to combine features. We just suppose to have an abstract union operator among features which is defined in terms of union operators for context-independent requirements, context-dependent requirement and implementation components. The actual implementation of these union operators will depend on the specific formalisms that may be used to express these three elements. 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: $f_1 \cup_f f_2 =< R_1 \cup_R R_2, I_1 \cup_I I_2, C_1 \cup_C C_2 >$. In the following we show a possible example on how to merge context requirements and implementation components for a system configuration starting from its features.

The union operator $\cup_C$ merges context requirements depending on the nature of resources. For example if we have two requirements demanding bandwidth for 20 kbps each one, their union will express a demand of bandwidth for 40 kbps. For the implementation portion I the software engineer combines the code artifacts in order to have a single access point to the whole configuration. Each configuration is composed by a Java class for each single feature plus a Java class which is the entry point for the system configuration. This class entails the method *execute* to trigger the execution of the configuration (e.g. see Figure 7).

Given a system definition it is necessary to model in which different future system configurations the system may evolve. The variability model that we have chosen is inspired by the *feature model* which has been first introduced in the Feature-Oriented Domain Analysis method [27]. Since then, feature modeling has been widely adopted by the SPL engineering community and a number of extensions have been proposed [45]. In our approach we consider a possible abstract syntax for the feature model defined starting from nodes (features) and arcs between nodes:

- The root node of the model is the label which stands for the system.
- Each node expresses a feature which can be either optional or mandatory.
- Each edge between two nodes expresses a decomposition relation (consist-of) between the parent node and the child node. It enables the possibility to add behavior to the parent feature. We consider two decomposition relations: AND decomposition and XOR decomposition.
- "Requires" constraint is a directed relation between two features. If one feature is present in the configuration the second has to be present as well.
- "Mutex" constraint enables the mutual exclusion between two features; therefore they cannot be in the system configuration simultaneously.

Starting from the feature model (abstract syntax), the feature diagram (concrete syntax) is commonly expressed as a tree structure. We adopt a subset of the syntax presented in [15]. In this diagram, features are represented in a tree-like format. Dark circles represent mandatory features, while white circles represent optional features. An inverted arc among multiple arcs expresses a XOR decomposition meaning that exactly one feature can be selected. Multiple arcs that start from a parent node express an AND decomposition.

Starting from the feature diagram, the set of possible system configurations is obtained by combining the features in subsets compliant to the diagram. The diagram shown in Figure 5 concerns our case study and contains 8 features which give rise to 10 system configurations. Each configuration contains only one feature to generate the image and only one feature to color it. An admissible configuration contains the features to download a predefined image from a remote server. We further discuss the features of the fractal application in Section 4.1.

Each system configuration is mapped to its implementation which enables its execution. Let us consider the system configuration $G_4 = \{f_{genPro}, f_{colB}\}$ implemented by the class diagram depicted in Figure 6. Each feature in G_4 is implemented as a single Java class. The class $MandelCanvas$, which implements f_{genPro}, provides the interface $generateProgressiveFractal$ that generates and draws the fractal image progressively a row at a time exploiting the operation $drawFractalPixel$. The class $Colouring$, which implements f_{colB}, provides the interface $pixelColourAsBands$ and the operation $initColourAsBands$ in order to color the image with different bands of colors. The only class which does not correspond to any of the features within the configuration is $LocalFractalApp$ that is the external interface to access the whole application variant. The configuration is enacted through its method *execute* which implements the logic of the variant. Figure 7 shows an excerpt of the Java specification for configuration G_4.

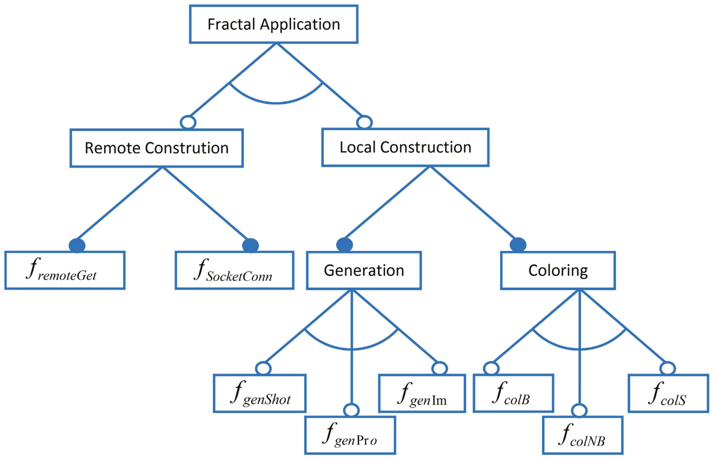

Fig. 5. Feature diagram

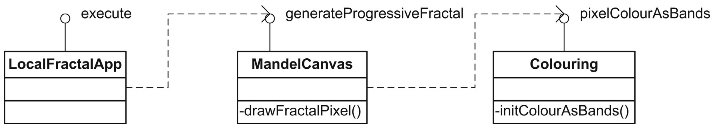

Fig. 6. Example: class diagram (G_4)

3.4 Consistency Checking

We propose a notion of consistency that is based on the notion of *feature interaction*. A feature interaction occurs when two or more features run correctly in isolation but they give rise to undesired behavior when jointly executed [1,9,38]. A certain system configuration is *consistent* if its features does not give rise to any feature interaction phenomenon. Following the Problem Frame approach [26] as exploited in [14], we formalize our notion of consistency for a certain configuration $G = (R_F, I_F, C_F)$ as:

$$I_F, C_F \vdash R_F \tag{2}$$

This definition entails three different problems:

- $(C_F)[\overrightarrow{c}/\overrightarrow{x}]$: this formula checks the joint context requirement (predicate) C_F assigning the current context values $\overrightarrow{c}$ to the formal parameters $\overrightarrow{x}$;
- R_F *is Satisfiable*: this formula checks if the joint context-independent requirement can be satisfied;
- $I_F \vdash R_F$: this formula validates the joint implementation with respect to the joint requirement either by means of model checking or through a testing process.

```
public class LocalFractalApp extends MIDlet {
 MandelCanvas mandelCanvas;
 ...
 public LocalFractalApp(){
  mandelCanvas = new MandelCanvas();
 }
 protected void execute(){
  currentDisplay = Display.getDisplay(this);
  currentDisplay.setCurrent(mandelCanvas);
  mandelCanvas.generateProgressiveFractal();
  exitAction();
 } ... }
public class MandelCanvas extends Canvas {
 ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
    column_ArrayCanvas[y] =  pixelColor(pixel_ArrayCanvas.isInsideFractal(),
    pixel_ArrayCanvas.getIterations(),pixel_ArrayCanvas.getDistance());
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas, 1, height, false);
   repaint();
 }} ... }
public class Colouring{
 ...
 private int pixelColourAsBands(boolean interno, int iterazioni,
 double dist){
  int tmp= (interno ? 0 : colors[iterazioni % paletteNumColors]);
  return tmp;
 }
 private void initColourAsBands(){
  int[] tmpColors_IterationsLimitedPalette = { -256, -16711681,-65281,-256,
  -4194304, -16728064, -16777024, -8323073, -32513, -128 };
  colors = tmpColors_IterationsLimitedPalette;
  paletteNumColors = colors.length;
 }... }
```

Fig. 7. Example: implementation (G_4)

Since our aim is to support a notion of consistency that can be performed at run-time we should take into account the computational effort for the corresponding three algorithms. Among them, checking the context requirement against a certain context state is the less expensive in terms of time and space. Although it is only a necessary but not sufficient condition for a complete notion of consistency it plays a key role to check the consistency of ubiquitous applications. Indeed, the serendipity of the environment that characterizes this kind of systems makes them very vulnerable to context variations. Therefore a weak notion of consistency can be based only on context requirements satisfiability:

Definition 1. G is *weakly consistent* in $\overrightarrow{c}$ iff $C_G[\overrightarrow{c}/\overrightarrow{x}]$ is True

Let us consider the configuration $G_x = \{f_{getRem}, f_{sockConn}, f_{tiffViewer}\}$ where each feature is characterized by the correspondent context requirement:

(i) $C_{tiffViewer} ::= \mathit{cRate} \geq 300 \wedge \mathit{mem} \geq 35$
(ii) $C_{getRem} ::= \mathit{mem} \geq 100$
(iii) $C_{sockConn} ::= \mathit{conn} = 1$

We assume that G_x has to be checked at the context state $\overrightarrow{c} = (100, 300, 4096, 1)$. This state provides 100 Kb of memory, a CPU clock rate of 300 Mhz, a screen device with 4096 colors and an Internet connection. Although each context requirement is weakly consistent separately at $\overrightarrow{c}$, the whole configuration is not weakly consistent because of the limited availability of free memory. Indeed, if we combine the request of memory coming from the context requirement (i) and (ii) we obtain a total request for $135Kb$ of memory that cannot be satisfied at the context state $\overrightarrow{c}$. Therefore it is not possible to execute the features $f_{getRem}, f_{sockConn}$ and $f_{tiffViewer}$ together at $\overrightarrow{c}$.

In our previous paper [23] we have extended this notion of weak consistency by defining a mechanism to check the configuration requirements with respect to the implementation artifacts. This enables us to catch also interactions that arise at the code level.

4 Software Development Process

In this section we describe how we support the development of a context-aware adaptive application. We have defined a software lifecycle process which follows the structure presented by Autili et al. [6]. Our software process implements four different activities, namely Explore, Integrate, Validate and Evolve as shown in Figure 8. The *exploration* phase exploits a feature library containing the code implementation and the correspondent requirements descriptions. The *integration* phase takes these features as input and it produces the space of the system variants as a feature diagram. Each variant is checked though a validation phase which performs the context analysis [24] and model checking [23]. Finally, the *evolution* phase reconfigures the system by switching from the current configuration to the new one.

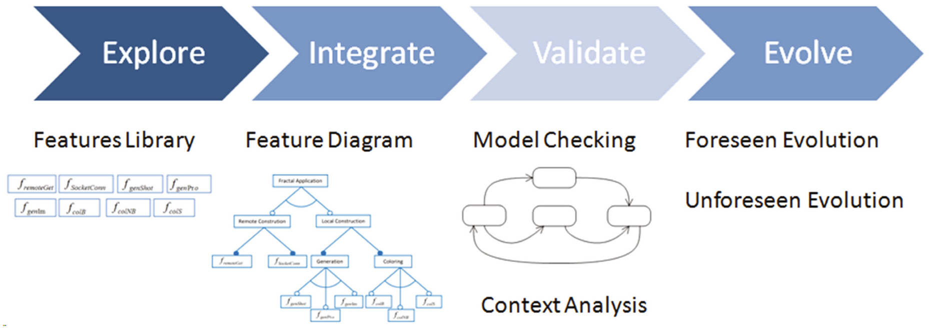

Fig. 8. Software process

In the remainder of this section we describe how our generic evolution framework supports the software process. The problem we face is the complexity for the software engineer to specify the context conditions under which each system configuration is admissible. Given n features it could be required to set the context conditions for 2^n configurations in the worst case. Our methodology makes the generation of the system configuration automatic by exploiting the models provided in the SPLE as described in Section 3.

At the *exploration* phase the software engineer defines the set of features of interest. Starting from a standard component it is possible to define a feature $f = (R, I, C)$ by considering the requirements of the component and its code. The feature code will be exactly the same as the code of the component. R will contain the requirements of the component that are not context-dependent. In general the requirements of the component will always contain requirements about the execution context, thus they will be added to C. Further context requirements, for example concerning resources consumptions can be obtained through suitable static code analysis. For example in our environment we use the Chameleon framework [5] in order to extract the consumption of resources caused by I (e.g. memory and CPU clock rate). At the end of the exploration phase we obtain a set of features defined in terms of their basic components, i.e. $A = \{f_1, .., f_n\}$.

At *integration* phase the software engineer combines the features in A through the feature diagram definition. Architectural constraints will be defined here at the integration phase. Starting from the feature diagram an automatic process generates all the system configurations:

$$G = \{G_1, G_2, ..., G_m\} \text{ s.t. } m \leq 2^{|A|} \qquad (3)$$

We assume that the requirements belonging to each configuration imply the system requirements. We further assume that each configuration satisfies its requirements: $I_{G_i} \vdash R_{G_i}\ \forall i = 1, .., m$. An automatic process generates the context structure and the context space S considering the context entities exploited by the context requirements belonging to the created configurations.

At *validation* phase we create the data structure to support the evolution. This phase takes place by means of two main steps. The first step consists in labeling each context state $\overrightarrow{c}$ in S with all the features which are consistent in $\overrightarrow{c}$ (Eq. 2). The *feature consistency table* is built inserting value 1 each time a feature is consistent in the corresponding context state. The second step consists in labeling each context state $\overrightarrow{c}$ in S with all the system configurations that are consistent in $\overrightarrow{c}$. The *configuration consistency table* is built inserting value 1 each time a configuration is consistent in the correspondent context state. Finally, we aggregate the context states that make the same set of configurations consistent. Nevertheless, we do not address the scalability problems arising from the number of context states and configurations within the mentioned tables. Different approaches [11] have been presented to reason about the configurations belonging to the feature diagram. Moreover the exponential growth of context states could be mitigated by clustering the states [16].

The *evolution* phase reconfigures the system whenever either a foreseen or an unforeseen evolution is required. In the first case we query the configuration consistency table to retrieve the space of the admissible configurations. Among them we select the most suitable one based on the data structures provided at the validation phase. In the second case we have to re-iterate the first three phases of our software process in order to evolve the system. We query a remote feature library to retrieve the feature implementing the new requirement and we integrate the new feature with the current configuration. Finally, we have to validate the new unforeseen configuration before we can add it to the configuration consistency table. The evolution processes are further discussed in Section 5.

4.1 Working Example

In this section we show how we design and develop the adaptive application to visualize a Mandelbrot fractal. To this end, the software engineer defines the set of features A in terms of requirements and code implementations:

$$A = \{f_{genShot}, f_{genPro}, f_{genImm}, f_{colB}, f_{colNB}, f_{colS}, f_{remGet}, f_{sockConn}\}$$

The set A contains the features to generate and color the fractal pixels and the features to download a standard fractal image from a remote server. The generation may be performed by visualizing a pixel at a time (f_{genImm}), a pixel row at a time (f_{genPro}) or the whole fractal image at the end of the drawing process ($f_{genShot}$). The pixel colors are defined following three different schemas: f_{colB} colors pixels as bands exploiting a limited number of tones; f_{colNB} colors pixels as bands exploiting a wide spectrum of tones while f_{colS} follows a smooth schema to color pixels exploiting a wide spectrum of tones. Finally, $f_{sockConn}$ connects the device to the Internet whereas f_{remGet} retrieves and views a standard fractal image from a remote server.

Figure 9 shows an excerpt of the features entailed in A. It is possible to define the context requirement of each feature by exploiting the Chameleon framework in order to obtain the consumption of resources, e.g. CPU clock rate and memory. Further, context requirements can be defined by extracting the requirement on the number of screen colors derived from the requirement of the component.

In order to design the fractal application the software engineer combines the features and produces the feature diagram as shown in Figure 5. The logic operators in the feature diagram guide the automatic generation of 10 system configurations as shown in Table 1. The first nine configurations are obtained combining the three different building mechanisms with three different coloring schemas. The last one simply gets an already defined fractal image from a remote server. Each configuration is characterized by the context requirement and by the offered qualities. The $DisplayModel$ quality represents the modality of showing the fractal while $ColorModel$ quality expresses the coloring modalities.

After creating the configurations, the integration phase generates the context model which contains the relevant resources for the fractal application as shown in Figure 3. In our example the context space will be defined as $S = mem \times cRate \times sc \times conn$.

$\mathbf{f_{genPro}} = (\mathbf{R_{genPro}}, \mathbf{I_{genPro}}, \mathbf{C_{genPro}})$
$\mathbf{R_{genPro}}$: Compute each fractal pixel and show it a pixel row at a time
$\mathbf{I_{genPro}}$:

```
public class MandelCanvas extends Canvas{ ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas,1, height, false);
   repaint();
  } } ... }
```

$\mathbf{C_{genPro}} : \mathbf{mem} \geq \mathbf{200}$

$\mathbf{f_{colS}} = (\mathbf{R_{colS}}, \mathbf{I_{colS}}, \mathbf{C_{colS}})$
$\mathbf{R_{colS}}$: Paint the fractal pixels as smoothly nice colored bands
$\mathbf{I_{colS}}$:

```
public class Colouring{...
 private int pixelColorSmoothly(boolean interno, int iterazioni, double dist){
  if (interno) return 0;
  iterazioni = iterazioni + 2;
  double mu_IterationsDistance = iterazioni-
  (Float11.log(Float11.log(dist))) / log2;
  int tmp= DBL_ToRGB(mu_IterationsDistance);
  return tmp;}
 private void initColorsSmoothly() {
  log2 = Float11.log(2.0);
 }...}
```

$\mathbf{C_{colS}} : \mathbf{crate} \geq \mathbf{500} \wedge \mathbf{sc} \geq \mathbf{4096}$

$\mathbf{f_{remGet}} = (\mathbf{R_{remGet}}, \mathbf{I_{remGet}}, \mathbf{C_{remGet}})$
$\mathbf{R_{remGet}}$:Retrieve and view the fractal image from the server
$\mathbf{I_{remGet}}$:

```
public class RemoteViewer extends Canvas{...
  public void viewRemoteFractal(){
   this.image = getFractal(startTime*1000,maxExecutionTime);
   repaint();
  } ...}
```

$\mathbf{C_{remGet}} : \mathbf{mem} \geq \mathbf{100}$

Fig. 9. Application features

As far as the validation of the fractal application is concerned we only show the consistency based on context analysis (Def. 1). The validation phase creates the feature consistency table (Table 2) by checking the weak consistency for each feature at each context state in S. It evaluates the validity for the context requirements (predicates) of each feature by assigning all the possible context values. The table assigns value 1 if it is possible to select a feature in a certain context state and 0 otherwise. After defining the feature consistency table the context analysis phase creates the configuration consistency table (Table 3) by considering the features included in each configuration. This table contains value 1 only if all the features in a certain configuration are jointly weakly consistent at a certain context state. The process checks the validity of the joint predicate as shown in Section 3.4.

Table 1. System configurations

System Configuration	**Context Requirement**	**Offered Quality**
$G_1 = \{f_{genShot}, f_{colB}\}$	$mem \geq 300 \wedge cRate \geq 100$	$DisplayModel = Shot$ $ColorModel = BandOfColors$
$G_2 = \{f_{genShot}, f_{colNB}\}$	$mem \geq 300 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = NiceBandOfColors$
$G_3 = \{f_{genShot}, f_{colS}\}$	$mem \geq 300 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = SmoothyBandOfColors$
$G_4 = \{f_{genPro}, f_{colB}\}$	$mem \geq 200 \wedge cRate \geq 100$	$DisplayModel = Progressive$ $ColorModel = BandOfColors$
$G_5 = \{f_{genPro}, f_{colNB}\}$	$mem \geq 200 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = NiceBandOfColors$
$G_6 = \{f_{genPro}, f_{colS}\}$	$mem \geq 200 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = SmoothyBandOfColors$
$G_7 = \{f_{genImm}, f_{colB}\}$	$mem \geq 120 \wedge cRate \geq 100$	$DisplayModel = Immediate$ $ColorModel = BandOfColors$
$G_8 = \{f_{genImm}, f_{colNB}\}$	$mem \geq 120 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = NiceBandOfColors$
$G_9 = \{f_{genImm}, f_{colS}\}$	$mem \geq 120 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = SmoothyBandOfColors$
$G_{10} = \{f_{remGet}, f_{sockConn}\}$	$mem \geq 100 \wedge conn = 1$	$DisplayModel = Shot$ $ColorModel = BandOfColors$

Table 2. Feature consistency table

$C(mem, cRate, sc, conn)/f_j$	$f_{genShot}$	f_{genPro}	f_{genImm}	f_{colB}	f_{colNB}	f_{colS}	f_{remGet}	$f_{sockConn}$
$C_0 = (100, 200, 256, 0)$	0	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	1	1	1	0	1	1
...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	1	1	1	0	0	1	1
...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1

Table 3. Configuration consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1

5 System Evolution

Our development process supports the system evolution required by the context variations. In the following we show that in the foreseen evolution system and context models are queried to support the reconfigurations, while in the case of unforeseen evolution the same models may have to be refined as a consequence of incoming user needs.

5.1 Foreseen Evolution

In the foreseen evolution we consider only configurations that have already been proven weakly consistent. A monitoring process notifies the context variations which invalidate the context requirement belonging to the running configuration. Whenever such a new assignment of resources is discovered, the framework queries the configuration consistency table to get the possibly new admissible configurations. In order to perform the static decision-making process among weakly consistent configurations we take into consideration context and user preferences. Since we want to make our mechanism resilient to future contexts we take into consideration which is the probable future evolution for each context state. We consider the predictions for the user centric information (user task, user mobility) and the predictions for the evolution laws of resources obtained as explained in [40]. Exploiting such information we build a probabilistic automaton according to the approaches in [34,8]. Each different state corresponds to a different context and each arc expresses the probability to move from a context to another (e.g. Figure 10).

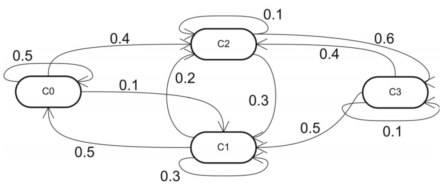

Fig. 10. Probabilistic evolution automata

If the user preferences are not fixed but they change over the execution we can include the possible preference variations within the automaton. Then we exploit the probabilistic model to evaluate the degree of suitability of each configuration to the context and to the user preferences. In [36] we have formalized and we have experimented a decision mechanism process that considers both factors within our probabilistic model.

5.2 Working Example

In the following we show a possible decision-making process that considers fixed user preferences and probable context evolutions in order to evaluate the overall fitness of each configuration G_i. Starting from the automaton in Figure 10 we evaluate the steady-state probability vector $\overrightarrow{p} = [0.2794\ 0.2794\ 0.2647\ 0.1765]$

which expresses how often the context belongs to a certain state. Then we obtain the *context fitness vector* by multiplying the vector $\overrightarrow{p}$ with the matrix m representing the configuration consistency table:

$$f = p \cdot m \tag{4}$$

This vector assigns a fitness value at each configuration that depends on the number of states in which the configuration is admissible and on the relevance for the states as evaluated by the steady-state probability vector. This ranking mechanism considers only how often the context belongs to a certain state whereas it ignores which is the current state and its future transitions thus leading to globally optimum solutions. Parallel to f we also evaluate a *user fitness vector t* expressing how each configuration is suitable with respect to the user preferences. We express preferences as weights over the quality attributes which characterize the variants. Each weight w_q (from 0 to 1) indicates the interest for the user towards a certain quality q. We use a predefined utility function $u_q(G_i)$ to assign a value from 0 to 1 at each quality dimension q provided by each G_i. The software engineer defines the utility functions and the weights for each quality since they are strictly application dependent. The *user fitness vector* is evaluated as:

$$t(G_i) = \sum_{q \in Qualities} w_q \times u_q(G_i) \tag{5}$$

Our decision-making process will consider together the user fitness $t(G_i)$ and context fitness $f(G_i)$ to evaluate the overall fitness of each configuration G_i.

Let us consider the scenario as depicted in Table 3 and let us suppose that the configuration $G_4 = \{f_{genPro}, f_{colB}\}$ is running at the context state $C_{43} = (350, 200, 4096, 1)$ whereas the user preferences assign higher weight to the $DisplayModel$ quality. The system is producing a fractal image drawing a row at a time and coloring pixels as bands of colors. Let us now suppose that because of a new application started on the mobile device, the current memory availability changes and the monitoring detects a context variation. By looking at the new context state $C_{33} = (150, 400, 4096, 1)$ in Table 3 we obtain the set of admissible (weakly consistent) configurations. Among them we select the one with the highest overall fitness. Therefore the current fractal application is stopped and it is evolved towards the configuration $G_7 = \{f_{genImm}, f_{colB}\}$ which represents the best trade-off between user and context fitness.

5.3 Unforeseen Evolution

Let us assume that during the execution phase the set of requirements the system needs to satisfy evolves because of changing user needs. For example the user has to deal with a new context situation that has not been foreseen by the software engineer at design time. Since a new behavior may have to be injected into the system it is necessary to modify at run-time the context-based decision table presented in the earlier sections. In addition also the models related to the system variability and context may have to be refined at run-time. Two different cases can arise: either a new requirement has to be added to the current configuration or an already existing requirement has to be deleted from the current configuration. We suppose that the requirement to add or to delete does not imply other requirements causing side effect phenomena to be managed. Thus, in order to evolve the application with a new requirement we augment the current selected configuration with a new feature implementing the new requirement. This leads to a new configuration that has not been anticipated at design time. Adding new requirements is more problematic than deleting requirements, thus we only discuss the first. Further, adding new behaviors seems to be appropriate for facing unforeseen situations.

In our approach we only evolve the current selected configuration whereas we do not consider how to augment the whole space of variants with the new requirement. We neither discuss how the addition of a new requirement to a configuration may affect the qualities attributes offered from the configuration.

The user may press a specific button within the application interface in order to communicate to the framework the variation of his/her needs. Then the user should specify the new requirement R_{New}, for example in natural language. The unforeseen evolution phase has to upgrade the running configuration with a new feature implementing the requirement R_{New}. We assume to have a search engine that given a requirement is able to return the set of features implementing it (exploration phase). Among them, we select the first feature $f_{New} = (R_{New}, I_{New}, C_{New})$ that is weakly consistent with the current running system configuration $G_F = (R_F, I_F, C_F)$ at the current context $\overrightarrow{c_{curr}}$:

$$(C_F \cup_C C_{new})[\overrightarrow{c_{curr}}/\overrightarrow{x}] \tag{6}$$

The integration phase creates the new configuration $G_F \cup_f f_{New}$ and the validation phase checks the weak consistency of the configuration at the current context state. The configuration is added to the configuration consistency table and since new resources may be required by the new feature it could be necessary to augment the context. Also the feature diagram is kept up-to-date by adding the incoming feature. We recall that in our approach, the integration of a new feature to the feature diagram only leads to a new configuration. We do not consider how to perform the integration of the new feature with all possible configurations since we only evolve the current configuration.

5.4 Working Example

Let us suppose that at the context state $C_{33} = (150, 400, 4096, 1)$, our framework completes a foreseen evolution for the fractal application. It puts in execution the configuration G_{10}=$\{f_{remGet}, f_{sockConn}\}$ which visualizes a precomputed fractal image after it has been downloaded from a remote server. The retrieved image complies to the TIFF image format. Because of unforeseen characteristics of the mobile device, the user cannot visualize the retrieved image. The device cannot decode TIFF images and therefore the fractal application has to be upgraded.

To this end, the user interacts with the framework to add a new requirement in the application. After accessing to the upgrading wizard, he/she specifies the new requirement in natural language:

$$R_{New} = \textit{The system shall visualize TIFF format images} \tag{7}$$

This requirement has not been foreseen at design time but arises only at run-time when the unforeseen device characteristics (context) make the fractal visualization impossible. Thus after the evolution process, we have to re-iterate the exploration, integration and validation phases at run-time in order to evolve the application with the feature (i.e. the software codec) to view TIFF format images. This will lead to a new configuration with same features of the current configuration plus the new feature.

The exploration phase queries the search engine in order to retrieve a feature which implements the new requirement, e.g. see Figure 11.

$I_{tiffViewer}$:
```
public class Viewer{ ...
 public RenderOp tiffViewer(Object stream){
  ParameterBlock params = new ParameterBlock();
  params.add(stream);
  TIFFDecodeParam decodeParam = new TIFFDecodeParam();
  RenderedOp image = JAI.create("tiff", params);
  return image;
}...}
```
$C_{tiffViewer}$: $cRate \geq 300 \wedge mem \geq 35$

Fig. 11. Example: new feature

The integration phase augments the feature diagram with the new feature as shown in Figure 12. An optional feature $f_{tiffViewer}$ is added to the diagram which only leads to a new configuration $G_{New} = \{f_{remGet}, f_{tiffViewer}, f_{sockConn}\}$.

For the validation phase we consider how the new context requirement affects the context requirements provided at design time. The new context requirement $C_{tiffViewer}$, that we consider for weak consistency, refers to the resources *cRate* and *mem* which have been already foreseen at design time; thus a context model extension is not required. To establish if the new configuration $G_{New} = G_{10} \cup_f f_{tiffViewer}$ is weakly consistent we evaluate the new context requirement jointly with the context requirement for G_{10}, i.e.:

$$C_{New} = cRate \geq 300 \wedge mem \geq 135 \wedge conn = 1$$

This predicate is true at the context state C_{33} since this state provides enough memory, cpu speed and an Internet connection. Only if the new predicate is false the framework does restart the evolution process by the exploration phase in order to consider other features. Finally, if the configuration is weakly consistent (the new predicate is true with the current context values), the validation phase adds the new configuration G_{New} to the consistency table as shown in Table 4 by checking the weak consistency property also for the other context states.

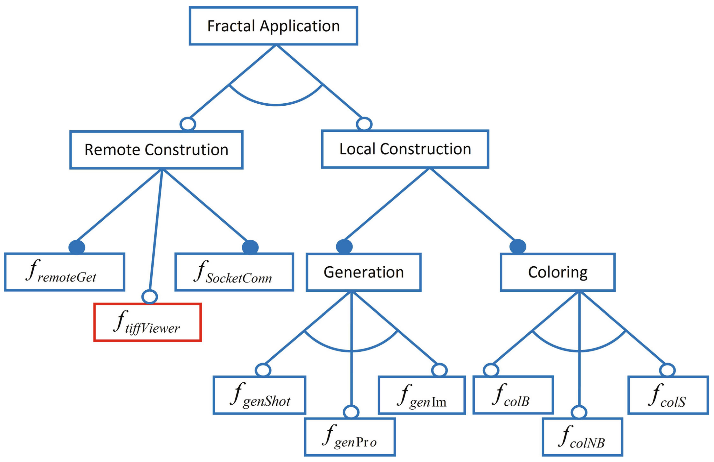

Fig. 12. Refined feature diagram

Even if it is not shown in the example, a new feature may also require new unforeseen context entities in its context requirements. Thus, it may be necessary to refine also the context model in order to consider the values for the new resources. As a consequence it would be also necessary to augment the consistency table with the new context states arising from the augmented context space.

Table 4. Refined configuration consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{New}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1	1

6 Evolution Framework Architecture

In Figure 13 is shown the architecture that can be implemented by any evolution framework in order to support the development and the execution of adaptive applications. This architecture implements the MAPE cycle as described in Section 1 and it supports our lifecycle process for context-aware adaptive systems. The application, configuration and feature blocks represent the basic components. They enable the definition of the application along with its variability. The context manager component is able to monitor the resources and to manage their definitions and values by accessing to the context model component. It performs the monitoring phase and it triggers the required evolution phases. The decision-making component maintains the context-based tables and the probabilistic automaton in order to support the decision-making mechanisms; it also supports the consistency checking phase and the ranking process for the configurations. A component for each kind of evolution is provided in the framework. As shown by the arrows, while the foreseen evolution accesses the decision-making component to select the most suitable configuration, the unforeseen evolution interacts with the user who specifies variation to the requirements. Finally the execution component enacts the system reconfiguration for both evolutions.

6.1 Framework Instantiation

We have instantiated the architecture in Figure 13 by exploiting current practices technologies available in the literature. We represent requirement R as Linear Time Temporal Logic expressions [39], whereas we represent the context requirements as predicates. We evaluate the context states in which a configuration is admissible by formalizing and solving a Constraint Satisfaction Problem (CSP) [35] by using the Java API available with the JaCoP tool[1]. Implementation artifacts are coded in Java, thus making it possible to verify the implementation components I with respect to the requirement R. To this end, our approach proposed in [23] defines a model checking phase which exploits the Java Path Finder tool[2].

Our framework supports the foreseen evolution by deciding which is the most suitable variant to execute whenever the current context state makes the running configuration not anymore admissible. To this end, the framework stops the execution of the running configuration and it puts the target variant in execution. Our earlier approach [36] describes how to select the most suitable variant based on the trade-off between user benefit and reconfiguration cost. The framework presented in this paper also supports the unforeseen evolution by exploiting a mechanism for the dynamic loading of Java classes. The user interacts with the application to specify a new requirement in a similar way as a programmer can add a new plug-in to the Eclipse or NetBeans IDE. If there is no configuration that can satisfy the augmented set of requirements, then the framework searches for a feature which implements the new requirement by interacting with a remote library of features. It creates a new configuration that contains the same set of features of the current configuration plus the new feature. The framework checks if the new set of features is free from interactions by evaluating context requirements. Once the framework has found such a new feature it gives as result the implementation for the new configuration. The framework supports the code replacement for the configuration and a mechanism for re-loading the new compiled classes (based on Javeleon[3]). Finally, the new configuration will be enacted trough its entry point method.

[1] `http://jacop.osolpro.com/`
[2] `http://babelfish.arc.nasa.gov/hg/jpf/jpf-core`
[3] `http://javeleon.org/`

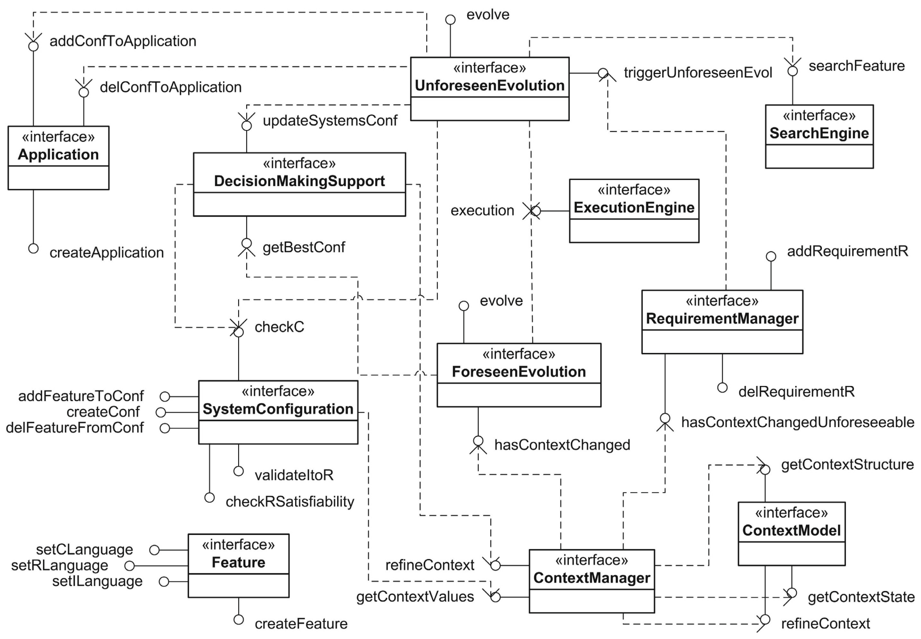

Fig. 13. Evolution framework architecture

7 Conclusion and Future Work

We have defined a generic model-centric software lifecycle process for context-aware adaptive systems. Our process supports concrete mechanisms to achieve consistent evolution both at design time and at run-time through a static and a dynamic decision-making procedure. We have proposed feature-oriented models to represent the system along with its variability and we have modeled context entities as the basis for the notion of weak consistent evolution.

We have defined a generic evolution framework in order to support the software process for adaptive systems. We have implemented a possible instance of the evolution framework applying current practice technologies.

As for future work, we will carry out extensive experimentations in order to evaluate advantages and disadvantages of adopting the framework to develop adaptive applications.

Acknowledgments. This work has been partially supported by the EU IST CONNECT (http://connect-forever.eu/) No 231167 of the FET - FP7 program and the EU IST CHOReOS (http://www.choreos.eu/) No 257178 of the FP7 program.

References

1. Alférez, M., Moreira, A., Kulesza, U., Araújo, J., Mateus, R., Amaral, V.: Detecting feature interactions in spl requirements analysis models. In: FOSD, pp. 117–123 (2009)
2. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15(4), 439–458 (2010)
3. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: SEAMS, pp. 27–47 (2009)
4. Autili, M., Di Benedetto, P., Inverardi, P.: Context-Aware Adaptive Services: The PLASTIC Approach. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 124–139. Springer, Heidelberg (2009)
5. Autili, M., Benedetto, P.D., Inverardi, P.: Hybrid approach for resource-based comparison of adaptable java applications. Journal of Science of Computer Programming (SCP) - Special issue of BElgian-NEtherlands software eVOLution seminar (BENEVOL) on Software Evolution, Adaptability and Maintenance (2012)
6. Autili, M., Cortellessa, V., Ruscio, D.D., Inverardi, P., Pelliccione, P., Tivoli, M.: Eagle: engineering software in the ubiquitous globe by leveraging uncertainty. In: SIGSOFT FSE, pp. 488–491 (2011)
7. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. IJAHUC 2(4), 263–277 (2007)
8. Berardinelli, L., Cortellessa, V., Di Marco, A.: Performance Modeling and Analysis of Context-Aware Mobile Software Systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 353–367. Springer, Heidelberg (2010)
9. Bisbal, J., Cheng, B.H.C.: Resource-based approach to feature interaction in adaptive software. In: WOSS, pp. 23–27 (2004)
10. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42(10), 22–27 (2009)
11. Brataas, G., Hallsteinsen, S.O., Rouvoy, R., Eliassen, F.: Scalability of decision models for dynamic product lines. In: SPLC (2), pp. 23–32 (2007)
12. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
13. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
14. Classen, A., Heymans, P., Schobbens, P.-Y.: What's in a *Feature*: A Requirements Engineering Perspective. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 16–30. Springer, Heidelberg (2008)
15. Czarnecki, K., Eisenecker, U.W.: Generative programming: Methods, Tools and Applications. Addison-Wesley (2000)
16. Dorn, C., Dustdar, S.: Weighted fuzzy clustering for capability-driven service aggregation. In: SOCA, pp. 1–8 (2010)
17. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
18. Ghezzi, C., Inverardi, P., Montangero, C.: Dynamically Evolvable Dependable Software: From Oxymoron to Reality. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 330–353. Springer, Heidelberg (2008)

19. Glinz, M.: On non-functional requirements. In: RE, pp. 21–26 (2007)
20. Gregersen, A.R., Jørgensen, B.N.: Dynamic update of java applications - balancing change flexibility vs programming transparency. Journal of Software Maintenance 21(2), 81–112 (2009)
21. Hong, J., Suh, E., Kim, S.-J.: Context-aware systems: A literature review and classification. Expert Syst. Appl. 36(4), 8509–8522 (2009)
22. Inverardi, P., Mori, M.: Feature oriented evolutions for context-aware adaptive systems. In: EVOL/IWPSE, pp. 93–97 (2010)
23. Inverardi, P., Mori, M.: Model checking requirements at run-time in adaptive systems. In: Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems, ASAS 2011, pp. 5–9 (2011)
24. Inverardi, P., Mori, M.: Requirements models at run-time to support consistent system evolutions. In: Proceedings of the 2nd International Workshop on Requirements@Run.Time, pp. 1–8 (2011)
25. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009)
26. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
27. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21 SEI Carnegie Mellon University (1990)
28. Kapitsaki, G.M., Prezerakos, G.N., Tselikas, N.D., Venieris, I.S.: Context-aware service engineering: A survey. JSS 82(8) (2009)
29. Keck, D.O., Kühn, P.J.: The feature and service interaction problem in telecommunications systems. a survey. IEEE TSE 24(10), 779–796 (1998)
30. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Software Eng. 16(11), 1293–1306 (1990)
31. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE, Washington, DC, USA, pp. 259–268 (2007)
32. Mancinelli, F., Inverardi, P.: A resource model for adaptable applications. In: SEAMS, New York, NY, USA, pp. 9–15 (2006)
33. Mandelbrot, B.: The fractal geometry of nature. Freeman (1982)
34. Marco, A.D., Mascolo, C.: Performance analysis and prediction of physically mobile systems. In: WOSP, pp. 129–132 (2007)
35. Marriott, K., Stuckey, P.: Programming with Constraints: An introduction. MIT Press (1998)
36. Mori, M., Li, F., Dorn, C., Inverardi, P., Dustdar, S.: Leveraging State-Based User Preferences in Context-Aware Reconfigurations for Self-Adaptive Systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 286–301. Springer, Heidelberg (2011)
37. Osterweil, L.: Software processes are software too. In: ICSE, Los Alamitos, CA, USA, pp. 2–13 (1987)
38. Parra, C., Cleve, A., Blanc, X., Duchien, L.: Feature-Based Composition of Software Architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 230–245. Springer, Heidelberg (2010)
39. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
40. Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., Sousa, J.: Leveraging resource prediction for anticipatory dynamic configuration. In: SASO, Washington, DC, USA, pp. 214–223 (2007)

41. Pukall, M., Grebhahn, A., Schröter, R., Kästner, C., Cazzola, W., Götz, S.: Javadaptor: unrestricted dynamic software updates for java. In: ICSE, pp. 989–991 (2011)
42. Qureshi, N., Perini, A.: Requirements Engineering for Adaptive Service Based Applications. In: RE, pp. 108–111 (2010)
43. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009)
44. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for re for self-adaptive systems. In: RE, pp. 95–103 (2010)
45. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)
46. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Software Eng. 33(12), 856–868 (2007)
47. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE, New York, NY, USA, pp. 371–380 (2006)
48. Zowghi, D., Gervasi, V.: The three cs of requirements: Consistency, completeness, and correctness. In: REFSQ (2002)