# Evolutionary Software Life Cycle for Self-Adapting Software Systems

Ahmed Ghoneim, Sven Apel and Gunter Saake

*Department of Computer Science,*
*Otto-von-Guericke-University, Magdeburg, Germany*
*Email: {ghoneim|apel|saake}@iti.cs.uni-magdeburg.de*

Keywords: Reflection, Runtime Adaptation, Software Evolution, Meta-Data Techniques, UML.

Abstract: Reactive non-stoppable software systems perform tasks continually to face environmental changes. These changes require for adapting strategies of the set of behaviors, or to add new ones according to the ability of the underlying hardware capabilities. Current approaches to runtime adaptation focus only on information which is provided by the implementation phase. Design information is not involved, even though it is extremely useful for adaptation, because they holds the semantics of the regarding software system. We present an evolutionary life cycle for self-evolving software systems by combining the traditional spiral life cycle model, known from software development with a reflective architecture which processes design information. This allows the iterative evolution of software systems at runtime. The reflective architecture (especially the meta-level) evolves the behavior and structure of the software system using its design information. Furthermore, we propose the concept of meta-feedback to react flexibly on the continuous changes of the environment. The proposed life cycle practicability is illustrated through a case study, a robot control software.

## 1 Introduction

A major issue in the software engineering research area is to develop software systems which can be adapted and customized to fit the environment and requirements. A special class, reactive non-stoppable software systems, needs to autonomously adapt itself to runtime changes of the requirements and runtime events, occurring in the environment. One prominent approach to implement self-adaptable software systems are reflective architectures (Maes, 1987; Dowling and Cahill, 2001; Roman et al., 2001). Reflective architectures provide information about their internal structure and behavior. This information can be inspected and adapted to react to changes of the environment or application requirements.

Our approach is based on our previous work on *architectural reflection* (Cazzola et al., 2002; Cazzola et al., 2004). Architectural reflection map the knowledge from the design phase, e.g. UML diagrams, to the running phase of the application, represented as meta-data. This mapping makes it easier to decide which runtime events are relevant and how to adapt. The contribution of this article is the combination of architectural reflection with the spiral life cycle model for software development (cf. Sec. 2): In short, the spiral software life cycle evolves a software system by passing through different development phases (e.g. analysis, design, implementation, etc.) iteratively several times. Each phase modifies and evolves the software system. Our approach transfers this iterative evolution to runtime. Doing so, we combine the spiral life cycle model with a reflective architecture and an evolution based on design information (architectural reflection). This makes software evolution possible at runtime, in order to adapt to changes of the environment and requirements.

To accomplish that, we divide the software system into a base-level and a meta-level: The base-level contains the running application and harvests and stores the design information. The design information is represented as *UML/XMI documents* (OMG, 2002). The meta-level evolves and validates the systems behavior. It reacts to runtime changes, indicated by runtime events, by triggering meta-cycles. Each meta-cycle evolves and validates the behavior according to one runtime event. The evolution and validation takes place in the UML/XMI document using graph transformation (cf. (Cazzola et al., 2004)). If multiple runtime events occur the meta-cycles are processed sequentially, one by one. To reduce the gap between the currently processed meta-data and the base-level state, the meta-data are updated between two cycles at a time. The updates, named *meta-feedback*, are reified continuously from the base-level. If no more runtime events occur, the meta-data are propagated back to the base-level. Having done so, the behavioral changes are applied to the base application at runtime.

The remaining paper is organized as follows: Section 2 provides a brief overview of the methods and tools we have adopted in our work; Section 3 describes the evolutionary life cycle; To exemplify, Section 4 applies the proposed life cycle to a robot control software. Finally, Section 5 concludes.

## 2 Background

## 2.1 Computational Reflection

Computational reflection (Maes, 1987) is a technique for inspecting the current structure and behavior of a software system. When using reflection, the system is able to reason about its own behavior and perform selected changes at runtime. A reflective system is divided into two levels, a base-level and a meta-level. The base-level is the part of the system that performs processing for the application, for instance controlling transaction executions. The objects at the base-level provide meta-interfaces that at the meta-level gives access to the internal representation of the system. The services provided by the meta-interfaces, often referred to as the meta-object protocol (MOP), allow inspection and modification of system behavior and structure (Kiczales et al., 1991).

Reflection is a technique that allows a software system to maintain information about itself (*meta-information*) and using this information to change (adapt) its behavior. This is implemented by a causal connection between base- (*monitored system*) and meta-level (*monitoring system*).

### 2.1.1 Architectural Reflection

In (Cazzola et al., 2002; Cazzola et al., 2004) we have proposed an infrastructure that dynamically adapts software systems using *architectural reflection*. The key idea of architectural reflection is to reuse the knowledge of the design phase to evolve and validate the structure and behavior of a software system at runtime. The base-level consists of the running application as well as of design information in form of UML/XMI documents. The meta-level is composed of an interpreter engine for managing the evolution and validating consistency processes for runtime changes. The evolution and validation is based on graph transformation (Cazzola et al., 2004) which takes place on the reified design information (UML/XMI documents).

## 2.2 Software Engineering Models

Software engineering models define how to develop a software system. There are two schools of thought in software engineering: first, linear thinking fostered by the *waterfall life cycle* (Royce, 1970). The waterfall life cycle is divided into sequential phases analysis, design, implementation and testing phase, which evolve the software system sequentially. Second, iterative or evolutionary thinking, fostered by the *spiral life cycle model* (Boehm, 1988; Cotton, 1996; Gilb, 1988). It divides the software engineering space into four quadrants: management planning, formal risk analysis, engineering, and customer assessment. The development cycle is divided into n-cycles. Each cycle consists of waterfall phases. The result of each cycle is a prototype, until the final version of the product is reached. Furthermore, user feedback is considered at the output of each cycle. Therewith, new additional features are included into the next cycle.

## 3 Evolutionary Life Cycle for Dynamic Adaptation

The structure of the evolutionary life cycle is based on the spiral life cycle model. The life cycle is composed of two cycles: the base-cycle and one or more meta-cycles, as shown in Figure 1. The processes of the base-cycle execute in sequential flow. At the base-cycle, the *base-engine* extracts the design information in form of UML/XMI documents from the base application to constitute the base-data. Adopting the ideas of the spiral life cycle model, we instantiate multiple meta-cycles, each for one runtime event. These meta-cycles evolve and validate the behavior (using the reified design information) in order to react to the runtime events. The processes of the meta-cycles execute in an incremental flow. The meta-processes start by the reification of the base-data to constitute the meta-data at meta-level. Runtime events trigger the instantiation of meta-cycles. Each meta-cycle evolves the meta-data representation of the design information using the evolutionary engine. The evolutionary phase inside a meta-cycle is responsible for building new meta-data including the runtime changes. Afterwards, the validation phase checks the consistency (*consistency checker*) of the evolved meta-data according to the effects of the runtime events. If a runtime event occurs during a running meta-cycle, the event is cached and a new cycle is instantiated after

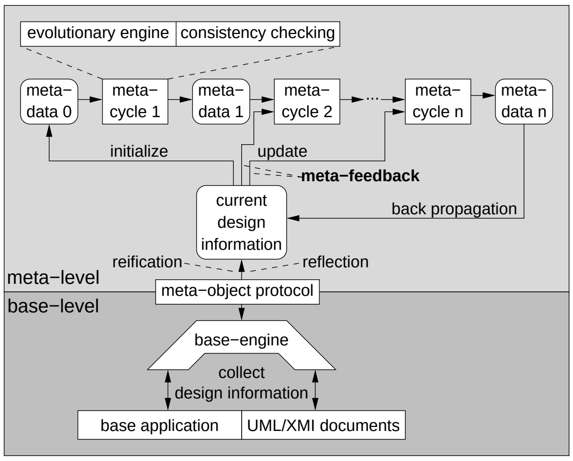


Figure 1: The Evolutionary Life Cycle.

the former cycle has finished. It starts after the finishing of the previous one. From the predecessor cycle each meta-cycle gets the resulting meta-data as input. To minimize the gap between the evolution phase using multiple meta-cycles and the base-level we propose the concept of *meta-feedback*. As mentioned this idea is adopted form the spiral life cycle model. The input meta-data of a new meta-cycle have to be merged with the current base-data. Finally, if no new event occurs in a defined time interval the modified meta-data are reflected back to the base-level. In the following we discuss both cycle types in more detail:

**Base-cycle.** The base-cycle includes two linear phases: The first phase contains all base implementation units as requirements; that includes objects, attributes, methods, states, and their graphical representation by using UML. By using the base-engine the design information will be extracted. The second phase: includes the UML/XMI documents. UML/XMI provides a simple translation of UML diagrams which are more suitable for runtime manipulation. The internal representation of the base cycle is illustrated in Algorithm 1.

```
Data   : objects, states, collaborations, UML representations
Result : UML/XMI schemas
repeat
    if object, collaboration or state has changed then
        create new UML/XMI document;
        provide documents via meta-object protocol;
    end
until base application is finished;
```

**Algorithm 1**: Base-cycle flow

**Meta-cycle.** The meta-cycles can access design information of the base-level (*base-data*) using the meta-object protocol. First, UML/XMI documents are reified to constitute the meta-data. In fact they are transferred from base- to meta-level at runtime. Afterwards, multiple meta-cycles are instantiated, each for processing one runtime event. If a runtime event occurs during a running meta-cycle, the event is cached. When the meta-cycle has finished, a new cycle is instantiated. This new cycle processes the cached event. Each meta-cycle evolves and validates the meta-data in order to react to runtime events, while the base-level system runs in parallel. After each meta-cycle the resulting meta-data are merged with the updated base-data, received from the base-level. This reduces the gap between the time used for the meta-cycle to modify reified data and the changes for the running base application during the meta-cycle processes. After the last meta-cycle the meta-data are propagated back to the base-level. The internal representation of the meta-cycles is illustrated in Algorithm 2.

Evolution and validation as well as the merging and update operations are based on graph transformations. These take place on the UML/XMI documents (cf. (Cazzola et al., 2004)). The following section describes the UML operators which work on UML/XMI documents and the overall evolutionary life cycle.

```
Data   : meta-data, runtime events
Result : new meta-data includes the runtime changes
repeat
    if meta-data reified then
        meta-level starts its processes to evolve and
        validate consistency for the reified meta-data;
        update meta-data;
        for each runtime event do
            evolutionary engine evolve the meta-data;
            validate the modified data by using the
            consistency checker;
        end
    end
until runtime events detected;
reflect modified meta-data back to the base-level;
```

**Algorithm 2**: Meta-cycle flow

## 3.1 Formalized Design Information

This section introduces two operators, which are used in the evolutionary life cycle to process design information: (1) the UML update operator which updates one UML/XMI document using up-to-date design information. This operator is used to merge the reified meta-data with the meta-feedback. (2) The UML intersection operator, which processes the intersection between two UML/XMI documents is used to get insight if consistency problems occur. To describe these operators and clarify the internal processes, we formalize the specification of the design information (UML/XMI documents):

**General definitions.** Starting point is the general definition of a *tree*: $G = \{V, E\}$. The nodes of the graph correspond to all possible tags of UML/XMI documents (OMG, 2002), except the relations (e.g. inheritance, association, etc.). The edges correspond to the relations between the UML entities (e.g. classes, objects, etc.). In the following we call these graphs UML trees. We define two operations on UML trees:

**UML intersection operator.** This operator determines the intersection of two UML trees. The result is a tree with the intersection of the node and edge set. To allow empty intersections the root node $v_r$ is left (the root node of UML/XMI documents is not an UML entity. It serves only as root for all UML entities.):

$$G_1 \ominus G_2 = \{(V_1 \cap V_2) \setminus \{v_r\}, \{e \mid e \in E_1 \cap E_2 \wedge e \notin \{v_r\} \times (V_1 \cup V_2)\}\} \quad (1)$$

**UML update operator.** The UML update operator defines how two UML trees are merged. Thereby, the first operand is dominant. That means, if in both node or edge sets equal elements are included, the element of the first operand is chosen to become part of the resulting set. Therefore this operand is not commutative:

$$G_1 \oplus G_2 = \{\{v, w \mid v \in V_1, w \in V_2 \wedge w \notin V_1\}, \{e, h \mid e \in E_1, h \in E_2 \ \wedge \ h \notin E_1\}\} \quad (2)$$

The intersection operator is used to find conflicts between two versions of design information, e.g. the currently processed meta-data conflict with the currently occurred changes at the base-level (meta-feedback). The update operator is used to merge the meta-data and the meta-feedback.

## 3.2 Formal Description of the Evolutionary Life Cycle

The evolutionary life cycle consists of several meta-cycles ($\Phi_i$ with $i = 0, 1, \ldots, n$ and $n \in \mathbb{N}$). Each meta-cycle gets reified base-data in form of an UML tree as argument. It modifies/evolves and validates these input data according to its corresponding runtime event. If a meta-cycle is active and a runtime event occurs, it is interrupted and a successor cycle is invoked. All meta-cycles except the first ($\forall \Phi_i$ with $i > 0$) get the modified data as input from its predecessor cycle. We define a meta-cycle $\Phi_i$ as a function which transforms the input graph $G_i$ to $G'_i$:

$$G'_i = \Phi_i(G_i) \quad (3)$$

The input data of the first meta-cycle ($\Phi_0$) is the reified (by the base-engine) UML tree which describes the current base-level state. For every other meta-cycle ($\Phi_i$, with $i \neq 0$) the input tree is the result of $\Phi_{i-1}$ merged with new reified UML tree $U_i$ of the base-level (meta-feedback) using the UML update operator:

$$G_i = U_i \oplus G'_{i-1} \quad (4)$$

In the following a sequence of two meta-cycles is depicted:

$$\begin{aligned} G'_i &= \Phi_i(G_i) \\ G_{i+1} &= U_{i+1} \oplus G'_i \\ G'_{i+1} &= \Phi_{i+1}(G_{i+1}) \end{aligned} \quad (5)$$

A consistency problem occurs if the resulting tree ${G_i}'$ of meta-cycle $\Phi_i$ overlaps the tree $U_{i+1}$ with updated base-level state. The problem can be found out using the UML intersection operator. In the current form UML update operator replaces all nodes and edges of ${G_i}'$ with equal ones of $U_{i+1}$. Several others are thinkable.

To clarify the functionality of the evolutionary cycle we present the required algorithm for the presented algebraic operations:

**UML intersection operator.** The intersection operator gets two UML trees as input and returns a graph as output which contains all nodes and edges which are in both input trees except of the root node $v_r$ and the edges which are connected to $v_r$. The algorithm is depicted in Algorithm 3:

```
Data   : firsttree, secondtree
Result : resulttree
for each node ∈ firsttree do
    if node ∈ secondtree and node ≠ $v_r$ then
        add node to resulttree;
    end
end
for each edge ∈ firsttree do
    if edge ∈ secondtree and edge is not connected to $v_r$ then
        add edge to resulttree;
    end
end
```

**Algorithm 3**: Algorithm of the UML intersection operator

**UML update operator.** The update operator gets two UML trees as input and returns a tree whose nodes and edges belong to the first input tree and the nodes and edges which belong to the second input graph and are not part of the first input graph. The corresponding algorithm is depicted in Algorithm 4:

```
Data    : firsttree, secondtree
Result  : resulttree
for each node ∈ firsttree and edge ∈ firsttree do
    add node and edge to resulttree;
end
for each node ∈ secondtree and edge ∈ secondtree do
    if node and edge ∉ firsttree then
        add node and edge to resulttree;
    end
end
```

**Algorithm 4**: Algorithm of the UML-update operator

**Evolutionary life cycle.** The evolutionary life cycle reifies base-data (see Alg. 5). In the main running phase it instantiates multiple meta-cycles (*init*). Runtime events can occur during a running meta-cycle. In this case the event is cached and after the finished cycle an new one is instantiated. The new cycles get the meta-data merged with the updated base-data as input (*update*). If the last meta-cycle has finished the meta-data are passed back to the base-level.

```
Data    : event, reified inputtree
while event && current metacycle not finished do
    init(metacycle, event);
    modifiedtree = startmetacycle(metacycle,
    inputtree);
    if metacycle not finished then
        inputtree = update(reify(), modifiedtree);
    end
end
```

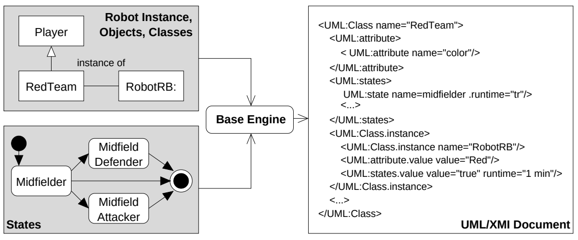
**Algorithm 5**: Algorithm of the evolutionary life cycle

## 4 Case Study: Robot Control Software

Robot control software is one representative example for a reactive non-stoppable software system. Robots interact strongly with their environment and react continuously to external changes and events. In this section we discuss how to apply the proposed evolutionary life cycle to robot systems, e.g. working Sony Legged Robots (SLR) (Fujita et al., 1999). For explanation, we use a robot soccer scenario: Each robot team consists of three robots including a goalkeeper. Robots can move, shoot and have sensors to observe the field, the ball and the other robots.

To clarify the appliance of our approach to this scenario, we discuss the internal phases of the evolutionary life for robot control software:

**Base-cycle.** In Figure 2, we illustrate the representation of the UML diagrams for a robot. As one can see design information consists of class/object diagrams and state charts. This design information is transformed by the base-engine to UML/XMI documents. The meta-level can access these documents using the meta-object protocol.



```
<UML:Class name="RedTeam">
  <UML:attribute>
    < UML:attribute name="color"/>
  </UML:attribute>
  <UML:states>
     UML:state name=midfielder .runtime="tr"/>
    <...>
  </UML:states>
  <UML:Class.instance>
    <UML:Class.instance name="RobotRB"/>
    <UML:attribute.value value="Red"/>
    <UML:states.value value="true" runtime="1 min"/>
  </UML:Class.instance>
  <...>
</UML:Class>
```

Figure 2: A robots base-cycle

**Meta-cycle.** When the base-engine has finished, the extracted data will be reified to constitute the meta-data. In Figure 3, we illustrate the structure of meta-cycles in detail. The evolutionary controller waits for runtime events, e.g., the ball or the goal is near. If a runtime event occurs the meta-cycle evolves the system behavior, e.g., pass the ball to another robot. If there is no new runtime event the modified meta-data are reflected back to the base-level and therewith the behavior of the robot changes. But in dynamic environments like the robot soccer scenario, runtime events occur frequently. Because of this fact, we cache runtime events which occur during a running meta-cycle. After the cycle has finished, a new one is instantiated to react on the new event, e.g. the shooting line to the goal is blocked. To take the current base application state into account, e.g. new knowledge about the robot player distribution over the field, the output meta-data from the finished cycle are merged (using the UML update operator) with the meta-feedback from the base-level. These merged data are the input of the new cycle. If no more event occur the meta-data are reflected back to the base-level.

For a better understanding, imagine the following example: A robot senses an opponent robot near the ball. What is the best behavior for the robot? The robot can use the following knowledge in form of meta-data to make decisions:

- position of the ball, and distance to the ball
- distribution of the robots over the field
- distance to the goal and opponents

Based on this knowledge about the environment, a robot can make several decisions, e.g.:

- move to the ball autonomously
- let another team member move
- block opponents to reach the ball
- move to a defender position

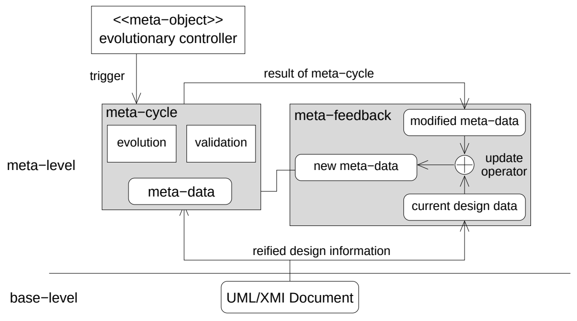


Figure 3: Evolutionary life cycle and meta-feedback concept

```
Data      : OppPos, BallPos, GoalPos, MyPos
if dist(OppPos, BallPos) < dist(MyPos, BallPos) then
    move to defender position; switch to defender state;
else
    if fireline is blocked then
        pass to nearest team member;
        switch to attacker state;
    else
        shoot the ball; switch to kick-off state;
    end
end
end
end
```

**Algorithm 6**: Rules to adapt the behavior of a robot.

Algorithm 6 shows possible rules to make a decision, based on the reified sensor information. If the opponent is closer to the ball, the robot moves to a defender position to block the offense. If the robot itself is closer to the ball, it has two alternatives: If the fire line to the goal is blocked he passes the ball to a team member; if there is no opponent robot in the line to the goal it can shoot directly. It can be seen that the robot uses reified sensor information to make decisions. Moreover, robots use the reified design information to switch between states (defender, attacker, kick-off).

## 5 Conclusion

We have proposed an evolutionary life cycle that is suitable for self-adapting object-oriented information systems. The key idea of our approach is to combine the concepts of the spiral life cycle model and architectural reflection, to map the ability of continuous evolution from the development phase to runtime execution. In this context, we have proposed the meta-feedback concept to update the meta-data iteratively in order to consider the current base-level state. Furthermore, we have illustrated the formal description of evolutionary life cycle. Finally, we have presented an application of our evolutionary life cycle on a working Sony Legged Robot (SLR).

In future work, we will provide a consistency runtime formal framework for all phases of the proposed life cycle as well as extensive experiments.

## REFERENCES

Boehm, B. (1988). A Spiral Model for Software Development and Enhancement. *IEEE Computer*, 21(5).

Cazzola, W., Ghoneim, A., and Saake, G. (2002). Reflective Analysis and Design for Adapting Object Run-time Behavior. In *Proc. of the 8th Int. Conf. on Object-Oriented Information Systems (OOIS'02)*.

Cazzola, W., Ghoneim, A., and Saake, G. (2004). Software Evolution through Dynamic Adaptation of Its OO Design. In *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, LNCS. Springer-Verlag.

Cotton, T. (1996). Evolutionary Fusion: A Customer Oriented Incremental Life Cycle for Fusion. In *Hewlett-Packard Journal*.

Dowling, J. and Cahill, V. (2001). The K-Component Architecture Meta-Model for Self-Adaptive Software. In *Proc. of 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01)*.

Fujita, M. et al. (1999). Vision, Strategy, and Localization Using the Sony Legged Robots at RoboCup-98. *AI Magazine*.

Gilb, T. (1988). *Principles of Software Engineering Management*. Addison-Wesley.

Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*.

OMG (2002). OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications.

Roman, M., Kon, F., and Campbell, R. (2001). Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online (Special Issue on Reflective Middleware)*, 2(5).

Royce, W. W. (1970). Managing the Development of Large Software Systems: Concepts and Techniques. In *Proc. of WESCON*.