Evolutionary Software Life Cycle
for Self-Adapting Robot Software Systems

Keywords: Dynamic Architecture, Adaptability, Evolution, Software Development Models, UML.

Abstract: Robot software systems perform tasks continually to face environmental changes. These changes in the environment require to adapt the strategies of the set of behaviors or to add new ones according to the ability of the robot’s hardware capabilities. We present an evolutionary life cycle for self-evolving robot software systems. The life cycle applies within a reflective architecture, that provides the ability to automatically trap the design information in form of UML/XMI documents of the base-level systems. The life cycle is composed of two cooperating cycles: the base-cycle which includes the running application and base-engine for getting the internal representation; and the meta-cycle which provides the adaptation engine for the base application. The evolutionary life cycle main features are highlighted as follows: First, it allows to extract the robots design information from UML models. Second, by using MOP capability the extracted data are trapped to constitute the meta-data. Third, incremental meta-cycles are applied to evolve and validate runtime changes. Finally, the modified meta-data are reflected to the base application and leaves it consistent with these changes. The proposed life cycle practicability is illustrated through a case study.

1 Introduction

A robot’s software system nowadays needs to autonomously adapt itself for the runtime environmental changes. The robot’s behavior changes during runtime by adding new ones or modifying the strategy for existing ones. A topical issue in software engineering research area is to produce software systems able to adapt themselves to environmental changes by adding new and/or modifying existing functionalities. During a Robot’s system lifetime, we want to be able to dynamically adapt the software of these systems for runtime events. Reflective system architectures have been built without the application of software engineering models. Some software engineering techniques have been used to assist the development of the reflective systems such as design patterns (Gamma et al., 1995; Buschmann et al., 1996), and aspect-oriented programming (Kiczales et al., 1997).

In (Cazzola et al., 2002b; Cazzola et al., 2004) we have proposed an infrastructure to dynamically adapt software systems. In our approach, we have presented a reflective system, that allows to render self-adaptable at runtime the base-level system. The meta-level systems is composed of an interpreter engine for managing the evolution and validating consistency processes for runtime changes. The meta-level behavior is described by a family of patterns (Cazzola et al., 2002a), the meta-level manages both the evolution and consistency of the base-level system.

A software life cycle model is an illustrative characterization of how software is or should be developed. We present an evolutionary life cycle for the robot to self adapt its behavior with the current state environment. The proposed life cycle defines two cooperative cycles (base and meta) through a reflective architecture. The base-cycle involves the engine to extract the data from the running base application. The meta-cycle involves the engine to evolve and validate consistency semantics at runtime. The meta-cycle involves the engine to extract the data from the running base application. The meta-cycle involves the engine to evolve and validate consistency semantics at runtime. The meta-cycle involves the engine to extract the data from the running base application. The meta-cycle involves the engine to evolve and validate consistency semantics at runtime.

Each meta-cycle is responsible for adapting and validating the reified meta-data for runtime events. The output of each meta cycle is the input for the next cycle. The state of the base level-system is changing while the meta-cycle is running it’s processes. So there the new base data must be merged with the out-
put data from the meta-cycle before using it as a new meta-data for the next cycle. This is done to reduce the gap between the time used for the meta-cycle to modify reified data for runtime events and the changes for running base application during the meta-cycle processes. We are using the meta-feedback concept for getting the current base-data, that will be merged with the modified data to represent the new meta-data as a requirement for the next meta-cycle.

The rest of the paper is organized as follows: section 2 provides a brief overview of the tools we have adopted in our work; section 3 describes the evolutionary life cycle for dynamically adapting the software of the robot’s system for runtime changes; section 4 describes the application of proposed life cycle to software evolution by an example. Finally in section 6 we draw our conclusions and present some future work.

2 Background

2.1 Computational Reflection

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

Reflection is a technique that allows a system to maintain information about itself (meta-information) and using these information to change (adapt) its behavior. This is realized through the casual connection between the base- (the monitored system) and the meta-level (the monitoring system).

2.2 Software Engineering models

All software engineering models define how to build a changeable software system for any change in its requirements. So, there are two schools of thought in software engineering: first, linear thinking fostered by waterfall life cycle (Royce, 1970). The waterfall life cycle is divided into sequential phases analysis, design, implementation and testing phase. In each phase we use special techniques and tools. When a phase ends the next phase will start in a sequential flow, until production code. The rudiment for this model is the simple requirement and the life span for the system software remains long without changes. When the requirement specifications are complex, we use prototype (Curtis et al., 1987) to simplify it. Second, iterative or evolutionary thinking, fostered by the spiral model (Boehm, 1988). This model integrates the waterfall with prototyping for producing a model dealing with the complex requirements. It divides the software engineering space into four quadrants: management planning, formal risk analysis, engineering, and customer assessment.

The spiral model for software engineering is currently the most realistic approach to the development for large-scale systems and software (Cotton, 1996; Gilb, 1988). There are two archetypes of the spiral model: the incremental life cycle. This model separates the development cycle into n-cycles, each cycle consists of waterfall phases, with the end of each cycle we have a prototype until we reach the final version of the product. Second archetype is the evolutionary development. This model improves the incremental flow by adding the user feedback at the output of each cycle and document his view and new additional functions into the next development cycle.

3 Evolutionary life cycle for dynamic adaptation

We present an evolutionary life cycle to automatically adapt and validate consistency of the base robot’s system for environmental changes. Each robot in the system must be able to adapt itself according to the data from its sensor, their digital eyes, and the data from other collaboration robots. The life cycle is to be built to guarantee the evolution and validate the consistency processes for the base system. The application of the proposed life cycle to each robot in the system includes the mechanism to evolve its behavior at runtime. The structure of the life cycle is based on Evolutionary life cycle concepts (Cotton, 1996). The life cycle is composed of two cycles: base and meta, as shown in figure 1. The processes of the base-cycle execute in sequential flow. At the base cycle, the base engine extracts the UML/XMI documents from the robot base application to constitute the base-data. The processes of the meta-cycle execute in an incremental flow. The meta-processes start by reifying the base data to constitute the meta data at meta-level, then remote the runtime events from the environment. The evolutionary engine evolves the data for the changes by using the evolutionary interpreter, which is responsible for building a new meta-data inclusive the runtime changes. After that the validatory engine checks
the consistency between both meta-data according to runtime events effects. Finally the modified meta-data reflected to the base level of the robots to currently execute the action. The evolutionary life cycle consists of two cycles:

**Base-cycle** includes two linear phases: The first phase: contains all base components 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 from its corresponding the UML. The second phase: includes the UML/XMI documents. UML/XMI provides a simple translation of UML diagrams in a form more suitable for run-time manipulation. The internal representation of the base cycle is illustrated in algorithm 1.

**Evolutionary-cycle** consists of three phases: first is the reified UML/XMI documents to constitute the meta-data. The second consists of n-meta cycles, the runtime changes interact with the result of each cycle. The evolutionary meta cycle (or cycles) consists of: (i) the reified UML/XMI documents, (ii) Evolutionary engine, and (iii) the validatory engine. Each meta-cycle needs time to evolve and validate the meta-data for runtime changes, while the base level system also runs. The evolutionary cycle involves the meta-feedback concept, which is able to integrate the output of each cycle with the current data from base-level. The last phase is the reflection phase which is responsible for reflecting the meta-changes(runtime modification) to the base UML/XMI documents. The internal representation of the meta cycle is illustrated in algorithm 2.

### 3.1 Formalized Design Information

In this section we introduce two operators which are used in the evolutionary cycle to modify the design information. For that we formalize the specification of the design information (UML/XMI documents).

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 document (OMG, 2002). In the following we call these graphs UML trees. We define two operations on UML trees:

**The UML-intersection 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 intersection 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.):

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

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

$$G_1 \oplus G_2 = \{\{v, w \mid v \in V_1, w \in V_2 \land w \notin V_1\}, \{e, h \mid e \in E_1, h \in E_2 \land h \notin E_1\} \}$$  \hspace{1cm} (2)

### 3.2 Formal Description of the Evolutionary Cycle

The evolutionary cycle consists of several meta cycles. All meta cycles are invoked by a corresponding a runtime event. Each meta cycle gets as argument reified base data in form of a UML-tree. It modifies these
Data : meta-data, runtime events
Result : new meta-data includes the runtime changes
repeat
  if meta data reified then
    meta engine starts its processes to evolve and validate consistency for the reified meta-data;
  for each runtime event do
    evolutionary engine evolves the meta-data;
    validate the modified data by using validatory engine;
    finally reflect the modified data to base system,
    to give the robot the execution
  end
until runtime events detected;

Algorithm 2: Meta-cycle flow

input data accordingly to its corresponding runtime event. If a meta cycle is active and runtime event occurs it is interrupted and a successor cycle is invoked. All meta cycles except the first (Φi with i > 0) get additionally the modified data of its predecessor meta cycle. We define a meta cycle Φi as a function which transforms the input graph Gi to G′i:

\[ G′_i = \Phi_i(G_i) \]  

The input data of the first meta cycle Φ0 is the reified UML tree which describes the current base level state. For every other meta cycle Φi, with i ≠ 0 the input tree is the result of Φi−1 merged with new reified UML tree Ui of the base level using the UML-update operator:

\[ G_i = U_i \oplus G′_{i-1} \]  

The UML-update algorithm gets as input two UML-trees 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 4:

Algorithm 3: Algorithm of the UML-intersection operator

4 Case Study: Robot systems

In this section we want to clarify the benefits of our proposed evolutionary life cycle model. For that we introduce an application scenario and present the evolutionary framework which implements the evolutionary life cycle.

The UML-update algorithm gets as input two UML-trees and returns as output a graph which contains all nodes and edges which are in both input trees except of the root node vroot and the edges which are connected to vroot. The algorithm is depicted in 3:

Algorithm 4: Algorithm of the UML-update operator

Using these two algorithms it is possible to describe the evolutionary cycle. It creates the different meta cycles and retrieves the updated base level information (see algorithm 5).
Data : event
handled = false;
inputtree = reify();
while handled == false do
    init(metacycle, event);
    modifiedtree = startmetacycle(inputtree);
    if ishandled(metacycle) then
        handled = true
    else
        inputtree = update(reify(), modifiedtree);
    end
end

Algorithm 5: Algorithm of the evolutionary cycle

4.1 Application Scenario

Robot soccer contests are a popular method to evaluate and compare new research results in the domain of robotics (Lenser et al., 2001). Recently a lot of different variants are used to evaluate the skills of modern robot technology – hardware as well as software. To evaluate the benefits of our evolutionary framework we decided to use the following imaginary scenario: Two concurrent soccer teams have the primary objective to bring the ball into the opponent’s goal. The ball and the robots play on a hedged soccer field. To bring the ball into the goal, the robots of one team can collaborate. The opponent team must prevent the team to score.

To observe the environment each robot has a set of sensors. For simplification we do not specify each sensor type. We assume that a robot can observe the environment and infer the position of other robots, their actions (e.g., movement, shooting, etc.) and it own position relatively to the soccer field. A set of cameras and adequate image processing software can accomplish that. To interact with the environment, each robot has a set of actors. Actors are a leg for shooting the ball, a locomotor system and the motor which adjusts the position of the sensors (e.g. for moving and rotating cameras). In figure 2 the concept of robots is depicted.

It can be seen that each robot uses the sensors to observe the environment, the movements and other action of the other robots. The actors are used to perform actions.

Some simply build-in rules control the progression of the match. The main rules are:
- bring the ball near to the opponent goal
- score a goal
- drive round obstacles
- hinder the opponent team to get the ball
- hinder the opponent team to score

The processing of these rules results in a not trivial control of actors and sensors. In the following we want to show how the evolutionary life cycle approach and the evolutionary framework can used as a basic software architecture to adapt the behaviour of the robot (sensor and actors). This contribution has not the aim to provide solution for decision making, sensor data processing or real-time problems. Instead of that we want to provide a solution for the evolution of the robots behavior at software level.

4.2 Design Overview

In figure 3, the UML diagram of the class structure of the robots software is depicted. The classes are divided into meta level and base level classes. The meta level classes constitute our evolutionary framework. The base level classes provide the core functionality of the robots application. The classes Sensor and Actor are base classes for sensors and actors. To communicate with concrete sensors/actors we have to provide sub-classes. Thereby it is possible to compose several basic sensors to create a virtual sensor (class VirtualSensor) which combines the abilities of all included sensor information. The same is possible for actors. The Behavior class abstracts the possible behavior of a robot. The possible behaviors are high-level abstractions and result in a complex control of several actors and sensors and in a lot of processing and decision making tasks see subsection 4.3. The RobotController is the main class and entry point for the robot software and controls the behavior of the sensors and actors. All other classes belong to the evolutionary framework and are located at the meta level. Recall that the task of the evolutionary framework is to process the reflective life cy-
Figure 3: UML framework for the Robot’s systems to evolve the robots behavior in reaction to external events. Incoming events are scheduled by the EventGate. The RuntimeEvent class is the base for all specific events. The specific event instances describe itself in form of how they depend on specific sensor and actor runtime-configurations. For instance the event GoalCloseness occurs if the robot position falls below a defined threshold. The event GoalChance depends additionally on the holding of the ball. The EvolutionaryController receives from the EventGate these incoming events and creates for each an EvolutionaryCycle. Each cycle uses the DecisionMaker and the BehaviorPlanner to evolve the behavior of the robot. The DecisionMaker makes strategic decisions according to the rules, to the current behavior and environmental circumstances (e.g. position of other robots, position on the field, goal closeness, etc.). The BehaviorPlanner uses the strategy to create a new behavior pattern and change the current behavior accordingly. In this contribution we focus on the evolution of the behavior using the current game strategy. The function of the DecisionMaker is out of the scope of this paper. In this context we refer to (Lenser et al., 2001; Lenser et al., 2002; Lenser and Veloso, 2003).

4.3 An Example Situation

In this subsection we apply the proposed evolutionary life cycle to the working Sony Legged Robot (SLR) (Fujita et al., 1999). Each team consists of three robots including a goalkeeper. The teams have different colors to separate the robots, one of teams has red stickers (Defender RA, midfielder RB, Attacker RC) and the other teams has blue stickers (Defender BA, midfielder BB, Attacker BC). The field is 1.8 x 2.8 meter with six landmarkers, one in each corner and two in the middle side. the landmarkers consists of two colors distinctive combination at fixed position used by the robots to identify their position on the field.

In the following we demonstrate the internal phase of the two cycles: base and meta.

Base cycle:

In figure 4, we illustrate the representation of the UML diagrams for the robot RB. All the design data stored in the object model and statechart diagrams translated to XMI schemas by using the base engine, which is responsible for getting the design data from UML. The base engine extracts the design data from each robot and allocates its in one XMI schemas.

Evolutionary cycle:

When the base engine finishes its processing for extracting the UML/XMI documents, then the extracted data will be reified to the meta-level to constitute the meta-data. In figure 5, we illustrate the evolutionary cycle in details. The evolutionary controller looking for runtime events. It then generates meta-cycle, which is responsible for evolving and validating the reified design data. If the evolutionary controller generates a new metacycle then we used the meta-feedback to update the metadata else the modified data reflected to the base systems. The meta-feedback uses the update operator to merge the output from the meta-cycle with the current data from base-level system. The proposed meta-data is the input data for the next cycle.
Suppose the following situation: the robot RC is near from the ball, what is the best behavior for robot RB? (see algorithm 6).

The evolutionary cycle reifies the current data to lead the robot RB is able for taking decision without outside control:

- allocate the sensor and cameras reading;
- specify the position of the robot RB, and the localization of other mates;
- localize the position of the opponents.

\[
\text{Data} : \text{OpponentsLocalization} \\
\text{opponentGoalPosition}; \\
\text{IsNear(}\text{Robot RC, position}); \\
\text{if } \text{stand}\_\text{opponent}=false \text{ and} \\
\text{matching(}\text{opponent-goal})=true \text{ then} \\
\hspace{1cm} \text{attacker\_kick(change\_mode)} \\
\hspace{1cm} \text{ball\_kickable(}\text{robot RC, GoalAngle}) \\
\hspace{1cm} \text{else} \\
\hspace{2cm} \text{pass\_ball(}\text{midfielder RB, change\_mode}); \\
\hspace{1cm} \text{end} \\
\text{end}
\]

Algorithm 6: Robot RC behave

5 Related Work

Several other researchers have proposed a mechanism for dynamic evolution by using:

Software development and evolution: The development of software systems iterates over analysis, design, implementation, and deployment. Software architectures have the potential to provide a foundation for systematic runtime software evolution. (Santis et al., 1997) presented, a description of the process used to support the evolutionary development of advanced information retrieval systems. (Boehm et al., 1998) proposed the Win-Win Spiral Model as a new version of the traditional spiral model, the Win-Win version strives to involve all stakeholders in the development process. It involves a collaborative engine that establishes “win” conditions set by users, customers, developers, and system engineers in order to evolve and reprioritize requirements throughout the process. The proposed model adds three activities to the front end of each spiral cycle:

- Identify the system or subsystem’s key stakeholders.
- Identify the stakeholders’ win conditions for the system or subsystem.
- Negotiate win-win reconciliations of the stakeholders’ win conditions.

Reflective architecture and design information:
The system we consider in this short overview are: first, UML virtual machine, (Riehle et al., 2001) presented the architecture for a UML virtual machine. Second, The K-Component Architecture, (Dowling and Cahill, 2001) proposed a meta-model framework named K-components, that realizes a dynamic, self-adaptive architecture. Third, Architectural Reflection, (Cazzola et al., 1999) presented a novel approach to reflection called architectural reflection which allows dynamic adaptation of a system through its design information. Finally, design enforcement, (Simhi et al., 1996) presented a method for design enforcement, based on a combination of reflection and state machine diagrams.

Aspect-oriented programming: Aspect-oriented programming (AOP) was first introduced by Kiczales (Kiczales et al., 1997). The aim of AOP is to modularize and separate crosscutting concerns. AOP can be used to evolve software systems in a non-invasive way (Kiczales et al., 1997). (Zhang and Jacobsen, 2004) has shown how aspect-oriented programming can be used to evolve software systems by adding new aspects. Currently a lot of research effort has been made in runtime weaving of aspects (Popovici et al., 2003a; Sato et al., 2003). In (Falcarin and Alonso, 2004) the use of runtime weaving for software evolution is presented. In (Popovici et al., 2003b) it has shown how to use runtime aspect weaving to evolve a middleware.

Reflective middleware: In the domain of middleware for distributed systems reflection has been proposed to evolve middleware behavior. OpenCorba (Ledoux, 1999), OpenORB (Blair et al., 2002) and dynamicTAO (Kon et al., 2000)
extend CORBA by a reflective architecture. These systems \textit{reify} important characteristics of the behavior and of the structure of the middleware, such as scheduling strategy and resource management. The application can access and modify this information using a meta-interface. This allows to customize the middleware at runtime. A similar reflective CORBA-independent approach is CARISMA (Capra et al., 2003). Its focus is on context-awareness and on policy conflict resolution.

\textit{UIIC} (Ronn et al., 2001) and \textit{ReMMoC} (Grace et al., 2003) are two examples of middleware which utilize reflection to evolve middleware to cope with fluctuating requirements due to device heterogeneity. Both assume that different devices use different middleware technologies, e.g., \textit{SOAP}, \textit{CORBA}, \textit{Java RMI}, and provide mechanisms to deal with this heterogeneity. \textit{UIIC} is based on dynamicTAO. It implements a reflective architecture and a minimal core of functionality. If the reflective architecture detects the presence of a remote device, it loads the adequate middleware component. \textit{ReMMoC} uses a similar approach.

\section*{6 Conclusion}

We proposed an evolutionary life cycle that would be suitable for self-adapting object-oriented information systems for the runtime changes. We applied the software development models through the reflective architecture, to satisfy the dynamic adaptation. We used the meta-feedback concept to remodify the metadata. We then illustrated the formal description of evolutionary cycle. Moreover, we 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.

\section*{REFERENCES}


