Change and Role as First-Class Abstractions for Realising Dynamic Evolution

Yin Chen, Xin-jun Mao

College of Computer, National University of Defense Technology, China
E-mail: chaanyean@163.com, mao.xinjun@gmail.com

Abstract

Today many software systems must undergo continuous dynamic evolution to adapt to variable environment while having minimal impact on their normal servicing. Programming dynamic evolution requires integrating evolution logic with business code, so that each single change as a stage of evolution can closely react to its context including objects and other concurrent changes. However, integrating evolution logic can lead to a closely coupled system which is less maintainable. In order to balance evolution logic integration and separation of concerns, so as to facilitate dynamic evolution at language level, we present a meta-model which takes change and role as first-class abstractions to model different degrees of such integration. It uses roles to model variation points and uses role enactment to realise changes in a modular and interactive manner. We extend conventional languages with primitives for describing dynamic changes and with atomic operations for manipulating changes according to the structure of target system. Our approach will hopefully facilitate the realisation of dynamic changes and well-structured, highly evolvable systems.

1. Introduction

A deployed software system should evolve continuously to react to contextual and requirement changes. For many real world systems which need to be highly available, such evolution should be performed dynamically to reduce impacts on the system’s normal servicing [3]. Evolution process is made up of phased changes, whose programming involves two aspects: (1) strategy of change, i.e., the systematic process (or its pattern) to realise a change, and (2) target of change, i.e., objects involved in the change. Effectively supporting these aspects at language level by explicit representation of change-related information relieves the effort of programming dynamic evolution [4]. Programs specifying targets of change should previously arrange variation points (as the join points of evolution logic and application logic) which clarify the dynamic structure of target system and facilitate its manipulation. Strategies of change then exploit these variation points to realise final changes.

Many efforts are made to enable programming dynamic evolution. Architecture description languages (ADL) [1] specify dynamic software architecture with graphs, process algebras, etc. Aspect-oriented programming (AOP) encapsulates evolution concerns as aspects and realises fine-grained evolution actions [5]. They are basically external approaches [6] that enhance separation of concerns by encouraging application-independent evolution mechanisms rather than intertwining them with application code. However, due to contextual complexity, change strategies must be application-dependent and closely integrated with target systems, because: (1) instead of designing all-powerful change strategies, many changes must be temporarily designed according to runtime situation; (2) in dynamic evolution, a change strategy needs to adapt to its context, and concurrent changes need to interact for synchronisation and data sharing, so as to ensure global requirements and minimal side effects. Context-oriented programming (COP) [7] and dynamically typed languages [4] integrate evolution capability at language level, enabling dynamic activation of codes via contextual events, but at the cost of separation of concerns and global coordination.

In order to facilitate programming of dynamic changes, we propose an approach to explicitly represent evolution from both external and internal perspectives with an attempt to balance and separate evolution concerns. We introduce role as first-class entity to model internal variation points and to combine evolution concern with target system. Roles enable a separation of concerns between business logic, and coordination or evolution logic, and promote a global view of target system [2]. We introduce change as first-class entity to model external change strategies. Changes are deployed according to the role-based structure of target system and manipulate roles for their realise. We propose explicit primitives for programming these concepts and atomic operations for bridging and manipulating internal and external perspectives of dynamic evolution.
The rest of this paper is organised as below. Section 2 presents the concepts for describing strategies and targets of changes. Section 3 presents atomic operations for specifying role-based interaction during change realisation, and the way these operations are organised into a concrete change process. Conclusion and outlook are outlined in section 4.

2. Meta-Model for Dynamic Evolution

We present a meta-model which provides following concepts for describing dynamically evolvable system: organ, role, change, and message, respectively modelling dynamically evolvable objects, variation points, change strategies, and information exchanged in interactive change realisation. Fig. 1 shows these concepts and their relations.

![Figure 1. Meta-model for dynamic evolution](image)

1. **Organ**: An organ is an evolvable object attached with changes and roles. Each organ $\alpha$ can be represented by tuple $\langle R, M, C, \sigma, R' \rangle$. $R$ is the set of roles enacted by $\alpha$. $M$ is a queue of messages received but not yet processed by $\alpha$. $C$ is the set of changes deployed to $\alpha$. $\sigma$ is an assignment which determines the values of $\alpha$’s field. $\varsigma$ is the sequence of statements to be executed by $\alpha$ at the current time. $R'$ is a set of roles owned by $\alpha$ to be enacted by other organs.

2. **Role**: A role is a reference to an organ which plays a part in another organ owning the role. Each role $\rho$ can be represented by tuple $\langle \alpha, \kappa, C, \alpha' \rangle$. $\alpha$ is the unique organ which enacts $\rho$. Its value is $\perp$ if no organ is currently enacting $\rho$. Each role is allowed to have at most one enactor, and is syntactically taken just as a reference to that enactor. $\kappa$ is the class specifying the behaviour and properties that can be implemented by $\alpha$. $C$ is the set of changes deployed to $\rho$ and $\alpha$. $\alpha'$ is the unique organ that owns $\rho$. We use $\text{in}(\rho)$ (or $\text{out}(\rho)$) to represent the algorithm which determines whether an organ satisfies the requirements for entering (or quitting) $\rho$, and then reacts accordingly.

For an organ $\alpha = \langle R, M, C, \sigma, \varsigma, R' \rangle$, roles in $R$ decompose $\alpha$’s tasks to the organs enacting these roles. Each role $\rho$ specifies entrance (or quitting) constraints for its enactor $\alpha_{\rho}$ using $\text{in}(\rho)$ (or $\text{out}(\rho)$) and attaches methods (defined in $\kappa$) to $\alpha_{\rho}$ for it to interact with other roles in $\alpha$. In terms of change realisation, roles provide variation points that combines organs in different ways, regulate changes using these constraints, and specify protocols about how their enactors coordinate to realise a change.

3. **Change**: A change is an executing entity which is deployed to the target—an organ $\alpha$ (or a role $\rho$) to make change to $\alpha$ (or $\rho$’s enactor) and which can deploy more changes. One instance of a change can occupy multiple threads. Each change $\gamma$ can be represented by tuple $\langle \iota, \sigma, \varsigma, C, \gamma' \rangle$. $\iota$ is an identifier of the target to which $\gamma$ is deployed. $\sigma$ is the assignment which determines parameters of $\gamma$, $\varsigma$ is the statement to be executed by $\gamma$. $C$ is the set of changes which are deployed by and are hence subordinate to $\gamma$. $\gamma'$ is the change which has deployed $\gamma$. Use $\text{init}(\gamma)$ to represent the algorithm executed by $\iota$ to initialise $\gamma$. We use $\text{chk}(\gamma)$ to represent the algorithm executed in a top-down manner after $\text{act}(\gamma)$’s completion to check whether the designed goal of $\gamma$ has been achieved. The execution of $\text{act}(\gamma)$ and $\text{chk}(\gamma)$, and the handling of their results, are automatically performed by the underlying framework.

4. **Message**: A message is an object used for asynchronous interaction among organs. Each message $\mu$ can be represented by tuple $\langle \alpha, \iota, \nu, \sigma \rangle$. $\alpha$ is the organ which has sent $\mu$. $\iota$ is the identifier of the organ or role to which $\mu$ is sent. $\nu$ is the local name of $\mu$ assigned by $\alpha$. $\sigma$ is an assignment which determines the value of $\mu$’s each parameter. Both $\nu$ and $\sigma$ are used by $\alpha'$ to locate the method or the statement (within a method) that handles $\mu$.

Fig. 2 shows how introducing role and change as first-class abstraction into object-oriented paradigm facilitates programming dynamic evolution. As shown by path A, in terms of change realisation, roles provide variation points that combines organs in different ways, so that simply by enacting a different role can an organ realise a change. These variation points can relate to business logic (which concerns how each organ functions or multiple organs coordinates, as shown by path B) or evolution strategy (which concerns how the changes exploits variation points and interact with each other to achieve their goals, as shown by path C). Role enactments provide managed variation points, as shown by D. Both concerns involves two aspects: coordination, that focuses on how multiple individuals work together to achieve system-level goals, and execution, that focuses on how each individual performs some low level operations to achieve local goals. About the coordination aspect: each role $\rho$ specifies entrance (or quitting) constraints for its enactor $\alpha_{\rho}$ using $\text{in}(\rho)$ (or $\text{out}(\rho)$) and attaches methods (defined in $\kappa$) to $\alpha_{\rho}$ for it to interact with other roles in $\alpha$. About the execution aspect: roles enable each organ to combine different services provided by other organs by delegating tasks to them through role enactment. The loop formed by paths A, C, D, and also the fact that one change can deploy other changes, indicate that there are infinite possi-
3. Change-Realising Atomic Operations

In a typical process for realising the changes using our meta-model, each single change needs to undergo a 4-phase process to be realised: (1) get deployed to the target object, (2) change the target object, (3) check whether the designed goal has been achieved, and deal with exceptions, (4) report the result of the change to who has deployed the change. Such process is usually iterative. For example, if (4) finds the goal unachieved, it may reconfigure and restart the change. This process can have multiple stages of preparation for completion. For example, some preparatory changes can be implemented to be used later and be dismissed once the final goal is achieved. Tasks for changing a higher level organ is decomposed and delegated to the subordinate organs through the “deploy” links. After the subordinate organs have completed and checked the changes, they report back to the higher level organ. Following syntax shows atomic operations used to realise such change-realising process.

$$\varsigma ::= \epsilon \mid \varsigma_{atom}; \varsigma \mid \varsigma \parallel \varsigma$$

$$\varsigma_{atom} ::= l.v(\sigma) \mid \text{recv}\{\ell_i!v_i(\sigma_i) : i \in I\} \mid \text{dply}(\gamma, i) \mid \text{chk}(\gamma) \mid \text{in}(\rho)(\alpha) \mid \text{out}(\rho)(\alpha) \mid \text{hire}(\rho)(A) \mid \text{fire}(\rho)$$

where $\varsigma$ denotes a statement which can be formed by a sequence of atomic statements or the parallel combination of multiple statements (e.g., $\varsigma_1 \parallel \varsigma_2$) which represents concurrent threads. $\varsigma_{atom,ori}$ denotes a statement defined by the original language, e.g., Java. Our extension provides 3 sets of atomic operations: message passing, change manipulation, and role manipulation. Fig. 3 presents the operational semantic rules of atomic operations for guiding design of dynamic evolution enabling languages, in terms of effects of these operations on configurations of organs, roles, or changes involved.

4. Conclusion and Outlook

We present a meta-model for describing a dynamically evolvable system, and language-independent atomic operations for manipulating a system complying with this model. Our approach balances between external and internal. It uses roles to manipulates the structure of target system. It provides language-level primitives to describe the strategies of change (i.e., change classes). Compared with COP, which uses layer to modularise dynamically switchable behaviour, we use role as a medium between target system and strategies of change, to add variation points encapsulating behavioural extension of an organ, inter-organ relations, and even application/evolution logic. Both changes and roles can be used for dynamic evolution. Changes are externally built, deployed to roles and organs, forming a network alongside role enactment network, exploiting roles to achieve their goal. Roles are relatively internal and more closely bound to the business logic. Organising changes with roles facilitates the global coordination of distributed changes. Using explicit change manipulation rather than implicit invocation of these additional concerns (as in AOP) facilitates the anticipation of the result of change. We plan to develop a framework supporting features of our model and language-level primitives, and provides facilities for the management, serialisation, and exception handling, of distributed runtime changes.
\[ \mu = (\alpha_1, i, \nu, \sigma), (i = \alpha_2 \lor i = \rho \in R_2 \lor i = \bot) \]

\[ \alpha_1 = (R_1, M_1, C_1, \sigma_1, i, \nu; (\gamma_1, R'_1), \alpha_2 = (R_2, M_2, C_2, \sigma_2, \nu, R'_2) \Rightarrow \alpha_1 = (R_1, M_1, C_1, \sigma_1, R'_1), \alpha_2 = (R_2, \mu; M_2, C_2, \sigma_2, R'_2) \]

\[ j \in I, t(\sigma_i) \prec t(\sigma), \mu = (\alpha_0, a, \nu, \sigma_j), (i = \alpha_0 \lor i = j = \rho \in R \lor i_j = \bot) \]

\[ \alpha = (R, M \cdot \mu; M', C, \sigma, \text{recv}(i, t(\sigma_i) : (\gamma_i)_{i \in I}; \gamma_i) \Rightarrow \alpha = (R, M \cdot M', C, \sigma, \gamma_i, \gamma_i, \gamma_i) \]

where for organ \( \alpha \) or role \( \rho = (\alpha, \kappa, C, \alpha'), (t(\alpha) \text{ or } t(\rho)) \) returns the type of \( \alpha \) or \( \rho \).

\[ \gamma = (i, \gamma_0; \text{dply}(\gamma_0); \gamma_i, \sigma_i); \gamma, \gamma_i) \Rightarrow \gamma = (i, \gamma_0; \gamma_i, \sigma_i, \gamma) \cup \{\gamma_0\}, \gamma' \]

\[ \gamma = (i, \sigma_i, \gamma_0 := \text{chk}(\gamma_0), \text{exc}(\gamma_0, \gamma_0); \gamma_i, \sigma_i), \gamma = (i, \gamma_0, \gamma_i); \gamma = (i, \sigma_i, \gamma_0 := \text{chk}(\gamma); \text{exc}(\gamma, \gamma) = 0, \gamma \}

\[ \gamma = (i, \sigma_i, \gamma_0 := \text{chk}(\gamma_0), \text{exc}(\gamma_0, \gamma_0); \gamma_i, \sigma_i), \gamma = (i, \gamma_0, \gamma_i); \gamma = (i, \gamma_0, \gamma_i) \]

\[ \rho = (\bot, \kappa, C, \alpha'), \alpha = (R, M, C, \sigma, \chi := \text{in}(\rho)(\alpha); \gamma, \gamma_i) \Rightarrow \rho = (\alpha, \kappa, C, \alpha') \]

\[ \rho = (\alpha, \kappa, C, \alpha'), \alpha = (R, M, C, \sigma, \chi := \text{out}(\rho)(\alpha); \gamma, \gamma_i) \Rightarrow \rho = (\bot, \kappa, C, \alpha') \]

\[ \kappa \prec t(\alpha), \alpha \in A, \rho = (\bot, \kappa, C, \alpha_0) \in R_0 \]

\[ \alpha_0 = (R_0, M_0, C_0, \sigma_0, \text{hire}(\rho)(A); \gamma_0, R'_0), \alpha = (R, M, C, \sigma, \gamma) \Rightarrow \alpha_0 = (R_0, M_0, C_0, \sigma_0, \gamma_0, R'_0), \alpha = (R, M, C, \sigma, \gamma := \text{in}(\rho)(\alpha); \gamma, \gamma_i) \]

\[ \rho = (\alpha, \kappa, C, \alpha_0) \in R_0 \]

\[ \alpha_0 = (R_0, M_0, C_0, \sigma_0, \text{fire}(\rho)(A); \gamma_0, R'_0), \alpha = (R, M, C, \sigma, \gamma) \Rightarrow \alpha_0 = (R_0, M_0, C_0, \sigma_0, \gamma_0, R'_0), \alpha = (R, M, C, \sigma, \gamma := \text{out}(\rho)(\alpha); \gamma, \gamma_i) \]

\[ \rho = (\alpha, \kappa, C, \alpha_0) \in R_0 \]

Figure 3. Operational semantics of change realising atomic operations

5 Acknowledgement

This research is founded by NSFC under granted number 61070034 and 61379051, Program for New Century Excellent Talents in University under granted number NCET-10-0898, and Open fund of State Key Laboratory of Software Development Environment under granted number SKLSDE-2012KF-0X.

References


