Dynamic Component and Code Co-Evolution

Markus Pizka
Institut für Informatik
Technische Universität München
Germany - 80290 Munich
pizka@in.tum.de

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement —*Extensibility, Restructuring*

Keywords
Software Evolution

ABSTRACT
This papers presents a radically new approach for the dynamic evolution of long-lived systems that can not easily be shut-down for maintenance and restarted afterwards. Conventionally, the source code of a software system is viewed as a static entity and separated from the system at runtime. This seems intuitive as a single piece of code is usually associated with multiple components at runtime. Obviously, this viewpoint is a major obstacle for dynamic evolution during runtime as it raises difficult consistency issues concerning the relationship between static code and the dynamically executing system. The evolution approach presented in this paper takes a completely different direction by seamlessly integrating static code with dynamic execution. By this and sound concepts for component categories, incompleteness and dynamic completion, software can be generalized and adapted during runtime in a highly flexible way.

1. DYNAMIC EVOLUTION
The process of software evolution often applies the following strategy one way or the other: *save-state, shutdown, perform-modification, restart, restore-state*. Despite of its simple structure, this pattern already raises numerous non-trivial questions, such as how can the state be captured at an arbitrary point in time and how can quality be reassured. Regardless of these questions, this static view of evolution assumes that it is possible to shut the system to be modified down. In cases of continuous operation, such as large-scale mission-critical databases, central components within a network or process steering systems (e. g. chemical processes) this requirement must be considered at least strongly undesirable.

International Workshop on Principles of Software Evolution In conjunction with the International Conference on Software Engineering (ICSE-2002), May 2002, Orlando, Florida, USA

1.1 Code Co-Evolution
Thus, dynamic evolution, i. e. adapting the system during its runtime without (or hardly any) disruption of its operation is strongly more favorable. Additionally, the conventional evolution strategy tends to blur the important link between the source level program[1] and the system in execution. This has several drawbacks. E. g., there is no evolvable representation of the system in execution if the code is modified while the system is executing. Furthermore, individual modification of two instances a and b of the same class c, specified by a single piece of static code, requires at least expensive versioning. Our approach to circumvent these problems, is to view the code as an integral part of each component in execution. By this evolution always affects code and component simultaneously.

1.2 Scope
We investigated the question of how to dynamically modify systems in the context of an effort to develop a general purpose distributed operating system (OS) [1, 7, 6, 4]. Our research goal was to build a distributed system with the requirement that the complete system, including its potential evolution, has to be fully known by the distributed OS at any time. The reason for this unusual requirement was the observation that a distributed OS must possess extensive information to be able to automatically select suitable resource management techniques, such as replication or migration for remote object access [10]. If a distributed OS fails to dynamically switch to an adequate management strategy, the performance delivered is usually unacceptable due to the enormous gap between the speed of local and network accesses ($\approx 10^5$).

To approximate the requirement for complete knowledge we employed an integrated, single system view. The OS and all applications within the distributed system are regarded as a single structured system. Consequently, to be able to introduce new applications into the already running single system or to evolve parts of the OS we needed new concepts for dynamic evolution.

[1] we further-on abbreviate source code with "code"

The results achieved within this context are not limited to the scope of distributed systems. The concepts and implementation techniques developed can easily be transfered to other environments where dynamic evolution of software at runtime is either needed or desirable.

1.3 Outline

Section 2 introduces our notion of flexible incremental evolution based on the two concepts *completion* and *generalization*. Section 3 defines the term evolvable "component" by distinguishing three different categories of components, which is a prerequisite to be able to fully model dynamic evolution. Based on this conceptual framework we are able to precisely define possible evolution steps during the lifetime of a dynamically evolving system in section 4. Finally, we briefly discuss implementation aspects of our proof of concept prototype in section 5.

2. INCREMENTAL COMPLETION

Figure 1 sketches our notion of flexible stepwise evolution. System[2] V_1 is developed as a solution to problem P_1. Afterwards, V_1 becomes generalized to V_2 in step A_1 by performing an abstraction. Completely specified components are replaced with incomplete ones. Step C_2 completes the generalized variant V_2 according to the requirements of the new problem P_2 forming variant V_3, which will be subject to further completion and generalization steps over time. The fundamental concepts completion and generalization are detailed in [5].

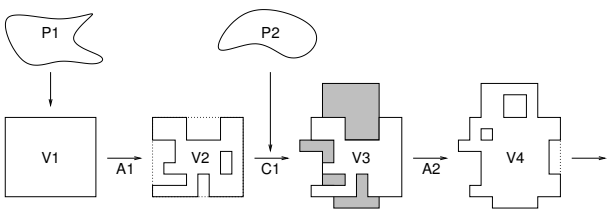


Figure 1: Incremental Evolution

- A *generalization* of a system V is the detachment of bindings within V, i. e. the replacement of bound identifiers with free variables. The resulting more incomplete derivate V' is a flexibilized *abstraction* of V.
- The binding of free variables of system V is called a [partial] *completion* of V. Degrees of freedom are removed and a more concrete system V' produced.

The specification of flexibility by means of incompleteness and the determination of bindings during completion steps is based on formal language concepts. In fact, the principles of completion and generalization somehow already exist in practice in several peculiarities. Examples are:

- input of data at runtime
- substitution by preprocessors
- parameterized subroutines
- inheritance
- design patterns

Incomplete specification combined with static or dynamic completion are integral parts of efforts to construct reusable and evolvable systems. However, the primary goals and problems of incremental evolution seem to have taken a back seat in the past. Although many details, such as features of macro substitution were improved (e.g. [2]), we argue that the actual characteristics of incremental evolution have not been systematically investigated, yet. This supposition is supported by the fact that the correlation between statics and dynamics seem to be hardly understood.

Here, completion and generalization is defined at a conceptual level. This has the advantage that these concepts are not a priori influenced by the artificial border between statics and dynamics, which is introduced by the conventional compilation, linking and execution cycle. In addition to this, our scheme allows to introduce new degrees of freedom into an already complete system in a defined way by means of generalization steps. We believe that this is especially important for software evolution since future requirements are in general unknown and can thus not be expected to be respected in advance.

3. COMPONENT CATEGORIES

To be able to relate code with dynamic execution we distinguish three different component categories:

3.1 Incarnations

The set X_t of *incarnations* are procedures, functions, or data objects executing at time t. An incarnation $a \in X_t$ usually contains other components of possibly varying categories and possesses a transient state.

3.2 Generators

Generators define classes of incarnations, such as a class of procedure incarnations. $\mathcal{G}_t$ denotes the set of generators within the system at time t. A generator is always considered a local component of an incarnation; i. e. for each generator $A \in G_t : \exists b \in X_t : A \in L(b)$.

Hence, generators are similar to classes in OO languages [9] but differ in being integrated into the dynamics of execution.

3.3 Generator-Families

Generators are not equal to code but dynamic entities because generators are local components of incarnations. We therefore need a third category of components to be able to explain the origin of generators. We call this category *generator-families*.

$\mathcal{G}_t$ denotes the set of generator-families at time t. Each generator-family $\mathcal{A} \in \mathcal{G}_t$ describes a class of generators. But in contrast to incarnations and generators, which are partially ordered by the flow of control, the set of generator-families is partially ordered by the nesting of code.

[2] "system" subsumes subsystems, components, and objects

Thereby, a hierarchy of generator-families may exist independent of incarnations and generators. Notice, generators can only exist within incarnations and are therefore not suitable to model code as part of the dynamic system. A result of this consideration is that the intuitive code-is-class understanding in OO languages is misleading and an obstacle for dynamic evolution.

3.4 Dependencies

This briefly sketched conceptual framework allows individual completion and generalization of incarnations, generators and generator-families. Hence, the properties of an incarnation $a \in X$ need not be identical with the properties of its generator $G(a)$[3].

Obviously, certain restrictions for completion and generalization must apply to ensure consistency and controlled, comprehensible evolution. The required regulations are defined as life-time and property dependencies between the components of the system, which are briefly sketched, below.

3.4.1 Lifetime Dependencies

- Each incarnation $a \in X$ is created by executing the *create* operation of the corresponding generator $A_a \in G$ (def.: $A_a \overset{c}{\Rightarrow} a$).
- A generator $A \in G$ emerges in turn of the elaboration of the declaration part of an incarnation. If the declaration part of $b \in X$ contains a declaration of a generator A then b *elaborates* the corresponding generator-family $\mathcal{A}_A \in \mathcal{G}$ (def.: $\mathcal{A}_A \overset{e}{\Rightarrow} A$). After its elaboration A can be used by b to create incarnations $a \in X$. Hence, generators are dynamically elaborated on the basis of generator-families during the evaluation of the declaration parts of incarnations. Generators are deleted with the deletion of the surrounding incarnation.

Analogously to the creation of a component, all incarnations must be deleted before the corresponding generator can be deleted and all generators must be deleted ahead of the corresponding generator-family.

3.4.2 Property Dependencies

The lifetime dependency induces a reasonable order on the creation and deletion of components of different categories but it does not define predicates concerning properties of the components. In the presence of individual component completion and generalization we therefore need further regulations to avoid anarchy and achieve controllable evolution with well-defined and structured properties. To achieve this, we introduce additional property dependencies.

- The set of properties $E(\mathcal{A})$ of a generator-family $\mathcal{A} \in \mathcal{G}$ is specified by means of the chosen programming language. $E(\mathcal{A})$ defines invariants for all generators $A \in G : \mathcal{A} \overset{e}{\Rightarrow} A$.
- The set of properties $E(A)$ of a generator A is specified by means of the chosen programming language. $E(A)$ defines invariants for all incarnations $a \in X : A \overset{c}{\Rightarrow} a$.
- The set of properties $E(a)$ of an incarnation a is specified by means of the chosen programming language.

Hence, let $a \in X, A \in G, \mathcal{A} \in \mathcal{G}$ be an incarnation, a generator and a generator-family, with $\mathcal{A} \overset{e}{\Rightarrow} A \overset{c}{\Rightarrow} a$, then

$$E(\mathcal{A}) \sqsubseteq E(A) \sqsubseteq E(a).$$

This conceptual dependency defines a minimal requirement for consistent transitions and must hold at all times, independent of the degree of completeness of a component. By this, it is for example not possible to have incomplete incarnations of a complete generator, whereas the opposite is possible and also desirable.

3.5 Example

We illustrate our component and evolution concept with the aid of the code excerpt shown in figure 2. After introducing this code as a hierarchy of nested generator-families into an initial boot process execution starts with the elaboration of the outermost declaration part. Afterwards there is one incarnation *system* which possesses a generator G_a as a local component. As soon as execution reaches the call `a(42)` G_a is used to create an incarnation a_1. a_1 in turn owns a generator $G_b^{a_1}$ after elaborating the corresponding generator-family in its declaration part. a_1 uses $G_b^{a_1}$ to create an incarnation $b_1^{a_1}$ before recursively creating a_2 using G_a and $b_2^{a_1}$. Now, a_2 elaborates a new generator $G_b^{a_2}$, which will be used to create an incarnation $b_1^{a_2}$.

```
PROCESS system IS
   PROCEDURE a(I : IN INTEGER) IS
       PROCEDURE b(J : IN INTEGER) IS
       BEGIN
          ...
       END;
   BEGIN
       b(I);          -- create 1st b incarnation
       IF I > 0 THEN
          a(I-1);
       END IF;
       b(I);          -- create 2nd b incarnation
   END;
BEGIN
   a(42);
END;
```

Figure 2: Evolution & Recursion

If we assume that the generator-family $\mathcal{G}_b$ is initially incomplete then we are now able to perform a range of different evolutions during the recursive computation of the system. Some examples:

- The most coarse evolutionary step would be to complete $\mathcal{G}_b$ before the first b-generator $G_b^{a_1}$ becomes elaborated. This would have the effects that both $G_b^{a_i}$ generators would be complete, $E(G_b^{a_1}) = E(G_b^{a_2})$, and all

[3]for simplicity, we assume a snapshot at time t and omit index t

future b-incarnations would have identical properties leaving neither flexibility nor the necessity for future adaptions. Thus, completion of the generator-family gives little control over evolution.

- Another option is to individually complete the incomplete generators $G_b^{a_1}$ and $G_b^{a_2}$ in a_1 and a_2 before a_1 resp. a_2 starts executing its statement part. This would have the effect, that the two b-incarnations created in each a would have identical properties although the pair of b-incarnations of a_1 might differ from the pair of b-incarnations of a_2. I. e., the subcomputations of a_1 and a_2 are able to evolve differently.

- Obviously, the greatest flexibility is achieved by individual evolution of the created b-incarnations. This is accomplished by leaving $\mathcal{G}_b$, $G_b^{a_1}$ and $G_b^{a_2}$ incomplete and performing an individual completion for each created b-incarnation. Notice, this means that even within a single a-incarnation such as a_1, the two b-incarnations created in the statement part can and must be completed separately.

4. TRANSITIONS

The state transition diagram shown in figure 3 summarizes all possible evolution steps of the system, based on the component category concept and the life-time and property dependencies introduced in section 3.4.

Assuming the existence of an incarnation $b \in X$ we focus on a generator-family $\mathcal{A}$ and its descendants. Each node of the state diagram describes the set of components descended from a generator-family $\mathcal{A}$. Horizontal transitions represent the creation (resp. elaboration) and deletion of components whereas vertical transitions represent completion and generalization steps. E. g. starting bottom left b may elaborate generator A on the basis of the complete generator-family $\mathcal{A}$ and later on use generator A to create an incarnation a resulting in the state bottom right, with three complete components a, A, $\mathcal{A}$. This sequence of states corresponds to the conventional execution of non-evolvable system.

Additionally, our flexible conceptual framework also allows the user to perform controlled incremental evolution during execution. Dependent on the current state of the system and the property-dependency, it is possible to perform a generator-family generalization (e. g. from bottom left to state above), a generator completion on A (e. g. from state in the middle to state below) or evolution steps on incarnations (from/to top most state).

5. CONCLUSION

This paper describes a radically different approach to encapsulate static code with its incarnations. The main idea is to differentiate between three different categories of components to be able to model the relationship between code and instances at runtime. These three categories are instances, generators and generator-families. While instances and generators are also common in other approaches, we additionally identified the category generator-families, discussed its role for dynamic system evolution and proposed a new concept for the systematic integration of these three component categories into the dynamics of the system.

We furthermore sketched a conceptual framework for the incremental generalization and abstraction of software components. By means of a simple example, it was demonstrated how the scope of a (de-)completion step relates with the three component categories. However, program code is always directly attached to a component and therefore participates in the dynamics of the computation.

It seems important to mention, that we also implemented this conceptual framework. The prototype shows that the high degree of freedom for evolution described in this paper can be achieved without significant constant performance degradation. This was mainly achieved by providing an incremental dynamic link loader and modifications of the stack frame layout within the compiler [8, 3].

A major problem remaining is the question of how to specify incompleteness. In this paper, we simply assumed that it is possible to specify incompleteness. As a matter of fact, we do not know how to integrate the large number of different aspects of incompleteness, such as paramters, macro expansion and design patterns, within a single sound specification concept, yet. Our future work aims at clarifying this question.

6. REFERENCES

[1] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. *The Journal of Supercomputing*, 13(1):33–55, Jan. 1999.

[2] S. Krishnamurthi, M. Felleisen, and B. F. Duba. From macros to reusable generative programming. In *GCSE*, pages 105–120, 1999.

[3] M. Pizka. Design and implementation of the GNU INSEL-compiler gic. Technical Report TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[4] M. Pizka. Distributed virtual address space management in the modis-os. Technical Report TUM-I9817, Technische Universität München, 1999.

[5] M. Pizka. *Integrated Management of Extensible Distributed Systems*. PhD thesis, Technische Universität München, June 1999. german.

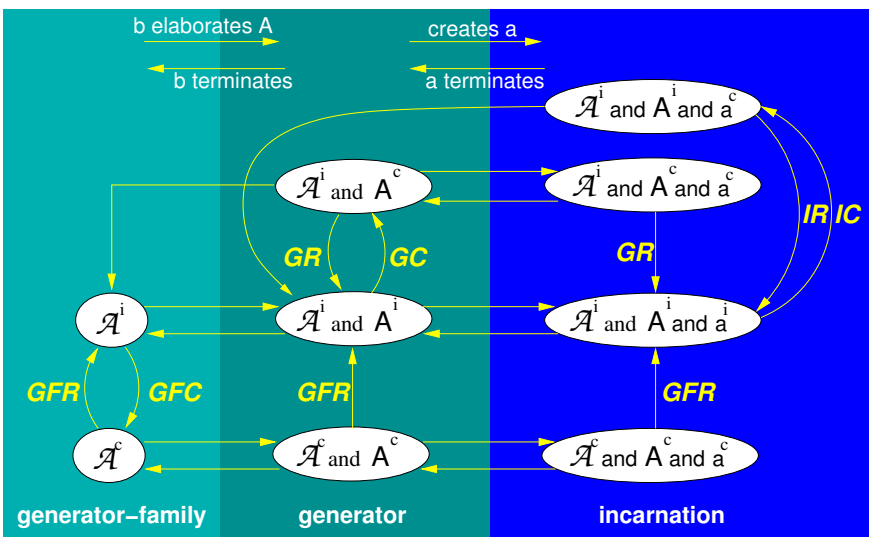
[6] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *HICSS-30*, pages 130–139, Maui, Hawai, Jan. 1997. IEEE CS Press.

[7] M. Pizka, C. Eckert, and S. Groh. Evolving software tools for new distributed computing environments. In *PDPTA'97*, pages 87–96, Las Vegas, NV, July 1997.

[8] C. Rehn. Incremental and dynamic linking in a distributed environment. Master's thesis, Technische Universität München, Institut für Informatik, 1998.

[9] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, MA, 2. edition, 1991.

[10] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In H. R. Arabnia, editor, *Proc. of PDPTA*, pages 115 – 131, Nov. 1995.



$\mathcal{A}$ resp. prefix GF: generator–family
A resp. prefix G: generator
a resp. prefix I: incarnation

index ^{i} : incomplete
index ^{c} : complete

suffix C: completion
suffix R: generalisation

Figure 3: State Transitions