STA - A Conceptual Model for System Evolution

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

Abstract

A great deal of work on software maintenance focuses on source code analysis and manipulation. Code is viewed as a static entity that is – more or less – separated from the system at runtime. Although there are certainly important open questions in this field, the separation of code and runtime imposes itself an obstacle for the evolution of continuously functioning systems. The goal of the work presented in this paper is to blur the separation between statics and dynamics of a software system on a conceptual level. To achieve this, we propose a system model that combines space, time and level of abstraction and a conceptual framework for incremental evolution of systems during execution. Based on this continuously functioning systems can be maintained in a highly flexible and conceptually sound way.

1. Extending Software Evolution to Systems

Software evolution often follows the pattern: save, shutdown, modify, restart, restore. Despite of its simple structure, this maintenance cycle already raises numerous non-trivial questions, such as how can the system state be captured at an arbitrary point in time, and how can quality be reassured after modification. Independent from these questions, this static view of evolution assumes that it is possible – and reasonable! – to shut the system to be maintained down. In cases of continuous operation, such as large-scale mission-critical databases or central components within a network this assumption does not hold. Thus, dynamic evolution, i. e. adapting the system during its runtime without (or hardly any) disruption of its operation is strongly more favorable.

1.1. The Relationship Between Source Code and Execution

We argue, that the main obstacle for more flexible maintenance techniques is the separation of source code[1] and system in execution[2]. Code is viewed as a static entity with few conceptually sound links with the system. As a consequence, the important link between the source level program and the system is lost. This has several negative consequences. E. g. there is no evolvable representation of the system if the code is modified while the system is executing and individual modification of two instances a and b of class c, specified by a single piece of code, requires versioning.

Indeed, most contemporary work on system design and modeling such as the UML concentrates on static properties of a software system although the separation between static code and dynamic execution is artificial. It was introduced with the compilation and linking techniques of the early 1960's. Interpreters, for example, do not impose a barrier between code and execution. The source code can be freely modified while the system is executing and changes take immediate effect.

Obviously, conventional interpreters can as well not be considered a reasonable choice for system evolution. First, their performance is unsatisfactory and second, being able to perform arbitrary modifications of the source code at any time without control on the semantic effects of the changes eventually produces chaos.

Our approach to eliminate the boundary between code and system is based on the vision that each component[3] could be transformed into a piece of code and vice versa. The only difference between code and system is the level of abstraction. Whereas code is a high-level and human-readable representation of a component, values in registers and in main memory provide an identical low-level repre-

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project ViSEK (Virtual Software Engineering Competence Center).

[1] "code" further-on abbreviates source code

[2] "system" further-on abbreviates system in execution

[3] "component" refers to any conceptual entity that is part of the execution of the system; i. e. object instances, data items, functions, etc.

sentation. Thus, code is regarded as the description of state and behavior of dynamically executing components and every component at runtime is described by its own code. Evolution always affects code and system simultaneously.

1.2. Related Work

We investigated the question of how to dynamically evolve systems in the context of an effort to develop a general purpose, language-based, distributed operating system (OS) [7, 13, 12, 11].

But what do distributed OS have to do with system evolution? Because of the enormous gap between the speed of local and remote accesses ($\approx 10^5$) performance of a distributed is usually unacceptable if the OS fails to dynamically switch to an adequate management strategy. A distributed OS must therefore posses extensive information on the current state of the system to be able to apply suitable strategies, such as replication or migration of remote objects [17]. But this can only be achieved, if the complete system, including its potential evolution is known to the OS at any time. To approximate this complete knowledge we employed an integrated, single system view. The OS and all applications within the distributed system are regarded as a single system. Clearly, to be able to introduce new applications into the single system or to evolve parts of the OS we needed new concepts for dynamic evolution of the running system.

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

Due to the broad scope of the OS research project, the work on system evolution described here, was itself influenced by various fields. Extensible and adaptable OS kernels [1, 4, 15] and incremental linking [5] were of strong interest to the OS community from the mid 80s to the mid 90s. Among the major results of this research direction are sophisticated implementation techniques for limited extensibility of OS kernels. Unfortunately, most of the work within this field focused strongly on technical issues neglecting semantic aspects of system evolution. We aimed at combining these implementation techniques with the more conceptual results achieved within the field of program transformation, such as in the early CIP project [3] and generative programming approaches [2, 6].

1.3. Outline

Section 2 sketches the STA system model that integrates space, time and level-of-abstraction. In Section 3 we discuss our notion of flexible incremental evolution based on the two concepts completion and generalization. Section 4 characterizes three different categories of components, which is a prerequisite to achieve sound semantics for system evolution. Based on this framework we are able to precisely define possible evolution steps during the lifetime of a dynamically evolving system in section 5 before we conclude in section 6.

2. The Integrated STA System Model

Conceptually sound system evolution requires an integrated system model that allows us to handle the various components and dependencies within a system in a flexible yet systematical way. We argue, that conventional models do not provide the required degree of integration but concentrate on specific aspects of a system that are not primarily related with evolution. For example, no system model known to the author considers both, level of abstraction and time. But both dimensions are relevant for system evolution. Some abstract notion of time is needed to record evolution histories and to be able to specify availability intervals for evolvable components. Levels of abstractions are needed to maintain the relationship between code and system in execution.

To improve this situation and to gain a better understanding of system evolution, we developed the STA system model. It integrates all dimensions relevant for system evolution within a single, homogeneous system model.

2.1. Dimensions of the STA Model

The STA system model embraces three dimensions:

- s: space
- t: time
- a: level of abstraction

The integration of these three dimensions accommodates the fact, that all levels of the system and all different representations of it – textual or the contents of machine level registers and main memory – are inherently related with each other. Usually this interconnections are just not systematically regarded, which in turn leads to inconsistencies between code and system and similar phenomenons.

At each point of time t the system is represented by a complete snapshot within the s/a-plane. This snapshot includes all artifacts contributing to the execution of the system; e. g. language-level classes, objects, instances, OS data structures, and hardware registers. All operations of the system, such as dynamic creation and deletion of data objects or modifications of the code are performed on t-snapshots.

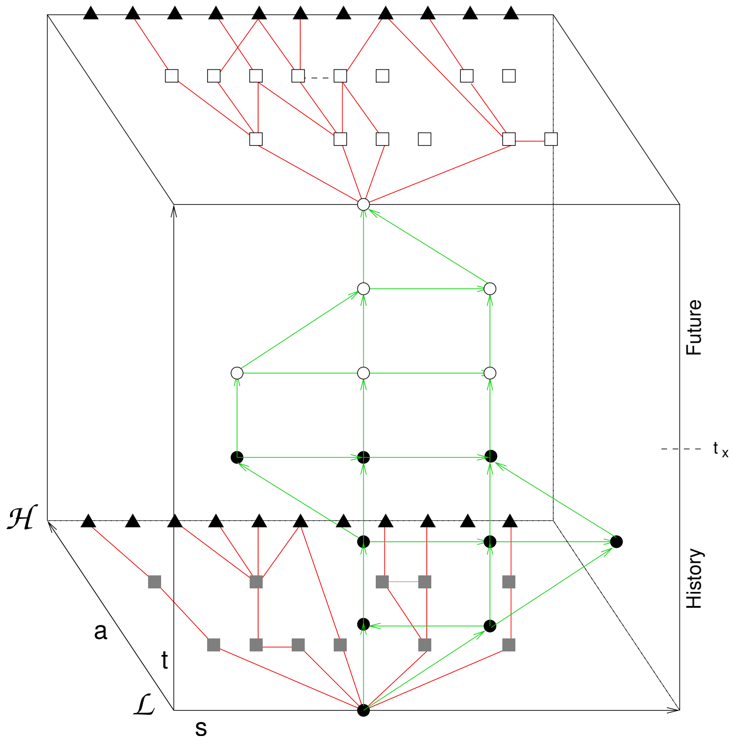

Figure 1. STA system model

2.2. Predefined Resources

Within these three dimensions a system is modeled as a set of interrelated *resources*.

Def. 1 *$\mathcal{H}$ is the set of physical resources such as registers, machine level instructions, processors, hard disk blocks or network interface; $\mathcal{H} \approx$ hardware facilities.*

Def. 2 *$\mathcal{L}$ is the set of predefined language level resources, such as the predefined type integer, conditional statements, or the grammar rules for the definition of functions; $\mathcal{L} \approx$ available programming language concepts.*

$\mathcal{H} \cup \mathcal{L}$ is the set of predefined resources $\mathcal{D}$. $\mathcal{D}$ predetermines the set of feasible systems given a hardware platform and a programming language. $\mathcal{H}$ and $\mathcal{L}$ furthermore constitute the two outer levels of the STA model in dimension a. Clearly, a software system specified by means of $\mathcal{L}$ is implemented by mapping its $\mathcal{L}$ resources to $\mathcal{H}$ resources. Thus, to comprehend a complete t-snapshot of a system during execution we have to regard multilevel bindings between $\mathcal{L}$ and $\mathcal{H}$ in addition to $\mathcal{D}$.

2.3. Intermediate Resources

For the construction of these bindings we need an additional class of resources to model abstract resources that are neither language nor hardware resources. We call these resources intermediate resources.

Def. 3 *Let $\mathcal{R}$ be the set of all resources. An intermediate resource is either*

- *a* reference *pointing to another resource $\hat{r}, r \in \mathcal{R}$*
- *a* tuple *of resources $(r_1, \dots, r_n), r_i \in \mathcal{R}$*
- *the* empty *resource ϵ*

Each resource is an instance of a class of resources and has a well-defined lifetime. Notice, lifetime establishes dependencies in dimension t.

This basic set of concepts is sufficient to model any real or abstract entity of a system, such as a main memory cell, stack frame or data object.

2.4. Bindings

Reference resources are the means to establish and detach bindings between resources.

Def. 4 *A binding between two resources r and s exists, if r references s at time t; $\rho_t(r, s) \Leftrightarrow r = \hat{s}$ at time t*

Similar to resources, each binding has a well-defined lifetime, too. The lifetime of a binding is crucial and is called *obligation interval*. For example, choosing a long-term upper bound for an obligation interval, will impede evolution steps at this part of the system for a long time whereas a very short term upper bound delivers greater flexibility for re-bindings but degrades system performance as bindings must be established more frequently.

The concepts of intermediate resources and bindings are powerful instruments. With these concepts efficient transitions between the language level concepts defined by $\mathcal{L}$ and the supply given with $\mathcal{H}$ can be constructed in a flexible and systematic manner.

3. Incremental Evolution

Figure 2 sketches our notion of stepwise evolution. System 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, i. e. completely specified components are replaced with incomplete ones. Step C_1 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.

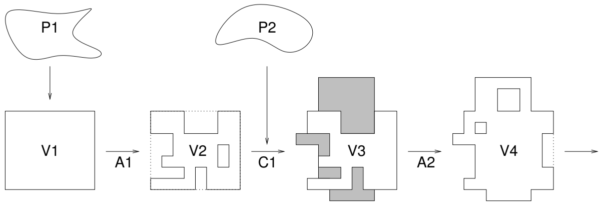

Figure 2. Incremental Evolution

The completion and generalization concept is detailed in [10]. Here, we give simplified definitions. Let $\mathcal{L} = \mathcal{L}^i \cup \mathcal{L}^c, \mathcal{L}^i \neq \mathcal{L}^c, \mathcal{L}^i \neq \emptyset$ and $\mathcal{L}^c \neq \emptyset$. $\mathcal{L}^c$ denotes the set of complete language resources, $\mathcal{L}^i$ the set of incomplete language resources, i. e. resources that need further bindings in dimension s for complete specification.

Def. 5 *A* generalization *of a system V is the replacement of one ore more bindings of intermediate resources $r \in \mathcal{I}$ to complete languages resources $\rho(r, c), c \in \mathcal{L}^c$ with bindings to incomplete resources $\rho(r, i), i \in \mathcal{L}^i$. The resulting more incomplete derivate V' is a flexibilized* abstraction *of V.*

Def. 6 *The transition from bindings $\rho(r, i), i \in \mathcal{L}^c$ to bindings $\rho(r, c), c \in \mathcal{L}^c$ is called a [partial]* completion *of V. Degrees of freedom are removed and a more concrete system V' produced.*

In fact, the principles of incomplete language resources and completion and generalization somehow already exist in practice in several peculiarities. Examples are:

- input of data at runtime
- parameterized subroutines
- substitution by preprocessors, design patterns, etc.

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. [8]), 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 are defined at a conceptual level without independent from 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.

4. Component Categories

A component is either an resource or a collection of interrelated resources. For a semantically sound association of code with dynamic execution three different component categories must be distinguished:

1. 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 posses a transient state.

2. *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 A is always considered a local component of an incarnation b; i. e. for each generator $A \in G_t$:

   $$\exists b \in X_t : G \in L(b),$$

   where $L(b)$ denotes the set of local components of b. Hence, generators are similar to classes in OO languages [16] but differ in being integrated into the dynamics of execution.

3. 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.

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 a misconception and an obstacle for dynamic evolution.

This briefly sketched 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 $G(a)$[4].

4.1. Dependencies

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.

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.

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 a set of complete language level concepts $\mathcal{L}_\mathcal{A} \subseteq \mathcal{L}^c$. $E(\mathcal{A})$ 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 $\mathcal{L}_A \subseteq \mathcal{L}^c$. $E(A)$ defines invariants for all incarnations $a \in X : A \overset{c}{\Rightarrow} a$.

- The set of properties $E(a)$ of a generator a is specified by means of $\mathcal{L}_a \subseteq \mathcal{L}^c$.

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. For example, it is not possible to have incomplete incarnations of a complete generator, whereas the opposite is possible and also desirable.

4.2. Example

We illustrate our component and evolution concept with the aid of the code excerpt shown in figure 3. 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 posses 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 after elaborating the corresponding generator-family in its declaration part. a_1 uses G_b to create an incarnation b_1 before recursively creating another a_2 using G_a. Now, a_2 elaborates a new generator G'_b, which will be used to create an incarnation b'_1.

The interesting part is, if we assume that the generator-family $\mathcal{G}_b$ is incomplete then we are able to perform different evolutions for the recursive computations of a_1 and a_2, because $E(G_b)$ must not be identical with $E(G'_b)$. It is even possible to chose between either individual evolutions

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

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

Figure 3. Evolution & Recursion

of the created b_i incarnations or evolution of the whole set of upcoming b_i incarnations by a completion of the generator-family $\mathcal{G}_b$.

5. Transitions

The state transition diagram shown in figure 4 summarizes all possible evolution steps of the system, based on the component category concept and the life-time and property dependencies introduced in section 4.1. 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).

6. Conclusion and Future Work

This paper presented a radically new vision for the evolution of dynamic systems. We introduced a conceptual framework, consisting of the three-dimensional STA system model that embraces space, time and level of abstraction. In addition to this we differentiated complete and incomplete language concepts and three different component categories to be able to support incremental generalization and completion of systems during execution in a semantically sound way. Within this integrated framework, code is directly attached to components and therefore participates in the dynamics of the computation.

Obviously, the conceptual STA model raises many new questions, which could not be answered within this paper and need further investigations. For example, what degree of flexibility can be achieved by incompleteness at the language level and which new language level concepts are desirable for this purpose? We believe, that the identification of this and other interesting questions is a major contribution of the STA model.

Finally, it seems important to mention, that we also implemented this conceptual model. The code excerpt shown in figure 3 is indeed part of the test suite of this implementation. [14] shows that the increased flexibility for evolution described in this paper can be achieved without significant constant performance degradation by means of an incremental dynamic link loader and modifications of the stack frame layout within the compiler [9].

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[2] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[3] F. L. Bauer et al. The munich project CIP, vol. 1: The wide spectrum language CIP-L. volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1985.

[4] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. Spin – an extensible microkernel for application-specific operating system services. In *Proceedings of the 6th ACM SIGOPS European Workshop, Matching Operating Systems to Application Needs*, pages 68 – 71, Dagstuhl Castle, Germany, September 1994.

[5] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems*, Annapolis, 1996.

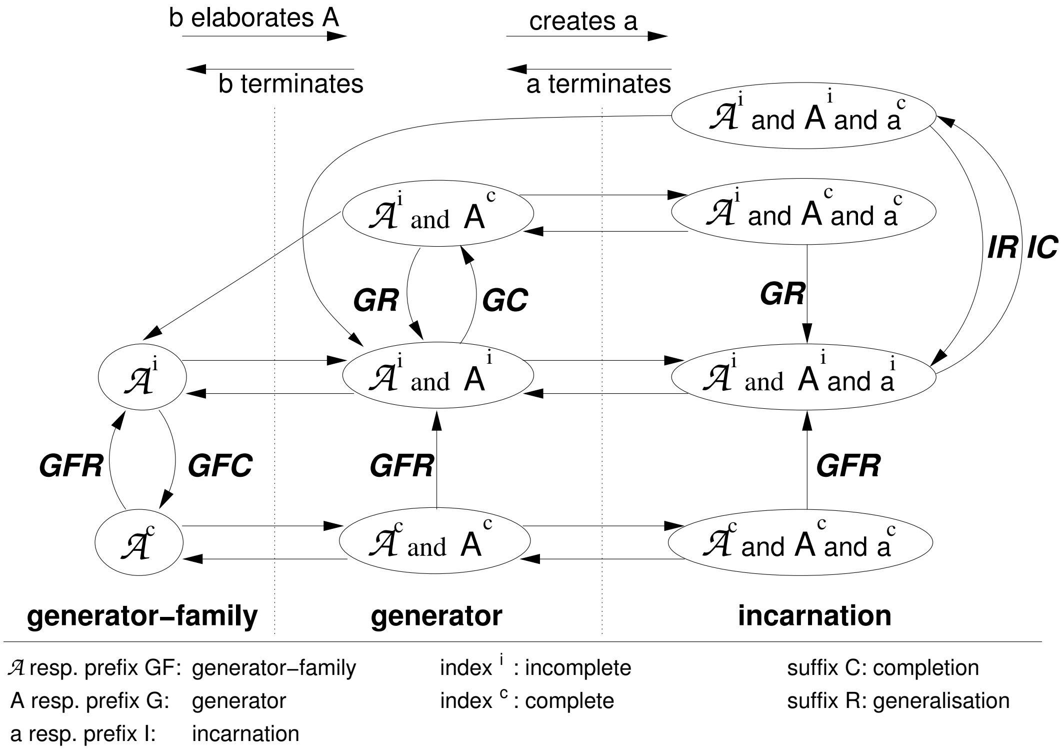
b elaborates A
b terminates
creates a
a terminates
$\mathcal{A}^i$ and A^i and a^c
$\mathcal{A}^i$ and A^c
$\mathcal{A}^i$ and A^c and a^c
IR
IC
GR
GC
GR
$\mathcal{A}^i$
$\mathcal{A}^i$ and A^i
$\mathcal{A}^i$ and A^i and a^i
GFR
GFC
GFR
GFR
$\mathcal{A}^c$
$\mathcal{A}^c$ and A^c
$\mathcal{A}^c$ and A^c and a^c
generator–family
generator
incarnation
$\mathcal{A}$ 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 4. State Transitions

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[7] 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.

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

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

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

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

[12] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, Jan. 1997. IEEE CS Press.

[13] M. Pizka, C. Eckert, and S. Groh. Evolving software tools for new distributed computing environments. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'97*, pages 87–96, Las Vegas, NV, July 1997.

[14] C. Rehn. Inkrementelles und dynamisches binden in einer verteilten umgebung. Master's thesis, Technische Universität München, Institut für Informatik, Feb. 1998.

[15] C. Small and M. Seltzer. A comparison of OS extension technologies. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 41–54, Berkeley, CA, USA, Jan. 1996. USENIX.

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

[17] 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.