Activity Diagram Inheritance[1]

Arnd Schnieders, Frank Puhlmann
Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam
{schnieders, puhlmann}@hpi.uni-potsdam.de

Abstract

This paper outlines the ongoing work on the realization of a flexible inheritance mechanism for Activity Diagrams that assures the maintenance of syntactical correctness for the derived Activity Diagrams. The objective is to support the reuse of process models especially by applying Activity Diagram inheritance as a variability mechanism in the context of product line oriented software development.

Keywords: Activity Diagrams, domain engineering, process inheritance, variability mechanism

1. Introduction

In industry similar products are frequently developed and produced as product lines. One of the main advantages is a gain of efficiency in development and production since parts, which are common for several product line members, can be reused optimally. This approach has been transferred successfully to software development and is also known by the name domain engineering. Variability mechanisms are thereby important for the effectiveness of domain engineering. A great number of variability mechanisms has already been published [5, 9, 11, 13, 18]. Unfortunately, existing variability mechanisms only refer to the static aspects of a software system's design while the impact of variability mechanisms on the process view on the system has been strongly neglected. Therefore, the first contribution of this paper is to contribute to closing this gap by making the important variability mechanism inheritance available for process design models in order to derive process model variants. The second contribution of this paper is to show how the defined process inheritance mechanism is realized concretely for UML 2.0 Activity Diagrams. UML Activity Diagrams are part of the Unified Modeling Language standard [14], which is the most important standard for describing software systems today. Activity Diagrams are suitable for the description of different kinds of processes like for example technical processes. Third, Activity Diagram inheritance is defined in a way that the preservation of

[1] The work reported in this paper has been supported by the German Ministry of Research and Education by the PESOA project [8]

the syntactical correctness of a process diagram variant is assured upon derivation from the original diagram. Thus using Activity Diagram inheritance, process design diagrams can be reused for similar products assuring the maintenance of syntactical correctness. Thereby development effort can be saved. The classification of this work in our long-term research activities, where a system-requirement driven reutilization of process design models is desired, can be found in 6.1.

This paper is structured as follows: Section 2 gives a definition of inheritance as used in this paper. Section 3 defines inheritance for Activity Diagrams, narrows down the class of Activity Diagrams regarded here and describes what will be considered as a syntactically correct Activity Diagram. Section 4 outlines the syntax-maintaining inheritance transformations deduced from the Activity Diagram inheritance definition and explains their application on the basis of a real motor control unit process. Section 5 gives an overview of existing work on process inheritance. Section 6 summarizes briefly this paper and gives an outlook for further investigations. Since this is a work in progress paper, there are still some important issues to be dealt with.

2. Clarification of the Inheritance Term

The first logical step in transferring an inheritance mechanism on Activity Diagrams is to clarify what inheritance actually means. Doing so it turns out that there is no unique normative definition of inheritance but that there are numerous ones, which can be quite different like for example the inheritance definition in [6] versus the inheritance definition in [12]. According to [20] there are actually three different conceptions that are typically mixed up with the term inheritance: subclassing, subtyping, and conceptual specialization. A definition of specialization is given by [15].

Subclassing is the type of inheritance that allows for the arbitrary redefinition, addition and cancellation of properties in subclasses. It is the type of inheritance that will be taken as a basis for the definition of an Activity Diagram inheritance mechanism because it offers the greatest flexibility in the reutilization of Activity Diagrams, which is our main objective. According to [20] subclassing can be formally defined as follows:

$$R = P \oplus \Delta R,$$

where R corresponds to the derived subclass, P denotes the ancestor class R is derived from, ΔR designates the properties in which R differs from P and $\oplus$ denotes the operation that in some way combines P with ΔR yielding R. The parts ΔR newly added to P may overlap with properties of P resulting in these properties being overwritten or cancelled.

3. Inheritance in Activity Diagrams

The inheritance mechanism for Activity Diagrams presented here is a mapping of the subclassing mechanism presented in Section 2 to processes. Thus inheritance between Activity Diagrams shall be defined as follows: a subactivity CA inherits from its superactivity PA according to the following schema:

$$CA = PA \oplus \Delta CA$$

ΔCA comprises elements that shall be newly added or that are already present in PA and shall be modified. By default modification of an element here means its replacement by another component. The replacing component may either be a single element or a subprocess, while the replaced element may only be a single element. Since a single element can invoke a new subprocess on his part, also entire subprocesses can be replaced. $\oplus$ designates the combination of PA with ΔCA that adds the new elements and replaces existing ones which are subject to modification.

3.1. Class of Activity Diagrams Regarded in this Paper

This paper will restrict itself on the IntermediateActivities compliance level of the UML 2.0 Activity Diagram specification. Accordingly the regarded Activity Diagrams consist of elements for modeling sequential, alternative, and concurrent control and data flows. The reduction on Intermediate Activities aims to achieve a reduction of the complexity of realizing inheritance rules for Activity Diagrams while they comprise modeling elements, which are absolutely sufficient for many applications.

3.2. Valid Activity Diagrams and Constraints for the Derivation of Subactivities

For the Activity Diagrams considered here it is assumed that they are correct according to the UML specification [14]. Nevertheless Activity Diagrams may exhibit intolerable errors without violating the UML specification. It shall be ensured that these kinds of errors are not introduced in accurate Activity Diagrams by using the Activity Diagram inheritance mechansm introduced here, i.e. the inheritance mechanisms shall be correctness-preserving. In this paper Activity Diagrams will be considered to be syntactically correct if they are in addition to the assumed correctness according to [14] free of:

- deadlocks
- livelocks, i.e. loops that don't terminate
- dead nodes, i.e. nodes that can never be reached during process execution

These correctness criteria shall be defined formally in the context of further investigations. In addition to the abovementioned syntactical constraints some notational restrictions shall be imposed for the sake of simplicity and clearness.

Constructions like the following (multiple edges flowing into or leading from the same Pin) shall be forbidden:

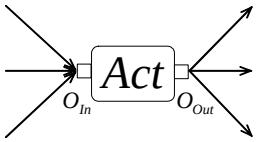

Figure 1. Multiple edges per pin are not allowed.

Instead the following semantically identical representation shall be used:

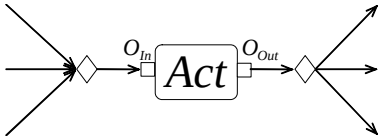

Figure 2. Construction to be used instead of multiple edges per pin

Incoming control flows are joined with the incoming data flows. If there is more than one incoming data flow it doesn't matter with which one the control flow is joined. Not allowed:

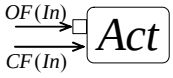

Figure 3. Actions shall not have incoming data and control flow edges at the same time.

Semantically identical representation to be used instead:

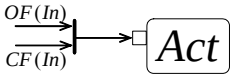

Figure 4. Construction to be used for Actions with incoming data and control flow

4. Realization of the Inheritance Mechanism

Table 1 and Table 2 summarize which process modifications are possible during inheritance according to the Activity Diagram Inheritance definition in 3.

Table 1. Possible replacements for Activity Diagrams

replaces	Simple Element	Subprocess
Simple Element	X	X
Subprocess	(X)	(X)

Table 2. Possible additions for Activity Diagrams

is added	
Simple Element	?
Subprocess	?

The third row in Table 1 indicates that Activity Diagram subprocesses may theoretically be replaced by simple Activity Diagram elements or subprocesses during inheritance. The crosses are thereby parenthesized because subprocesses are in fact treated as simple elements. Therefore the subprocess to be replaced is represented as a separate Activity and substituted by a Call Behavior Action invoking the new Activity. So arbitrary subprocesses, which can be encapsulated in a separate Activity can be replaced according to the rules for the substitution of Actions during inheritance. The addition of simple elements and subprocesses is still a matter of further investigations as indicated in Table 2.

4.1. Substitution of Simple Elements by Simple Elements

The tables below summarize, what types of simple Intermediate Activity Diagram elements can be overwritten by what other simple element types. The tables are to be read from left to right. An arrow pointing to the right means "can replace", an arrow pointing to the left means "can be replaced by". The braces indicate that a substitution is subject to certain constraints, which will not be discussed here in detail for spatial reasons.

Table 3. Action Substitutions

	Action	**CentralBuffer Node**	**DecisionNode/ MergeNode**	**ForkNode/ JoinNode**
Action	(↔)	(↔)	(→)	(↔)

Table 4. ObjectNode substitutions

	Pin	**DecisionNode/ MergeNode**
Pin	(↔)	
CentralBufferNode		←

Table 5. ControlNode substitutions

	Initial Node	Activity FinalNode	Flow FinalNode	Merge Node	Fork Node	Join Node
InitialNode	↔					
ActivityFinal Node		↔	→			
FlowFinal Node		←	↔			
MergeNode				↔		
ForkNode					↔	
JoinNode						↔

Table 6. ActivityPartition substitutions

	ActivityPartition
ActivityPartition	(↔)

4.2. Substitution of Simple Elements by Subnets

Up to now the correctness-preserving substitution of Actions with arbitrary numbers and types of input and output pins by structurally typical subprocesses has been investigated. Also Activity Diagram elements replaceable by Actions according to 4.1 can be subject to the same subprocess substitutions. Some of the investigated substitutions are presented in 4.4.

In principle for substitution any subprocess can be used that has the same input and output interface (i.e. the same numbers and types of Pins and incoming and outgoing edges) as the substituted component, that follows the restrictions for Activity Diagrams defined in 3.2 and that does not duplicate or delete any tokens in comparison to the substituted node. Moreover for every replacing subprocess it must be assumed that the execution time of the contained Actions is always finite, that the contained Activity Edges never dispose of unsatisfiable Guard-Expressions and that any Decision Node is constructed in a way that for every incoming token always exactly one of the edges running out of the Decision Node accepts the offered token.

4.3. Substitution of Subnets

Substitution of subnets shall be realized by reducing it to the substitution of Call Behavior Actions, which invoke the subnet to be substituted and that has been sourced out into a separate Activity. Therefore it has to be investigated what kinds of subnets can be sourced out into a separate Activity and thus may be replaced. This question is not trivial at all and will be subject to further investigations. In order to avoid problems (deadlocks, syntax violations, etc.) the subprocess encapsulation shall only be allowed according to the following rule for the moment: in an Activity Diagram A a subnet B can be sourced out into an Activity that is invoked by a Call Behavior Action t, if for any Activity Edge

$x \rightarrow y$ from B to A' (and for any Activity Edge $x \leftarrow y$ from A' to B) the following applies:

– x is an Action in B and y is an Action in A'
– any edge $x \leftarrow y$ leading from A' to B meets in the same Action x_s
– any edge $x \rightarrow y$ leading from B to A' runs out of the same Action xe

(A' corresponds to net A without subnet B). Figure 5 illustrates the abovementioned rule.

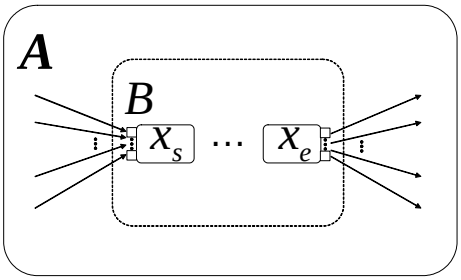

Figure 5. Allowed subprocess encapsulations

4.4. Example for the Application of Activity Diagram Inheritance

Due to a lack of space mechanisms for the replacement of subprocesses by other subprocesses will be introduced only by means of an example. The example shows a cutout of a motor control unit initialization process from which the process of a motor control unit with an integrated immobilizer system is derived using Activity Diagram inheritance. The constraints which have to be regarded for every substitution are omitted. Figure 6 shows the initial motor control unit initialization process without immobilizer system.

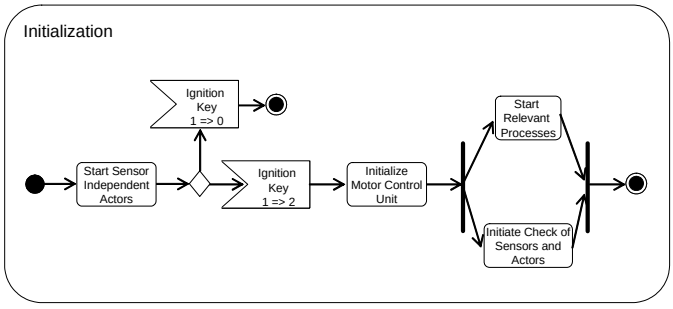

Figure 6. Initial motor control unit process

Figure 7 depicts the first inheritance step, where the Action "Initialize Motor Control Unit" is replaced by an Action sequence adding the new Action "Check Immobilizer Sensors".

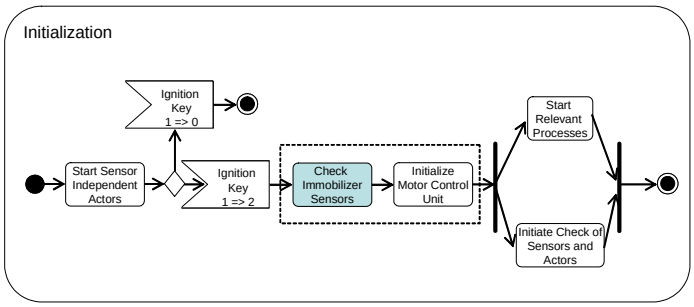

Figure 7. Motor control unit process after first inheritance step

This substitution follows the general schema for the substitution of an Action (and therefore also subnets that can be encapsulated as a separate Activity) with arbitrary numbers and types of input and output pins by an Action sequence as shown in Figure 8.

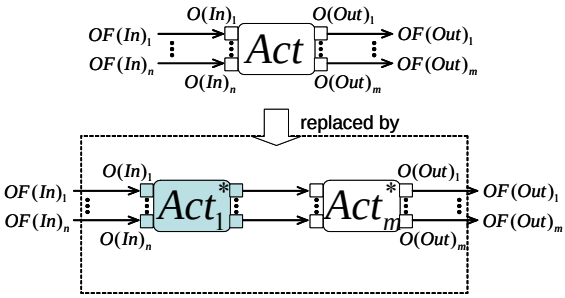

Figure 8. General schema for first inheritance step: sequence

In the next step shown in Figure 9 for the Action "Check Immobilizer Sensor" a subprocess with two parallel Actions is inserted adding the Action "Initiate Immobilizer Check".

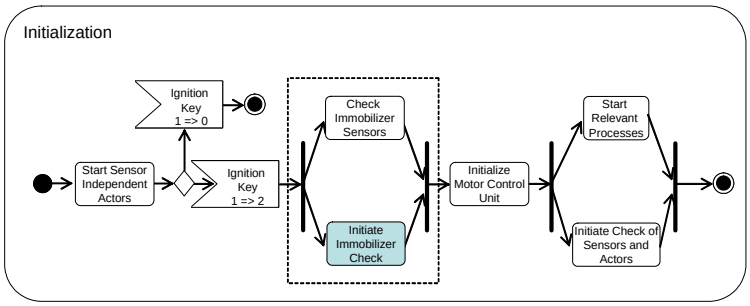

Figure 9. Motor control unit process after second inheritance step

The general substitution schema for the inheritance step shown in Figure 9 is depicted in Figure 10.

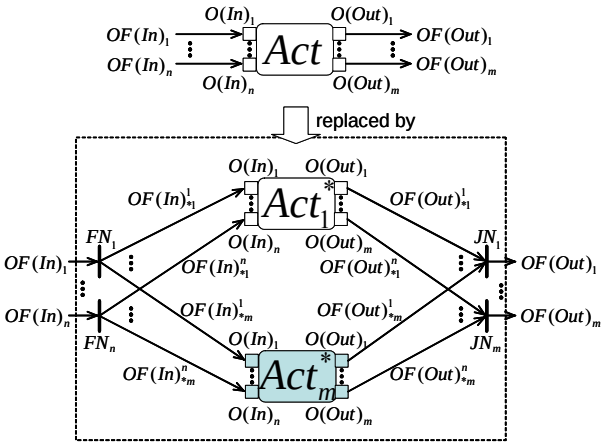

Figure 10. General schema for second inheritance step: parallelism

Figure 11 shows the final inheritance step, where the subprocess enclosed by the Actions "Start Sensor Independent Actors" and "Initialize Motor Control Unit" is replaced by the same process which will now be executed multiple times depending on whether the immobilizer has been deactivated or not.

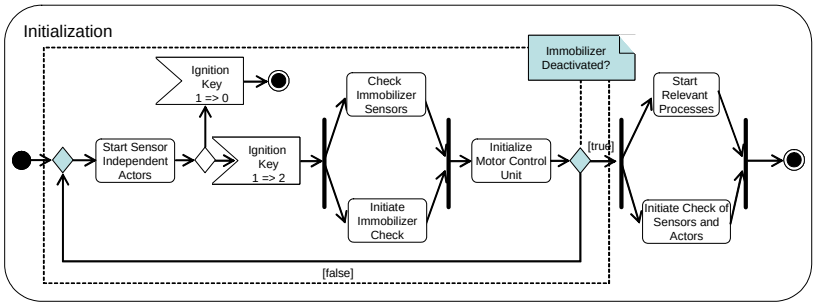

Figure 11. Motor control unit process after third inheritance step

The modification leading to the diagram in Figure 12 follows the general schema as shown in Figure 11.

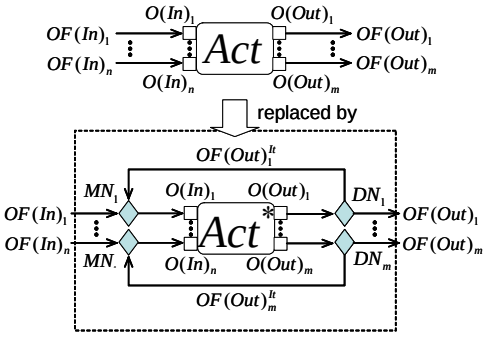

Figure 12. General schema for third inheritance step: iteration

5. Related Work

One approach for process inheritance has been presented in [2, 3, 1]. This approach is closer related to specialization and is much more restrictive than the inheritance approach introduced in this paper. The transformation rules it is based on can also be expressed by the inheritance transformations introduced in this paper. Another definition of process inheritance, which is much more generic and which is similar to the inheritance definition given in this paper, is introduced in [10]. However, this definition is given for arbitrary processes and it is not discussed how it shall be realized for a concrete process modeling

language. Other process inheritance approaches known to the author have been suggested for example by [19, 21] but since they are too specialized to a certain process modeling language they will not be regarded here. An overview of several process inheritance approaches can be found in [7].

Concerning the analysis of syntactical correctness of processes, one soundness definition has been given in [1], which could be used as a starting point for the definition of soundness for Activity Diagrams.

6. Conclusion

This paper has given a definition of Activity Diagram and has outlined syntax-maintaining inheritance transformations deduced from the Activity Diagram inheritance definition. What will be considered as a syntactically correct Activity Diagram has been stated informally. The formal definition of syntactical correctness for Activity Diagrams is an open issue of further investigations. Therefore a formalization of the Activity Diagram semantics is required. Moreover an example has been given in this paper of how Activity Diagram inheritance can be applied to derive syntactically correct process diagram variants. Concerning the syntax-maintaining inheritance transformations outlined in section 4 a more general mechanism for the encapsulation of Activity Diagram subprocesses has to be developed in the context of further investigations in order to make Activity Diagram inheritance more powerful. Also the syntax-preserving addition of Activity Diagram elements has to be analyzed.

6.1. Outlook

The long-term goal of our research is to define various variability mechanisms for the process design and to categorize them according to the relevant non-functional characteristics of their modification like the maintenance of the syntactical correctness of a process or the modifiability of the process according to respective metrics [16]. The idea is that using the right variability mechanisms the requirements of a system, which are realized by a corresponding system design [17], can be maintained while deriving process design variants for similar software products by means of the suitable variability mechanisms.

Figure 13 visualizes the abovementioned ideas. The upper ellipse shows possible non-functional requirements for software products. Syntactical correctness is the non-functional requirement relevant for software product A and A' whose process design model shall be derived from the process design model of software product A. Therefore syntactical correctness is highlighted. The lower ellipse contains various variability mechanisms with different non-functional properties they preserve if being used for the derivation of process variants. An inheritance mechanism shall for example have the property to be correctness-preserving. Therefore it is selected to derive a process design model for software product A' from software product A as shown in the lowest part of

the figure. This is possible, since inheritance shall be correctness-preserving and because the process of A shall also be, according to the requirements of A, syntactically correct.

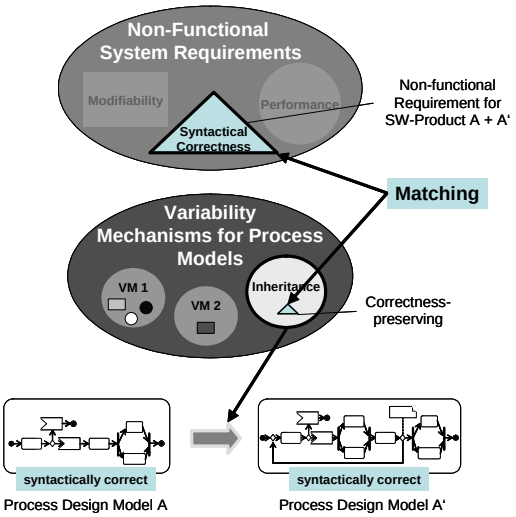

Figure 13. Requirement driven derivation of process variants by means of appropriate variability mechanisms

7. References

1. W.M.P. van der Aalst, K. M. van Hee. Workflow Management: Models, Methods, and Systems. The MIT Press, Cambridge, etc., 2002.
2. W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based approach. In P. Azema and G. Balbo, editors, Application and Theory of Petri Nets 1997, volume 1248, pages 62-81. Springer-Verlag, Berlin, 1997.
3. W.M.P. van der Aalst. Inheritance of business processes: Four problems – one solution. In Weber, H., Ehrig, H., and Reisig, W., editors, Proceedings of the Colloquium on Petri Net Technologies for Modelling Communication Based Systems, pages 1-28, Berlin, Germany, 1999. Fraunhofer-Institute for Software and Systems Engineering (ISST).
4. W.M.P. van der Aalst. Inheritance of Dynamic Behaviour in UML. In: D. Moldt (ed.), Proceedings of the Second Workshop on Modelling of Objects, Components and Agents (MOCA 2002), DAIMI 561, 105–120, Aarhus, Denmark, August 2002.
5. M. Anastasopoulos and C. Gacek. Implementing product line variabilities. IESE Report No. 089.00/E, November 2000.
6. Helmut Balzert. Lehrbuch der Software-Technik: Software-Entwicklung. Spektrum Akademischer Verlag, Heidelberg et al., 2. edition, 2000.
7. T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. Journal of Logic and Algebraic Programming, 47(2): 47-145, 2001.

8. BMBF-Project PESOA (Process Family Engineering in Service-Oriented Applications) Homepage. Internet-URL: www.pesoa.de [last accessed in December 2004]
9. Jan Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Harlow, England et al., 2000.
10. Christoph Bussler. Process Inheritance. Lecture Notes in Computer Science, 2348/2002:701, May 2002.
11. P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, Upper Saddle River, NJ 07458, 2002.
12. Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. ACM Computing Surveys 10(1): 29-72, 1988.
13. Ivar Jacobson, Martin Griss, and Patrik Jonsson. Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley Longman, Harlow, England et al., 1997.
14. OMG: Unified Modeling Language: superstructure. Version 2.0. 2003. Internet-URL: http://www.omg.org/cgi-bin/doc?ptc/2003-08-02 [last accessed in December 2004]
15. C. H. Pedersen. Extending ordinary inheritance schemas to include generalization. In Conference proceedings on Object-oriented programming systems, languages and applications, pages 407-417, 1989.
16. H.A. Reijers and I.T.P.Vanderfeesten. Cohesion and Coupling Metrics for Workflow Process Design. In J. Desel, B. Pernici and M. Weske, editors, Proceedings of the 2nd International Conference on Business Process Management (BPM 2004), Lecture Notes in Computer Science 3080, 290-305. Springer Verlag, Berlin, 2004.
17. Software Engineering Standards Committee of the IEEE Computer Society. IEEE Recommended Practice for Software Design Descriptions, Std 1016-1998
18. Mikael Svahnberg and Jan Bosch. Issues Concerning Variability in Software Product Lines, volume June of 146. Lecture Notes in Computer Science, 2003.
19. Markus Stumptner and Michael Schrefl. Behavior consistent inheritance in UML. Lecture Notes in Computer Science, 1920/2000:527-542, July 2003.
20. Antero Taivalsaari. On the notion of inheritance. ACM Comput. Surv., 28(3):438-479, 1996.
21. Guangxin Yan. Process Inheritance and instance modification. In Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work, pages 229-238. ACM Press, 2003.