ISSN 0105-8517

Position Papers from The 7th Workshop for PhD Students in Object-Oriented Systems

Frank Gerhardt
Lutz Wohlrab
Erik Ernst
(Eds.)

DAIMI PB – 526
September 1997

DATALOGISK AFDELING
AARHUS UNIVERSITET
Ny Munkegade, Bygn. 540
8000 Århus C

Position Papers from The 7th Workshop for PhD Students in Object-Oriented Systems

Frank Gerhardt, Lutz Wohlrab, Erik Ernst (Eds.)

Contents

Preface

This book contains the position papers accepted at The 7th Workshop for PhD Students in Object-Oriented Systems, which took place June 9-10, 1997, in Jyväskylä, Finland, in connection with the ECOOP'97 conference.

It is a tradition at ECOOP conferences to have a workshop for PhD students, conducted by the network of PhD Students in Object-Oriented Systems (PhDOOS). The purpose of this network is to help leveraging the collective resources of young researchers in the object community by improving the communication and cooperation between them. In a year of the PhDOOS network the workshop is the main event where we meet face-to-face. Between workshops we stay in touch through our mailing list. More information on the PhDOOS network can be found at `http://purl.org/net/PhDOOS`.

A conference workshop typically concentrates on a few topics chosen at the outset. For this workshop, the technical topics covered were derived from the research interests of the participants. Since the workshop had 36 participants, we partitioned the main group into several subgroups, each having a more focused research area as topic. The work in these subgroups had been prepared extensively by the participants. A little less than half of the participants had submitted a position paper. Everybody had prepared a presentation of his or her research work—a longer presentation for those participants with a position paper, and a shorter one for those who just provided a short abstract of their research work. The position papers are presented in this report. A comprehensive workshop report containing a short presentation of the research work of each of the 36 participants appear in the ECOOP'97 Workshop Reader.[1]

The technical sessions in subgroups were an important part of the workshop, but there were also other activities. In plenary sessions we heard two

[1]Unfortunately, the precise ISBN and LNCS numbers of the ECOOP'97 Workshop Reader were not available when this was printed

invited speakers, had a writer's workshop, and discussed issues related to the network itself. We also had a discussion about the conditions of being a doctoral student in various countries, as a followup to an email based discussion about this shortly before the workshop.

Our invited speakers were Prof. Mehmet Aksit from the University of Twente and Prof. Peter Wegner from Brown University. They spoke about their academic lives in retrospect, their current and future research, and the PhD-getting process in general. We were impressed by wide range of experience they demonstrated and thankful for the personal remarks they made regarding our profession. We think it is invaluable to get this insight when starting a career in academia or industry.

There were a couple of plenary sessions dealing with the network itself. We felt that the network is too inactive during the year, and that communication needs to be improved. The ECOOP workshop is good, but there ought to be more of other things, too. To make this happen, the activities in the network should become a natural and indispensable part of the daily work of the members, as opposed to a beautiful idea that we can play with after having finished our real work...

We picked up the idea from the year before to review each other's papers. While the previous approach intended to have reviews only before each ECOOP conference, we now want to start a continuous review process. It should be convenient and a good habit for members of the network to receive valuable feed-back from other members of the network about articles, books, or selected parts of such written work, before submitting them to a conference or publishing them. We also have to make sure that the authors feel assured their work is not "stolen" by anybody in this process. Since cooperation is a basic tool in research today, keeping the work secret is not an option. On the contrary, as soon as many people know that a particular idea or approach originally came from one group of persons, it will in fact be *better* protected against "theft" than without this community awareness. The network is a great resource of knowledge and inspiration, we just have to push a little bit to make it visible, accessible and useful for each member.

Another idea was to use the Internet more intensively to get in touch on a regular basis. Real meetings are great, but difficult to arrange. Therefore we want to try out "vitual meetings" using technologies like IRC or conferencing groupware. Whether in real life or via network cables, meeting other people and getting to know them is a necessary precondition for good, lively cooperation.

Finally we had to find the organizers of next year's workshop. Erik and Frank will continue for one more year. They are joined by Luigi Benedicenti (`Luigi.Benedicenti@dist.unige.it`). The homepage of the 1998 workshop is `http://purl.org/net/PhDOOS/1998`. If you want to join the network, take a look at `http://purl.org/net/PhDOOS`.

The following sections contain the position papers in a revised version which the authors produced shortly after the workshop. The articles are also available electronically, on the URL `ftp://ftp.daimi.aau.dk/pub/empl/eernst/phdws97/positionpapers`.

A Synchronization-Scheme Using Temporal Logic

László Blum
University of Veszprém
Department of Mathematics
and Computer Science
e-mail: bluml@almos.vein.hu

Abstract: A new synchronization scheme based on the reflective model of objects and propositional temporal logic with operator only for the past is suggested. Using the proposed scheme, not only state partitional and state modification anomalies can be avoided, but history-only sensitiveness inheritance anomalies [Mat93b] can be reduced as well. The scheme provides the separability between implementation and synchronization code in order to overcome the above anomalies.

Introduction

The central concept of concurrent object-oriented programming is the sharing of knowledge beside concurrency. The essence of knowledge-sharing is the re-use of the descriptions of objects. The advantage of knowledge-sharing is in part that modularity increases, and also that a possibility of hierarchical structuring opens. The tools of knowledge-sharing are subtyping and inheritance. These concepts mix strongly in sequential object-oriented languages. The difference between the two concepts is in the difference between the levels of abstraction. The inheritance is the concept of the level of implementation and means code-sharing, while the subtype is a concept of the specification, the behaviour description level and the subtype hierarchy is based on the behaviour of the object instances [Ame87].

The reflective model of objects supports inheritance in a natural way, because we can subdivide the description in the reflective model into small components, which can be united with other components to form a subclass easier. On the other hand, the monolithic handling of description leads to the delegation protocol that, as we have seen, does not result in the most effective way of code-sharing.

The parallel object-oriented languages offer language primitives and/or general object level schemes to program synchronization constraints. The most often used synchronization schemes in concurrent object-oriented programming are the following:

- A logical expression belongs to each method in the case of guarded methods, and only those methods can be selected for execution, the logical expressions of which are true.
- Synchronization with enabled sets. After the execution of a method, the object always specifies the set of those messages, the answer for which can be generated in the next cycles.

In case of the application of different synchronization-schemes, during the reuse of the code, various difficulties may arise, which are called inheritance anomalies in the literature. Generally, we talk of inheritance anomaly if some kind of difficulty arises out of the synchronization in the re-use of the implementation code. We can find several solutions for getting rid of the inheritance anomalies in [Mat93]. The solutions are based on the fact that the occurrence of the inheritance anomalies depends on the applied synchronization schemes. Using only one synchronization scheme, the anomalies can occur easily, while changing the schemes, they can be avoided. The localization of the synchronization code and scheme to the given object gives an opportunity for this. Thus we can apply a completely different synchronization-scheme in the sub-class than in the parent class. The distribution of the synchronization code between the objects can be done similarly to the inheritance of the methods. The above purpose can be reached for example with the use of synchronizers and transition specifications as synchronization schemes.

In the present work, a new scheme with high abstraction level is proposed, which, first of all, can be used as a specification tool, but it can be used as an implementation tool as well.

The setup of this paper is the following: in the first part, we shall give a description of the object model we work with, then a new synchronization scheme, called synchronization set is introduced and tested on new or known problems that lead to inheritance anomalies, and finally, a proposal for implementing PTPTL formulas used in the scheme, is given.

The Object-Model

In our paper, we shall refer to the reflective model of objects of the kinds described below. In the reflective model, [Tom89] every object consists of the (recursive) composition of four objects: Meta Object, Container Object, Processor Object, Mailbox Object:

- The Meta Object manages the three other objects.
- The Container Object stores the acquaintances of the object.
- The Processor Object can change the state of the object upon receiving an enabled message from another object.
- The Mailbox Object stores asynchronously received messages (requests for method-executions) from other objects and synchronizes the object: uses a policy for choosing the next enabled message.

When the object executes a request, the Processor Object will be blocked, and no other request can be enabled until the execution has been done.

Assigning temporal logical atomic formulas to actions

In our model, like in [Ara95], a truth-value for each atomic formula will be given to every method request and method execution. For a method *m1*, *m1* means both the name of the method and an atomic formula having a truth-value that corresponds to the execution state of the method in each time-point. In addition, we introduce an atomic formula *req_m1,* which describes whether there has been a request for the method *m1* or not. An atomic formula *req_m1* is true until there is a request for method *m1* in the Mailbox. A temporal formula of a method expressing that it can be executed if there is no request for method *m3* and the previously executed one was method *m4*, is the following,

$$\neg req\_m3 \wedge \bullet m4$$

Since we want to use temporal expressions for synchronization, we have to define the time-points of the Kripke-structure of the object. Since only past-time temporal operators are used, it is enough to build up a Kripke-structure up to the present. Taking this into account, the next time-point to the Kripke-structure of the object is given when a request for executing the method is satisfied. The period between two executions can be looked at as a container period, during which the atomic formulas assigned to messages are given a value representing the next time-point.

Choosing requests from the Mailbox

In our model, the Meta Object of an object tries to send a request to the Mailbox to get the next accepted request for a method after every request-execution or arrival of a new message. If successful, it makes the Processor Object execute the request.

A New Synchronization-Scheme

Further in this section, a new synchronization-scheme will be presented[Blu96]. The abstraction-level of the scheme is rather high, because it uses temporal logical formulas for synchronization. The scheme is an extension of the well-known guarded methods, where the constraints of a method are collected in a set so that they can be expanded when inherited. Using the scheme, the state modification and the state partitioning anomalies [Mat93b] can be resolved and the history-only sensitiveness anomalies radically reduced.

Past-Time Propositional Temporal Logic (PTPTL)

In the model, PTPTL formulas are used to give constraints for method-execution. Past-time operators of PTPTL are similar to those used by [Arapis], extended with operators *atprev*, *punless*, *pwhile* and *after*. For a Kripke-structure *K* [Kröger] and a time-point *i*, the semantics of the operators can be defined in the following way:

$K_i(a\ atprev\ b) = t$	iff for the largest $j<i$ where $K_j(b)=t,\ K_j(a)=t$.
$K_i(a\ punless\ b) = t$	iff $K_j(b)=t$ for some $j<i$ and $K_k(a)=t \quad \forall k: j<k<i$
	or $K_k(a)=t\ \ \forall k: 0 \le k < i$
$K_i(a\ pwhile\ b) = t$	iff $K_j(b)=f$ for some $j<i$ and $K_k(a)=K_k(b)=t \quad \forall k: j<k<i$.
$K_i(a\ after\ b) = t$	iff for all $K_j(b)=t$ (j<i) there is $j<k<i$ such that $K_k(a)=t$.

Synchronization Sets

In the model developed, the key structure is a set consisting of the elements (called synchronization element),

[*method_name*, *tf_set*]

where *method_name* denotes a method of the object and *tf_set* is a set of PTPTL formulas. Every object has exactly one synchronization set by which the methods of the object can be synchronized.

Definition 1: A set *tf_set* of PTPTL formulas is called true at a time-point of a Kripke-structure, if each formula in the set is true.

Taking an object *O* with a synchronization set *S,* a request *req_method1* in the Mailbox may be satisfied:

1. if there is an element in *S* the method of which is *method1* and its formula set is true,
2. if there is no element in *S* the method of which is *method1*.

Evaluating Temporal Formulas

A PTPTL formula can be easily evaluated at the present without changing the synchronization variables. In our model, a temporal formula can change its present value, because requests for methods can change the values of atomic formulas belonging to them (See Figure 10.). But this change is not confusing, since most of the temporal operators are affected by the past time-points and the values of atomic formulas at the past time-points are unambiguous. We can view it as a trying period of a time-point, where we can test how the formulas are changed by atomic formulas. When one of our requests is satisfied, we view it as a new time-point to which the values set during the period belong.

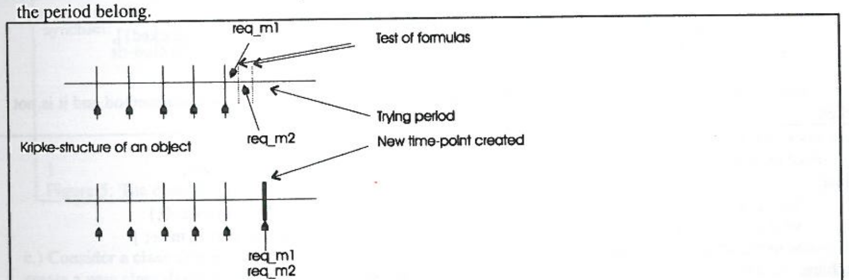


Figure 10. Assigning atomic formulas to a new time-point

Operations with Synchronization Sets

In case of inheritance, the synchronization set of the descendant objects can be established by using the following operators on synchronization sets:

Let *S1* and *S2* be two synchronization sets and MS is a set of method names and *tf_set* is a set of PTPTL formulas, then

a. *S1+S2* means the union of sets *S1* and *S2*
b. *S1++[MS, tf_set]* denotes the union of the formula sets of those elements of *S1*, that first element of which are members of *MS*. If *MS* contains all of the methods of *S1*, then this may be denoted in short by *S1*++tf_set*.

For example, let *S1={[method1, {form1, form2}], [method2, {form3, form4}]}*
S1++[{method1},{form5}]={[method1, {form1, form2, form5}], [method2, {form3, form4}]}

Applying the operator + , the descendant object can revise the synchronization set of the ancestor object by adding constraints to the methods that are independent of those used in the ancestor object. With the operator ++, more constraints can be added to the constraints of the ancestor.

Remark 1: Combining operators + and ++, we can absolutely revise the constraints inherited from the ancestor, if we use the operator ++ with set {False} and establish the new constraints using operator +.

Using Synchronization Sets to Avoid Inheritance Anomaly - some examples

Partitioning of Acceptable States Anomaly

Since the synchronization set is an extension of guarded methods, this kind of anomaly can not occur [Mat93].

Modification of Acceptable States Anomaly

The classic example for this kind of anomaly is in [Mat93b]. Consider an abstract mix-in class *Lock* capable to lock and unlock. Inheriting the class *Lock* together with another class *C*, the descendant object can lock and unlock its methods. In the example, the class *b-buf* is an implementation of the bounded buffer, which we want to be lockable. The two ancestor classes are described by an ABCL-like syntax (see figure 1. and 2.). We introduce the notation *synch_set:* to define the synchronization set of an object. The synchronization set LockS describes that method *lock* is enabled if *locked=false* and *unlock* is enabled if *locked=true*. Combining classes *b-buf* and *Lock* we would like to get a descendant class *lb-buf*, which is a lockable bounded buffer (see figure 3.). The synchronization set *lb-bufS* is established by expanding the formula sets of each element of *b-bufS* by formula *¬locked* and expanding this set by *LockS*:

lb-bufS={[put, {in<out+SIZE, ¬locked}], [get, {in>=out+1, ¬locked}], [lock, {¬locked}],
[unlock, {locked}]}

In this way, in the class *lb-buf,* the method *get()* is enabled, when it is enabled in the ancestor method and it is not locked.

```
Class Lock: ACTOR {
        bool locked
public:
        void Lock() { locked=0;}
        void lock() {locked=1;}
        void unlock() {locked=0;}
synchset:
        LockS={[lock, {¬locked}], [unlock, {locked}]}
}
```
Figure 1: The class lock, synchronized by synchronization set

```
Class b-buf: ACTOR {
        int in, out, buf[SIZE]
public:
        void b-buf() { in=out=0;}
        void put( int item ) {in++; }
        int get () {out++;}
synchset:
        b-bufS={[put, {in<out+SIZE}], [get,
                                        {in>=out+1}]}
}
```
Figure 2: The class bounded buffer

```
Class lb-buf: b-buf, Lock {
public:
        void lb-buf();
synchset:
        lb-bufS=b-bufS*++{¬locked}+LockS
}
```
Figure 3: The lockable bounded buffer

History-Only Sensitiveness Anomaly

Since the scheme developed gives a stronger tool for this type of anomaly, more examples will be given.
a.) Consider a subclass of *b-buf* (Figure 2.), *gb-buf* [Mat93], that has only one new method *gget()*. *gget()* works as the method *get()*, but is enabled only if *put()* has not been invoked right before it. *gb-buf* is shown in figure 4.

```
Class gb-buf: b-buf {
public:
        int gget() {out++;}
synchset:
        gb-bufS=b-bufS+{[gget, {in>=out+1 ¬●put}]}
}
```
Figure 4: The class *gb-buf*

b-bufS was expanded only with one element belonging to *gget()*. The temporal formula holds if no *put()* was executed at the previous time-point. We can see how easy it is to resolve anomalies like this, putting temporal operators in guards.

b.) The next subclass, *sb-buf*, of *b-buf* adds a new method, *is_full()*, that indicates whether the buffer is full. In *sb-buf*, the pattern of three *get()* methods immediately followed by two *put()* methods repeats, but there is an exception: *is_full* can interrupt the chain of three *get()*-s and two *put()*-s and get the chain start again.(See Figure 5.) It can be seen how great of the expressive power of PTPTL is. In *sb-bufS*, the first part of the PTPTL formula set of *get()* expresses that *get()* is enabled if two *put()*-s were executed after a *get()* or *is_full()* was executed last or the method was called after the creation of the object (because of the definition of ◗).

```
Class sb-buf: b-buf {
public:
        void Sb-buf{ }

synchset:
        sb-bufS=b-bufS+{[get, { (◗put and ◗●put and ◗●●get or ◗is_full) or
                                (●get and ●◗put or ◗is_full) or
                                (●get and ●●get and ●●◗put or ◗is_full)],
                          [put, {(◗get and ◗●get and ◗●●get or ◗is_full) or
                                (●put and ●◗get or ◗is_full)]}
}
```
Figure 5: The class sb-buf

c.) Consider a class *democ* with methods *m1* and *m2* and synchronization set *democS* (Figure 6.). We would like to create a new class *desdemoc* with the same methods and with the same synchronization, but in addition *m3* is enabled if and only if *m2* and *m1* were last requested at the same time.

```
Class democ: ACTOR {
public:
            void Democ() {...}
            void m1(){...}
            void m2(){...}
            void m3(){...}
synchset:
            democS={...}
}
```
Figure 6: The class democ with some methods m1, m2 and m3

```
Class desdemoc: democ {
public:
            void Desdemoc() {...}
            void m3(){...}
synchset:
desdemocS=democS++[{m3},{◆req_m1 and ((req_m1∧req_m2) atprev (req_m1∨req_m2))}]}
}
```
Figure 7: The class desdemoc

We can see in Figure 7. the new class *desdemoc*. We only added one more strict constraint to the formula set of method *m3*. The formula holds if *m1* has been requested, and if *m1* or *m2* have been requested recently, then both were requested at the same time.

Application of synchronization sets

In [Blu97], we gave an implementation method for temporal formulas in the synchronization sets. The method is based on the recursive expressiveness of temporal operators, and on the fact that we can evaluate these formulas storing the previous values of formulas in newly introduced variables. Using the method, the number of new variables is three times the number of binary temporal operators in formulas plus the number of unary operators.

In the future we want to examine how to prove the correctness of a synchronization of an object and how to describe an object using the above-mentioned scheme.

Conclusion

In the paper, a new synchronization scheme was developed. The abstraction level of the scheme is rather high, so we can look at the scheme as either specification or implementation of synchronization of methods. The scheme is an extension of guarded methods, where the constraints belonging to methods are contained in a set (called synchronization set). The synchronization and implementation codes are separated, so both codes can be inherited separately.

In [Fer95], the known inheritance anomalies are suggested to be solved by (nested) Conditional Critical Regions. We can easily find the relationship between nesting the CCR for inheriting the ancestors' synchronization and using the operator ++ for adding new synchronization constraints to the parents'.

In the future, what we are also developing is to decide whether a synchronization defined by a synchronization set contains a contradiction and whether it can be a correct synchronization. With the method, we can answer the first question based on the fact that a synchronization set can be expressed by one large PTPTL formula. Extending PTPTL with future-time temporal operators [Ara95][Kes93], we can label a formula expressing that for every time-point there will be a combination of messages satisfying the synchronization criteria. We can find arbitrary Kripke-structures satisfying the synchronization criteria by using the tableau method for full propositional temporal logic [Kes93].

References

[Agh86] G. Agha: Actors: A model of concurrent computation in distributed systems, MIT Press, Cambridge, 1986.

[Agh90] G. Agha: Concurrent object-oriented programming, Comm. of the ACM Vol. 33 No. 9, pp. 125-141, 1990.

[Ara95] C. Arapis: A Temporal Perspective of Composite Objects, In: Object-Oriented Software Composition Ed. by O. Nierstrasz & D. Tsichritzis, pp. 123-152, Prentice Hall 1995.

[Ame87] P. America: Inheritance and Subtyping in a Parallel Object-oriented Language, LNCS Vol. 276, pp. 234-242, 1987.

[Ame89] P. America: Issues in the Design of a Parallel Object-oriented Language, In: Formal Aspect of Computing, pp. 366-411, 1989,.

[Baq94] C. Baquero, F. Moura: Concurrency Annotations in C++, ACM SIGPLAN Notices Vol. 29, No. 7, pp.61-67, 1994.

[Blu96] L.Blum, L. Kozma: Inheritance in Object-Oriented Programming, I. National Conference on Object-Oriented Programming, 1996. (In Hungarian)

[Blu97] L.Blum, L. Kozma: Implementation Problems of a New Synchronization Scheme, Fifth Symposium in Programming Languages and Software Tools, Jyvaskyla 1997.

[Bri87] J-P. Briot, A. Yonezawa: Inheritance and Synchronization in Concurrent OOP, LNCS Vol. 276, pp. 32-40, 1987.

[Cre91] S. Crespi Reghizzi, G. Galli de Paratesi, S. Genolini: Definition of reusable concurrent software components, LNCS Vol. 512, pp. 148-166, 1991.

[Dec90] D Decouchant, P. le Dot, M. Riveill A Synchronisation Mechanism for an Object Oriented Distributed System, Bull IMAG, Z.I. de Mayencin - 2, rue Vignate 38610 Gieres -France,1990.

[Hew77] C. Hewitt: Viewing control structures as patterns of passing messages, J. Artifi. Intell. Vol. 8, No. 3 , pp. 323-364, 1977.

[Kes93] Y. Kesten, Z. Manna, H. Mcguire, A. Pnueli: A Decision Algorithm for Full Propositional Temporal Logic. In 5th Conference on Computer Aided Verification, LNCS 697, Springer-Verlag, 1993. pp. 97-109.

[Krö87] F. Kröger: Temporal Logic of Programs, Springer-Verlag, Berlin, Heidelberg, 1987.

[Neu91] C. Neusius: Synchronising Actions, In: Proc. of the ECOOP'91, pp. 118-132, 1991

[Mey92] B. Meyer: Eiffel, the Language, Prentice Hall, Englewood Cliffs. N. J.,1992.

[Yon86] A. Yonezawa, J-P. Briot, E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1 Association for Computing Machinery, pp. 258-268, 1986.

An Object-Oriented Approach to Software Architecture Specification *

Carlos Canal **Ernesto Pimentel** **José M. Troya**

Dept. of Computer Science and Languages, University of Málaga

Campus de Teatinos, 29071 Málaga, Spain

{canal,ernesto,troya}@lcc.uma.es

Abstract

Architectural specifications of software systems show them as a collection of interrelated components, and constitute what has been called the Software Architecture level of software design. It is at this level where the description and verification of structural properties of the system are naturally addressed. Besides, the use of explicit descriptions of the architecture of software systems enhances system comprehension and promotes software reuse. Despite several notations and languages for architectural specification have been proposed, some important aspects of composition, extension and reuse have not been properly addressed, and deserve further research. Our approach tries to address some of these open problems by combining the use of formal methods, particularly process algebras, with concepts coming from the object-oriented domain, such as inheritance, polymorphism, and parameterization.

Keywords: Software Architecture, formal methods, π-calculus, compatibility, inheritance

1 Introduction

The term Software Architecture (SA) has been recently adopted referring to the level of software design in which the system is represented as a collection of computational and data elements, or components, interconnected in certain way. From this point of view, we can consider SA as the level where the architecture and structural properties of software systems are described. SA focuses in those aspects of design and development which cannot be suitably treated inside the components that form the system [18]. Among them are included those which derive from the structure of the system, i.e. from the way in which its different components are combined.

The significance of explicit architectural specifications of software systems is twofold. First, they raise the level of abstraction, facilitating the description and comprehension of complex systems. Second, they increase reuse of both architectures and components.

*This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0930-C02-01.

However, effective reuse of a certain architecture often requires that some of its components can be removed, replaced, and reconfigured without perturbing other parts of the application [14].

Although object-orientation can be applied to all levels of software design, in SA the more general term *component-oriented* is preferred, allowing to consider not only objects but architectures, interaction mechanisms and design patterns as first-class concepts of a software architecture [13]. However, most concepts coming from the object-oriented paradigm can be almost directly applied to SA. In particular, we are interested the application of inheritance, parameterization and polymorphism to the specification of software architectures.

Despite of the importance of architectural aspects in software development, their descriptions have been traditionally limited to the use of certain idioms [18], such as *client-server architecture*, *layered architecture*, etc. These textual indications were generally accompanied with informal box-and-line diagrams. However, these descriptions lack of a precise meaning, which limit their utility to a great extent [1].

Only in the 90's appeared the first Architectural Description Languages (ADLs). ADLs address the need for expressive notation in architectural design. They try to provide precise descriptions of the *glue* for combining components into larger systems. However, most of the work is to be done yet [6]. While the proposed notations seem useful for the description of complex software systems, most of them are not formally based (see Section 6), which prevents the analysis and proof of the properties of the systems and architectures described. In addition, several significant issues, such as parameterization or inheritance, are not usually addressed.

2 The Role of Formal Methods

Our interest focus on the application of formal methods to SA. Formal specifications have a precise meaning derived from the semantics of the notation used and they admit several forms of reasoning, providing a formal basis for ADLs and allowing the development of verification tools.

The success on the application of formal methods to SA depends on the ability in finding models and formalisms adequate to this level of the development process. To this effect, process algebras are widely accepted for the specification of software systems, in particular for communication protocols and distributed systems. The systems so specified can be checked for equivalence, deadlock freedom, and other interesting properties.

In particular, we propose the use of the π-calculus [11] for the specification of software architectures. The π-calculus is a simple and powerful process algebra which can express directly *mobility* [5], making easier the specification of dynamic systems. Mobility is achieved in the π-calculus by passing channel names as arguments of messages. Since the names received can be used as channels for future transmissions, this allows an easy and effective reconfiguration of the system.

The calculus permits the specification of both *local* and *global choices*. Local choices indicate that a process may decide, not depending on its context, to commit to a certain transition or not, and model nondeterministic behavior. Global choices indicate that a process only commits to a transition after an agreement with another process performing the complementary action (either by local or global choice), and model deterministic

behavior. Thus, using the π-calculus, we are able to express whether a process can initiate a certain communication (and then it *requires* that its environment could follow its decisions), or it just *offers* a certain behavior to its environment.

The formal basis of the π-calculus permits the analysis of the specifications for bisimilarity, deadlock and other interesting properties, and also the development of automated verification tools [19]. However, the π-calculus is a low level notation, which makes difficult its direct application to the specification of large systems. Hence, a higher-level notation is required. Formal specifications in π-calculus can be incorporated into the description of components by extending one of the existing ADLs. As it will be shown in the following sections, the use of formal specifications in our approach address issues of system composability, extensibility, and parameterization in a similar way to their treatment in the object-oriented paradigm.

3 Composability

One of the properties that we may analyze in a software architecture is *composability*, which we could define as the capability of the system of being composed by combining its components as indicated in its architecture. Composability can be checked by determining whether the components of the system are *compatible* or not. Furthermore, in order to enhance reusability, we should be able to check if a certain existing component can be used in a new system where a similar function is required. Again the intuitive notion of *compatibility* arises.

System compatibility could be determined by composing in parallel the elements under checking. Then, the resulting system would be analyzed for deadlock. However, this would be impractical for complex systems, as it requires the analysis of all the interaction traces of large specifications. Instead of that, we propose the use of explicit interface specifications, or *roles*, for each connection or *attachment* between components of the system, indicating the behavior of those components as seen from outside. Then, each attachment between roles is checked locally for compatibility. This reduces the complexity of the analysis to a great extent.

In our approach, a software component is specified by a set of roles, which describe its behavior in relation to the other components it is attached to. Thus, roles may be considered as partial specifications of the interface of a component. As we usually want to attach roles that match only partially, equivalence checking, using for instance the bisimilarity relations established for the π-calculus, is not well suited for our purposes. Thus, we have defined a relation of protocol compatibility in the context of π-calculus. This relation ensures that two components, represented by a pair of roles, will be able to interact without deadlock until they reach a well-defined final state. In this context, the analysis of the composability of a software architecture is reduced to local analysis of compatibility. Compatibility ensures that any software system built according to the specifications of the architecture will not produce a deadlock caused by the interaction in any attachment between its components.

A formal proof of the properties of compatibility is out of the scope of this paper, but it can be found in [4]. Compatibility analysis can be easily automated in a similar way to the characterization devised by Sangiorgi for the bisimilarity relations [16].

4 Extensibility

In order to promote effective reuse of both components and architectures a mechanism of redefinition and extension of roles and components is required. In the object-oriented paradigm reuse is achieved by inheritance and polymorphism. Inheritance refers to a relation among object classes by which a heir class inherits the properties (methods and attributes) of its parent classes, while it can extend them by adding its own properties. The inherited properties may be redefined, usually under certain restrictions, but roughly speaking, we may say that the interface of the heir class includes those of its parents. Inheritance is a natural condition for polymorphism, as it ensures that the derived classes will have at least the same properties than their parents. Thus, a relation of inheritance would be also of use for specifications of software components.

However, in our context the interface of a component is defined not only by the signature of its properties (i. e. the signature of its roles), but this interface also includes the behavioral patterns described in the roles. Thus, redefinition of behavior is restricted by several conditions, which define what we have called a relation of inheritance among roles. These conditions ensure that role compatibility is closed under inheritance, in the sense that if two roles are found compatible, any derived role related to one of them by inheritance will be also compatible. Once again, we refer to [4] for a formal definition of role inheritance and its properties.

Role inheritance can be easily extended to components. A child component inherits its roles from its parents, while redefinition is restricted by the conditions defined for role inheritance. Since the relation ensures the maintenance of compatibility and deadlock freedom, it defines the conditions for polymorphism to take place, allowing the substitution of a component in any system by a specialized version which inherits from the former, with no need of checking the compatibility of the modified attachments. This gives place to a mechanism of architecture instantiation, by which a software architecture can be considered as a generic *framework* [14] which can be partially instantiated and reused as many times as needed.

Component frameworks derive from the idea of design patterns, and they represent the highest level of reusability in software development: not only source code and single components, but also architectural design is reused in applications built on top of the framework [15]. Since our specifications can be verified for compatibility, this promotes both software reusability and quality.

5 Example

We will show the implications of our approach by means of a well-known example. Consider the architecture of a typical `Producer-Consumer` system. This architecture is built from three interconnected components: a `Producer`, a `Consumer`, and a temporal store that we can describe as a `Buffer`. Suppose that the `Producer` generates several items and sends them to the `Buffer`, using an operation `in`, until it decides to quit, which is notified by sending a predefined event `wquit`. On the other hand, the `Consumer` retrieves items from the `Buffer`, using and operation `out`, and performs some computations with them. The `Consumer` ends when it has got all the items generated by the `Producer`, which is notified by the `Buffer` with an event `rquit`.

```
component Producer-Consumer {
  interface empty;
  composition
   p :  Producer;
   c :  Consumer;
   b :  Buffer;
  attachments
   p.Writer(in,wquit) <> b.Input(in,wquit);
   c.Reader(out,rquit) <> b.Output(out,rquit);
  }
```

```
component Producer {
  interface role Writer(w,q) = ···;
  }

component Consumer {
  interface role Reader(r,q) = ···;
  }
component Buffer {
  interface
   role Input(i,iq) = ···;
   role Output(o,oq) = ···;
  }
```

Figure 1: Architecture of a Producer-Consumer system

From the description above, we can derive the specification of the interface of the components. Events `in` and `wquit` form the interface between the `Producer` and the `Buffer`, while `out` and `rquit` form the interface between the `Buffer` and the `Consumer`. Figure 1 (left) shows what could be the specification of our system in a typical ADL. The `Producer-Consumer` system is a composite which contains three sub-components, each one instance of a certain component class. These subcomponents (to be more exact the roles that represent their interfaces) are attached to each other, making the previously described interfaces.

The correctness of the attachments can be determined by static analysis, checking the conformance of the names of the operations and the type of the parameters, but mere type checking doesn't ensure that the system will not crash during execution. For instance, the `Consumer` can invoke an `out` operation when the `Buffer` is empty and the `Producer` has quitted the system. This behavior will lead to a deadlock or a run-time error, depending on the actual implementation of the operations. Thus, a correct performance of the system requires that the components involved follow a certain protocol.

In our example, the `Producer` may invoke `in` an undefined number of times, but it must finish sending an `wquit` event. This behavior must be specified by the role `Writer`. On the other hand, the `Buffer` is defined by two roles. Role `Input` must indicate that the `Buffer` is able to accept either `in` or `wquit` at any time, but always ending in `wquit` after a sequence of `in` operations, while role `Output` must specify that the `Buffer` will perform `out` operations only when it is not empty, and that a `rquit` event will be sent when the `Buffer` empties after the `Producer` has quitted. Since the `Consumer` doesn't know whether these conditions have occurred or not, this will be specified as an nondeterministic behavior of the `Buffer`, using local choices. Finally, the `Consumer` must perform `out` operations until it receives a `rquit` event, this being described in role `Reader` as a deterministic behavior, using global choices. Figure 1 (right) shows the specification of system sub-components and their roles.

Role protocols can be specified in the π-calculus (but they are not included here to avoid detailed explanations about the syntax and semantics of the calculus), and we could analyze the compatibility of the attachments `Input` <> `Writer` and `Output` <> `Reader`, finding out that the architecture is composable. Thus, an instance of this architecture, such as

```
prodcon : Producer-Consumer;
```

will be deadlock-free.

However, not every Producer-Consumer system follows exactly these protocols. Let's consider a certain Producer' which generates exactly four items before quitting. Let role Writer' describe this behavior. Using the relation of role inheritance, we can verify that Writer' inherits from Writer, and thus Producer' inherits from Producer. Hence,

```
prodcon' : Producer-Consumer(p : Producer');
```

an instance of Producer-Consumer in which the Producer has been replaced with an instance of Producer', will be also deadlock-free.

Similarly, we could extend our Buffer with an operation flush, that empties the Buffer, and add this new behavior to the Input role. Once again, we can verify that the new Buffer' inherits from Buffer. Thus, Buffer' can be used in any system containing a Buffer, as in

```
prodcon'' : Producer-Consumer(p: Producer'; b : Buffer');
```

with no need of checking the compatibility of the extended role Input' with the roles representing the particular Producers in these systems.

In conclusion, a single proof of role inheritance allow the replacement of a component in any system belonging to a family of related (but not identical) systems. Liveness properties of the architecture are ensured by static checking even in a scenario of dynamic, i.e. at run-time, binding.

6 Discussion

In the last few years several proposals related to the specification of software architectures have been presented. However, most of them are not formally based, which prevent any kind of analysis of the systems so specified. All these proposals are compositional, in the sense that they consider a software system as a composition of several more elemental units. However, they differ in which are considered the elemental units for the composition of software systems. Some of them, like Darwin [9], consider that systems are built from components, while others, like Wright [3] or UniCon [17], also includes connectors as first-order elements of the notation. In [3], this distinction leads to the use of *ports* for components interfaces, and *roles* for connectors. This causes an asymmetric interpretation of compatibility which would lead to different relations of inheritance for ports and roles.

Besides, the distinction of components and connectors at language level does not scale very well. If we think of composite structures, formed by several components and connectors, the distinction vanishes, as usually these structures share characteristics of components (computation, data storage) and connectors (they serve to interconnect other components). The composition of components and connectors would lead to hybrid composites with free ports and roles which could not be classified either as components or as connectors. In order to maintain regularity and simplicity, we do not distinguish at language level between these categories, and all system components are specified the same, as a set of roles which represent their interface.

If we focus on formal methods, several papers [2, 7] have already proposed the use of formal notations, such as CSP or the Chemical Abstract Machine, for architecture specification. In [8] the π-calculus is used for defining the semantics of Darwin, where direct expression of mobility in the calculus is used to endow this language with lazy instantiation and direct dynamic instantiation mechanisms. However, the modeling of the interactions among components is not considered, and type checking is reduced to name equivalence. Aspects of inheritance and extension are not considered either.

With regard to compatibility, our proposal follows the ideas developed in [2], which uses CSP to determine compatibility of ports and roles. However, as it is stated in [8], CCS or CSP do not seem appropriate for the description of evolving or dynamic structures. Our proposal tries to solve this problem by using the π-calculus. Direct expression of mobility in the calculus makes easier the architectural specifications for dynamic systems. Furthermore, [2] does not address inheritance or specialization.

In [20], finite-state diagrams are used for the specification of what they call *protocols*, and relations of compatibility and protocol subtyping are also provided. Our approach differs from theirs in several relevant characteristics. Some of them derive from the richer expressiveness of process algebras with respect to state diagrams, as indicated in [12]. First of all, the use of the π-calculus allows the specification of dynamic systems simply by sending channel names as arguments of messages, while their proposal refers only to static architectures. Second, using the π-calculus, every message is send or received through a certain channel. Scope rules permit the restriction of channels to a set of processes, resulting in specifications more robust, modular, and also closer to software implementations, while in [20] channels are not considered, and every message is sent to a common *pool*, from which it could be retrieved by any component in the system; this being easily error-prone. Third, the π-calculus distinguish between global and local choices, allowing us to express scenarios in which a component may decide locally to commit to a certain transition or not (asynchronously), or in which the components involved in a communication globally agree in the commitment. However, their approach only takes into account synchronous global choices.

References

[1] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proc. SIGSOFT'93: Foundations of Software Engineering*, December 1993.

[2] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie-Mellon University, March 1994. Available at <ftp://reports.adm.cs.cmu.edu>.

[3] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. ICSE'94*, pages 71–80, Sorrento (Italy), May 1994.

[4] C. Canal, E. Pimentel, and J.M. Troya. A formal definition of compatibility and inheritance for software architectures. Technical report, Dept. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, June 1997.

[5] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical Report DAIMI PB-208, Computer Science Dept., University of Aarhus, 1986.

[6] D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, June 1995.

[7] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[8] J. Magee, S. Eisenbach, and J. Kramer. Modeling darwin in the π-calculus. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 133–152. Springer Verlag, 1995.

[9] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of ACM Foundations of Software Engineering*, pages 3–14, San Francisco, October 1996.

[10] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[12] F. Moller and S. A. Smolka. On the computational complexity of bisimulation. *ACM Computing Surveys*, 27(2):287–289, June 1995.

[13] O. Nierstrasz. Requirements for a composition language. In *Proc. of ECOOP'94 workshop on Models and Languages for Coordination of Parallelism and Distribution*, number 924 in LNCS, pages 147–161. Springer Verlag, 1995.

[14] O. Nierstrasz and T.D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–262, June 1995.

[15] W. Pree. *Framework Patterns.* SIGS Publications, 1996.

[16] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, University of Edinburgh, June 1993. Available at <http://www.dcs.ed.ac.uk/publications/lfcsreps>.

[17] M. Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[18] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In J. van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 307–323. Springer Verlag, 1995.

[19] B. Victor. A verification tool for the polyadic π-calculus. Master's thesis, Department of Computer Systems, Uppsala University (Sweden), May 1994. Available as report DoCS 94/50 at <http://www.docs.uu.se/docs/reports>.

[20] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.

Generic Reusable Object-Oriented Vertical Business Object Model[1]

Islam Choudhury
South Bank University. School of Computing, Information Systems and Maths
103 Borough Road, London SE1 0AA
(email- choudhia@sbu.ac.uk-

Abstract
This paper outlines a framework for building Generic Reusable Vertical Business Object Models (GRVBOM). A GRVBOM is a conceptual model of a Specific Industry or a blueprint of that industry capturing the essence of the industry. The paper presents an approach to tackle and resolve the problems inherent in doing so, and suggests that the resulting Vertical Industry Reference Model[2] can be applied to assist Information Systems developers to rapidly provision solutions as business changes. The main idea is to accurately reflect the business in a model, capturing the core business objects and processes of a particular industry so that this model will evolve and can be reused to develop solutions to multiple problems either within the same organisation who built the model or by different organisations of the same industry.

1. Introduction

Business information systems continue to be developed on the basis of business requirements which are only a snapshot of a business's dynamic life. This type of development, not taking into account overall conceptual and clear understanding of the business has resulted in large amounts of waste in terms of continually developing new systems from scratch (Jacobson 1995, Taylor 1995, Partridge 1996, Graham 1994, Martin 1992). It is argued that these systems have failed because of a fundamental lack of understanding of business systems as living systems.. We suggest that the business system must be understood in context to other businesses and in relation to the industrial sector that business belongs to. The anomaly in business is that there is a lack of prevalence of generic models for vertical industries However, this type of understanding and development of vertical industry models is a very difficult task Systems Analysts, Software Engineers, Information Systems developers face. This is because they have minimum resources of time and money with which to develop, efficient, fully functional, maintainable systems that captures the business requirements for many projects. We suggest that a separate group of developers should spend an initial overhead of time and money to develop a reference model that captures the overall core business and once developed this would be an extremely useful tool that these developers can use as the starting point to several business application developments. These developers can then work within this business framework to make decisions and to develop specific systems that meet the requirements in multiple projects, hence saving time and money in the long term. This reference model would be a Generic Reusable Vertical Business Object Model (GRVBOM).

[1] This research is funded with an award from British Telecommunications plc.
[2] A Vertical Industry is a family of organisations within the same industrial sector, e.g. Telecommunications, Banking, Manufacturing , etc.

It is difficult to understand why something as common-sensible as building Generic Reusable Business Object Models of Vertical Industries has not become a central and essential feature of business modelling before. If getting such significant improvements in building such models were easy, it would have been done long ago. One of the problems is that a Generic Business Model is usually misunderstood. It is assumed that the model defines the business as generic, however this is not the case. There are core, common and generic aspects of a business that can be captured which is applicable to many organisations within the same Vertical Industry. Another problem is that it is difficult to see general patterns for the high level classes that will supersede the original lower level classes. The secret is in the object paradigm and the categorising of specific businesses into specific industrial sectors. This gives a much better, much more conceptually accurate, understanding of a business in context and in relation to other businesses. Digre (1995) has highlighted the need to build vertical industry models and Rutt and Stringer (1996) of the OMG Vertical Industry Domain Task Force (OMGVIDTF) have started work on a Telecommunication Vertical Industry Model.

The paper is structured as follows. Section 2 identifies and describes some of the characteristics of a GRVBOM. We then develop a framework within which generic objects and processes within vertical industries can be captured, analysed and represented. These generic activities represent the core business object and processes for that particular industry.

In section 3, we highlight the implementation of the GRVBOM in its application in the Telecommunications Industry and the Finance Industry. We conclude with the suggestion that the GRVBOM is a viable modelling approach and present an overview of further work in progress to evaluate the viability, usefulness strengths and weaknesses of a GRVBOM.

2. The Generic Reusable Vertical Business Object Model

This section gives an explanation into some of the characteristics of a GRVBOM. At the heart of the model is a Core Business Model (CBM). The core business model is built within the framework of generality and ability to be reused within a Vertical Industry. Most importantly, it presents an approach to model the conceptual understanding of the business using Jacobson's Use Case Engineering and Rumboughs OMT. The core business model can be mapped onto uses cases and a business object model. It must be stressed that the GRVBOM is built independently of any application development.

2.1 GRVBOM- Core Business Model

At the centre of the GRBOM is the Core Business Model. This is built by analysing the existing documentation, systems and operations within the domain of interest of a particular industry. Then the generic business processes, business objects and the interconnections are captured and represented in a GRBOM - Core Business Model (CBM). Figure 1 shows the GRBOM-CBM.

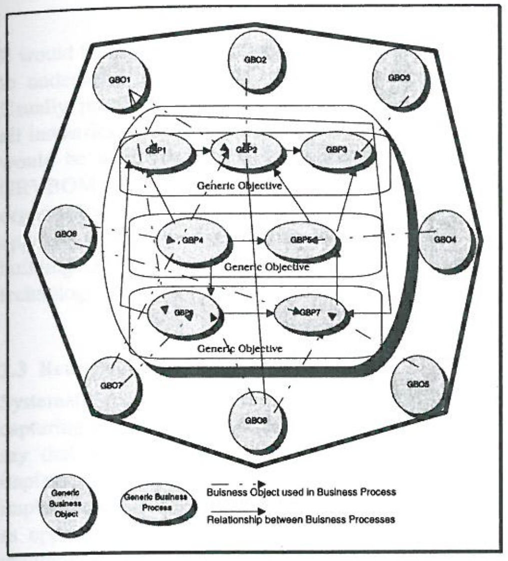


Figure 1 The GRVBOM- Core Business Model (CBM)

2.2 GRVBOM- Generic Characteristic

When using the word generic you must qualify it with the level of genericity you are dealing with. The idea is to appropriately distinguish the general from the specific. Generality is a relative term and the next few paragraphs will explain this in the context of this research.

Genericity is the staged refinement of entire models (Vernadat 1992). The concept uses stepwise instantiation to go from aggregations of generic components, through increasing specialisation's of business domains, to enterprise, facility, and work area implementation. Genericity is a controlled process which utilises principles of specialisation, inheritance, relationships, and contexts to leverage reusable industrial models and rapidly provision tailored solutions.

The concept of genericity should be used to specialise the language and semantics appropriate for each business domain. The business language should be defined in terms of higher level generic constructs. Subdomain specialisation will in turn be defined in terms of domain constructs.

Figure 2 is a genericity diagram which utilises the above mentioned ideas and is an important component of the GRVBOM. A core business model can be built for each level.

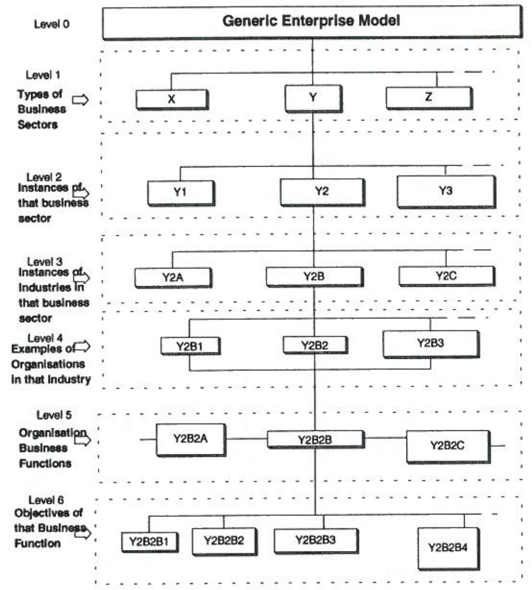

Level 0
Generic Enterprise Model
Level 1
Types of Business Sectors
X
Y
Z
Level 2
Instances of that business sector
Y1
Y2
Y3
Level 3
Instances of Industries in that business sector
Y2A
Y2B
Y2C
Level 4
Examples of Organisations in that Industry
Y2B1
Y2B2
Y2B3
Level 5
Organisation Business Functions
Y2B2A
Y2B2B
Y2B2C
Level 6
Objectives of that Business Function
Y2B2B1
Y2B2B2
Y2B2B3
Y2B2B4

Figure 2 GRVBOM- Genericity Diagram

Generic Models in the context of this research describe certain kinds of scenarios which are common to vertical industries. Hence integrating the understanding of a family of industries. The similarities of these scenarios are reflected in the corresponding business object models, i.e. they have the same or similar structure. A generic model can help the modeller to develop and analyse his/her own model. To take advantage of a repository of generic models there must be an understanding of these models and a way of analysing the user specific model and a generic model. There must be a way of comparing and contrasting existing generic models and newly designed user models (Digre 1996).

Sutherland (1996) forcefully pointed out that it would be wiser to build vertical industry models that fit specific industries as opposed to Fowler's (1996) idea that there should be a master model that can be adapted to any industry. Sutherland (1996) argues that each industry provides different services, functionality and mechanisms to run the business. The complexity of all the different objects and behaviour interactions within an industry is so great that it would be more realistic and effective to build a model to suit a particular industry. It is with this in mind that this research is justified in the building of a GRBOM of a vertical industry..

It would be much easier to understand one particular industry in detail rather then trying to understand all industries and capture the common core element of all industries. Usually, practitioners have expert knowledge of one industry as opposed to knowledge of all industries and this expertise can be drawn upon in building a GRVBOM. GRVBOM would be a reference model when building new systems for that particular industry. GRVBOM of a vertical industry will provide strategic competitive advantage to an organisation within a particular industry that posses and makes use of a GRVBOM as opposed to an organisation that does not posses this model. The critical enablers for building GRVBOMs are generalisation-specialisation, use case engineering and object technology.

2.3 Reuse Characteristic

Systematic reuse is a general principle that is instrumental in avoiding duplication and capturing commonalty in inherently similar tasks and domains. Frakes and Isoda (1994) say that companies that have a better understanding of the business domain and implement systematic reuse will have a powerful competitive advantage. Reuse usually implies software reuse but this project will address reuse at the business knowledge level as opposed to just software reuse. There is a notion that every business has unique requirements and therefore unique solutions have to be proposed for the information system of each specific business. However, this is not necessarily the case. There is an understanding that vertical industries have much in common, and there is a need to exploit this commonalty therefore avoiding arbitrary duplication when developing similar systems for a particular industry. There is a clear market trend that organisations cannot afford this bespoke systems development approach in today's turbulent business environment and must develop systems that can survive when businesses change. One approach to achieving systematic reuse is to build business objects (Arrow et al. 1995) that are generic, stored in a enterprise-wide repository as assets and therefore can be reused as required. This involves enterprise modelling which is a major area of application for object technology and software reuse (Fraser and Macintosh 1994).

The GRVBOM is a reference model that can be reused by systems developers in the development of multiple projects. Figure 3. shows the GRBOM reuse dimension. The diagram shows the positioning of the CBM in relation to business modelling and Information Systems modelling. The CBM is designed for reuse (DfR) and then applications are developed with design with reuse (DwR) in multiple projects.

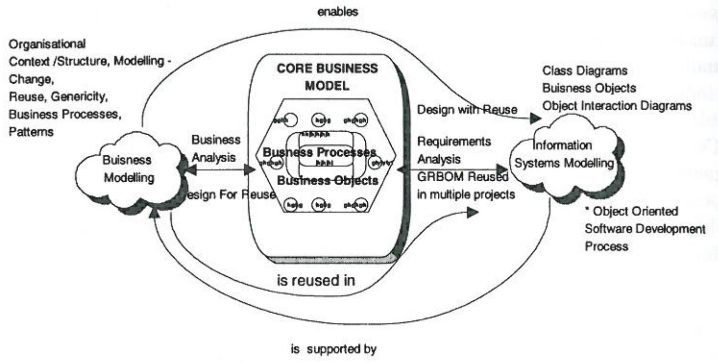

enables
Organisational Context /Structure, Modelling - Change, Reuse, Genericity, Business Processes, Patterns
CORE BUSINESS MODEL
Business Processes
Business Objects
Buisness Modelling
Business Analysis
Design For Reuse
Design with Reuse
Requirements Analysis
GRBOM Reused in multiple projects
Information Systems Modelling
Class Diagrams
Buisness Objects
Object Interaction Diagrams
* Object Oriented Software Development Process
is reused in
is supported by

Figure 3 The GRVBOM- Reuse Characteristic

2.4 Business Object Model

Jacobson's Use Case Engineering (Jacobson *et al.* 1995) and Rumbough's OMT (Rumbough *et al* 1991), were used to build an Object-Oriented Model of the above CBM. Figure 4 shows the overall business object modelling dimension. The Generic Processes in the Core Model were represented as a series of Use Cases containing various Business Objects. Object Management Group Business Object Domain Task Force (OMGBODTF) (Arrow et al. 1995) have defined Business Objects as "a representation of a thing in the business having attributes, behaviour, relationships and constraints. Select OMT was used as a CASE tool to support the building of GRVBOM models and because it provides a repository to store the Business Objects and Models. Select OMT is a fully functional CASE tool that supports Rumbough's Object Modelling Technique and Jacobson's Use Cases and Object Interaction Diagrams. The business processes and business objects of the Core Business Model are mapped to use cases and business objects of the business object model, see figure 5. The business objects are made up of interface objects, control objects and entity objects.

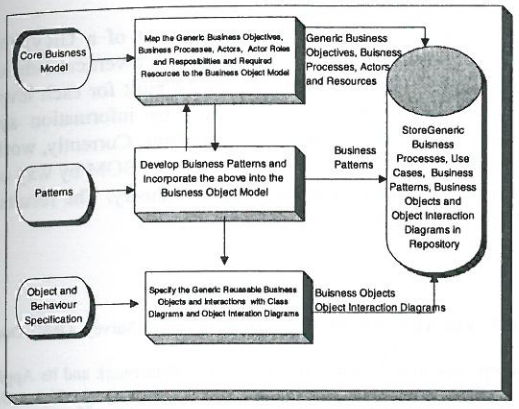

Core Buisness Model
Patterns
Object and Behaviour Specification
Map the Generic Business Objectives, Business Processes, Actors, Actor Roles and Resposibilities and Required Resources to the Business Object Model
Develop Buisness Patterns and Incorporate the above into the Buisness Object Model
Specify the Generic Reusable Business Objects and Interactions with Class Diagrams and Object Interation Diagrams
Generic Business Objectives, Buisness Processes, Actors and Resources
Business Patterns
Buisness Objects
Object Interaction Diagrams
StoreGeneric Buisness Processes, Use Cases, Business Patterns, Business Objects and Object Interaction Diagrams in Repository

Figure 4. GRVBOM - Business Object Modelling Approach

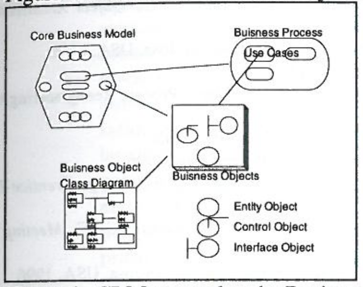


Figure 5 - CBM mapped to the Business Object Model

3. GRVBOM Implementation

The GRVBOM was applied in the development of a GRVBOM of the Customer Services Domain of British Telecommunication plc GRVBOM-CSD-BT (Choudhury and Sun 1997). This showed that GRVBOM is a viable modelling approach. This model is being evaluated against BT's existing object-oriented customer models, which have already been analysed by previous analysts to develop various systems for BT applications, for example customer billing systems, customer enquiry systems, customer marketing systems, etc. The generic nature of the GRVBOM is being evaluated by instantiating the model in different domains for example the Finance domain. We are currently carrying out a Case Study, similar to the one in the Telecommunication Industry, in the Customer Services domain of a Credit Insurance company which can be generalised to the Insurance Industry which in turn can be generalised to the Finance Industry sector. In order to justify the usefulness of the GRVBOM, evaluation criteria are currently being developed and these will be used in subsequent testing and further development of the GRVBOM framework.

4. Conclusion and Further Work

We have presented and explained the need for the development of a GRVBOM. A GRVBOM is an object-oriented business model that is general to a vertical industry. At the heart of the model is the Core Business Model that can be built for each level of a Genericity Diagram. This model can be used and reused by information systems developers as the starting point of several application developments. Currently, work is in progress to evaluate the appropriateness and usefulness of the GRVBOM by way of case studies in the Telecommunications Industry and the Finance Industry. The results from this evaluation will be presented in a future paper.

5. References

Arrow, L., Barnwell, R., Burt, C. and Anderson, M. OMG Business Object Survey. OMG Document 95-6-4. 25 May 1995

Choudhury, I. And Sun, Y. Generic Reusable Business Object Model- A Framework and its Application in British Telecommunication plc. *Submitted to the 4th International Conference on Object-Oriented Information Systems OOIS97*-Brisbane, Australia, November 1997.

Digre, T. Business Application Components *OOPSLA 95 Workshop on Business Object Architecture*. Austin Texas, USA. 1995

Fowler, I. Business Object Architecture Workshop at the *OOPSLA 1996* at San Josa, USA, 1996.

Graham, I. Migrating to Object Technology, 1994, *Addison Wesley.*

Jacobson, I., Ericson, M. and Jacobson, A. The Object Advantage. Business Process Reengineering with Object Technology. *ACM Press.* 1995.

Martin, J. and Odell, J. Object Oriented Analysis and Design 1992, *Prentice Hall.*

Partridge, C. Business Objects Reengineering for Reuse. Butterworth-Heinmaan 1996

Rumbaugh, J., Blaha, M., Premerlani, W., *et al* Object-Oriented Modelling and Design *Prentice Hall* 1991

Rutt, T., and Stringer, D. The OMG Telecom Domain Task Force *First Class : OMG: Meeting the challenge of Vertical Industries.* August/ September 1996

Sutherland, J. Business Object Architecture Workshop at the *OOPSLA 1996* at San Josa, USA, 1996

Taylor, D. A. Business Engineering with Object Technology. *John Wiley and Sons, Inc.* 1995

Vernadat, F.B. 1992. CIMOSA - A European Development for Enterprise Integration. Part2: Enterprise Modelling. ESPIRIT Consortium AMICE. *Proceedings of the First International Conference on Enterprise Integration.* Petrie, C.J. 1992, pp 179-188

Constraint Based Inheritance

Erik Ernst
eernst@daimi.aau.dk
Department of Computer Science
University of Århus
8000 Århus C, Denmark

Abstract

This paper presents my PhD research work in very general terms. My main interest is programming language design. Since language design is so hard, I've chosen to generalize and regularize the design of an existing, high-quality OO language, namely BETA. A good language combines good expressiveness through few, well-designed abstraction mechanisms with a high degree of safety through compile-time analyzability. I'm using two main tools in this process, an actual implementation of the generalized language, and formal semantics. A formal language specification helps in spotting inconsistencies and bad language design, enables strict conformance checks on an implementation, and supports the further development of the language in light of the insight gathered. The purpose of an implementation is to discover the pratical properties of a language, that is how it actually behaves as a tool for a programmer. The generalized language embodies a notion of inheritance built on constraint solving, and this enables an unusually profound separation of concerns, and also improves on the possibilities to specify and enforce multiclass type consistency constraints.

Keywords: language design, language implementation, formal semantics, type systems, inheritance, abstraction mechanisms, separation of concerns

1 Introduction

An important part of the history of computer science is devoted to the design and specification of programming languages. There is an analogy between the notion of a progamming language and a fractal like the Mandelbrot set: it yields an unfolding of a seemingly small and simple seed (e.g. $z \mapsto z^2 + c$) into a very complex entity; in this case the syntax and semantics of the programming language is the simple seed, and the entire set of realized and potential programs is the complex unfolding of it.

It is hard to evaluate a seed with such a large and complex unfolding as a programming language. It may be, loosely, considered as a chaotic system, since even small changes in the seed may have a profound impact on the unfolding. Two approaches to evaluating it are to use previous experiences about particular aspects of the seed, and to actually do some unfolding. The first approach amounts to choosing language constructs and combinations of constructs which are "known" to work well, and possibly to work well together; for the second approach one must experiment with carefully selected example problems, investigating how solutions to them may be expressed. Or, alternatively, one may use the language in a lot of mission critical real-life projects and then conclude in rich detail why it is no good. ;-)

My approach is to take a language—BETA—which I find quite well-designed already, and then to adjust the design in ways that improve the expressiveness and flexibility of the language without loosing the safety of strict, static type checking, and ensuring that it is still possible to create tools—compilers, debuggers etc.—and programs with good performance. When the language was first designed in the 70'ties, it was a prime consideration that it should consist of few, but very expressive, orthogonal concepts. This has also been a primary criterion in my process of generalization.

Section 2 describes the current language BETA; section 3 outlines and motivates the changes I've made to the language, and section 4 presents a progression of language mechanisms which should hint at the expressive power at hand. Finally section 5 concludes.

2 The Language BETA Today

BETA [9] is a modern, strictly typed, object-oriented programming language, coming from the Scandinavian tradition [8, e.g.] which produced the first object-oriented language, Simula [2]. The language has a somewhat unusual syntax which may at first confuse some people, but it is actually both simple and consistent. A good example is that assignment and parameter passing is written in the evaluation order and using an arrow, which is natural but

very unusual. For example, what is usually expressed somewhat like `y:=x` is written `x->y` in BETA, and `aProcedure(x,y,aFunction(u,v))` is written `(x,y,(u,v)->aFunction)->aProcedure`. Note that this allows you to return more than one result, and to deliver parts of such a return list to different receivers: `returnsTwoResults->(takesOneArg,takesOneArg)`.

A more profound—but equally unusual—property of BETA is the unification of many programming language concepts into one, more general concept, the *pattern*. The pattern concept subsumes and generalizes the concepts of class, procedure, function, coroutine, process, exception, and more. It describes structure in terms of attributes, input and output properties, and behaviour. Think of it as a class concept equipped with behaviour, which you might just describe as a default method or function. Focus on the behaviour of a pattern, and it may be used as a procedure or function, depending on the input/output properties; describe some state using attribute declarations, and the pattern may be used as a class; patterns nested in the "class" may be used as methods. Patterns are *descriptions* and objects are *substance* created according to such descriptions. If the substance is used for an extended period of time it is like an "object" in traditional languages; if the substance only survives briefly, it is more like a procedure invocation.

There is general block-structure, i.e. support for unlimited nesting of patterns and objects. This means that whenever you create an instance of a pattern P' textually nested within the declaration of a pattern P, the instance of P' is immutably associated with one particular enclosing instance of P, and the P' instance has access to that enclosing instance of P. This makes it possible to use an instance of a nested pattern as a method invocation for the enclosing object—a method usually wants to read or write the state of "its" object. But it also lets you save such a "method invocation" for later or repeated execution, which is needed for futures and callbacks, or in general for "closures" as they are usually designated within the functional language community.

The notion of *subpatterns*—and the mechanism *inheritance* which produces them—enables the construction of class hierarchies as known from other languages; but since a pattern is also a procedure we get the unusual but very useful notion of *pre-methoding*, i.e. inheritance hierarchies of procedures and functions. The *virtual* pattern concept of course supports similar usage as the concept of virtual member functions in C++ [3, 13] and (non-frozen) method features in Eiffel [10]. But since a pattern is a class, it also supports type-safe constrained genericity as do parameterized classes in

Eiffel, and it supports the equivalent of Eiffel's declaration by association.[1] Note that a BETA program does not need a universe-wide "system-validity check" to ensure that a program will never give rise to a type error at run-time. Nevertheless, a programmer may choose to write BETA programs that may produce run-time type errors at certain well-known places (the compiler emits a warning for each place), effectively accepting an unsafe program in return for the greater flexibility.

There is a working implementation of BETA [5] developed by some of the original language designers among others. The actual semantics of the implementation along with informal language descriptions has been the starting point for my specification and generalization of the language. There has been some attempts at describing the formal semantics of (parts of) BETA [1, 7]. These specifications have not centered on describing the language but rather on slightly different things, such as comparing various variants of inheritance, or using a particular formalism. Consequently, they have not yielded a description which is sufficiently close to BETA as to be useful for a language development or implementation effort. For example, they describe a language in which the semantics of name lookup and/or assignment and parameter transfers depend on the actual type of an object, not on the statically known type. Such a language probably cannot be typechecked statically, and it certainly behaves a lot differently than BETA. One could have the suspicion that semantic descriptions of languages tend to describe Smalltalk-like languages, since such a "typeless" semantics is much easier to specify. In my opinion, both the static checking of programs and formal specifications of languages are so valuable facilities that they should be reconciled.

3 Generalization and Regularization

From this already very general language I have worked towards a more general and regular one. The regularization part has to do with those parts of the language where the implementation is best explained in terms of pragmatic decisions taken in order to implement an actual compiler within reasonable time, producing programs with a good performance. For example, the Mjølner BETA implementation does not support obtaining a dynamic reference to an integer ("to take its address"), or of instances of other basic, predefined patterns. Should we formalize this into a description of simple non-object entities and complex, user defined objects? Should `false` and `true` be patterns? Should they be specializations of `boolean`? Should there be a notion of *the* object "one" and *the* object "two" and so on, as in Smalltalk [4], such that computations on integers and other simple values would be computations on special references to unique, always-existing, immutable objects, or how else should we get started in the description of *state*?

[1] "Anchored" declarations, as in `x: like y`

I've arrived at a description which uses both values and objects, and hence is more in line with traditional formal semantics, and which consequently goes against the "everything is an object" philosophy known from informal descriptions of object-oriented languages [4, in particular]. The description is using the formalism Action Semantics [11, 12]. It is still under construction, though.

The generalization part has to do with the type system, particularly inheritance—or *specialization*, as it is normally designated in the Scandinavian OO tradition. Normally, specialization is an operational mechanism, in the sense that it supports the construction of a derived entity from an existing entity and an difference specification. For example, using the name of some particular class you may specify a subclass of it by listing the enhancements (e.g. some new attributes and enhanced versions of existing attributes). The class being built on is known precisely, and the enhancement is known precisely at the place of specialization. Building on more than one base class (multiple inheritance) does not change this.

My model supports a more declarative approach in which specialization is defined by a *constraint solving* process. The pattern denoted by a particular pattern attribute is computed by collecting all constraints on that attribute in scope. The combination of two patterns entails the combination of the constraints on common, nested patterns, so the specialization operation is *implicit* in many cases. It is this implicitness that allows the construction of separate patterns which may by weaved in a delicate way through multiple inheritance, in particular because the combination of behaviour is very expressive. The combination of block structure, specialization, and virtuals makes for a quite complicated setting from a type-checking point of view, but for the programmer this generalization appears as the removal of a number of limitations. Among other things it becomes possible to inherit from a virtual pattern. Examples of the new possibilities and the reasons why we want them will be given at the workshop.

With constraint based specialization, a new level of "static properties" emerges: traditional static knowledge says something at compile-time about all potential runs of a program. The new level says something at the time of library construction about every possible library application.

Here's a longer explanation: Since a type checker is a theorem prover for a special class of theorems, type checking supports establishment of some precise statements about the run-time properties of a given program. Now, for reuse-oriented libraries there is yet another "time," namely library construction time. As long as a library is always used as-is, this presents no new challenges. With inheritance and genericity, however, there could be errors that make it impossible to *compile* a program using a library, and there could be problems at compile-time or at run-time arising because of contract violations.[2] The problem is that specialization of classes and instantiation of parameterized classes (e.g. C++ templates) is a more *intrusive* usage of library code than just calling some functions i a C library. More expressiveness goes along with more fragility.

We need more expressive constraints on library code, such that library users can tailor a library extensively, guarded against contract violations by a good checking system, and not impeded by superfluous constraints.

The constraint based specialization mechanism in my generalization of BETA is one step towards that impossible goal, expressing "the right" properties that hold or should hold for all specialized or instantiated versions of a reuse-oriented piece of code. In particular, it is able to express type relations within a group of classes that should hold across specialization. This it not in any way a special property of libraries, that is just a convenient frame to give motivation for the usefulness. It is a general property of patterns.

4 Some Levels of Expressiveness

When designing a template class in C++, one builds a description entity which will generate a somewhat homogeneous set of instances. If an attribute of such a class has the same type designation as a formal argument of one of its member functions, they will have the same type in all instances, which gives some nice, simple consistency guarantees. But a template cannot be thoroughly type checked, most of the type checking must be done individually for each instance. This may give rise to problems when using the template: perhaps a class parameter is not acceptable, because the template accesses

[2]Using Eiffel parlance

a member which is not present in that class. C++ templates are little more than textual macros.

In Eiffel, constrained genericity allows real type-checking of the parameterized class, such that no instantiations of a parameterized class will be able to provoke type errors within the source code of the parameterized class. But if you have a set of interrelated, parameterized classes, you will have to specialize them one-by-one even if the specialization follows simple, strict and well-known rules. From the language point of view you have specialized them independently, and any regularity in this is treated as purely accidental. When you start using the result, any mistakes in this regular, manual undertaking may give rise to compile-time errors or, worse, run-time errors.

In BOPL [6], a family of languages constructed for the presentation of object oriented type systems theory, you may specialize a set of classes as a whole; the type relationships are kept intact by a mechanism called capturing. Specialization of one class implicitly creates a whole family of specialized classes by finding the transitive closure of type dependencies and recreating the entire, reachable graph of type relationships "at a higher level." The simplest example is the `ListNode` class whose `Next` member of type `ListNode` gets specialized along with the class; this special case works like a `like Current` declaration in Eiffel. Another example: assume that you have a `Tree` class associated with a `Node` class, and the `Node` class has a `Value` member of type `Object`.[3] If you derive a `TextNode` class having a `Value` of type `Text`, a class which is a specialization of `Tree` will be created implicitly. This implicitly constructed class would for instance have an `insert` method taking an argument of type `Text` in stead of an original `insert` taking an argument of type `Object`. The details are a little more complicated, but this is the general picture. This actually realizes a multiclass type consistency support mechanism, and that is an important step forward. A problem is that the implicitly derived classes are available for use by the implementation of the derived classes themselves, but they have no (known) names and hence cannot be used in all the ways normal classes can. Another problem is that you may not *always* want to have the semantics of such a generalized version of `like Current`—some types should not be captured.

In today's BETA, specialization (i.e. inheritance) is expressive enough to subsume constrained genericity, and you may use nesting to ensure that a group of classes are specialized as a group, so you can have capturing for

[3]The most general type in that type system

exactly the patterns you want to capture. There is no type algebra, however, allowing you to specify the constructions of patterns in terms of other patterns. The only facility of this kind is the single inheritance mechanism which demands that both the base class and the enhancement specification are compile-time constants. The generalized BETA language supports the specification of relative constraints on patterns, such that a pattern may be known to be an enhancement of one or more other patterns even though none of them need to be known at compile-time. As a result of this you may for instance specify some design patterns[4] as working, reusable, and checkable code, not just as rules of thumb.

5 Conclusion

This is a very brief presentation, and hence it concentrates on the large picture and the motivation of the work I've done during the PhD getting process sofar. Inevitably, the properties of the work seem like unsupported assertions, but I hope to make up for this at the workshop! The main topic of this work is programming language design, development, and application of formal semantics. Starting from the nice language BETA I've developed a more general and regular language by means of an actual implementation, and work is in progress on an action semantic specification of the result. The type system has been generalized considerably, allowing for a constraint based notion of inheritance which has the traditional inheritance mechanism of BETA—already unusually expressive by the way—as a special case. The generalized language supports the expression of multiclass type relation constraints, and in combination with pre-methoding it allows an unusually effective separation of concerns. For more information please check `http://www.daimi.aau.dk/~eernst/gbeta`.

[4]One example I'll give at the workshop is about the observer design pattern

References

[1] W. R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[2] O.-J. Dahl and K. Nygaard. The development of the Simula languages. In R. W. Wexelblat, editor, *History of Programming Languages*, New York, 1981. Academic Press.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, USA, 1990.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, Reading, MA, USA, 1989.

[5] Mjølner Informatics. *The Mjølner BETA System.* WWW, http://www.-mjolner.dk, 1997.

[6] Michael I. Schwartzbach Jens Palsberg. *Object-Oriented Type Systems.* John Wiley & Sons, New York City, 1994.

[7] Jens Palsberg Jørgensen. *An Action Semantics for Inheritance.* Datalogisk afdeling, AArhus Universitet, Århus, Denmark, 1988. Arkiv: 119607.

[8] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments - The Mjølner Approach.* Prentice Hall, Hertfordshire, GB, 1993.

[9] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, MA, USA, 1993.

[10] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1991.

[11] Peter D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, GB, 1992.

[12] Peter D. Mosses. *Action Semantics Home Page.* WWW, http://www.-daimi.aau.dk/BRICS/FormalMethods/AS/index.html, 1997.

[13] Bjarne Stroustrup. *The C++ Programming Language (Second Edition).* Addison-Wesley, Reading, MA, USA, 1991.

Requirements for an Object-Oriented Language for the Design and Implementation of Telecommunication Systems

Nils Fischbeck

Humboldt-University of Berlin - Dept. of Computer Science,
Axel-Springer-Str. 54a,
10117 Berlin, Germany
e-mail: fischbec@informatik.hu-berlin.de

The design and implementation of telecommunication systems requires special features from an object-oriented language like real-time constraints, concurrency, dedicated object states and asynchronous communication. This paper defines terms which classify these and other requirements and evaluates the object-oriented language SDL.

1. Introduction

A telecommunication system like an ISDN network comprises a great variety of software components. It includes software for switches, the terminal equipment (like a simple telephone) but also for the management of resources and the provision of high level services like time dependent call forwarding. Software for telecommunication systems like other software must be fast, robust and easy to maintain (e.g. extensible).

Like in many other application domains telecommunication software has to comply to standardized interfaces in order to enable interoperability between software of different vendors. The international standardization body ITU-T (International Telecommunication Union - Telecommunication Sector) accounts for the definition of interfaces using formal description techniques. The main language to define the behaviour of telecommunication systems is SDL (Specification and Description Language). The current version SDL'96 [Z.100] provides abstract data types, concurrency, asynchronous communication and type based definitions. The use of SDL is not confined to the standardization area. The language was successfully applied for the design and implementation of large communicating systems[PL112A]. However the language could not keep up with recent developments in the domain of distributed (telecommunication) systems (ODP [X.901], TINA-C [CM94]). The following sections motivate what language concepts are necessary for the design and implementation of telecommunication systems and how SDL fulfills these requirements.

2. Application area

The term telecommunication system as used in this paper denotes all kinds of software necessary to provide a telecommunication service. Examples for telecommunication services are

- Basic Telephony,
- Conference Call,

• File Transfer.

These services may be based on existing telecommunication networks like

• PSTN (Public Switched Telephony Network),

• B-ISDN (Broadband Integrated Services Digital Network) or

• TINA-C Distributed Processing Environment.

Telecommunication systems can be distinguished from other distributed applications since they

• must satisfy real time constraints,

• must comply to standardized interfaces,

• use specialized hardware (e.g. language recognition devices),

• mostly use asynchronous communication and

• involves the transport of a variable or constant stream of data (e.g. a telephone call).

The experiences presented in this paper are results of projects for the design and implementation of telecommunication systems at Humboldt-University Berlin, mainly:

• PLATINUM [PL112A],

• OOSPEC II [NF96],

• PT87 [PT87] and

• CAMOUFLAGE [HK97]

3. The object-oriented language SDL

The following description of SDL is a simplification, for a complete reference, the reader is referred to the ITU recommendation Z.100 [ITU96].

SDL is the ITU Specification and Description Language. It is standardized in the ITU Recommendation Z.100 and it is based on the concept of communicating extended finite state machines. Each state machine works in a stimulus/response fashion. SDL is mostly used for the specification of telecommunication systems, but can be used to describe any discrete system which exhibits an event-driven control flow.

With the 1992 revision of SDL, object-oriented constructs were introduced into the language. SDL now contains a typing concept for structuring the system, and concepts of virtuality and inheritance to allow for hierarchies of types.

SDL provides concepts to describe both the structure and the behaviour of a distributed system. On the structural level, a system or system type is subdivided into blocks. The concept of block is usually used to express physical distribution, meaning that the processes inside different blocks are executed at different physical locations. A block consists of sets of processes, which define the behaviour of the block. The behaviour of a process is described using a state-transition-diagram, which is shared by all processes of a set. All processes execute independently from each other, and possibly in parallel. They communicate by exchanging signals. The signals are delivered via signal routes and channels, and end up in the infinite input port of the receiving process.

To further substructure the behaviour of a process, services and procedures may be used. A service can be used to group the transitions of a process by the signal that is consumed as a stimulus. Procedures define transitions that can be invoked during a transition. Procedures are similar to the subroutines found in other programming languages, whereas services are similar

to the concept of coroutines.

In SDL-92, the structural concepts were extended with a type concept, including system, block, process and service types. A type definition can be used to create instances of the type, as well as to serve as a base type of a derived type definition. A derived type definition may add new local entities, as well as redefine virtual entities. For example, a block type definition B containing a virtual process type P can be specialized to a block type B1, which replaces the process type with a different one. All instances of the process type P then become instances of the process as defined in B1.

On the behavioural level, transitions can be declared as virtual. A transition definition consists of a state, a signal input (or any other stimulus) and a sequence of actions. If, in that state, the signal is received, the process will perform the associated actions. If an input is marked as virtual in a process type, a specialized process definition might redefine the input and associate different actions with it.

4. Structuring

Telecommunication systems consist of several entities which, for example, process a call, provide algorithms for the evaluation of call parameters, or map data between different formats using a data base. To cope with the complexity of these tasks a telecommunication system must be structured. There are two complementary concepts to structure a system:

• structural composition

• hierarchical composition.

Structural composition means that an entity can contain several other entities. Hierarchical composition allows that the contained entities are structured itself hence contain other entities. Both concepts used together yield a *structural and hierarchical composition* (SHC) [TM96]. SDL supports SHC by allowing that a *system* contains *blocks*, a *block* contains *processes* and a *process* contains *services*.

5. Objects

The base entity a telecommunication system is built on shall be called *object* throughout the paper. An object is a capsule for attributes which are either data attributes (variables), stream attributes or executable attributes (operations).

5.1. Active Objects

Telecommunication systems require *active objects*. Active objects can change their states (defined by the value of their attributes) without external influence. That is the main difference to passive objects which change their state by message passing or method invocation. The cause for the state change of an active object can be the expiration of a timer. Active objects are essential for the behaviour of the telecommunication systems (e.g. for the clearing of a partial established call after a certain time).

In SDL every *process* and every *service* is an active object since these entities support timer driven execution of code.

5.2. Passive Objects

Passive objects do not change its state without external invocation. Passive object are associated to an active object. An attribute of a passive object which is executable (an operation) can be invoked from the associated active object or from another passive object which is associated to the active object.

SDL supports passive objects with algebraic defined abstract data types using ACT ONE. Operations of passive objects can be defined with executable code too.

6. Threads

A telecommunication system must support different *threads* of execution. With the term thread an execution context is denoted. Every active object has its own thread. Executable attributes of passive objects which are associated to an active object are executed in the same thread as the active object. Two threads can be executed in two modi relative to each other:

- concurrent: both threads can be executed at a given time,
- alternative: only one thread can be executed at a given time.

Concurrent thread execution models the existence of multiple processors which execute threads simultaneously. Alternative execution requires a scheduling mechanism which grants execution time to a thread. Alternative scheduling can be

- non-deterministic: external scheduling - e.g. from timer events or
- deterministic: the executed code determines what thread is executed next [Mad93].

Often non-deterministic alternative scheduling on a single processor is used as approximation of concurrent execution for simulation purposes.

Concurrent and alternative execution are both necessary in telecommunication systems. Concurrent execution is provided by a multi-processor architecture. Alternative execution is necessary to guarantee a fixed order of execution of threads. This could be for instance necessary to have exclusive access on data in a special part of the transition or to assure a special order of signal sending which are sent from different threads. Although this could also be accomplished by means of synchronization between concurrent threads it is more convenient to simply mark the threads as alternative threads.

The classification of active objects in a structural and hierarchical composition and the classification as alternative or concurrent are different kinds of classification. That means that three active objects a, b and c contained in an entity e should not necessarily have the same execution relation between each other. If a and b are executed concurrently it is not necessary that a and c are executed concurrently. Often the SHC is also used for the classification of the execution relation because such a classification is easy to understand as there is only one classification for two purposes. The language SDL yields an example: a *process* composed of *services* specifies alternative execution of all contained *services* but a *block* composed of *processes* specifies concurrent execution of all contained *processes*.

6.1. Time

A language suited for the design and implementation of telecommunication systems should support at least two time models. An implementation of a telecommunication system will run under real time conditions and therefore support real time values, e.g. for timer. The design of the telecommunication involves another model of time: model time. In this time model one can assign a certain amount of time to an action or to a signal transport (e.g. no time progress during signal transport but a time progress of one during execution of an action). The assigned time has nothing to do with the time needed to execute the actions during simulation. Instead it indicates time progress under certain conditions.

SDL has no means to specify the duration of an action or a signal transport. Timers are associated with a duration in SDL. This duration implies a partial order of timer expirations. SDL tool providers give additional timing semantics to SDL specifications, e.g. the timer duration is used as real time value of a certain unit.

7. Communication

An active object must be able to communicate with other active objects. Communication can take place when one active object is the source for the communicated information, another object accepts the communicated information and there is a way to transport the information between the objects. Two kinds of communication should be supported:

- stream communication
- operational communication

Stream communication denotes a continuous exchange of entities from the *source object* to a *sink object*. This kind of communication is necessary to model a continous data flow like a speech channel in a telecommunication system. Stream communication takes place between special attributes of an object: *stream attributes*.

Operational communication denotes the *invocation* of an operation (an executable attribute) at a target object. An operational communication comes in two flavours:

- asynchronous communication and
- synchronous communication.

The source of an *asynchronous communication* does not wait until the operation is finished. Asynchronous communication can be devided into two groups:

- communication with assured reception,
- communication with assured invocation and
- communication with non-assured reception.

Using *assured reception* the source of the communication waits until it is assured that the operation reaches the target and that the target can execute such an operation. It is not assured that the operation is executed (the active object can decide to die before the operation is executed). *Assured invocation* implies assured reception. It guarantees in addition that the operation was started. *Non-assured reception* does neither guarantee that the operation reaches the target nor that it is started.

The three kinds of asynchronous communication are necessary to mirror the high time and security constraints of telecommunication systems. For instance telecommunication systems must assure that a communication which delivers the billing information reaches the object which provides billing, but the source object should not wait until the billing information is actually stored. This requires asynchronous communication with assured reception.

If the target can terminate before all operations are executed that are already delivered to the target and one wants to assure that the operation is executed assured invocation must be used. This may be necessary in systems in which an object is replaced by another object, but the pending operations are not transferred to the new object. The old object terminates but notifies all sources of assured invocation operations about the failure of execution.

Communication with non-assured reception is necessary to communicate logging information which can be discarded because an object terminated. SDL provides this kind of communication through *signal* sending.

Using synchronous communication the source object waits until the operation is finished. This kind of communication also provides results of the operation to the source object. SDL supports synchronous communication through *remote procedure calls*.

Communication between two active objects should be allowed only if a communication path exists which supports a special kind of communication. These communication paths can be established explicitly and implicitly. Explicit communication paths improve reliability because during analysis and execution tools can check whether the specified communication paths are used. Explicit communication paths can also improve readability because the reader can recognize the communication structure of the SHC without viewing the code which invokes communication. Specifying communication paths has the disadvantage that it is tedious work, that it can deteriorate readability because it enlarges the specification and that it does not improve security if all objects have disjoint sets of communication interfaces (see next section about interfaces).

SDL allows explicit and implicit communication paths.

8. Interfaces

The implementation of an operation is hidden from the outside world of an object. It is also hidden how a stream is produced or consumed. This information hiding serves to protect the programmer from writing code which depends from implementation details. Only attributes exposed in *interfaces* of an object can be accessed from outside the object. An interface contains a fixed set of attributes of an object. The interface contains information how an object can access attributes of another object. Communication can take place only if an object has access to an interface and to an attribute in this interface of another object. Objects provide the implementation of an interface.

There are two kinds of interfaces:

- *stream interfaces*: which contain stream attributes only and
- *operational interfaces*: which contain data attributes and executable attributes.

During the design of a telecommunication system the programmer can write down the interfaces of an object without caring about the implementation of the interface. This simplifies the task of writing distributed systems. It hides unnecessary implementation details when a task is devided into smaller tasks

Gates, exported variables and exported procedures are the attributes visible from outside a process in SDL. However there is no notion of an interface in SDL. The description of external visible attributes and the implementation are not separated.

9. States

Telecommunication protocols are described with finite state machines. Active object should therefore have a special data attribute with a fixed set of values which serves as *state* of the object. An interface is tied to a subset of all defined states. Access to an interface is only possible if the active object is in one of the states which are tied to the interface.

Objects in telecommunication systems communicate only if the object is in a certain state. For instance in the B-ISDN UNI Q.2931 [Q.2931] protocol the reception of an alerting signal (asynchronous communication with non assured reception) is only allowed if the object is in

state „setup delivered".

SDL has state associated operations since it allows selected signals to be received in selected states only.

10. Predefined Passive Objects

Every language used for implementation needs simple passive objects like integer values or real values. Languages for telecommunication systems need furthermore time values to denote the expiration of timers. More sophisticated objects are necessary for mappings which store associations between two objects. Such an associating is needed to tie call specific information to a call number.

SDL has a great variety of predefined passive objects (SDL uses ACT ONE to define data types). SDL can be combined with ASN.1 [X.208] to be able to use additional values like sets or sequences.

11. Summary

From experiences with the design and implementation of telecommunication systems language constraints are defined. Such a language needs active and passive objects to provide a telecommunication service. Active objects are mapped to threads which can be executed concurrently or alternatively. Objects provide data attributes, executable attributes and stream attributes. Attributes are grouped in interfaces. Interfaces are necessary to establish communication. Communication can take place asynchronous and synchronous.

The ITU-T specification and description language SDL is designed primarily for telecommunication systems. However it does not provide all required features. The main restrictions are

- no interface definitions,
- sophisticated hierarchy definition,
- no asynchronous communication with assured reception and assured invocation and
- limited use of alternative threads.

12. Outlook

This paper does not elaborate requirements about object references, real time constraints or partitioning of language units. It is the starting point for a deeper analysis of language requirements. Other languages like CHILL, LOTOS or ROOM will be evaluated as done for SDL in this paper.

13. REFERENCES

[CM94] M. Chapman, S. Montesi: Overall Concepts and Principles of TINA, TINA Consortium 1994.

[HK97] E. Holz, O. Kath: Abschlußbericht Einsatz von B-ISDN Netzen in einer TINA konformen Kommunikationsplattform, Projekt CAMOUFLAGE/S, Humboldt-Universität Berlin 1997.

[NF96] Nils Fischbeck: Anwendung formaler Beschreibungstechniken in der Standardisierung, Humboldt-Universität Berlin, Institut für Informatik 1996.

[TM96] Toshimi Minoura: Structural active-objects systems fundamentals; in H. Kilov, W. Harvey: Object-Oriented Behavioral Specifications, Kluwer Academic Publishers Norwell, 1996.

[PT87] ETSI: SDL Validation Model for B-ISDN DSS2 Point-to-Multipoint (network); DTR/SPS-05128; December 1996.

[RBP91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: Object Oriented Modelling and Design. Prentice Hall, International Editions (1991).

[Q.2931] ITU-T: B-ISDN DSS 2 UNI Layer 3 Specification for Basic Call/Connection Control, Recommendation Q.2931. ITU-T, 1995.

[X.901] ITU-T Recommendation X.901 / ISO/ IEC 10746-1: Open Distributed Processing - Reference Model Part 1: Overview", ITU-T 1995.

[Z.100] ITU-T: Z.100 (1996): CCITT Specification and Description Language (SDL), ITU-T Jun. 1996.

"Is there life after death?": The Rejuvenation of Life-cycle in a Dynamic Business Object Architecture

Kitty Hung
E-mail: hungks@sbu.ac.uk
Homepage: http://www.scism.sbu.ac.uk/cios/hungks
South Bank University, UK

ABSTRACT

In recent years, Business Object technology is considered to be one of the ideal approaches to deliver solutions to achieve the objective of Software Best Practice (SBP). However, the current phenomenon has shown that the strategies proposed only see the business from the IT developers' pair of "tinted glasses" with the developers looking at the business from their own perspectives. The influence of business end-users over SBP has since been neglected. Business end-users hold business knowledge and they pose to be the most ideal candidates as business information providers and system testers and responders.

Dynamic Systems Development Method (DSDM) is derived from the concept of Rapid Application Development (RAD) with additional principles emphasising on user's involvement. DSDM provides an ideal environment to enable developers to produce quality software while deliver on time and within budget through the techniques of: joint requirement planning (JRP), joint application development (JAD), function points, time-boxing, clean room technique, feasibility studies, business studies, functional model iteration, system design and build iteration, implementation. The holistic approach of DSDM is to form a vehicle to drive the developers and end-users together. Traditionally, developers tend to put a subjective view on their work presuming this is what the real world needs. A fundamental assumption of DSDM is that nothing is built perfectly first time. As a result all steps can be revisited as part of its iterative approach. Therefore the current step needs be completed only enough to move to the next step. DSDM not only provides a life-cycle but also the necessary controls to ensure its success.

This paper attempts to integrate two of the existing techniques namely: (1) Business Object Architecture (BOA) and (2) Dynamic Systems Development Method (DSDM) life-cycle environment to develop a Dynamic Business Object Architecture (DBOA). The DBOA model contains business objects holding business knowledge. The architectural design of the BOA makes the business object components easy to be reused. The rejuvenation of life-cycle through different stages of prototyping is to enable the developers to build a model at an early stage of the project before any significant investment is incurred and allows the developers to modify the system throughout the development phases. The holistic approach of DSDM through substantial user involvement has brought the business end-users and software developers together to achieve the objective of SBP. CAD Consultants Ltd. (a credit insurance agent)'s system has been used in this paper as a case study of our development work.

1 INTRODUCTION

Software Best Practice (SBP) is not only the objective of the European Commission (EC) as a driving force of software initiative but also the objective of commercial organisations and as well as research and development bodies' aim to achieve [ESPRIT97]. Software developers are striking hard to bring new products and strategies to improve software quality.

Although there are many ways to achieve the objective of SBP, we believe that if we want to use software to solve business problems, we should understand the business first. A situation in the world can be in any shapes, any forms. It can be very fuzzy and complicated too. We need to make a structured model of the situation in order to make it easier to tackle the problems. There should be a balance between the business problems and software solutions. If the solutions are less than the problems, the software will not be able to meet the business requirements. However, if the solutions are more than the problems, we would be running at a risk of going over-budget or not meeting the deadline. Another thing is that the end-users and the software developers should pass the right messages to each other. All the end-users care about is the user interface, which reflects the performance of the system. End users do not have to know what programming languages are used nor whether the system is built using Object-oriented method or not. These jobs are passed to the software developers.

In order to deliver a system appropriate to the business needs, end-user's involvement is not only essential but also critical. There are four 'C' factors maintaining the relationship between the software developers and the business end-users namely: (1) communication; (2) co-ordination; (3) co-operation; and (4) compromise. There is no longer a vendor/purchaser relationship. It should be a partnership. Business and IT should set a common goal together and work hand-in-hand towards the goal. To fulfil this goal, the research work contained in this paper is focused on Business Objects (BO), Business Object Architecture (BOA) and the Dynamic Systems Development Method (DSDM) life-cycle environment. The paper is arranged in the following order.

In section two, we address the concept of BO, how it is constructed and its problems. Section three explains why an architecture is needed and the way to construct it. In section four, we discuss about life-cycle and the DSDM life-cycle approach. Section five describes the integration between BOA and DSDM to produce a Dynamic Business Object Architecture (DBOA). Section six describes the case study and evaluation. Section seven is the conclusion. Section eight is the future work.

2 BUSINESS OBJECT

The current climate of object-orientation is still predominantly focusing on the analysis and design methodologies, programming languages and user interfaces [Ramackers96]. However, these effortsonly emphasise on how to produce 'software systems' rather than how to produce 'business solutions'. We have adopted the Business Object technology as [Sutherland95] describes Business Object as *"where in the object are the business"*.

Business Object, as defined by the Object Management Group (OMG)'s Business Object Domain Task Force (BODTF) in their Common Facilities Request For Proposal (RFP CF-44) is: "A representation of a thing active in the business domain including at least its business attributes, behaviour, relationships and constraints. A Business Object may represent a person, place or concept. The representation may be in a natural language, a modelling language, or a programming language" [Shelton96]. In other words, business object is still an object but it extracts the abstraction from the business and simulates it to software components.

The standard definition of Business Object is yet to be finalised by OMG but we have proposed in this paper how we construct the Business Object.

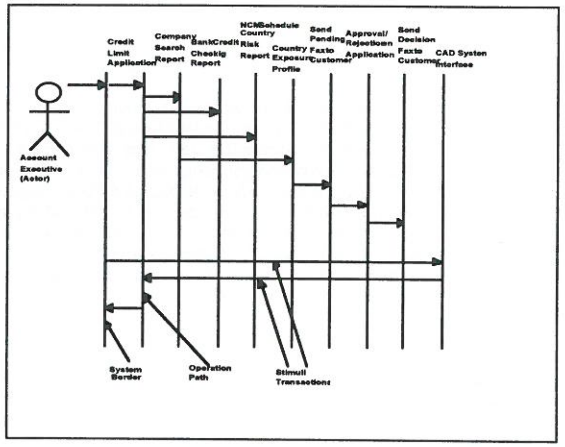

Credit Limit Application
Company Search Report
BankCredit Checkig Report
NCMSchedule Country Risk Report
Country Exposure Profile
Send Pending Faxto Customer
Approval/ Rejectionn Application
Send Decision Faxto Customer
CAD Systen Interface
Account Executive (Actor)
System Border
Operation Path
Stimuli Transactions

Figure 1 : Event Diagram

2.1 Modelling the Business Process

We have adopted Jacobson's Use Case Engineering (UCE) as a foundation to develop Business Objects as shown in Figure 1. UCE has provide a very powerful mechanism to force IT developers to understand the business before anything else. Our Business Object model starts with an Event Diagram to model the business processes on a high abstraction enterprise level.

2.2 Interaction Diagram (Flowchart)

The Interaction diagram shown in Figure 2 is used to define the business procedures in detail. There are actions, decisions, input and output nodes in the diagram.

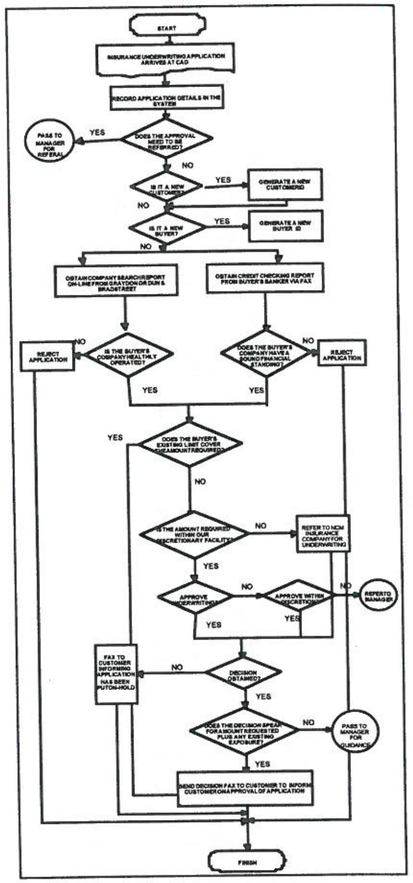

START
INSURANCE UNDERWRITING APPLICATION ARRIVES AT CAD
RECORD APPLICATION DETAILS IN THE SYSTEM
DOES THE APPROVAL NEED TO BE REFERRED?
YES
PASS TO MANAGER FOR REFERAL
NO
IS IT A NEW CUSTOMER?
YES
GENERATE A NEW CUSTOMERID
NO
IS IT A NEW BUYER?
YES
GENERATE A NEW BUYER ID
NO
OBTAIN COMPANY SEARCH REPORT ON-LINE FROM GRAYDON OR DUN & BRADSTREET
OBTAIN CREDIT CHECKING REPORT FROM BUYER'S BANKER VIA FAX
IS THE BUYER'S COMPANY HEALTHILY OPERATED?
NO
REJECT APPLICATION
YES
DOES THE BUYER'S COMPANY HAVE A SOUND FINANCIAL STANDING?
NO
REJECT APPLICATION
YES
DOES THE BUYER'S EXISTING LIMIT COVER THE AMOUNT REQUIRED?
YES
NO
IS THE AMOUNT REQUIRED WITHIN OUR DISCRETIONARY FACILITY?
NO
REFER TO NCM INSURANCE COMPANY FOR UNDERWRITING
YES
APPROVE UNDERWRITING?
NO
APPROVE WITHIN DISCRETION?
NO
REFERTO MANAGER
YES
YES
DECISION OBTAINED?
NO
FAX TO CUSTOMER INFORMING APPLICATION HAS BEEN PUTON-HOLD
YES
DOES THE DECISION SPEAK FOR AMOUNT REQUESTED PLUS ANY EXISTING EXPOSURE?
NO
PASS TO MANAGER FOR GUIDANCE
YES
SEND DECISION FAX TO CUSTOMER TO INFORM CUSTOMER ON APPROVAL OF APPLICATION
FINISH

Figure 2 : Interaction Diagram

2.3 Use Case Model

Having defined the business processes, we use the Actor and use Cases to define the relationships between the business processes and how the people handle the processes. As illustrated in Figure 3, each use case represents a task inside the business processes.

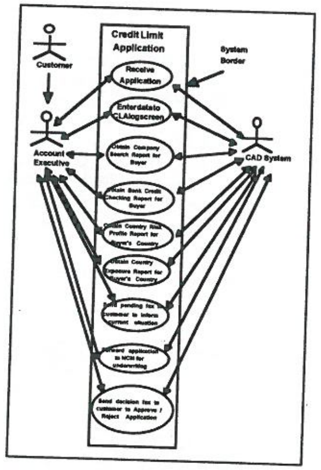

Customer
Credit Limit Application
System Border
Receive Application
Enterdatato CLAlogscreen
Account Executive
Obtain Company Search Report for Buyer
CAD System
Obtain Bank Credit Checking Report for Buyer
Obtain Country Risk Profile Report for Buyer's Country
Obtain Country Exposure Report for Buyer's Country
Send pending fax to customer to inform current situation
Forward application to NCM for underwriting
Send decision fax to customer to Approve / Reject Application

Figure 3 : Use Cases and Actor

2.4 Use Cases and Objects

In the Use Cases and Objects diagram showing in Figure 4, objects are derived from Use Cases. At this stage, we start to identify what objects should be included in the UCE based on the business processes. There are 3 types of objects namely: (1) Interface Objects 9communication boundary between the end-user and the system); (2) Control Objects (manipulating the functionalities and performanceof the system); and (3) Entity Objects (storing the static data). Each type of object carries out different duties.

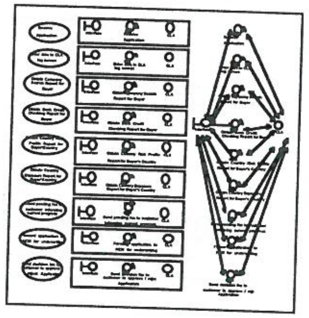


Figure 4 : Use Cases and Objects

2.5 Complete Use Cases Model

Figure 5 shows the communication between the:-

(1) Business End-Users and the Interface Objects;
(2) Interface Objects and the Control Objects;
(3) Control Objects and the Entity Objects;
(4) Entity Objects and the System Developers

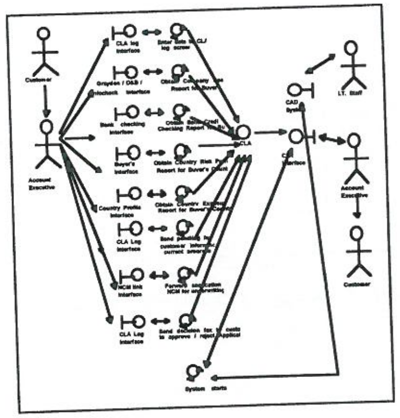


Figure 5 : Complete Use Cases Model

2.6 Business Use Cases and Business Objects

As shown in Figure 6, the InterfaceObject, Control Object and Entity Objects which are derived from the UCE have now formed Business Objects. In other words, Business Objects encapsulate Use Case Objects.

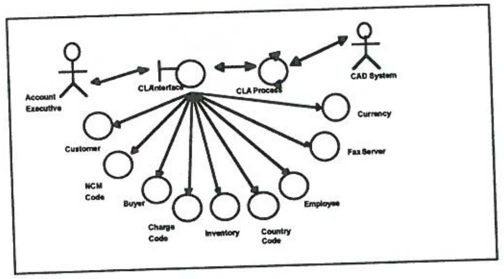


Figure 6 : Business Use Cases and Business Objects

2.7 Business Object encapsulates Entity Objects

Figure 7 expands from Figure 6. Having identified what Entity Objects to be included in the Business Object, we then define the attributes within each Entity Object for aggregation. The Business Object itself has its own attributes and operations that are outside the selection from within the aggregated Entity Objects.

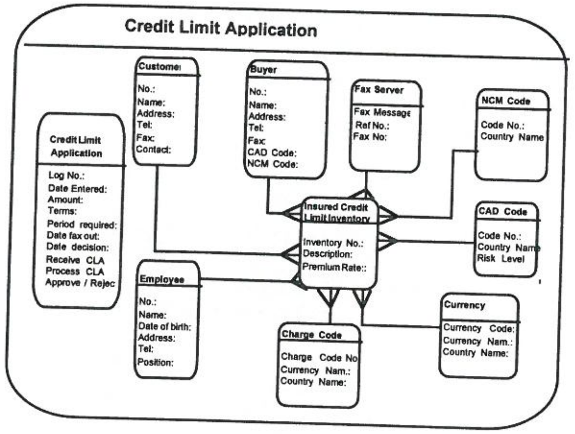


Figure 7 : A visualisation for a Business Object with attributes

2.8 A Road Map of Business Object

As shown in Figure 8, Business Object starts with business process and is followed by the interaction between the Actor and Use Cases. Objects are derived from Use Cases. The Interface Object, Control Object and Entity Object play differentroles. Business Object encapsulates the Use Case Objects in the form of a package consisting of business processes, functionalities, operations and static data.

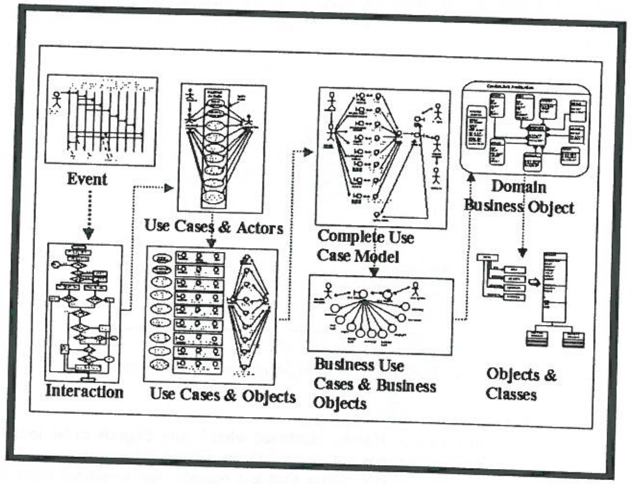


Figure 8 : A Road Map of Business Object

2.9 Problem of Business Object

When it comes to complexity management, no single definition of a Business Object can serve all purpose. One Business Object has only a single interfacedefinition and is unable to take

differentpoints of view [Daniels96]. Moreover, we do need workflow direction to describe different business transactions within an organisation.

3 BUSINESS OBJECT ARCHITECTURE (BOA)

The problem occurs in Business Object has prompted the adoption of Business Object Architecture (BOA). Cory Casanave - Chair of OMG Business Object Domain Task Force (BODTF) defines an architecture to represent the components that are used to 'model' the business problems and build the system. As shown in Figure 9, our proposed BOA frameworkis to adopt both Top-Down and Bottom-Up approaches. After defining the Business Processes from a high level abstraction, Entity Objects are identified Bottom-Up. Finally, Business Objects are formed which integrate the Business Processes, functionalities and operations together. Business Objects do not encapsulate the Entity Objects (as opposed to the conventional way of Object-oriented encapsulation). Rather they call the Entity Objects when they want to use them. The Entity Objects stay in the same position at the bottom of the framework. Benefit of which is that different Business Objects share a single Entity Object. Therefore, if we want to change the attributes of the Entity Object, we only have to change once. Another advantage is that not only we can reuse the Entity Objects but also we can reuse the Business Processes. Business Objects can also be reused as a package as well.

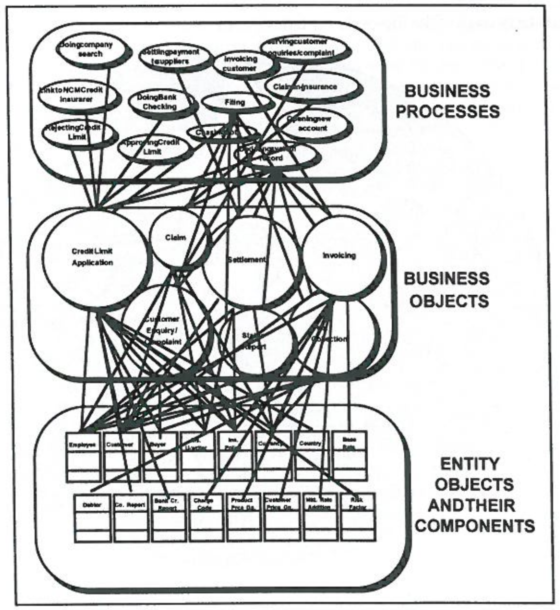


Figure 9 : BOA Framework

4 DYNAMIC SYSTEMS DEVELOPMENT METHOD (DSDM) LIFE-CYCLE ENVIRONMENT

Life-cycle methodologies are used by software developers and project managers in facilitating controllability for their project management. There are differenttypes of life-cycles such as Waterfall [Hargrave96], Spiral [Bohem86], Fountain, Pinball [Henderson-Sellers96], Rapid Application Development [Martin91], Dynamic Systems Development Method [DSDM95(a)]. DSDM provides an ideal environment to enable developers to produce quality software while deliver on time and within budget.

5 DYNAMIC BUSINESS OBJECT ARCHITECTURE (DBOA)

DBOA is a technique to develop a BOA using DSDM's holistic approach by means of substantial user involvement, frequent reviewing, testing, and identification of problems at the early stages of development. The iterative life-cycle also enables developers to review and modify the model even on a conceptual level. Figure 10 shows the DBOA model. The BOA is situated in the centre of the DSDM life-cycle model. Among each life-cycles, there is an incremental prototyping approach through these phases moving anti-clockwise from the top with feasibility studies, 1st phase functional prototype, 2nd phase functional prototype, design prototype and implementation. Black arrows show the transfer points from one phase to another and the grey ones show where the development can easily return to an earlier phase. The white arrow indicates that the BOA model can always be re-architectured at any stage of the life-cycle.

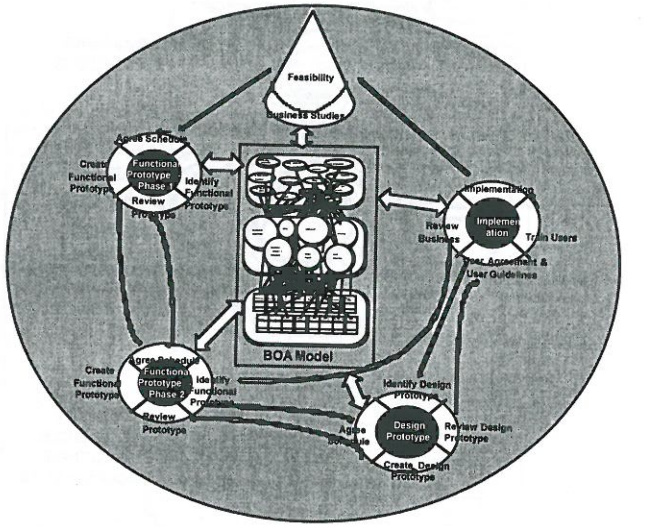


Figure 10 : DBOA Model

6 CASE STUDY AND EVALUATION

6.1 Case Study - A New Buyer System

The case study project is to combine the Domestic Buyer and Export Buyer's data files together to form a single file and single interface. The project spanned a five week period in terms of five time boxes as shown in Figure 11

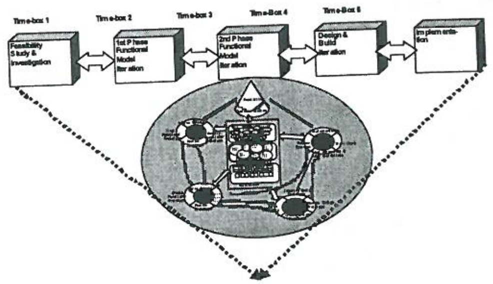

Time-box 1
Time-box 2
Time-box 3
Time-Box 4
Time-Box 5
Feasibility Study & Investigation
1st Phase Functional Model Iteration
2nd Phase Functional Model Iteration
Design & Build Iteration
Implementation

Figure 11 : Time-boxing in DSDM Life-cycle

6.2 Case Study Evaluation

The 'New Buyer System' project had been delivered on time and was operational. BOA model was considered to be satisfactory as a vehicle to communicate with the end-users and interpret the business requirements for the new system and how it linked with other business objects. At the end of the project, the user community was satisfied, and had clearly felt very much involved in the whole process. The final success of the project was felt by the complete 'team'; not only the developers, but also the equally essential end-users who had been so actively involved in the JRP and JAD sessions and the user acceptance testing.

The result of this case study has highlighted a few areas in which we feel that our adaptation of DBOA showed particular strength. None of them is particularly novel, but together they paint a picture of a successful interpretation of the method for a DBOA development in that:

- The gap between the conceptual model and software implementation had obviously been narrowed. Had it not been the DSDM approach, we would not be able to check whether our conceptual BOA model is the right model for the business .
- Communication between developers and end-users was much better. The users were very much involved, to the point where 'the team' was quite definitely a description applicable to the mix of people, developer and user, involved in the project. There was an integration of the two roles; a change of relationship from supplier/consumer to partnership. The final system was our system, not their system. Equally significantly, if not more so, the users enjoyed the experience of taking responsibility for their own system. It is also worth mentioning that the experience was (most of the time!) enjoyable for the developers.
- The holistic approach, as a result of the this partnership, has enabled the developers to obtain a better understanding of the business and its requirements. The intensity and effectiveness of the JRP and JAD sessions was beyond any doubt. The concept of getting the right people to concentrate exclusively on the problem, and of empowering them to make the right decisions, paid off. And because of the heavy involvement of the business end-users, an IT project has become more of a business project. This is consistent with the prototyping that the function of IT support is to solve business problems.
- The iterative prototyping approach worked. The rejuvenation of Life-cycle enabled us to revisit the BOA conceptual model and modify it in response to the circumstances changes and business changes. It is not practical to obtain a correct design from a conventional requirement. We did not even try. The first functional prototype was very much imperfect. But at least it was something for the user to work with. The process of refinement which went on through Time-boxes 2, 3 and 4 resulted in numerous opportunities to fix the imperfections.
- We met, with comparative ease, what would have been an impossible deadline using the conventionallife-cycle.

7 CONCLUSION

Although the result of the above case study is considered to be successful, DSDM is still not a mature technology. There are several 'challenging' areas where we would have to warn the developers when using the DBOA approach:-

- ***Friction between developers and end-users*** : there is always a situation where the developers and the end-users do not get along well.
- ***How to select the "right" people and to empower them to make "right" decision?***: this is more to do with business issue and it can only be improved through experience.
- ***Time-boxing Syndrome***: everything is set inside a time-scale agreed with the business end-users. If planning is insufficient, developers would juggle between time-boxes. They will be forced to omit some unfinished tasks if they overrun the time-boxes or get panic to catch up at later time-boxes or they might have to abandon project if under pressure.
- ***Work Pattern / Paradigm shift for developers*** : the boundary between IT and business world is taken away. Developers have to cross the border to communicate with the business end-users and to experience business environment rather than developing the system in their own environment.

8 FUTURE WORK

- ***Tackle the challenges***: Continue to research on the strategies to tackle the challenges as listed above.

Object Repository/Reuse Library: Object Repository and Reuse Library for managing reuse is to be developed through CASE tools. As the DBOA is developed under a rapid and iterative life-cycle, appropriate CASE environment is critical. High level development tools and 4GL implementation will continue to be used in the future projects using DBOA as a foundation. The combination of both of these technologies means that design/implementation iterations which a few years ago may have taken a matter of weeks, could now be achieved in hours.

9 REFERENCES

[Basson97] Basson, H. et al. Improvement of the Quality of OO Database Evolution in the *Proceedings of Software Quality Management '97 Conference*, Bath, UK, March 1997.
[Casanave95] Casanave, C. "Business Object Architectures and Standards" in the *OOPSLA'95 Conference Business Object Design & Implementation Workshop II*, San Jose, CA, October 1996. Source:http://www.tiac.net/users/jsuth/oopsla/oo95summary.html.
[Daniels96] Daniel, J. "The Reality of Business Objects Today" in the *OMG Building and Using Financial Business Objects Conference*, 23-23 Oct 1996, London.
[DSDM95] DSDM Consortium. *Dynamic Systems Development Method Version 2.0*. Tresseract Publishing, Surrey, UK, 1995.
[ESPRIT97] ESPRIT Project Programme Report on European Systems and Software Initiatives (ESSI). European Community, Brussels, 1997. Source: http://www.cordis.lu/esprit/src/wp.htm.
[Hung97] Hung, K. and Linecar, P. "Experiences In Developing a Small Application Using a DSDM Approach" in the *British Computer Society Software Quality Management '97 Conference Proceedings*. Mechanical Engineering Publications, London, 1997, pp 165-178.
[IFPUG96] International Function Point Users Group. Ohio, USA. Source: http://cuiwww.unige.ch/OSG/FAQ/SE/se-faq-s-2.html#S-2.
[Jacobson94] Jacobson, I. et al. *The Object Advantages: Business Process Reengineering with Object Technology*. Addison Wesley, NY, 1994.
[Jacobson96] Jacobson, I. "Use Case Engineering Tutorial" in the *OOPSLA'96 Conference, San Jose, CA*, October 1996.
[Martin91] Martin, J. *Rapid Application Development*. Macmillan, NY, 1991.
[OMG95] Object Management Group. *Object Management Architecture Guide*. John Wiley & Sons, Inc., N.Y. 1995.
[Ramackers96] Ramackers, G. "BPR with Extended Use Cases and Business Objects" in the *Object World '96 UK Conference*, June 1996, London.
[Rumbaugh96] Rumbaugh, J et al. "The Unified Modelling Language Tutorial" in the *OOPSLA'96 Conference*, October 1996, San Jose, CA.
[Shelton96] Shelton, R et al. *OMG Business Application Architecture White* Paper. OMG Business Object Management Special Interest Group (BOMSIG). Sources: http://www.omg.org.
[Sutherland95] Sutherland, J. "Capturing Business Object Benefits in Corporate Information Systems" in the *IPP Interoperability Symposium*, Brown University, N.Y. Sources: http://www.mpgfk.tu-dresden.de/~weise/docs/ippbrown.html.

Design Patterns as Program Extracts

Eyðun Eli Jacobsen
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, Denmark
E-mail: jacobsen@cs.auc.dk

April 4, 1997

Abstract

We present a view on software systems and emphasize that a software system creates the user's conceptual model. Regarding a software developer as a user, we point out the existence of a gap between the conceptual model of a software development system and the software developer's conceptual model of a software system. To minimize this gap, extending software development systems to support architectural abstractions is motivated. A two level model consisting of a program level and an extract level is proposed.

1 Introduction

Traditionally software has been designed with focus on algorithms and on components making up the system. Today software is to an greater extent being designed with focus on the user's experience with the software. Attention is payed to the nature of human-computer interaction and the metaphorical spaces that users inhabit when using a piece of software—the design of software is the design of the user's conceptual model [Winograd, 1996].

A software developer is a kind of user who uses software to develop new software. If a development system is to be comfortable and productive to use we must, according to [Winograd, 1996], pay attention to the user's conceptual model when designing the software development system.

2 Architectural Abstractions and Development Systems

A software development system can be oriented towards several sets of concepts; examples include classes and objects, functions, and logical assertions.

Software development with classes and objects is now so well-understood that software developers are extending their design language. A software developer's design language is the set of abstractions over structures in a software system. The elements finding way to software developers' design language are new concepts that express archetypical patterns of class and object relations and collaborations. One kind of archetypical pattern are termed design patterns [Gamma et al., 1995]. Design patterns are not dependent on a specific software system, but occur across many software systems. This evolution is reflected in recent books on object-oriented programming, for example [Gamma et al., 1995, Buschmann et al., 1996].

The patterns of class and object collaborations are in [Kristensen, 1996] regarded as examples of *architectural abstractions* where the notion of an architectural abstraction is a more general concept regarding abstraction over structures in software systems, and as such it is a more convenient concept when discussing extensions of a design language as it covers other kinds of abstractions than just class and object relations and collaborations.

With the arrival of new architectural abstractions comes also an increased distance between a software developer's design language and the conceptual model represented by his software development system[1]. The more abstract or complex the new concepts are, the longer the distance between languages becomes. The distance between the languages is problematical since developing software involves both languages and translations between these, and the longer the distance between languages, the harder it is to develop software using the specific software development system.

We are therefore interested in an understanding of design patterns, and to describe a model for the these, and to describe a software development system in which design patterns are part of conceptual model represented be the software development system.

3 Program Level and Extract Level

We regard programs and design patterns to reside at two different levels. The programs reside at the *program level* and design patterns reside at the *extract level*. The overall idea is that the programs themselves are expressed at the program level, and our additional understanding of the programs is expressed at the extract level.

3.1 Program Level

A programming language is a language for describing a process/computation. The program is executed by an interpreter which is constructed to meet the semantic specification of the programming language. The structure of a program is given by an abstract syntax, and can be understood as a series of elements arranged in accordance with the abstract syntax.

A program can also be understood more independently of the abstract syntax—general patterns for organising elements and their relations can be identified and understood at a general level (we might think of composite and observer design patterns here). These patterns will of course contain elements that are part of the abstract syntax, but the organisation is different from what is dictated by the abstract syntax. A program element in a program can be understood through several patterns, and the patterns hence become perspectives on a program.

It is these different understandings of a program that we want to capture at the extract level.

3.2 Extract Level

At the extract level we want to be able to express how we perceive a program. The main element at the extract level is the extract abstraction which design patterns are special cases of. The idea behind extract abstractions is that we group together *relevant* program elements and consider these as a whole.

Two kinds of operations are relevant for extract abstractions:

1. Operations on the relations between extract abstractions and program elements. This is related to connecting program elements and extract abstractions, creating program elements from extract abstractions, and creating extract abstractions from program elements. We will discuss this in Section 3.3
2. Operations on extract abstractions themselves. As a starting point we use the model for abstractions presented in [Kristensen and Østerbye, 1994]. From here we obtain four operations on extract abstractions: aggregation, decomposition, generalisation, and specialisation. Aggregation means that we form a new extract abstraction on the basis on more simple extract abstractions. Decomposition means that we decompose a (more complicated) extract abstraction into more simple extract abstractions. Specialisation means that we form a new extract abstraction by adding properties to it. Generalisation means that we omit some properties from an extract abstraction. In order to understand these four operations on extract abstractions we must have an understanding of the notion of properties. Only then we will be able to outline an algebra for the operations.

[1]We use the terms 'construction language' and the conceptual model represented by the software development system interchangeably.

We consider an extract abstraction to be a sequence of extracts, where an extract can either denote a program element or an extract abstraction. The properties of an extract abstraction are the set of extracts it contains. In order to combine extract abstraction we need to define operations (addition, overwriting) on properties. What should happen when we add two method-extracts?, two class-extracts?

3.3 Relations

In this section we discuss various operations involving both program elements and extract abstractions.

Processes. We discuss the three above mentioned processes.

- Creating program elements from extract abstractions. The elements in the extract abstraction are copied and possibly renamed. The extract abstraction functions as a template for the program code.
- Creating program elements from extract abstractions where the new elements are coupled to already existing program elements. The existing program elements are extended by (a subset of) the elements in the extract abstraction. Here we need to define matching operation between program elements and extract abstractions.
- Marking existing program elements as instances of an extract abstraction. Here we realise that something existing can be seen as an instance of an extract abstraction. In practice, however, something existing will rarely fit to an extract abstraction, so some support/guidance for adjusting the program elements to the extract abstraction will be needed.

Types of references An extract abstraction can be related to two kinds of program elements:

- Concrete program elements. These program elements cannot be specialised, and the elements might, depending on the language, be imperatives, declarations, etc.
- Abstract program elements. These program elements can be specialised, e.g. classes and methods. By relating abstract program elements to extract abstractions we can couple the specialisation of program elements to specialisation of extract abstractions.

Also we can categorise extract abstractions into those that are self contained, and those that are dependent on program elements. The first group can be exemplified by the observer pattern and other general purpose patterns. The latter group consists of context-dependent patterns. As examples we might consider patterns related to a specific framework, which means that they are domain specific.

4 Summary

We presented a motivation for supporting higher level abstractions than object and classes in software development environments. The motivation was rooted in the idea of the user's conceptual model.

We suggested a two level view on software. The program level, which contains the operational description of the system, and the extract level, which contains descriptions of how we perceive parts of the system and how parts are combined.

Some work on software development environments supporting patterns has already been done, but much remains to be done on the conceptual level.

Acknowledgements

We thank Bent Bruun Kristensen for inspiring discussions.

References

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Kristensen, 1996] Kristensen, B. B. (1996). Architectural abstractions and language mechanisms. *Proceedings of the Asia Pacific Software Engineering Conference '96, Seoul, Korea.*

[Kristensen and Østerbye, 1994] Kristensen, B. B. and Østerbye, K. (1994). Conceptual programming. *SIGPLAN Notices*, 29(9).

[Winograd, 1996] Winograd, T., editor (1996). *Bringing Design to Software.* Addison-Wesley, Reading, Massachusetts.

Oberon-D

Adding Database Functionality to an Object-Oriented Development Environment

Markus Knasmüller

Johannes Kepler University
Institute for Practical Computer Science (System Software)
Altenbergerstrasse 69, Linz, 4040, Austria
knasmueller@ssw.uni-linz.ac.at

Abstract. While object-orientation has become a standard technique in modern software engineering, most object-oriented systems lack persistency of objects. This is rather surprising since most objects (e.g. objects in a graphical editor) have a persistent character, thus, they persist past the execution of some program. Nevertheless, most systems require the programmer to implement load and store operations for the objects. In this Ph.D. work we demonstrate the seamless integration of database functionality into an object-oriented development environment, in which the surving of objects is for free. A proof-of-concept prototype implementation has been done in the Oberon system, called Oberon-D.

Keywords: databases, typing

1 Introduction

In today's software engineering projects the advantages of object-orientation such as reusability and extensibility are well-known. While object-orientation is a very common feature of modern software development environments, persistence is not. This is rather surprising since in most applications the objects do not only exist temporarily but persist beyond the execution of the program that created them. Examples are user interface objects of a graphical editor, design objects of a CAD system, and document objects of a workflow system, to mention just a few. If persistence is not supported by the chosen development environment, the software engineer has two possibilities: On the one hand, he can change the environment. But this is often impossible, because high efforts and costs are combined with the used environment. On the other hand, he can try to add persistence to the chosen environment by implementing read and write mechanisms for various object types. It would be advantageous, however, to add persistence as a general feature to the object-oriented development environment, instead of reimplementing it repeatedly for every program that needs it. The main contribution of this paper is to demonstrate the ease of integrating database functionalities into an object-oriented development environment, in our case into the Oberon system [WiGu89].

Oberon-2 [MöWi91] is a general purpose programming language in the tradition of Pascal and Modula-2. It combines the well proven type system and module concept of its ancestors with the new concept of record type extensions. Additional improvements like basic string operations and type-bound procedures make the language more convenient to use. But Oberon is also a run-time environment providing Mark & Sweep garbage collection, dynamic module loading, run-time types, and commands. Commands are procedures that can be called interactively from the user interface by clicking on their names. They provide multiple entry points into a module.

In the project Oberon-D [Kna97] the following database features are added to the Oberon system:

- Persistence, a characteristic describing an object's lifetime.
- Schema Evolution, the possibility to change the type of the persistent objects.
- Query Languages, the possibility of a simple access to the objects.
- Recovery, the guarantee that a consistent state is recovered after a system crash.
- Concurrency

We divided our project into different parts. Each part adds a new database functionality to the Oberon system and is presented in a section of its own.

2 Persistence

Persistence is a characteristic describing an object's lifetime. In a language with persistence, objects may survive between program runs. In contrast to persistent objects, transient objects only exist during one run of a program.

Persistence in Oberon-D is obtained by a persistent heap on the disk. Persistent objects are on this heap, while transient objects are in the transient memory. Transient and persistent objects can access each other mutually. Accessing a persistent object leads to loading the object into the transient heap. If persistent objects are not accessed from transient objects any more, they will be written back to the persistent heap. Persistent objects which are not referenced by a persistent root are reclaimed by a garbage collector.

All objects are allocated with Oberon's standard procedure NEW. They become persistent as soon as they are referenced (directly or indirectly) from a persistent root. A persistent root is any object that has been registered using the procedure *Persistent.SetRoot (obj, key)*, where *obj* is (a pointer to) an object that should become a persistent root and *key* is a user-defined string that serves a unique name for the root.

Applications may access a root object with the name *key* by using the procedure *Persistent.GetRoot (obj, key)*. Having the root, persistent objects which are directly or indirectly referenced from the respective root object can simply be accessed by pointer dereferencing.

The following code fragment shows how to make a list of objects persistent by registering its root with the key "myroot":

```
TYPE
   Node = POINTER TO NodeDesc;
   NodeDesc = RECORD data: INTEGER; next: Node END;
VAR
   p, q: Node;
...
NEW (p); p.data := ...;
NEW (q); q.data := ...;
p.next := q; q.next := NIL;
Persistent.SetRoot (p, "myroot")
```

This persistent list can be accessed as follows:

```
Persistent.GetRoot (p, "myroot");
WHILE p # NIL DO
   Out.Int (p.data, 10);
   p := p.next
END
```

Another central point is the deallocation of unused persistent data. Transient data is reclaimed automatically by a garbage collector that frees programmers from the non-trivial task of deallocating data structures correctly and thus helps to avoid errors. We also use a garbage collector for persistent data. All disk objects that are not accessible from a persistent root are garbage and will therefore be removed in the next run of the garbage collector. The garbage collector is started explicitly by calling a command. A persistent root with memory name *n* can be removed by the function *Persistent.RemoveRoot (n)*.

After the presentation of the usage of persistence in Oberon-D the implementation is introduced: Each object is identified by a unique key, the object identifier OID. In order to make an object persistent it is necessary to map it into an external representation. There are three decisions involved: When should the object be externalized, where should it survive, and how should this mapping work?

• When?
This point was easy to decide. An object should be externalized when it is no longer referenced by a transient object. There is only one possible time to map such objects: between the Mark and the Sweep phase of the garbage collector. All persistent objects that are unmarked after the Mark phase are externalized. Externalizing an object earlier would allow that it could still be changed afterwards via references to it. Externalizing an unreferenced object after the Sweep phase is impossible, since the object does not exist any more.

• Where?
All persistent objects are stored in a single persistent heap. Their positions are reflected in their OID. In future versions we plan to investigate the use of multiple heaps (see also Section 5).

• How?
An externalization procedure is called for all objects, that should be externalized. This procedure maps them to their external presentation by writing the type of the object and the object's data to the disk.
Another interesting question is how to load persistent objects. The trap handler is responsible for reading persistent objects. Every time an OID is dereferenced, an illegal pointer trap is caused. The trap handler determines the register whose contents caused the trap. If the register contains NIL the standard trap handler is called, otherwise the absolute value of the register's contents is equivalent to a persistent object's OID. In this case the trap handler loads the object into the transient heap.

3 Schema Evolution

Persistent objects can exist for a long time, but in this time the environment may change, e.g. new experiences and new demands can influence the persistent objects. These environment changes can lead to new object structures to better meet the needs of the applications. Such changes are often not supported by the database system, so the user is himself responsible for a schema change, which means high efforts and costs for the user. It would be advantageous if the system supports the user by handling these changes automatically. Such a process - called schema evolution - is often missed, because of the high implementation effort and the produced additional delays. Our solution is to integrate the schema evolution process into the persistent garbage collector. So, no additional delays are produced, and the implementation is rather easy.

For the user of Oberon-D the schema evolution process is rather simple. He has to start the persistent garbage collector, which migrates all inconsistent objects to the new type automatically. So schema changes, like adding or deleting of attributes, renaming attributes, and modifying the inheritance hierarchy can simply be detected and repaired. There is also a possibility to handle schema changes in a manual way, thereby the user has to implement a transformation procedure, which reads the object from the disk, by using the special type *PersMaps.Map*. With this procedure changes such as summing up of two attributes to a new component are possible. The following code fragement shows such a procedure.

```
PROCEDURE TransProc* (map: PersMaps.Map; o: SYSTEM.PTR);
(* transformation procedure for type T *)
    VAR a: T; x, y: LONGINT;
BEGIN
    a := SYSTEM.VAL (T, o); (* o interpreted as of type T *)
    map.ReadLInt (x); map.ReadLInt (y); a.sum := x + y;
    map.ReadInt (dummy);
    map.ReadBool (a.cond)
END TransProc;
```

The schema change which is done with this procedure is shown in Figure 1.

Old Type Structure:	New Type Structure:

TYPE
  T = POINTER TO TDesc;
  TDesc = RECORD
    x, y: LONGINT;
    z: INTEGER;
    cond: BOOLEAN
  END
	TYPE
	  T = POINTER TO TDesc;
	  TDesc = RECORD
	    sum: LONGINT;
	    cond: BOOLEAN
	  END

Figure 1. Old and new type structure of type *T*.

Also the implementation of schema evolution is rather simple. It is done during a persistent garbage collection run. Oberon-D uses Stop & Copy [Wil92] garbage collection to delete obsolete persistent data. This algorithm uses two heaps (files) and copies all accessible objects from the heap *fromHeap* to the heap *toHeap*.

For each referenced object the garbage collector checks if its type was modified since the last garbage collector run. If so, the object is read from *fromHeap* using the old type definition and written to *toHeap* using the new type definition. Between reading and writing, a conversion from the old to the new format is done. The general idea can be seen in Figure 2.

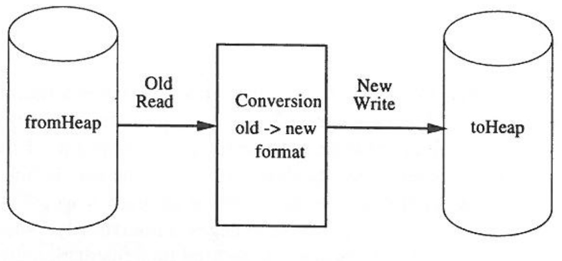


Figure 2. General idea of schema evolution process

In a first step, the garbage collector has to find out which types have been changed. It iterates over the list of persistent types and checks for each type if the old type structure still equals the

current type structure, which is obtained from the run-time type information (available by the Oberon system). Each type with a changed type structure is marked.

As the next step, an *attribute mapping list* is built for every marked type. It shows how to map the attributes of the old type to those of the new type. To build the mapping list, the fields of the old type and the new type are first collected in two separate field lists containing the names and the types of all fields including those of their supertypes (see Figures 3 and 4). The fields of types for which a transformation procedure was specified are not included in this field list.

Old Type Structure:	New Type Structure:

TYPE A = POINTER TO ADesc;
TYPE B = POINTER TO BDesc;
 	
TYPE BDesc = RECORD
  a: SHORTINT;
  b: LONGINT;
  c: A;
END;
 	
TYPE ADesc = RECORD (BDesc)
  d: INTEGER;
  e: B;
  x: PROCEDURE;
END;
	TYPE A = POINTER TO ADesc;
	TYPE C = POINTER TO CDesc;
	
	TYPE CDesc = RECORD
	  a: INTEGER;
	  c: LONGINT;
	  d: A;
	END;
	
	TYPE ADesc = RECORD (CDesc)
	  e: B;
	  x: PROCEDURE;
	END;

Figure 3. Example for type change

The attribute mapping list is then built as follows: For every field *y* in *Old* an entry is created in the mapping list if one of the following conditions hold:

- *New* contains a field *x* with the same name as the field *y* and *x* := *y* is a valid assignment.
- The yet unmapped part of *New* contains just one field *x* with the same type as *y* (*x* and *y* may have different names).

A mapping list entry contains the offsets and the types of *x* and *y*. Any fields of *New* that could not be mapped will get undefined values later. Figure 4 shows the mapping list for the example in Figure 3.

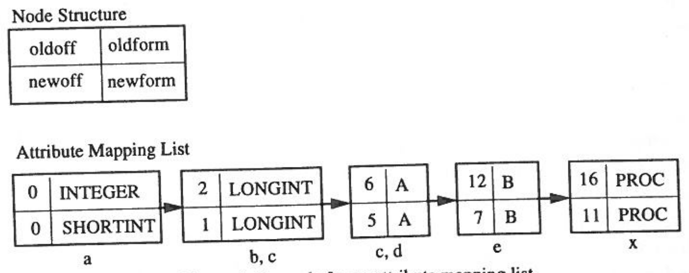


Figure 4. Example for an attribute mapping list.

The entry (*b*, *c*) was included, because when *b* was inspected in *Old*, *c* was the only unmapped LONGINT field in *New*. The same holds for the entry (*c*, *d*). The entry *a* was included because the corresponding fields in *Old* and *New* were assignment compatible. Field *d* from *Old* was not included.

If a transformation procedure was specified, this procedure does the mapping, otherwise the attribute mapping list is used. After the garbage collection, the type list is changed to contain the new type structures.

4 Query Language

Object-oriented databases support accessing to data by implementing programs. This is convenient for professional programmers, but not always for end users who prefer simple access via an SQL like language. Such an SQL like language for object-oriented databases, called OQL, is presented by the Object Database Management Group (ODMG) [Cat96]. In our work we implemented an Oberon-2 binding for the ODMG-93 standard.

In order to exchange data between the database and a program there must be a mapping between the object types in the database and the types in the Oberon program. Basic ODMG types such as integers or strings are mapped to standard Oberon types. Types for which there is no counterpart in the Oberon language such as relations and collections are mapped to abstract data types defined in a module *OML*. This module defines the collections *Set*, *Bag*, *List*, and *Array*. Before a collection is used it must be allocated and initialized by specifying its element type.

Objects in Oberon have a unique identity, and references to objects may appear in a variety of naming contexts. These names can be accessed from OQL. The object query language can be used to perform database queries. There are two ways to call OQL commands from Oberon programs: one can call a method of a collection object, i.e. *col.Select (cmd)*, *col.Exists (cmd)*, or *col.ForAll (cmd)*; alternatively, one can call the procedure *OQL.Query (cmd, res)*. The parameter *cmd* is a string containing an OQL command in which all visible Oberon variables can be used (i.e., local variables, global variables or persistent roots).

In both cases the OQL command is translated into an Oberon-2 program, which is compiled and executed "on the fly", which means that the user is not aware of the translation.

The *ForAll* method is a universal quantification, which returns TRUE if all the elements of the collection satisfy the query and FALSE otherwise. The *Exists* method is an existential quantification, which returns TRUE if one of the elements of the collection satisfies the query and FALSE otherwise. With the *Select* method all objects of the collection which satisfy the query can be fetched. That means *col.Select ("age = 15")* returns the same result as the OQL statement *select all from col where col.age = 15.*

The procedure *OQL.Query (cmd, res)* executes the OQL command *cmd*, and returns the result in the parameter *res*, which is a structure of type *OML.Value*. *Value* has an attribute *class* indicating the type of the result as well as attributes for each possible type of result. A very simple example of such a call is *OQL.Query ("3 < 4", res)* which retrieves a value *res* with *res.class = OML.Bool* and the boolean attribute *res.b = TRUE.*

As an example for the query language we show an Oberon-2 application that accesses an ODMG software database containing information about modules (type *Module*), procedures (type *Proc*) and variables (type *Var*). Each module has a *name*, a *codesize*, a set of procedures *procs*, and a set of variables *vars*. A global variable *mods* holds the set of all modules. The procedure *Query* in Figure 5 shows how to perform OQL queries from within Oberon (an asterisk after a declared name means that the name is exported).

```
MODULE SWDatabase;

IMPORT OML, OQL;

TYPE
  Module* = POINTER TO ModuleDesc;
  ModuleDesc* = RECORD
    name*: ARRAY 32 OF CHAR;
    codesize*: LONGINT;
    procs*: OML.Set;
    vars*: OML.Set;
  END;

  Proc* = POINTER TO ProcDesc;
  ProcDesc* = RECORD
    name*: ARRAY 32 OF CHAR;
    vars*: OML.Set;
  END;

  Var* = POINTER TO VarDesc;
  VarDesc* = RECORD
    name*: ARRAY 32 OF CHAR;
    type*: SHORTINT; (* unique number ident. type *)
  END;

  VAR
    mods: OML.Set; (* set of all modules *)

  PROCEDURE Query*;
    VAR c: OML.Collection; res: OML.Value;
  BEGIN
    (* -- e.g. all modules with codesize > 500 *)
    c := mods.Select ("codesize > 500");
    (* -- e.g. all local variables with name "i" *)
    OQL.Query ("select var from mods, mods.vars as var where var.name = 'i' ", res);
      (* => the value of res.class = OML.Pointer and res.o is a reference to the result set *)
    (* -- e.g. all procedures of module "Types" *)
    OQL.Query ("select m.procs from mods as m where m.name = 'Types' ", res);
      (* => the value of res.class = OML.Pointer and res.o is a reference to the result set *)
    (* ... *)
  END Query;

END SWDatabase.
```

Figure 5. Example

5 Future Work

Oberon-D is an ongoing project. The next steps will be to add the following items (see also http://www.ssw.uni-linz.ac.at/Projects/OberonD.html):

- Recovery: Another interesting point is the occurrence of a system crash. In that case the system must guarantee that the database is in a consistent state at the next system start.
- Concurrency: Concurrency control limits simultaneous reads and updates by different users to give all users a consistent view of the data [Cat94, p.69ff]. Although Oberon-D is a single-user database system, there may be different tasks working simultaneously on the persistent heap.

Additionally, the already implemented functionalities will be improved in the future. The most important points are:

- Optimization of the persistent garbage collection: Garbage collection can be optimized by caching the old and the new heap of the copy collector in transient memory. Nearly all of the memory can be taken for this purpose. Depending on the number of cache blocks and the size of each block the time profit is over eighty percent. The best values for these two parameters have yet to be found.
- Usage of more than one heap: In the current version all objects are stored on one persistent heap. In future versions the user should have the choice between different heaps, e.g., each application can have its own persistent heap
- Schema evolution should be supported by offering the user a schema editing tool. This allows the description and revision of the schema through a graphical user interface.
- Furthermore our work will be evaluated by implementing a huge example, probably a software database.

Acknowledgments

I wish to thank Prof. Hanspeter Mössenböck and Prof. Gerti Kappel for their support of this project. My thank goes also to my colleagues Markus Hof, Christoph Steindl, and Josef Templ for many stimulating discussions about Oberon-D.

References

[Cat94] R.G.G.Cattel, *Object Data Management*
Addision-Wesley, 1994

[Cat96] R.G.G. Cattel (ed.): *The Object Database Standard: ODMG-93*
2nd Edition, Addision-Wesley, 1996

[Kna97] M.Knasmüller, *Adding Persistence to the Oberon-System*
Proc. of the JMLC 97, Linz, Springer, 1997

[MöWi91] H.Mössenböck, N.Wirth: *The Programming Language Oberon-2*
Structured Programming, 12, 4, 1991, pp. 179 - 195

[WiGu89] N.Wirth, J.Gutknecht, *The Oberon System*
Software - Practice and Experiences, 19, 9, 1989

[Wil92] P.Wilson, *Uniprocessor Garbage Collection Techniques*
Lecture Notes in Computer Science 637, Springer 1992, pp. 1-42

On Testing of Object-Oriented Programs

Yvan LABICHE
LIS - Aérospatiale
LAAS - CNRS

7 Avenue du Colonel Roche
31077 Toulouse Cedex - FRANCE
Tel.: 33.5.61.33.62.05, Fax.: 33.5.61.33.64.11

ylabiche@laas.fr

Abstract

The object-oriented paradigm is a new technology for producing software. This new technology has many benefits for parts of the entire software development cycle (analysis, design and implementation phases) : the object-oriented development process is iterative, the object-oriented paradigm emphasize reuse, the items of interest are always the objects, ... Thus, engineers and managers want to use this technology in their own field. But, for critical systems, which need a certification, the testing process is an important task.
Then, the testing techniques for object-oriented programs should be studied even though some people can think the object-oriented paradigm seams to be the panacea. By applying usual testing techniques (i.e. those for procedural programs) for object-oriented programs one found two major problems. First, procedural testing techniques are not well-suited for object-oriented ones : there is an intrinsic difference between the procedural and the object-oriented approach (structure vs. behaviour). Second, the new useful features introduced (such as encapsulation, inheritance, or polymorphism) imply new problems for the testing process : problems of observability and undecidability, for example.
Then we are interested in studying these new mechanisms with the tester viewpoint in mind through three major questions. What are the new problems raised? How usual solutions for the testing process can be applied (or extended) or do we have to find new ones? and, How object-orientedness (the new mechanisms) can help us for the testing process?

1. Introduction

The testing process of a software is part of its verification process, which aims at verifying that its implementation meets its specification. Testing consists in exercising the software with input values. In practice, testing the software with all the input values from the input domain (exhaustive testing) is not feasible. Then the tester has to select a subset of the input domain that is well-suited for revealing the real, but unknown, faults. The selection is guided by test criteria that relate either to a model of the program (structural testing) or a model of its functionality (functional testing) [Beizer 1990].
Given a model and a criterion, there are two principles for generating test inputs : deterministic and probabilistic. In the deterministic approach, test inputs are selected from the input domain (generally by hand) in accordance with the criterion [Beizer 1990]. In random, or statistical, testing, test patterns are selected according to a defined probability distribution on the input domain : the distribution and the number of input data are determined according to the adopted criteria (see e.g. [Duran & Ntafos 1984, Thévenod-Fosse *et al.* 1995]).

Testing is then accomplished in three main phases (testing levels) :

- unit testing is the test of small building blocks (the unit is the smallest piece of software that can be independently tested). With "procedure-oriented" software in mind, unit testing focuses on testing subprograms (the work of one programmer which consists of several hundred or fewer, lines of code);
- integration testing is the test of an integrated aggregate of one or more units;
- system testing (software, library, ...).

Object-Oriented paradigm is a radically new approach to software construction [Korson & McGregor 1990, Hill 1996]. It introduces new features so that some people can consider testing useless. Even if object-oriented development methods increase reusability, testing is still a necessary step to produce highly reliable software because to err is human. An extensive survey of the literature on this topic can be found in [Binder 1996].

In the remainder of this article, we discuss the modifications introduced by the object-oriented paradigm in the testing process. We present shortly the problems raised by this new approach and some work on these topics. And finally, we describe directions for our further investigations.

2. Testing of Object-Oriented Programs

In object-oriented programs, subprograms (i.e. object methods) cannot be considered as the basic unit of testing. Indeed, object methods take part all together in the object behaviour. Hence, the smallest unit to be tested in an object-oriented program is the instanciation of one class. Moreover, due to this change of testing level and new features introduced by object-orientedness (like encapsulation, inheritance or polymorphism), usual models of programs and then associated test criteria cannot be applied directly. Nevertheless they are still well-suited for some parts of the test, typically algorithm testing. Then with object-oriented programs, there is a need to find new models and new criteria, or to modify usual ones.

To find new models, one have to understand the new features introduced by object-orientedness and the problems they raise.

Objects are the basic run-time entities in an object-oriented program. They are instanciation of classes and encapsulate both state and behaviour (i.e. attributes and methods). Testing a class, the basic unit in an object-oriented program, is impeded by three major mechanisms as explain in [Lejter *et al.* 1992, Turner & Robson 1993a, Barbey *et al.* 1994, Mc Daniel & Mc Gregor 1994, Kung *et al.* 1995] : encapsulation, inheritance and polymorphism.

- Encapsulation implies a problem of observability because the only way to observe the state of an object (during testing one want to check the coherence of the object's states) is through its methods. There is a problem if some attributes cannot be reached through a method. Proposed solutions are intrusive and consist in adding methods in the class under test for observability or deriving this class by new one with full visibility (one method per attribute).
- With inheritance, it is clear that added and overridden methods have to be tested but one can think that inherited methods don't need retesting (if the parent class is already tested). It is a mistake because invocations of a method and its results depend on the object's state. Furthermore the set of states may differ from the parent class from the child class, because among other things attributes may be different. Thus, every method of a class has to be tested according to the set of object's states.

- Polymorphism, and dynamic binding introduce a problem of undecidability. It is very difficult or even impossible to statically determine which method will be invoked in a given test case. The problem appears when a call to a polymorphic method is made or when a method has polymorphic parameters. Nevertheless, an intrinsic characteristic of polymorphic methods can help use : they all logically perform the same task (otherwise, they cannot be polymorphic). Then one can specify the expected minimal features of a polymorphic method (the task) and all its possible redefinitions.

Some work currently takes these mechanisms into account. They concentrate on testing levels, models and methodologies.

Classical testing levels (i.e. unit testing, integration testing, ...) are used with some changes to test object-oriented programs in [Fiedler 1989, Jorgensen & Erickson 1994]. And new testing levels suitable for object-oriented programs are studied : class testing (see e.g. [Murphy *et al.* 1994]), intra-class and interclass testing (see e.g. [Kung *et al.* 1993]).

Testing at the class level is also introduced through a new model. Because the class is the basic unit for testing strategies, one have to consider its behaviour. For this purpose the behaviour is break down into states, i.e. values stored in each of the variables that constitute the data representation. This is state-based testing [Kung *et al.* 1993, Turner & Robson 1993b, Kung *et al.* 1995]. State-based testing focuses on objects' state-dependent behaviour rather than the control structure and individual data.
With the new testing level (the class), Kung et al. present three complementary diagrams obtained from the reverse engineering of source code : the object relation diagram (displays inheritance, aggregation and association relationships between classes), the block branch diagram (a kind of control and data flow graph associated to each method), the object state diagram (hierarchical, concurrent, communicating state machine that models the object-state dependent behaviour).

Two important methodologies are introduced to take advantage of object-oriented features.
Incremental testing [Harrold *et al.* 1992] consists in constructing an inheritance hierarchy for testing history of classes in parallel with the class hierarchy. Depending on the inheritance relationship between the parent class and the child class, one can decide to use or not test cases from the parent class, or design new test cases.
The use of statistical (structural) testing for object-oriented programs is studied in [Thévenod-Fosse & Waeselynck 1996]. Emphasis is put on the inheritance mechanism so that progressive testing of small subsystems (class cluster) is facilitated.

Most of the work describe above is made from the software analysis (reverse engineering, ...). Another approach consists in testing from formal specification. From this point of view, work has been made for procedural programs and also for object-oriented programs (see e.g. [Doong & Frankl 1994, Barbey *et al.* 1996]).

Without doing formal specification, one can however use appropriate analysis and design methodologies with the aim of testing.
In some work, design models are used to help testing procedural software. Particularly, finite-state machines are used in [Chow 1978, Waeselynck 1993, Thévenod-Fosse *et al.* 1995] for testing purpose. Statecharts [Harel 1988], an extension of finite-state machines producing modular, hierarchical, and structured descriptions are also used to model behavioural aspects of a software : in [Thévenod-Fosse & Waeselynck 1993, Waeselynck 1993] the authors use Statemate, a tool based on Statecharts [Harel *et al.* 1990], for designing test cases.
Unfortunately, little work is made to use object-oriented analysis and design methodologies for the testing process [Binder 1994, Poston 1994].

3. On going work

In that context, our purpose is to study new models (and test criteria for these models) or adaptation of classical ones that take into account object-orientedness. New features like encapsulation, inheritance, polymorphism, dynamic binding or genericity have to be addressed in such models.This first step in our work will be made without being worried about the testing level (unit testing, integration testing, class testing, ...) or the testing technique which will be applied according to the studied model (deterministic testing, statistical testing, ...).

In that direction we are interested in extension of state-machines (for object-orientedness) so that testing techniques based on state-machine could be applied, or extended. At the present time, work is made to use Statecharts in object-oriented design. Coleman et al. [Coleman *et al.* 1992] extend Statecharts in Objectcharts so that they characterise the behaviour of a class as a state machine. Typically, Objectchart transitions correspond to state-changing methods of a class and object attributes define Objectchart states. With Objectcharts, one can described the inheritance relationships. In [Harel & Gery 1996] the authors introduce an UML-consistent (UML - Unified Modeling Language) extension of Statecharts named O-charts for modeling object-oriented systems. This approach addresses inheritance, dynamic changing, association relationships. These two approaches and the model introduced by Kung et al [Kung *et al.* 1993] are very close because they try to address the same things from the same graphical language : finite-state machines.
Furthermore, we are interested in using object-oriented analysis and design methodologies in general. From this point of view, the Object Modeling Technique (OMT) [Rumbaugh *et al.* 1991] and its union with the Booch's method [Rumbaugh 1996] seam to be good candidates. Indeed, in these methods, the behavioural aspects are designed with Statecharts.

Then we are interested in applying statistical testing to object-oriented programs because several case studies have already confirmed the high fault revealing power of this technique for procedural programs (see e.g. [Thévenod-Fosse *et al.* 1995]).

We also want to address the problem of the programming language. Each object-oriented programming language (like C++, Java or Ada95) implements differently some features of the object-oriented paradigm. Hence, the question of whether or not testing has to be different from one to another is still an open issue.

References

[Barbey *et al.* 1994] S. Barbey, M. Ammann and A. Strohmeier, "Open issues in testing Object-Oriented Software", in *Proc. of the European Conference on Software Quality,* pp.257-67, 1994.

[Barbey *et al.* 1996] S. Barbey, D. Buchs and C. Péraire, "A Theory of Specification-Based Testing for Object-Oriented Software", in *Proc. of the 2nd European Dependable Computing Conference,* (Italy), pp.303-20, 1996.

[Beizer 1990] B. Beizer, *Software Testing Techniques,* Van Nostrand Reinhold, New York, 1990.

[Binder 1994] R. V. Binder, "Design for Testability in Object-Oriented Systems", *Communication of the ACM*, 37 (9), pp.87-101, 1994.

[Binder 1996] R. V. Binder, "Testing Object-Oriented Software: a Survey", *Journal of Software Testing, Verification & Reliability*, 6, pp.125-252, 1996.

[Chow 1978] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines", *IEEE Transactions on Software Engineering*, SE-4 (3), pp.178-87, 1978.

[Coleman *et al.* 1992] D. Coleman, F. Hayes and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design", *IEEE Transactions on Software Engineering*, 18, pp.9-18, 1992.

[Doong & Frankl 1994] R. Doong and P. G. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs", *ACM Transactions on Software Engineering and Methodology*, 3(4), pp.101-30, 1994.

[Duran & Ntafos 1984] J. W. Duran and S. C. Ntafos, "An evaluation of random testing", *IEEE Transactions on Software Engineering*, SE-10 (4), pp.438-44, 1984.

[Fiedler 1989] S. P. Fiedler, "Object-Oriented Unit Testing", *Hewlett-Packard Journal*, pp.69-74, 1989.

[Harel 1988] D. Harel, "On Visual Formalisms", *Communications of the ACM*, 31, pp.514-30, 1988.

[Harel & Gery 1996] D. Harel and E. Gery, "Executable Object Modeling with Satecharts", in *Proc. of the 18th Int. Conf. on Software Engineering,* (I. Press, Ed.), pp.246-57, 1996.

[Harel *et al.* 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, "STATEMATE : A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, 16 (4), pp.403-13, 1990.

[Harrold *et al.* 1992] M. J. Harrold, J. D. McGregor and K. J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", in *Proc. of the 14th Int. Conf. on Software Engineering,* (Melbourne, Australia), pp.68-80, 1992.

[Hill 1996] D. R. C. Hill, *Object-Oriented Analysis & Simulation,* Addison-Wesley, 1996.

[Jorgensen & Erickson 1994] P. C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing", *Communications of the ACM*, 37(9), pp.30-8, 1994.

[Korson & McGregor 1990] T. Korson and J. D. McGregor, "Understanding Object-Oriented : a Unifying Paradigm", *Communications of the ACM*, 33, pp.40-60, 1990.

[Kung *et al.* 1993] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Design Recovery for Software Testing of Object-Oriented Programs", in *Proc. of the Working Conference on Reverse Engineering,* (IEEE, Ed.), pp.202-11, 1993.

[Kung *et al.* 1995] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim and Y. Song, "Developing an Object-Oriented Software Testing and Maintenance Environment", *Communications of the ACM*, 38, pp.75-87, 1995.

[Lejter *et al.* 1992] M. Lejter, S. Meyers and S. Reiss, "Support for Maintaining Object-Oriented Programs", *IEEE Transactions on Software Engineering*, 18(12), pp.1045-52, 1992.

[Mc Daniel & Mc Gregor 1994] R. Mc Daniel and J. Mc Gregor, *Testing the Polymorphic Interactions between Classes*, Department of Computer Science, Clemson University, Technical Report, N°TR94-103, March, 7 1994.

[Murphy *et al.* 1994] G. C. Murphy, P. Townsend and P. S. Wong, "Experiences with Cluster and Class Testing", *Communication of the ACM*, 37 (9), pp.39-47, 1994.

[Poston 1994] R. M. Poston, "Automated Testing from Object Models", *Communications of the ACM*, 37 (9), pp.48-58, 1994.

[Rumbaugh 1996] J. Rumbaugh, "To form a more perfect union : Unifying the OMT and Booch methods", *Journal of Object-Oriented Programming*, 8, pp.14-8, 1996.

[Rumbaugh *et al.* 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[Thévenod-Fosse & Waeselynck 1993] P. Thévenod-Fosse and H. Waeselynck, "Statemate Applied to Statistical Testing", in *Proc. of the International Symposium on Software Testing and Analysis,* (Cambridge, USA), pp.99-109, 1993.

[Thévenod-Fosse & Waeselynck 1996] P. Thévenod-Fosse and H. Waeselynck, *Towards a Statistical Approach to Testing Object-Oriented Programs*, LAAS, (To appear in Proc. 27th Int. Symposium on Fault-Tolerant Computing (FTCS-27), (Seattle, USA), June 1997), N°96481, December 1996.

[Thévenod-Fosse *et al.* 1995] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software Statistical Testing", in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), ESPRIT Basic Research Series, Springer Verlag, 1995.

[Turner & Robson 1993a] C. D. Turner and D. J. Robson, *Guidance for the Testing of Object-Oriented Programs*, Computer Science Divison, School of Engineering and Computer Science (SECS), University of Durham, Technical Report, N°TR 2/93, June 1993a.

[Turner & Robson 1993b] C. D. Turner and D. J. Robson, "The State-Based Testing of Object-Oriented Programs", in *Proc. of the Conference on Software Maintenance*, (I. C. S. Press, Ed.), (Los Almitos, California, USA), pp.302-10, 1993b.

[Waeselynck 1993] H. Waeselynck, *Vérification de Logiciels Critiques par le Test Statistique*, Doctoral Dissertation, Institut National Polytechnique de Toulouse, LAAS Report N°93.006, 1993.

Implementing Generic Types for SDL'92

Martin von Löwis
Harald Böhme
Humboldt Universität zu Berlin
Lindenstr. 54a
10117 Berlin
E-Mail: loewis@informatik.hu-berlin.de

SDL'92 supports the concept of parametric types to allow using a type in different contexts. The SITE generator produces C++ code for a given specification. This paper presents an approach to translate generic SDL types into C++.

Translating SDL to C++

The ITU-T Specification and Description Language [Z.100] is used to define telecommunication services and protocols. It supports structural and behavioural specification elements. In its 1992 revision, SDL introduced concepts for object oriented specifications and generic types. The SDL Integrated Tool Environment (SITE, [Sch95]) provides tools to analyse and execute SDL specifications. Execution of SDL is based on a translation process of a given SDL specification to a set of C++ classes. The generated C++ code is structurally similar to the original SDL specification: For a given SDL entity (system, block, process, procedure, service, datatype specification), a fixed number of C++ classes is generated, usually one.
The generated classes are derived from classes defined in the SDL runtime library. The interworking between the generated classes and the library classes is defined by means of method calls: The generated code calls methods defined in the library, and the library calls virtual methods implemented in the generated code. This set of methods defines the interface of an abstract SDL maschine. The interface allows the implementation of the SDL maschine in different ways. There are currently two SDL libraries: The simulation library allows the execution of a specification using a simulator, and the prototype library provides the interworking with a given target system.

SDL Context Parameters

The object oriented features were introduced into SDL in order to facilitate re-use and encapsulation. Instead of defining a system or process instances, a system type or block type can be designed and stored in a package. Applications can use these definitions either to define derived type definitions, or to instantiate these types in their target context. Virtual types allow one to replace a type with a different one in a redifinition of the enclosing type, and virtual transitions allow one to replace single transitions (i.e. action sequences) in the redefinition of a process type or procedure.
Unfortunately, these mechanisms are not sufficient for applications where instances of the type need to interwork closely with the environment where they are instantiated. Therefore, SDL'92 allows to parametrise a type definition with context parameters.

The following kinds of context parameters are supported in SDL'92:

- Process context parameters define a process instance set. They can be used to create new processes in the set, or send signals to the instance set (meaning that an arbitrary process in the set will receive the signal).
- Procedure context parameters allow to pass a procedure definition to a type definition (e.g. procedure or service type definition). For example, a generic procedure could apply a given procedure to each element of an array.
- Signal context parameters can be used to send and receive signals defined in the context. The context parameter can also be used to specify channels and signal routes that transport the signal.
- Variable context parameters can be passed to procedures and services, so that they can access and modify values of a variable of the context. For procedures, the same effect can be achieved by passing the variable as an actual *in/out* parameter to each call. By introducing a variable context parameter, it is specified that the same variable is accessed in each call.
- Timer context parameters can be passed to procedures so that the latter can set the timer and receive the timeout signal. This is desirable if a procedure with states should return either on reception of a signal, or reception of a timeout.
- Synonym context parameters allow the use of symbolic constants in a type definition, where the actual value of the synonym is available only in the context of the type instantiation.
- Remote variable and remote procedure context parameters allow the type to export and import a remote entity which is defined in the context of an instance.
- Sort context parameters allow to pass data type definitions to a type definition. A sort context parameter can be used almost everywhere where a sort definition can be used, including variable declarations, formal parameters to procedures, processes and signals, as well as in syntype definitions. Using the sort context parameter as a base type in sort specialization is not allowed.

Many of the context parameters come in three flavors: unconstrained, constrained by base type, and constrained by signature. For an unconstrained formal context parameter, a definition of the right entity kind can be passed as actual context parameter. If the context parameter is constrained by a base type definition (keyword *atleast*), the actual context parameter must be of a type that is a specialization of the context parameter constraint. If the context parameter is constrained by signature (keywords *fpar* and *returns* or *operators* and *literals*), the actual context parameter must support this signature. For procedure and process context parameters, this means that the formal parameter lists must match. For sort context parameters, the actual sort must provide the operators and literals defined in the constraint.

Translating Generic Types to C++

One objective of the SITE C++ generation is to allow separate compilation of packages. In other words, an SDL package should be compilable into a C++ library that can be linked with the code produced from an SDL system definition using that package. Current C++ compilers do not support code generation for C++ templates, instead, they instantiate the template for each parameter combination at compile time. This technique contradicts with the objective of separate compilation, thus translating generic types to C++ templates is not acceptable. In addition, some context parameters, especially process instance sets, are not adequately translatable to C++ templates.
Instead, actual context parameters are represented as instances of C++ classes throughout the SDL runtime system. For most parameter kinds, the usage of these instances in the generic type is straight forward and similar to the usage of those entities in non-context-parameter

statements. The differences are discussed below. Generally speaking, if the interface required from the context parameter is provided by the virtual SDL machine interface, the same interface is offered by the objects representing acontext parameter. For sort context parameters, the interface is specific to the sort, so a different mechanism must be used.

Implementing Genericity as Specialization

In the case the interface needed inside the type definition is available in the SDL runtime library, implementation of context parameters is straight forward. An example showing the usage of process context parameters should illustrate that approach. First, consider the following specification snippet which does not use context parameters.

```
signal p1_created(PId);
process p1; fpar i integer;
    start;
    stop;
endprocess;
process p2 referenced;

procedure proc;
    start;
    create p1(10);
    output p1_created(offspring) to p2;
    return;
endprocedure;
```

This defines a procedure which first creates a process in set *p1*, then sends a signal to *p2* notifying about the process creation. For the process creation, the following code is generated

```
P_p1_SetPtr ->
        create (my_process(), new arglist(1,SDLInt(10).addr()));
```

First, the instance set pointer is accessed. Then a method *create* is invoked on the instance set. This method expects the parent process and the argument list. The argument list is represented in an *arglist* object, which carries a variable number of arguments. The class *arglist* was introduced in the SDL maschine interface primarily to support inheritance and genericity.
The output statement is translated into

```
SDLSignal *sig = new SIG_p1_created;
sig->arguments()->put(1,SDLOffspring().addr());
output(sig, MY_B_y->P_p2_SetPtr);
```

This creates a new signal instance, sets the arguments, and calls the procedure's output function. The *put* method again belongs to an *arglist* object.
Now, if context parameters were involved, the procedure could be defined as

```
procedure proc <process cp1 atleast pt1;
    process cp2;signal cp1_created(PId)>;
    start;
    create cp1(10);
    output cp1_created(offspring) to cp2;
    return;
endprocedure;
```

The *create* action is now translated to

```
P_cpl->create (my_process(), new arglist(1,SDLInt(10).addr()));
```

and the output to

```
    SDLSignal *sig = SIG_cpl_created->copy();
    sig->arguments()->put(1,SDLOffspring().addr());
    output(sig, P_cp2);
```

In the create statement, the context parameter is accessed using the *P_cpl* member of the procedure. This points to the instance set that was passed as actual context parameter when creating the procedure. Because *P_cpl* is known to be of at least *Instance_set** (in C++), the call of *create* is possible. Since the actual parameters of the process are encapsulated in a special object, the same signature for process creation can be used for all process definitions. This allows to define the *create* function as virtual in the SDL runtime library. A similar discussion applies to the output function; this function expects an *SDLSignal** and an *Instance_set**. Since the context parameters are known to be derived from these classes, the call to the library function output is possible.
For the signal context parameter, things are somewhat different: It is necessary to create a new instance of the signal. However, the *new* operator cannot be used since the class name of the signal is not known when compiling the procedure. Instead, the *SIG_cpl_created* member carries an instance of the actual signal context parameter. Every derived *SDLSignal* is required to implement a *copy* function which returns the new signal. Using the *arglist* interface of this signal, it is possible to insert the parameters into the signal.
This approach can be extended to most other context parameter kinds. Procedures have a methods to create new instances, as well as to invoke an instance. Timers have methods to set and reset them, as well as to identify a given signal as timeout signal of the timer. Variables of well-known sorts have methods that are well-known to the SDL compiler. Synonyms behave like read-only variables. The remaining issue is the implementation of sort context parameters. Especially difficult is the implementation of sort context parameters with signature constraints, which is discussed below.

Sort Context parameters

The following example will help to identify the problems with implementing sort context parameters. First, a non-context-parameter example:

```
procedure proc;
    fpar i integer;
    returns integer;
    start;
    task i:=i+i;
    return i;
endprocedure;
```

This gives the following declaration for the variable:

```
class PROC_proc /* ... */ { /* ... */
SDLInt& VAR_i() {
    if(!_VAR_i)_VAR_i = new SDLInt();
    return *_VAR_i;
}
}
```

and the expressions are compiled into

```
VAR_i() = VAR_i().add(VAR_i());
VAR_PROC_Result()=VAR_i();
```

Each variable is represented by a dynamically allocated instance of the C++ class corresponding to the type of the variable. This class supplies all the operators that are defined for the type, as well as the literals of the sort. In the example, only the "+" operation is used, which is compiled into the *add* method.
Using sort context parameters, this specification could be extended to

```
procedure proc <newtype like_int
    operators "+": like_int,like_int -> like_int;
    endnewtype>;
    fpar i like_int;
    returns like_int;
    start;
    task i:=i+i;
    return i;
endprocedure;
```

This is a generic routine that ‚doubles‘ its argument, which can be of any type that supports "+". Among the predefined types of SDL, this includes Integer, Real, and Duration. In the discussion below, this example goes on by passing *Integer* as the actual context parameter.
When deciding how to implement sort context parameters, it is desirable that the expressions involving context parameters are compiled into similar code as the ones without context parameters. In particular, the expressions should still compile as

```
VAR_i() = VAR_i().add(VAR_i());
VAR_PROC_Result()=VAR_i();
```

In order to support this interface, *VAR_i* must return a reference to a class that implements an assignment operator and an *add* method. Following the approach taken for variables of non-context sort, this method is implemented as

```
class PROC_proc /*...*/ { /* ... */
TYPE_like_int& VAR_i(){
  if(!_VAR_i) _VAR_i = FCX_like_int->copy();
  return *_VAR_i;
}
}
```

Instead of initializing the variable by calling the operator *new* for a type, a virtual copy function of the actual context parameter is called to get a new instance.
In order to supply the signature of the context parameter, the class *TYPE_like_int* already

needs to support an *add* method. However, the implementation of the actual context parameter (e.g. *SDLInt*) cannot be called, since the actual context parameter is not known when compiling *proc*. In addition, *TYPE_like_int* cannot serve as a base class for SDLInt, because the base class of SDLInt is already defined elsewhere. Since inheritance and virtuality cannot be used directly, delegation is employed.
Delegation means that the class implementing the actual context parameter is not *SDLInt*, but some wrapper around *SDLInt*. This wrapper needs to implement the *add* method by delegating it to the *SDLInt*. Because *add* creates a new value, this wrapper can be implemented as

```
struct formal_wrapper{ //base class for all wrappers
    virtual formal_wrapper* copy()=0;
    virtual formal_wrapper* add(formal_wrapper* other)=0;
};
struct actual_wrapper:formal_wrapper{ //wrapper around SDLInt
    SDLInt value;
    actual_wrapper(const SDLInt& v):value(v){}
    formal_wrapper *copy(){return new actual_wrapper(value);}
    formal_wrapper *add(formal_wrapper *o){
        //this invokes SDLInt::add
        return new actual_wrapper(value.add(o->value));
    }
};
```

Because of the requirement to create new objects, and because the layout of these objects is not known in the base class, this wrapper class returns pointers to objects instead of references. Since the interface for all other values in the SDL runtime assumes references and automatic memory management, the actual context parameter is another wrapper around the *formal_wrapper*.

```
class TYPE_like_int{
    formal_wrapper *value;
    TYPE_like_int(formal_wrapper *v):value(v){}
    TYPE_like_int(const TYPE_like_int& o):
        value(o.value->copy()){}
    ~TYPE_like_int(){delete value;}
    TYPE_like_int* copy(){
        return new TYPE_like_int(*this);
    }
    TYPE_like_int& add(TYPE_like_int& other){
        return value->add(other.value);
    }
};
```

Using this approach, it is possible to implement all operators found in sort context parameter constraints. In order to implement literals as well, additional protocol has to be introduced, if the constraint includes an infinite number of literals, i.e. a literal pattern. In this case, the association between literal name and literal value must be communicated via character strings.

Summary

Generic types in SDL providea mechanism to support encapsulation and re-use of types. Compiling generic types to C++ is mostly straight forward, using abstract interfaces and factories.

In order to support operations that are not included in the abstract interface, a bridge pattern must be used. One wrapper class is needed to delegate the functionality, another deals with memory management. For those context parameters that use the interface as implemented in the library, execution speed is identical compared to not using context parameters. For sort context parameters, the amount of method calls for evaluating an expression increases by a factor of three due to the delegation.

Literature

[Z.100] CCITT: SDL - Specification Description Language, International Standard Recommendation Z.100, Genf,1992

[Sch95] Ralf Schröder, SDL Integrated Tool Environment, http://www.informatik.hu-berlin.de:80/Themen/SITE/

THE HERMENEUTIC NATURE OF AN ECODESIGN MODEL AND SELF

Albertina Lourenci
Al@sc.usp.br; Lourenci@lsi.usp.br
Faculty of Architecture and Urbanism- University of São Paulo- Rua Maranhão,88 - Higienópolis - São Paulo -SP- CEP: 01240-000

1 Transcendent and immanent aspects must be interwoven

Either software developers lament "If only software engineering could be more like X..." where X is any intensive-design profession says Dough Lea or experts in computer science spend lots of time discussing the pros and cons of different paradigms in computer science. They pay little attention to the intrinsic nature of the problem to be solved and even less to the global problems mankind has to face urgently.

Thus I introduce a problem and its intrinsic nature straightforwardly: the need to generate architectural/structural/landscape design within an urban ecosystem context enabling the design and planning of sustainable cities. There are fundamental processes underlying what I would call an ecodesign modeling. Basically what happens in a natural ecosystem also reflects in the urban ecosystem because we are embedded in the former. The green designer's goal is to create an environment to improve the physical, mental, psychical health of the human beings. Scientists have been showing certain environments help man to keep his cellular oscillations. R.S. Ulrich compared data of recovery for pairs of patients submitted to surgery who are expected to experience considerable stress. To carry out the experiment, pairs of patients with the same surgery, sex, weight, age, tobacco use and previous hospitalization were booked in identic rooms with exception of the view through the window. A member of each pair looked into a grove, while the other looked to a wall of brown bricks. Individuals with view for trees needed only aspirins, recovered sooner and left the hospital earlier. The wall patients received more analgesics, narcotics, complained quite a lot and spent more days at the hospital.

I am going to trace back the implicit origins of the latter behaviour. In archaic Greece, the notion *nomos-physis* (law-nature) formed a unity. In the feelings of the ancient Greeks, *nomos* is primarily the distributive justice from which nobody could escape. Each one had access to a site during his/her existence without the need for written laws. With the advent of currency and the shift from oral communication to writing, a new political space and time emerged accompanied with the dissolution of the *basileus* - central figure of the archaic communitariam power. The logic of *polis* prevailed and hence the dissociation from cosmos.

This opened the gate to the Platonic conception of cosmos creation by a Demiurge whose creation is *anthropoi*, human and male, making us view earth as a passive container or receptacle.The separation fro *physis* and the cosmos as well as the women segregation has led to an entropic behaviour towards the site and the planet.

Although this has been the mainstream trend, Aristotle's ideas have loomed on the horizon tuning with the scientific ideas of self-organization and pointing forward the environmental degradation even at Greek times. For him, physical research must deal with conditions and characteristics of physical objects without contrasting them with the properties of things eternal. This caused departmentalization but is more than a method. It carries with it a certain autonomy for the subject thus treated.

Summarizing, while Plato cares for transcendent reasons, Aristotle cares for immanent reasons. Our mandatory task is to reconcile both trends in the sense that time is ripe for attempting to incorporate at least embryonically both transcendent and immanent aspects into our approaches.

2 Homeostasis is common to the planet and to the human being

The question is how to introject such an encompassing vision into an ecodesign modeling? First we have to try to discern the fundamental dimensions embedded in the modeling.

James Lovelock in his theory of Gaia unravels the Earth's nature: *...the Earth's living matter, air, oceans and land surface form a complex system which can be seen as a single organism and which has the capacity to keep our planet a fit place to life. Gaia is a cybernetic or feedback system which seeks an optimal physical and chemical environment for life on this planet [Joer92].*

Likewise the ergonomist or the human factor engineer is well aware that the general thermal state of the body both in comfort and in heat or cold stress is dependent on an analysis of the heat balance for the human body:

Moreover the triad of temperature, potential evapotranspiration ratio and average total annual precipitation in mm determines a range of natural life zones or ecosystems in terms of world plant formations and their associated local biodiversity.

There are two fundamental processes associated to life: one "order from order"and the other "order from disorder". Hence information and energy are the two ingredients of life. Order from order is the information stored in the DNA molecule and responsible for the generation of myriad of life forms. Paradoxically, the more

we try to keep order, the more energy is necessary and the more stress (entropy) is thrown into the environment, because all energy transformations from one form into the other entail waste heat production due to the second law of thermodynamics.

However the recently discovered fourth law of thermodynamics states: *If a system has a through-flow of exergy (it is a measure of the potential of energy to perform useful work), it will attempt to utilize the flow to increase its exergy, i.e., to move farther away from thermodynamic equilibrium; if more combinations and processes are offered to utilize the exergy flow, the organization that is able to give the system the highest exergy under the prevailing conditions and perturbations will be selected* [Joer92].

I want to convey the idea the whole is much more than the simple sum of its part despite the second law of thermodynamics if we manage to create a link between the local and the global keeping the homeostasis of the organisms and of the planet (the great organism).

3 Formalist computer science

Ignoring these concerns, Mitchell, Stiny, Yoshikawa, Coyne [SC90] tried to advance software in design and CAD proposing the atomic model language. Mitchell says design can be described by words that shape a language and such descriptions of words can be formalized using the first order predicate calculus notation. Here design is an epistemological event. Its knowledge and application are separate and sequential. Knowledge precedes its application. The responses to the questions that happened in the situation are known previously. They do not change to adapt to peculiar demands. Theory precedes practice says Snodgrass and Coyne.

Likewise, the formalism of the shape grammars correspond directly to geometry; even nongeometric representations are mapped onto one or more geometric realizations. Shape grammars are applied to generate Palladian villas, Frank Lloyd Wright's prairie houses, Queen Anne houses, Japanese tea houses and other designs [Heis94].

Rosenman and Gero from the Key Centre of Design Computing, Department of Architectural and Design Sciences from University of Sydney generate architectural and structural design cooperatively. Inspired by Bobrow about the function, behaviour, purpose and structure of a watch, they develop a model integrating the architect, mechanical engineer and structural engineer's views. Their integration means juxtaposition of visions in the style of arranging blocks and decorating façades. They implement certain abstractions in the C programming language, utilizing the class concept in an arbitrary way. The modeling of the multiple views is implemented in CAD [RG96].

Richard Coyne, the only architect of the research team in their center after realizing the failure of his logic models to generate design reacts and recognizes the hermeneutic nature of design.

4 Hermeneutic computer science

Although the formalist computer science is still the mainstream, there is a growing opposing trend in the artificial intelligence meetings and more recently in the object oriented paradigm through the introduction of design patterns and prototype-based object oriented languages. The formalist paradigm in philosophy and science orbits around centralization, control, hierarchy, predictability and 'provability'. Its ideal model follows the atomic model language. Its visions are grounded in the philosophies of Descartes, Hobbes, Leibniz, Russell and Whitehead. Babbage, Turing and von Neumann illustrate the formalist philosophy in computer science. Newell, Simon and Minsky , in artificial intelligence.

Bo Dahlbom and Lars Mathiassen [DM97] adds that *the romantic view grew out of a reaction against mechanistic thinking and was formulated towards the end of the 18th century, primarily by German philosopher-artists like Herder and Hegel (.....) Where the mechanists saw structures and systems, the romantics saw processes and change. Likewise followers of the hermeneutic thinking emphasize autonomy, multiple perspective, self-organization, change, evolution, interpretation, malleability and flexibility. They argue that any formal syntax fails in grasping the intrinsic properties of the natural world [West97]. Husserl, Heidegger, Gadamer, Vygotsky and Foucault represent this trend in philosophy as well as the contemporaries Maturana, Varela, Prigogine and Gell-Mann. In computer science, it reflects in the viewpoints of Floyd and colleagues, Coyne, Dreyfus brothers and the post-artificial intelligence works from Winograd.*

In the sixties, this trend was represented by the LISP exploratory programming; in the seventies, it moved to Prolog and in the eighties, to Smalltalk. However, the object oriented paradigm behaves rather like a Janus head. Hermeneutization begins to be introduced in the formalist world in parallel to object-oriented methods not hermeneutic [Aksi97]. And the confusion arises.

I believe hermeneutics has the nature of the rainbow, its colour gamut varies from infrared to ultraviolet and appears in the sky after the rain due to the diffraction of light. To try to mimic its composition, first dealing with the primary colours and then the secondary colours orange, green an purple will not lead to its circular shape ranging from red to ultraviolet and even less ignoring the need of interaction with

light and water. An hermeneutic method has necessarily a holistic nature. It leads to diversity in unity.

5 The Model of Primary, Secondary and Tertiary Waves (MPSTW)

The design and planning within an ecological context is essentially an interpretative activity. It is an activity that belongs to the global understanding of a situation that must be aprehended while discrete entities defined by the situation and able to define the situation. It evolves in time and space. To keep pace with Gaia and shelter human life soundly, an urban ecosystem must care about the 'earth'from where comes everything (materials) and its surrounding environment that receives the incoming energy from the sun and the design of human behaviour. This it achieves by designing the environment in which behaviour occurs encompassing all environments form domestic, industrial to agricultural, medical, military, etc modeling basically the ergonomic and geotechnical dimensions. At the microlevel, the ergonomic dimension includes human-artifact interface technology or hardware ergonomics; human - environment interface technology or environmental ergonomics and user-system interface technology or software ergonomics and at the macrolevel, there is organization-artifact technology or macroergonomics. It is fundamental here that input comes from the layout of the plans pervaded by the needs of the human beings in interaction with the environment.

The geotechnical dimension guides the designer to receive the geological information given by first order observational maps (topographic, bedrock geology, geologic, tectonic and structure, agricultural soils, ground water, surface drainage); second-order (engineering) maps (unconfined compressive strength, rock quality, slope stability, excavation difficulty, infiltration capacity, corrosivity, soil quality, engineering soils classification, engineering resources) and third-order (interpretive) maps (geologic hazards, home site suitability, heavy construction suitability, subsurface installations suitability, resource suitability and waste disposal suitability) in terms of use suitability and not in terms of geologic processes or descriptions and thus information must be such that can be applied directly to sustainable policies of soil use management given by fourth-order (planning) maps (engineering geology recommended land use map).

The MPSTW (figure 1) derived from the application of catastrophe, semiotics and graph theories deals gracefully with these dimensions. The primary waves work like a genotype dealing with the processes of interaction of the elements of the architectonic object and the environment through the hypotheses of homeostasis, continuity, differentiation and repeatability. The design processes are dealt by the

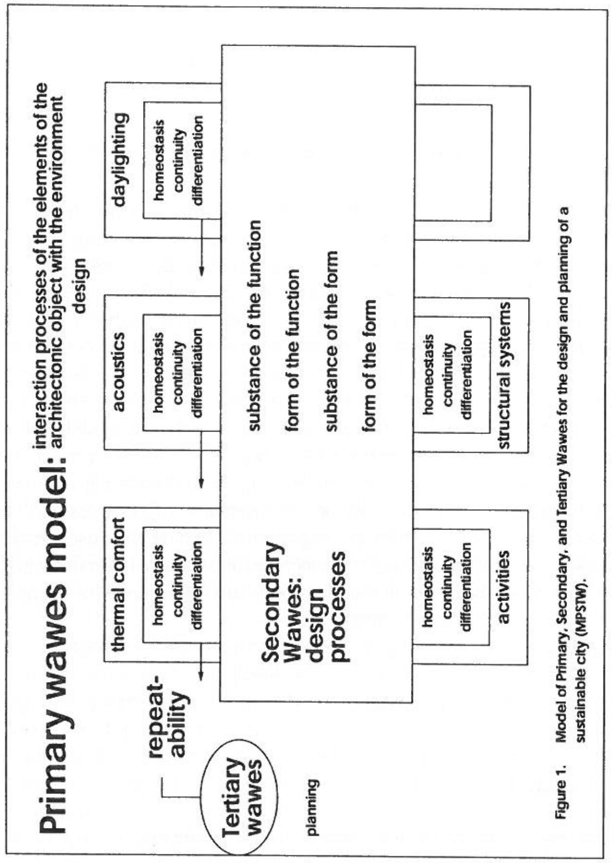

Primary wawes model:
interaction processes of the elements of the architectonic object with the environment design
thermal comfort
homeostasis continuity differentiation
acoustics
homeostasis continuity differentiation
daylighting
homeostasis continuity differentiation
repeat-ability
Tertiary wawes
planning
Secondary Wawes: design processes
substance of the function
form of the function
substance of the form
form of the form
homeostasis continuity differentiation
activities
homeostasis continuity differentiation
structural systems

Figure 1. Model of Primary, Secondary, and Tertiary Wawes for the design and planning of a sustainable city (MPSTW).

secondary waves and act like phenotypes and involve a geometric intuition that creates a specific geometry to model the urban morphogenesis through tilings, discrete groups of the plane and fractals. To develop this submodel I demonstrate taht architectural design may be considered a language with its planes function and form and their stratas substance of the function, form of the function, substance of the form and form of the form. All these processes are applied to the elements of the architectural design, namely, environmental comfort (thermal comfort, acoustics, daylighting), activities, structural systems, hydraulic

installations, etc. Its outcome is the eco-system with hyphen or the architectonic artifact. It has an autopoietic nature insofar as it grows from the smallest artifact (the building and support facilities) passing by the neighborhood and boroughs to the sustainable city and the bioregion. A finer granularity unravels through the subeco-system with hyphen or the sum of the processes applied to each element. They work out like the figures of a language (a sign is composed of figures). This partaking of the urban phenomenon especially through the hypothesis repeatability causes the emergence of the tertiary waves or the blending of design and planning. Or the eco-system with hyphen defines the the urban ecosystem and is defined by it. Moreover it allows the designer and the user to start modeling by any of the processes adding flexibility and malleability to the design and planning enhancing participation of the citizen.

6 Self

Hence the MPSTW falls within an hermeneutic trend. Indeed, its elements and processes behave themselves with dynamism and flexibility. They are not variations around a theme. Classes are useful when multiple instances of similar objects pervade the problem space. Hence sharing attributes among the objects as well as programming exploratorily is fundamental. The independence of each element or process suggests an object. Each object accepts or delegates tasks to the other. Each element and each process is unique. There is no need for classes No clear taxonomy for tasks is defined, hence little need for inheritance [Grog97]. Moreover, MPSTW is built for cooperative work among designers and citizens covering total synergetic interaction among its members.

Martin Abadi and Luca Cardelli [CA97] insist on that everything can be better represented in terms of objects, even functions and classes. The basic constructions are simpler, flexible and powerful. Hence naturally a prototype ased object oriented language tunes with the hermeneutic nature of design enhancing it Wegner insists on that objects, classes and inheritance are not orthogonal. Classes are defined in terms of objects, inheritance in terms of classes. The essence of a class can be defined independently of the object and the essence of inheritance independently of classes and objects [Wegn87].

Inheritance as a mechanism to share resources defined incrementally internalizing shared resources treat the latter as part of an extended self (identity). Hence the definition of inheritance in terms of a particular mechanism of self-reference enabling the internalization of remotely defined operations as part of the extended identity of the object is called delegation.

The delegation based languages allow the objects to share and internalize operations from ancestral objects called prototypes, that work as instances and templates for the descendents. Hence the prototypical languages are languages based on delegation that carry out delegation by prototypes. And the chosen language to implement MPSTW is Self, a prototype based object oriented language.

Ungar and Smith [SU95] emphasize the concreteness of the prototypes because they are examples of objects instead of format descriptions and initializations.

The shared behaviour by a family of objects is hold by a separate object that is the father of all the objects, even of the prototype. This way, the prototype is absolutely equal to any other member of the family. The object that contains the shared behaviour plays the role of a class, except that it only contains the shared behaviour without format information.These parent-objects are called traits-objects.

Self contains graphical objects called morphs that behave exactly like the object. Since MPSTW also has a graphical nature this feature is very relevant.

Acknowledgements

Joergen Lindskov Knudsen played a great role in paving the way for my firm introduction to hermeneutics. Indeed, the pattern construct in Beta and the Mjoelner environment put you nearer hermeneutic computer science than the other OO languages. Heartfelt thanks to the Beta Group. Erik Ernst and Paulo Ceneviva (EESC-USP) helped me an awful lot in the final version of this paper.

7. References

[Aksi97] Aksit, N.: Active software artifacts. Workshop Modeling Software Processes and Artifacts. *ECOOP'97.*

[Card97] Cardelli, L. and Abadi, M.: *A theory of objects.* Tutorial ECOOP'97.

[DM97] Dahlbom, B. and Mathiassen, L.: The future of our profession.*CACM* Vol.40, No.6, June 1997, 80-89

[Grog97] Grogono, P.: Messy solutions for messy problems. Paper position for Second Workshop on prototype-based object oriented programming . *ECOOP'97.* Finland. Jyväskylä.

[Heis94] Heisserman, J.: Generative geometric design. *IEEE Computer graphics and applications.* March 1994

[Joer92] Joergensen, S.E.: *Integration of ecosystem theories: a pattern.* Kluwer Academic Publishers. 1992

[RG96] Rosenman, M.A. and Gero, J.S.: Modelling multiple views of design objects in a collaborative CAD environment. *Computer Aided Design*, Vol.28, No.3

[Snod90] Snodgrass, A. and Coyne, R.: Is designing hermeneutical? *Working paper.* Faculty of Architecture and Urbanism. University of Sydney. 1990

[SU95] Smith, R.B. and Ungar, D.: Programming as an experience: the inspiration for Self. ECOOP'95. *LNCS 952*

[Wegn87] Wegner, P.: Dimensions of object-based language design. *OOPSLA'87.*

[West97] West, D.: Hermeneutic computer science. *CACM.* April 1997. Vol 40, N.4.

Frameworks — Representations & Perspectives

Position Paper, ECOOP'97

Palle Nowack

Department of Computer Science, Aalborg University

Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, Denmark

E-mail: nowack@cs.auc.dk

May 12, 1997

Abstract

In order to enhance the language support for the development and use of object-oriented frameworks, we propose to elaborate on the conceptual understanding of frameworks, especially regarding architectural issues. For this purpose we suggest the use of different perspectives on frameworks. The different perspectives supports different needs in a framework's lifecycle, and they should be used as inspiration for developing alternative representations of frameworks, e.g. languages. In particular we propose the idea of abstract frameworks and framework components.

1 Introduction

Within software engineering in general, *reuse* is considered to be a part of an effective development process. This effectiveness origins from the qualitative and economical benefits reuse accomplishes. Using previously developed and tested components when building an application saves development efforts and reduces the risk of introducing errors into the system.

Object-oriented software development furthermore benefits from the use of *abstraction*: A composite object encapsulates various parts, an abstract class encapsulates common properties (attributes and methods) of several classes. Abstractions and abstraction mechanisms make software development easier, because they let the developer work with fewer elements—abstracting from details. This also makes it easier to communicate about software. The goal of providing programming languages with better abstractions and modeling constructs has been pursued within *conceptual programming* (Examples include roles, complex associations, activities. See for example [Kristensen, 1994]).

The *software architecture* [Garlan and Shaw, 1996] level of design is concerned with the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In [Buschmann et al., 1996] software architecture is defined as: "A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and -non-functional properties of a software system."

In [Kristensen, 1996] the notion of *architectural abstraction* are discussed. *Design patterns* [Gamma et al., 1995] and *frameworks* [Johnson and Foote, 1988] are described as being two different examples of categories of architectural abstractions. The categories are characterized differently in the universe of architectural abstractions according to a set of dimensions, including Level of abstraction, Degree of domain specificness, Level of granularity, and Degree of completeness. In the same paper it is claimed that language mechanisms and architectural abstractions mutually influence each other; the available language mechanisms provide the means for expressing and implementing the architectural abstractions, and the architectural abstractions provides inspiration for the development of new language mechanisms.

In [Jacobsen and Nowack, 1997] it is argued that an architectural abstraction provides functionality, logic structure, abstraction, and reusability. Architectural abstractions are abstractions over *software* entities (design and implementations), and not necessarily domain concepts. If the latter is also the case we have a modelling situation, where concepts in the application domain is modelled by concepts in the software domain. Many types of patterns and frameworks adheres to the notion of architectural abstractions. Frameworks have the benefit of supporting the reuse of analysis, design, and implementation [Johnson and Russo, 1991], but they are a complex type of software, hard to design, maintain, and use. Design patterns on the other hand are not nearly as complex, they are smaller in scope, and they provide very useful abstractions when communicating about software. Patterns however, due to their nature, do not support the reuse of actual code, and the process of instantiating a pattern in a given context is non-trivial. Moreover many patterns do not deal with architectural issues, but rather various (sometimes language dependent) programming tricks.

Our Goal To manage the complexity of object-oriented software designs and implementations we propose to use architectural abstractions as means for decomposing complex software systems into more conceivable units. Frameworks are architectural abstractions, as they provide reusable abstract designs for specific domains. Furthermore different notions of patterns can be used as architectural abstractions, providing different perspectives on the design. However, both frameworks and patterns contain a lot more information than the architectural abstraction: code, documentation, diagrams, prose etc. We are interested in refining the architectural aspects of frameworks, and for this end, we propose the use of different representations of and perspectives on object-oriented frameworks.

2 Perspectives

Patterns as Perspectives

In general different notions of patterns provide different perspectives on software [Jacobsen and Nowack, 1997]. With regards to frameworks, the different notions of patterns support different stages in the development process [Jacobsen et al., 1997], as well as the framework application process. Some object-oriented patterns [Coad, 1992] facilitate the modelling of domains, design patterns [Gamma et al., 1995] provide abstractions to discuss designs, and meta patterns [Pree, 1995] provide framework users with a hot-spot perspective, indicating where and how to adapt a framework. Furthermore pattern languages used as framework documentation, such as the one proposed by [Johnson, 1992], provide an application developer with a perspective on the framework to be used (the product), as well as a perspective on how to use it (the process).

Software Architecture Level

All of the above perspectives are based the same representation: the objects and classes of the framework, and the individual types of patterns are a unit in the perspectives. Others are possible. For example the software architecture level of system design suggest the use of components and connectors as basic units [Garlan and Shaw, 1996]. Components and connectors are used to describe the high-level composition of systems independent of the individual components' and connectors' representations, which can be formal specifications as well as implementations.

We believe that a software architecture perspective on frameworks would be very useful, as it would make the architectural guidance provided by a framework explicit. We believe that the architectural constraints embedded in a framework, is its primary contribution and benefit (as opposed to reuse of analysis (domain concepts) and implementation (code reuse for instance by providing black box components in an accompanying class library). The architectural constraints are the hardest part to get right (requires iteration and/or experience), it should be fixed (it is the backbone of the application), and hence you will have to live it for a long time.

A problem with conventional frameworks is that the architectural constraints are not very visible, they are not first class entities. The rules and patterns of collaborations that objects must follow in order to adhere to the framework architecture is not made explicit. At best it is hidden in the methods of the abstract classes of the frameworks (for example a template method), but often is simply described in the framework documentation (for example as patterns).

One reason for this is that the medium for describing the architecture is the same as the medium for describing the actual code, namely an object-oriented programming language, which is basically designed for specifications of data structures and algorithms, and not for the high-level composition of software.

3 Representation

A framework is currently always expressed in an object-oriented programming language. Based on the different perspectives on frameworks, as the ones described in the previous section, alternative representations should be examined. It is useful to distinguish between different types of representations in framework development, evolution, and usage.

Levels of Abstraction and Genericness

Typically a framework is expressed in a specific object-oriented programming language. The application of the framework is also conducted in the programming language, and the resulting application can be compiled and executed. We term this type of framework a *conventional* (or concrete) framework.

A framework can also be expressed in a generic object-oriented programming language. Such a generic language is a suitable common extract of similar conventional existing object-oriented programming languages with a suitable syntax and semantics. By a straightforward translation it should be possible to translate a *generic* framework to a concrete framework. The purpose of a generic framework is to make the description of a concrete framework independent of a specific language and its corresponding environment.

Furthermore it should be possible to express a framework in an abstract notation, resulting in an *abstract* framework. This again would require a suitable notation. We believe that such a notation should be based on the object-oriented paradigm, and include aspects of conceptual programming and the software architecture level of design, see description of framework components below. An abstract framework is not complete as it can't be adapted, translated, and executed, because it lacks sufficient details.

Framework Components

A major part of the complexity of frameworks is caused by the lack of intermediate abstractions. This is part of the reason why patterns have been successfully applied in framework development, documentation, and use, as they provide this abstraction level. However, patterns are not made explicit in frameworks, and in order to benefit from the patterns used in in a framework, one must rely on the framework documentation to describe the applied patterns.

We want the possibility of representing an abstract framework by a collection of associated *framework components*.

According to the processes involved with framework-centered software development, and the different possible perspectives on frameworks we identify the following characteristics that a framework component should possess:

Characteristics A framework component is an architectural abstraction. That is, it can be used to describe the architecture of a framework by providing architectural rules and constraints that the framework implementations (classes, methods) must adhere to.

- Framework components can be specialized and composed like conventional classes and objects.
- Framework components can be parameterized and instantiated.
- Framework components support both framework adaption as well as framework evolution. Possibly by providing different interfaces.
- Framework components do not contain or specify methods or other imperative constructs. Instead they specify rules constraining the methods supplied by the framework.

Concrete & Abstract Parts

One approach to get a better understanding of framework components would be to extrapolate on the notion of abstract classes. In the following we characterize 'concrete' and 'abstract' object-oriented system parts.

Applications & Frameworks An object-oriented application is primary build out of concrete parts, although a mature and well-structured application also contains abstract parts.

An object-oriented framework consists of a set of abstract classes facilitating the reuse of architectural designs. Furthermore a framework typically contains concrete classes, describing possible ways of fleshing out the superstructure imposed by the framework's abstract design. The latter typically consists of black box components organized in a library. This is reuse of implementation.

We believe that the framework's abstract design is build from general abstract parts, and the framework's concrete implementations is build from general concrete parts.

Concrete Parts A concrete object-oriented system part is build out of objects & classes. This includes the use of instance variables, references, operations (terminology somewhat fuzzy), inheritance, nesting ; in general a very limited selection of language constructs. But the selection is well-known and supported by most object-oriented programming languages.

Abstract Parts An abstract part is build from abstract classes, virtual (deferred) methods, template methods, abstract coupling. It is important to note that all but virtual methods are conventions, –not supported by language constructs.

An abstract part is used to describe a partial generic software domain. Furthermore abstract parts are used as templates for concrete parts. This implies that they must posses properties supporting this.

Future work

Based on the different possible perspectives on frameworks, we need to decide what kind of architectural information a framework component should represent, how to represent it, and how to combine different framework components into a complete abstract framework.

References

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

[Coad, 1992] Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9).

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Garlan and Shaw, 1996] Garlan, D. and Shaw, M. (1996). *Software Architecture (Perspectives on an Emerging Discipline)*. Prentice Hall.

[Jacobsen et al., 1997] Jacobsen, E. E., Kristensen, B. B., and Nowack, P. (1997). Patterns in the analysis, design, and implementation of frameworks. In *Proceedings of the Twenty-First Annual International Computer Software and Application Conference, Washington, USA (COMPSAC'97) (to appear)*.

[Jacobsen and Nowack, 1997] Jacobsen, E. E. and Nowack, P. (1997). Frameworks & patterns: Architectural abstractions. *Submitted.*

[Johnson, 1992] Johnson, R. E. (1992). Documenting frameworks using patterns. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*.

[Johnson and Foote, 1988] Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 2(1).

[Johnson and Russo, 1991] Johnson, R. E. and Russo, V. F. (1991). Reusing object-oriented design. Technical Report UIUCDCS 91-1696, University of Illinois.

[Kristensen, 1994] Kristensen, B. B. (1994). Complex associations: Abstractions in object-oriented modelling. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*.

[Kristensen, 1996] Kristensen, B. B. (1996). Architectural abstractions and language mechanisms. In *Proceedings of Asia Pacific Software Engineering Conference (AISEC'96)*.

[Pree, 1995] Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Addison-Wesley.

Splitting Synchronization from Algorithmic Behaviour

An Event Model for Synchronizing Concurrent Classes

Stephan Reitzner

University of Erlangen-Nürnberg
Department of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
http://www4.informatik.uni-erlangen.de/~reitzner/
reitzner@informatik.uni-erlangen.de

Abstract

Synchronization of concurrent activities is a major issue of concurrent object-oriented programming languages, as we deal with fine grained synchronization involving objects and methods. In this paper I propose the separation of the synchronization code from the algorithmic code of a concurrent object, arriving at two objects: one that implements the algorithmic behavior, and the other one that is responsible for controlling concurrency. I introduce an event model to carry out the link between the two objects. I will also present a new way of deriving from classes with concurrency control. The inheritance step is split into two steps: one for the sequential behaviour and one for the concurrency control. This new mechanism together with my event model solves many problems of the well known inheritance anomalies.

1 Introduction

Parallel architectures force programmers to use new programming models. It is obvious that coarse grained concurrency on a process level is not enough: multithreaded applications are needed. For object-oriented programming this means that objects can be accessed concurrently by multiple threads of control. The language must provide means of protection for objects to prevent inconsistent object states in the case of concurrent access. This protection is called *concurrency control*, the implementation of which is called *synchronization code*.

In the following I will present a mechanism to model the concurrency control in object oriented languages. This model completely separates the synchronization code from the object and in turn encapsulates it in a so called synchronization object. The link between these two independent objects is established by an event model introduced in Section 3. One major problem with concurrency control in object-oriented languages is inheritance. Section 4 will show how inheritance works with my separation model. The problem of the so called "inheritance anomaly" [MaYo93][1] is addressed in that Section, too. Section 5 briefly describes the current and future work.

1. Matsuoka and Yonezawa evaluated the discrepancy between reuse of a class by subclassing and the annotation of classes for concurrency control. They showed that deriving from a synchronized class can lead to a reimplementation of algorithmic behaviour only because of incompatibilities of the concurrency control. They called this discrepancy the ***inheritance anomaly***.

2 Synchronization Objects

A common way to introduce concurrency control to a language is to separate the synchronization description from the sequential algorithm [Hal94]. The synchronization is not mixed into the code of the methods, such that the reusability of a class, for example by inheritance, is improved. The problem with most of these implementations is that the synchronization constraints are still too closely tied to the methods. This arises from the fact that the binding between methods and synchronization constraints is too fixed. It is not possible for subclasses to break these bindings in order to extend or modify the synchronization constraints of a base class. An example of this sort of synchronization is the *Synchronizing Actions* by Neusius [Neu91]. It turned out that such synchronization mechanisms tend to increase the number of anomalies that occur when deriving from a synchronized class.

The basic idea of my synchronization mechanism is to split the state of the concurrent object into a sequential and a concurrent part, as shown at position ① in Figure 1. The sequential part contains the actual state and the methods of the object that implement the algorithmic behaviour; the concurrent part contains the state and the behaviour necessary for the synchronization of the concurrent object. This leads to a complete separation of the sequential part from the synchronization part. These two independent parts are in turn modelled as classes of their own (position ② in Figure 1): the *sequential class* and the *synchronization class*. The logical union of the two classes is the *concurrent class*. Instances of these classes are called *sequential object, synchronization object,* and *concurrent object.* The synchronization object is responsible for controlling concurrency in the sequential object. The binding between a sequential class and a synchronization class is called the *synchronization relation,* which is realized by the event model. It is the most critical part of the design of an object oriented synchronization mechanism, and therefore I will focus on that in the next sections.

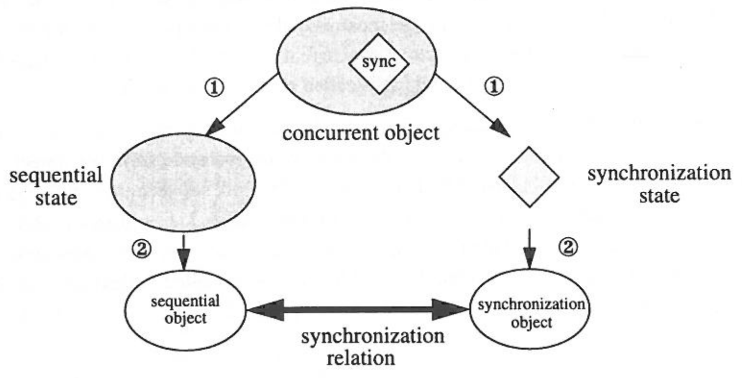


Figure 1: Separating the sequential state and the synchronization state of an object.

3 The Event Model

The synchronization relationship is based upon the event model. The event model raises several events when executing methods of a sequential object. The language model defines several important events like entering a method or exiting a method. If the programmer of the sequential class wants to synchronize on a specific event he only has to tell the the compiler that he wants to catch this event and name the occurence of that specific event. Program 1 shows an example of such an annotation in a language similar to C++: the **`raises`** statement is used to raise specific events (e.g., **`Entry`**) and to name this event (e.g., **`M1Entry(a)`**). Table 1 shows all existing event types.

Event name	*Event Description*	*Where to annotate*
Entry / Exit	entry/exit of a method	method declaration
Call / Return	outgoing call / return of an outgoing call	method invocation
IEntry / IExit	internal method enter / exit	method declaration
ICall / IReturn	internal call / return of an internal call	method invocation
Custom	raise a custom event while executing inside a method	anywhere inside a code block

Table 1: Available event types.

```
class Seq_Class {

    void M1(int a) raises  Entry    M1Entry ( a )
                           Exit     M1Exit ( )
    {
        ... Meth-Code ...
    }

    SyncMapping RW {

        SyncClass       RWSync;

        M1Entry (a) -> ReaderEntry ();
        M1Exit () -> ReaderExit ();
    }
};
```

Program 1: Annotation of a sequential class, Reader/Writer syncronization.

The actual relationship between the events defined in the sequential object and the synchronization object is established by the *event mapping*. Those events which are defined inside the sequential class are mapped to specific methods of the synchronization object within the sequential class. The programmer is able to specify several mappings inside the class to enable the use of the class in several different contexts. The concrete mapping is choosen when instatiating a class. If nothing is specified with instantiation, the *default* mapping[2] is applied. An example for such a mapping is shown in program 1.

Each time an event is raised by the sequential object, the execution is transferred to a specific method of the synchronization object. If the event cannot be processed by the synchronization object because of a blocking condition, the activity is blocked until the blocking conditions for the event no longer exist. These conditions are specified in the event-handling-methods of the synchronization object. My event model simplifies the task of deriving a subclass from a concurrent base class. The next section introduces this feature and takes a closer look at what is called the inheritance anomaly.

4 Separation and Inheritance

One of the most important issues when defining a concurrent object oriented language is inheritance. A programmer refines or specializes a class by subclassing. When inheriting from a **concurrent** base class, two classes are involved: the sequential class and the synchronization class. The normal inheritance mechanism must be split into two separate inheritance steps, one of the sequential class and one for the synchronization class. In the following I will call this sort of inheritance "*concurrent inheritance*".

Inheritance from the sequential class is used to refine the algorithmic behaviour of the sequential base class (*BC*). Embedding the extensions of the sequential subclass (*SC*) in the concurrent context implies that the refinements in the subclass must have their own synchronization code associated with the subclass methods. As shown in Figure 2, this is done by deriving another subclass from the associated synchronization base class (*syncBC*), resulting in the synchronization subclass (*syncSC*). These two subclasses are now one logical unit, the concurrent subclass. The methods from *BC* have a synchronization relation with *syncBC*, and *SC* has a synchronization relation with *syncSC*. In many synchronization schemes the relationships above the *border of modification* are fixed[3], which is the reason why inheritance anomalies occur [MaYo93]. Even if the code can be modified, this contradicts the basic object oriented ideas of inheritance. The number of inheritance anomalies is reduced by untying the fixed relations between the base classes. Figure 2 shows the general pattern of the relationships between classes if inheritance from a concurrent base class is employed. Looking more closely at concurrent inheritance, we can distinguish four specific configurations which allow us to categorize certain problems of inheritance.

2. the default mapping is the first mapping defined in a sequential class
3. The implementation of base classes is often imported from class libraries, which are delivered in binary form. Modifying these implementations and the associated synchronization specification is impossible.

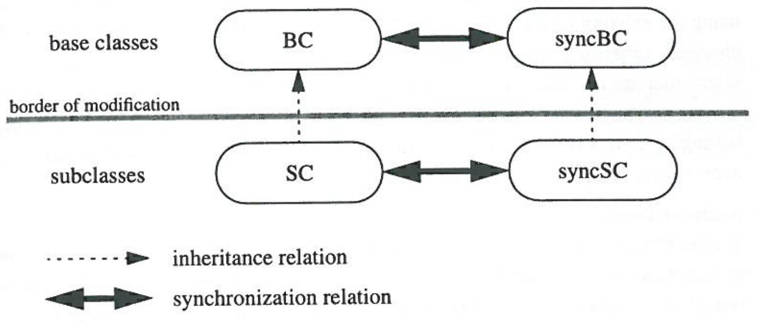


Figure 2: Inheriting from concurrent classes

Different configurations of concurrent inheritance

The four categories are shown in Figure 3. These configurations can be modelled with the aid of the presented synchronization scheme by using object-oriented concepts like inheritance and late binding. The inheritance anomalies described below do not occur with my synchronization scheme. The usage of the four categories are:

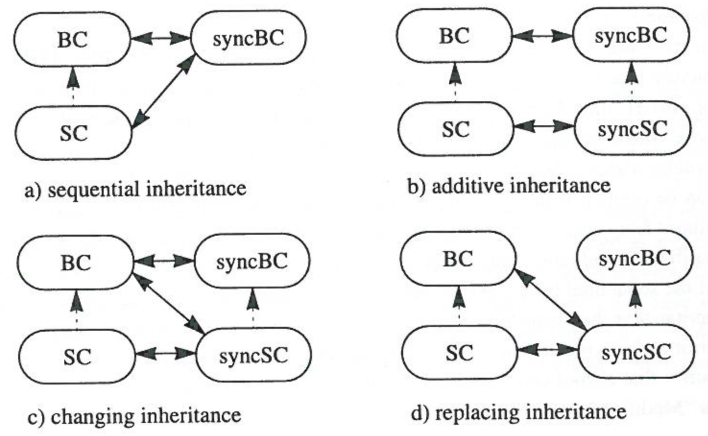


Figure 3: Different configurations when deriving from concurrent classes.

- **Sequential inheritance**
  The derived, sequential subclass *SC* offers additional methods that can be synchronized using the existing methods from *syncBC*. The relationship between the base classes is not changed. Inheriting from *syncBC* is not necessary, *syncBC* already contains all mechanisms that are necessary to synchronize the *SC*. The additional events from the subclass are mapped to existing synchronization methods from *syncBC*. Synchronization problems belonging to this category are, for example, reader/writer problems. This kind of inheritance normally does not show any anomalies.
- **Additive inheritance**
  We use this kind of inheritance for "backpack inheritance", where a concurrent class adds new, orthogonal functionality in the subclass. The synchronization of the additional methods of *SC* cannot be achieved with methods from *syncBC*. A *sync*SC must be derived to provide these methods. This inheritance does not change the synchronization relation of the base classes. An example of this kind of inheritance is the extended buffer from Matsuoka [MaYo93], which is derived from a bounded buffer and adds a `Get2`[4] method. This kind of inheritance can create anomalies called "Partitioning of Acceptable States".
- **Changing inheritance**
  The synchronization relation in the base classes changes, and some of the base class events must be ramped to methods from the synchronization subclass. When dealing with a problem that needs changing inheritance, it is not sufficient to implement additional synchronization methods into the subclass. There are two ways to achieve the necessary changes: Either the mapping of the base class events can be changed directly, or the method to which the event was mapped is reimplemented in the synchronization subclass. This constellation can raise anomalies called "History-only Sensitiveness of Acceptable States".
- **Replacing inheritance**
  In this case, the synchronization relation of the base classes has been exchanged completely. The relationship between *BC* and the *syncBC* has to be untied and all the methods of *syncBC* must be reimplemented in *syncSC*. This is the most powerful form of inheritance with concurrent classes, even if *syncSC* is not really derived from the *syncBC*. The synchronization behavior is fully replaced. The inheritance of the synchronization code can be omitted if the type system allows type compatibility between classes that do not inherit from each other. A language like C++, for example, does not allow such type compatibilities and inheritance is necessary. The type compatibility ensures that all mappings of the sequential base class are satisfied by the new class. An example of this kind of inheritance is the reader/writer problem for the case where the subclass wants to change the priority from reader priority to writer priority. Another example is a subclass of bounded buffer that allows concurrent access to the buffer. Matsuoka categorizes these examples as "Modification of Acceptable State".

4. The *Get2*-Method removes 2 elements from the buffer in one atomic operation.

5 Current work

Typing of annotated classes

The presented synchronization scheme has very much flexibility in combining sequential objects and synchronization objects. A subclass for example can make modifications in the synchronization relation of the baseclass. It would be useful if the programmer of a sequential class could specify the minimum needs of synchronization necessary for his object. One way to achieve that is typing. I am currently working on a type system for concurrent objects to secure the composition of a sequential object and a synchronization object. Both the sequential object and the synchronization object have a concurrent type (Figure 4): a sequential object specifies its synchronization requirements, whereas a synchronization object defines the synchronization mechanisms it tenders. A pair consisting of a sequential and a synchronization object is type conform with respect to a certain mapping, if the synchronization requirements of the sequential object are met. I have to find out what the term "*subtype relation*" means in this context.

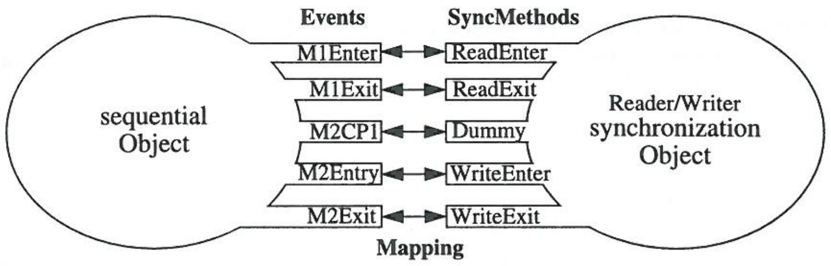


Figure 4: Concurrent type of objects.

Implementation with MetaJava

I am trying to implement the presented synchronization scheme into languages like Java. The main problem is to raise events like entering a method and to make these events visible in the language itself. This problem could be solved by using meta systems. We have developed a platform called "MetaJava" [KlG96]. It allows us to attach any meta objects to normal Java objects. The meta object then catches several events like method invocations to the attached object (see Figure 5). The above mentioned synchronization objects could be modelled as meta objects. The problem with these meta objects is, that they catch all events of a certain event class even if it is not specified inside the sequential object. Therefore the meta object must have all information from the raise statements and the complete mapping of the events in order to make the correct decisions. I am currently developing a MetaJava module to implement my event model for synchronization.

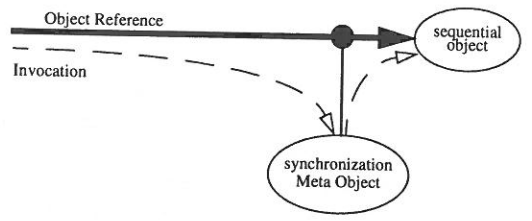


Figure 5: Synchronization event model with MetaJava (Method Enter Event).

6 References

Hal94 Ciaran McHale: *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* Ph.D., University of Dublin, Trinity College, October 1994

KlG96 Kleinöder, J.; Golm, M.: "MetaJava: An Efficient Run-Time Meta Architecture for Java", *IWOOOS '96 workshop*, Seattle, 1996

MaYo93 S. Matsuoka; A. Yonezawa: "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", In: G.Agha, A.Yonezawa, and P.Wegner [Ed.], *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993

Neu91 C. Neusius: "Synchronizing Actions". In: P. America [Ed.], *ECOOP '91, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, No.512, Springer-Verlag, Berlin, Heidelberg, 1991, pp. 118-132

Aspectual (de)composition for natural design of (operating) systems

Lutz Wohlrab

Chemnitz University of Technology
Department of Computer Science
Operating Systems Group
09107 Chemnitz
Germany

e-mail: lwo@informatik.tu-chemnitz.de
www: http://www.tu-chemnitz.de/~luwo/
phone: +49 371 531 1390

1 Introduction

To design an operating system solely based on state-of-the-art functional (object-oriented) analytic and constructive methods is always somewhat unnatural. It is difficult to answer questions like

- What about Threads? Are they objects? If so, how do other objects become active? What about critical sections in the code of the objects, parallelism? If not, what else and what does this imply?
- How about reflection? Do only single objects reflect or the whole system and how?
- How about distribution?
- Is a suitable error detection/handling possible without mixing up with the actual functionality?

This difficulty stems from the fact, that the operating system can be seen from different angles. The functional hierarchy model, the thread model, the reflection technique, and other views of an operating system are fairly orthogonal to each other. Current systems usually were designed with an overall model in mind, which was some mix of hese views. Such models tend to neglect other views beyond a chosen major one. Usually this is either the procedural/functional or the thread/process perspective.

To make things simpler, the (neglected) views are restricted to fit the model of the main one. For instance, to unite the functional (objects) and the activities (thread) view, the object definition is modified into "everything is an active object", which means each object has (mostly only) one thread somehow assigned to itself during its whole lifetime. Or, the other way 'round, threads are declared to be special objects and are temporarily attached to the (actual) objects. Models of the first kind do not allow objects to be as fine grained as known from software development, models of the second kind are usually very complicated to handle.

Beyond this, the mixing of views leads to a lot of tangling code whithin the components of the system which has nothing to do with their actual function. It is about the other views besides the functional: threads synchronisation, reflection, error detection/handling... This code is very difficult to further develop and maintain. This property of the views is called cross-cutting [KLM+97] and so they are to be regarded as aspects.

2 Object orientation is the most powerful integrating technology, but still not powerful enough

Object orientation was intended for breaking large systems and structures into peaces of conquerable, understandable size (regarding OOA), for constructing systems without being overwhelmed by their sheer size (OOD), and programming without continually reinventing the wheel (OOP). Exactly these areas mark its strengths. Therefore, object orientation can be used for integrating other technologies into large systems quite easily.

But as soon as one makes object-orientation a dogma, the resultant solutions are likely to suffer: imagine, for instance, a database into which lots of small, simple data entities like names and income figures have to be stored. Turning everything into object would make the little relational database inefficient, offering no other advantage than that in future the additional storage of multi media data would be easy. If that is not needed, the overhead which has to be paid is too high. The more efficient solution regards the whole database as a single object, which just happens to work internally with another (the relational) technology. The interface methods of this database object do the necessary integration (conversion) work between object and relational model. This way the (more efficient) relational internal the implementation would be hidden and be replaceable by an object-oriented one as soon as really needed.

The situation is a bit similar with the discussed views of an operating system: collapsing everything into a single (object) model resulted in complicating rather than simplifying things. But not every view is as easily integrated by conventional means as the relational data model in the example above.

The integration of the above-mentioned views of operating systems affect not just a single part identifiable in the functional (object) model, but a large part of the system. The way of the threads through the code has to be restricted reasonably, appropriate synchronisation enforced. Meta knowledge about the objects of each class to be collected and forwarded to an instance capable of reasoning about the whole operating system.

It has to be distinguished between local and remote actions. If all the code for this is hand-written into the various components of the system, it results in large portions of tangling code as mentioned in the introductory section.

To avoid tangling code, tools are needed which allow the analyst/programmer/designer to independently deal with the separate aspects or views, which bring them together, and turn the whole into a working system. The code which the human deals with then is not mixed and only about one aspect a time. Therefore, it is much easier to understand and maintain.

Object orientation focuses on functional (de)composition. But especially on large software systems, various technical constraints and peculiarities are imposed, resulting in the different views of the system. Dealing with these views is nowadays called aspectual (de)composition and performed additionally to functional (de)composition. Groundbreaking research in this direction is done under supervision of Gregor Kiczales at Xerox PARC [KLM+97, ILG+97, LK97, MKL97].

3 CHEOPS

The CHEOPS[1] object management hierarchy is depicted in Figure 1. The root of this hierarchy, the class object manager, will be part of the system core which is already running on bare hardware and bridges the semantical gap between hardware and objects.

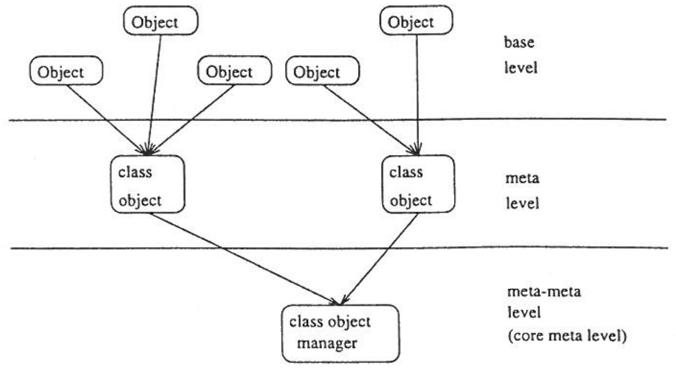


Figure 1: CHEOPS object management hierarchy

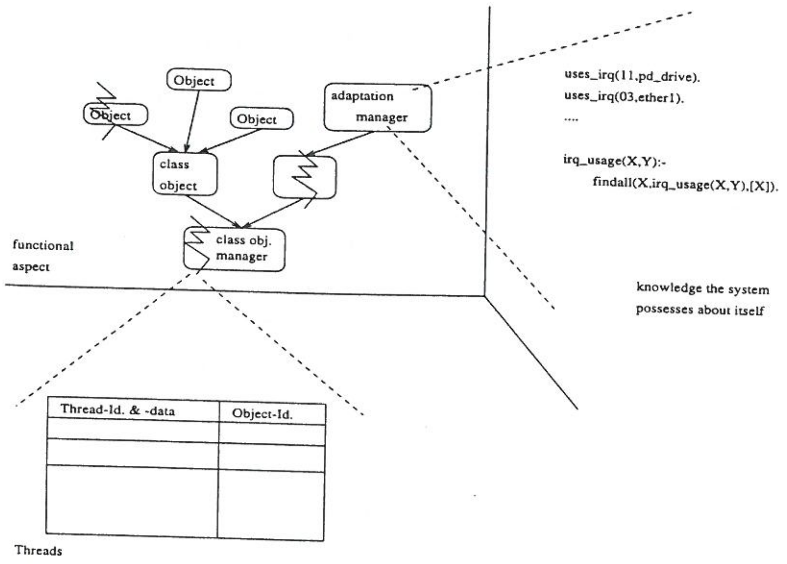
It is (among other tasks) responsible for loading the code of the class object and object classes into memory and instantiating the class objects. Hence, it acts as meta object for class objects. For each class, only one class object can be instantiated during runtime.

[1]CHemnitz OPerating System

A class object is responsible for (de)instantiating of the objects of the class it represents. Moreover, it provides the meta interface for this objects, tries to ensure that they get the best runtime environment which can be established, aids the migration of objects to other hosts by transparently setting up proxies and a lot more.

Meta object interfaces without a way to reason about the system's behavior and knowledge about the system itself would be useless. Since we want the operating system as a whole reason about its behavior, we decided to concentrate the knowledge as well as most of the reasoning within a single component, the adaptation manager. The knowledge about itself and the mechanisms connected therewith form an independent aspect, since it represents a view of the whole system as well as the object management hierarchy.

The knowledge we lay down in the shape of Prolog clauses. We use Prolog as an aspect language for this because it is an established language in the field of knowledge representation, and a number of reasoning systems have been successfully built using it. Additionally, its meta computation facilities allow for checking new knowledge before accepting/learning it, and thus keep the knowledge base free of avoidable conflicts.



```
uses_irq(11,pd_drive).
uses_irq(03,ether1).
....

irq_usage(X,Y):-
        findall(X,irq_usage(X,Y),[X]).
```

Figure 2: CHEOPS object management, threads, knowledge-about-itself aspects

But how are the knowledge-about-itself aspect and the object view brought together? The adaptation manager is programmed by hand, as an object encapsulating the inference mechanism. When describing the knowledge-about-itself aspect, the programmer writes down Prolog clauses. These clauses are to be picked up by the class object class generator tools. These tools add code to the class object classes, which forwards the knowledge

to the adaptation manager when the class object is instantiated. Other code has to be handwritten in C++, such as code for adaptation requests from the objects. How it can be suitably generated out of the knowledge-about-itself view is still subject to investigation.

The same technology applies to the threads aspect: The code for the thread management is a handwritten component of the system core, code needed for synchronization with regard to the instance variables is woven in by source code generators as above.

This resembles what is understood under partly compile-time, partly runtime weaving [KLM+97]. At compile-time, the above-mentioned generators add the knowledge forwarding code, for instance. When executed, this code adds class specific knowledge to the knowledge base during runtime.

The result is not a single program, woven, compiled, and executable. It rather is a set of components which got a few properties added during the weaving and compiling, and some solely hand-coded entities. Modern operating systems retain a lot of the modular structure from the source code level during runtime.

4 Related work

[Hec91] introduced the concept of adaptiveness in an operating system using a knowledge base. From his point of view, the system consists of execution base, observation base and knowledge base. In his concept, knowledge base and reflection form the main view onto the system, like objects or processes in others.

Apertos [YTT89, Yok92] as the first reflective object-oriented operating system is related to CHEOPS in that respect, that both of them are object-oriented operating systems enabled to reflect about their own behavior. However, they significantly differ in architecture: real Apertos objects are generally active, CHEOPS objects are passive until they get a thread attached temporarily. In CHEOPS, a single instance (the adaptation manager) does or controls reflective operations. In Apertos, this task is divided among reflectors.

Other than in Choices [MIKC92] CHEOPS class objects are instances of a special class object class tailored to the objects they maintain. Of each class object class, there is only one instance in the object management hierarchy. This enables us to more easily plug in filters, observer, guard, and proxy objects. Moreover, it offers a hook where to place the code stemming from other aspect descriptions.

The work of Calton Pu [PAB+95] we owe our awareness for the need of optimistic implementation alternatives, their shielding with guards outside the main computation path, and their reflectional exchange when guard conditions get violated.

Last, but most important: introducing aspectual decomposition in their white paper on the web [KLM+97], the team of Gregor Kiczales' at Xerox PARC decided the (rather ineffective) discussions in our team, how everything was to be brought into the shape of an object whilst it really was, for instance, a rule better written down as Prolog clause.

5 Future Work

The CHEOPS system is being implemented at three points in parallel. As of now, a part of the system core is already running on bare i486 hardware, the object management (tools for automatically generating raw class object classes and the hand-coded class object manager) is implemented as a prototype running on top of Linux. The current prototype of the adaptation manager is implemented as a combination of a daemon and a dynamically loadable Linux kernel module. After the evaluation phase, the last two components will be ported to the native kernel. Moreover, the class object class generator tools will be supplemented to generate code for the knowledge-about-itself aspect.

At least the adaptation manager prototype will be further developed under both systems. We are interested in how this new technology is usable under "old" operating systems especially how applications unaware of the existence of the adaptation manager can still use old configuration interfaces (/etc, for instance) and the system as a whole in spite of being old-fashioned can profit from adaptation management. Therefore, the most pressing work is to further populate the till now sparsely filled knowledge base. In order not to program the whole knowledge base manually, new interface and development tools (e.g. gateway functions for some /etc config files) will have to be designed and programmed.

Regarding aspect-orientation, our project is still very much at the stage which Kiczales et al. [KLM+97] describe as experimenting with metaobject protocols to get a feeling what design is most suitable for aspect language (-restrictions) and what weaving technique is best used. Step by step the gained experience will be reflected in future versions of our code generator utilities and runtime mechanisms.

References

[Hec91] A. Heck. *Adaptive Betriebssystemkonzepte.* PhD thesis, Technische Hochschule Darmstadt, Fachbereich Informatik, 1991.

[ILG+97] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales s, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. Technical Report SPL97-007 P9710045, Xerox PARC, February 1997. `http://www.parc.xerox.com/spl/projects/aop/tr-aml.html`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997. `http://www.parc.xerox.com/spl/projects/aop/tr-aop.html` submitted to OOPSLA'97.

[LK97] Christina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010 P9710047, Xerox PARC, February 1997. `http://www.parc.xerox.com/spl/projects/aop/tr-rg.html` submitted to OOPSLA'97.

[MIKC92] Peter Madany, Nayeem Islam, Panayotis Kougiouris, and Roy H. Campbell. Practical examples of reification and reflection in C++. In *Proceedings of the International Workshop on Reflection and Meta-Level Architectures*, pages 76–82, November 1992.

[MKL97] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, February 1997. `http://www.parc.xerox.com/spl/projects/aop/tr-rg.html` submitted to OOPSLA'97.

[PAB+95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, December 1995. `ftp://cse.ogi.edu/pub/dsrg/synthetix/sosp95.ps.gz`.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system: the concept and its implementation. Technical Report SCSL-TR-92-014, Sony Computer Science Laboratory, October 1992. *Proceedings of the Seventh Annual Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA'92)*, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/92/SCSL-TR-92-014.ps.Z`.

[YTT89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. Technical Report SCSL-TR-89-001, Sony Computer Science Laboratory, 1989. *Proceedings of the 3rd European Conference on Object-Oriented Programming*, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/89/SCSL-TR-89-001.ps.Z`.

Mobile Object Systems: The Next Generation

Michael J. Zastre

University of Victoria, British Columbia, Canada

Abstract. Mobile objects fit into the matrix of internetworks, large data sources, and powerful network clients. The union of mobile code, state plus data has many applications, such as distributing control in workflow or groupware systems. The next generation of these systems of travelling objects must address two groups of issues. *Mobile Objects in-the-small* is the name I give to concerns over mobile object construction and arrangement. *Mobile Objects in-the-large* is my term for the issues involving large systems of mobile objects (dozens to hundreds) interacting together. It includes concerns over how these systems can be validated from the specification and synthesis of the mobile and interacting portions of an object graph given a database of application domain knowledge. I explore these two aspects of next generation systems in more detail, using a workflow example for motivation. Standard meta-programming facilities are re-cast into the mobile agent support environment (e.g. for memory management, run-time stack examination, heap manipulation), and I propose additional facilities. Specification of mobile object interactions and expressibility of their properties, and currently technology's bearing on these, is discussed, and future work suggested. A schematic for a system synthesis tool is also presented. All discussions are ultimately aimed at the research and development of an integrated set of mobile object system construction tools.

1 Introduction

Mobile Object Systems are continue to receive increasing attention from the research and industrial development communities. The transport of code plus intermediate results (state), combined with plentiful bandwidth and spare computing capacity at servers and client workstations, is changing the face of distributed computing. First generation mobile systems have focused on solving important issues such as *how* mobility may be expressed, and the *ways* in which security is assured. Other characteristics are:

- application programming interfaces (APIs) hiding developers from important implementation decisions; and
- small low numbers of objects interacting with each other (less than 10)

Examples of these systems include Java Aglets [11], ML-based Facile [10], a derivative of Modula-3 and Obliq called Visual Obliq [3], Telescript [18], and a new language and environment from the University of Geneva named SEAL [16]. They are descendants from previous work on distributed systems and programming languages, such as the ground-breaking work of the late 1980s by the

Emerald Project [8] and that of the 1990s Sun's Java Language and especially the Java Virtual Machine and its sandbox security model.

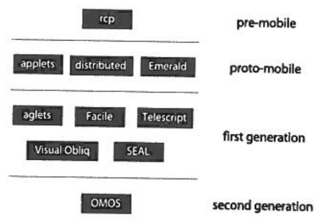


Fig. 1. Generations of mobile object systems

Mobile Object technology promises to become ubiquitous. Before this happens, however, I believe that at least two aspects require much further research:

- *Mobile objects-in-the small*: programming environment support for developing mobile object systems, such as for the construction, transmission, reception, and resource management of these objects, *where the system developer determines the implementation decisions underlying the mobile object system itself.* This constitutes an *Open Mobile Object System (OMOS)*.
- *Mobile objects in-the-large*: tools for specifying and exploring the interactions of large (dozens) and very large (hundreds) of systems of mobile objects whether these objects enter a network from outside an enterprise, or originate from within the same intranet.

These should be the characteristics mobile object's next generation of languages and systems (see Figure 1).

Mobile object research must ease the development and deployment of large distributed systems. The following section of this paper outlines a sample problem from the Very Large Database research community. This will place into context the research problems discussed throughout the text. The rest of the paper explores these two characteristics (*in-the-small* and *in-the-large*) of second generation mobile object systems, outlining the research problems I am investigating as part of my Ph.D. research.

2 Motivation

A combination of highly-available networks and plentiful bandwidth, plus powerful processors and large data capacity on each desk, has led to *groupware* and *workflow* systems. These combine the capabilities of distributed systems and very large databases. Operating systems such as COOL [12] have been developed for

this, and database researchers recognize workflow's unanticipated use of existing data stores and systems [2]. Mobile objects can be used at various levels to act as "glue" knitting together interactions across databases and as encapsulators of the data.

Ansari et. al. [1] have examined the use of workflow in a telephone company; a composite diagram of their diagrams appears in Figure 2. Database interactions with a central control are broken into sub-transactions; various databases are accessed more than once. Workflow systems are complex because of tension between the independence of each database simultaneously and the requirement sub-transaction results cannot be committed until the overall transaction is succeeds; the interval from start to finish may be several minutes or several days.

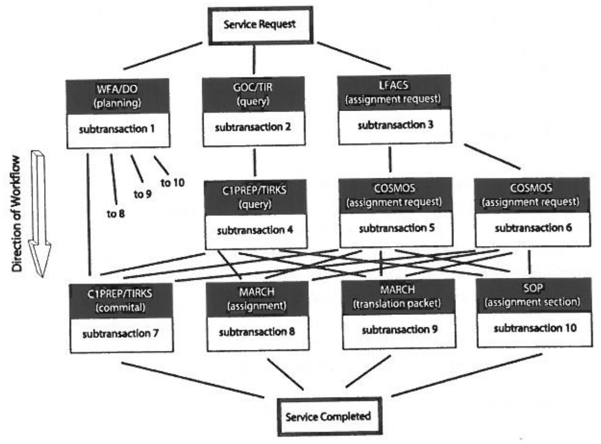


Fig. 2. Telephone company workflow (from Asari et. al)

Similar workflows may be seen in other office environments, such as the paper flow through an insurance company (adding new clients into information system, servicing claims, handling inquiries), government departments, or publishing houses. Mobile object systems are an appealing technology both data-centric and process-centric workflows by providing objects that:

- encapsulate the data item (form, database query, database query result);
- store state in the form of intermediate results to forward on to another site;
- combine methods and data to determine the next stage, step, or control or data site (i.e. person, machine or server) needed to complete current and

future stages in the workflow; and
- encapsulate code in the form of methods acting on both the data within the object and data at the site at which the mobile object currently resides.

This exposes weaknesses with current mobile object systems:

- (*in-the-small*) One size of object does not fit all; system developers may need some functionality of these systems but not all. Object data encapsulation and transport will depend on the application, the use of replication, and the degree of independence mobile objects should have from the executing platforms.
- (*in-the-large*) New objects and their interactions should be derived from some sort of specification; mobility could be "boilerplated" into the objects in the same way Visual programming systems automatically generate code for GUIs. These specifications of larger systems must be used to prove certain properties, such as correctness of interactions, freedom from deadlocks, and the confirmation that definite results will be obtained.

3 In-the-small: Current Work

A claim made by the meta-programming community is that the black box model of computation may front average case implementation decisions. These decisions are often best made by the software developer [9, 5]. A meta-interface is used to specify computation on the computation, as opposed to a regular interface which specifies computation on the data domain. Exposing the implementation details of mobile object system will require careful design of the meta-interface.

J. Templ's dissertation [15] demonstrated that a modern object-oriented imperative programming language can benefit from meta-programming facilities. In this case, Oberon is extended with facilities for:

- traversing (or "riders") through heap space and its data structures;
- traversing through the run-time stack for both examining the data and modifying it;
- introduction of "filters" at run-time which stand between an event and the procedure servicing that event, with the filters in effect acting as a "call-back" mechanism that may extend the functionality of already compiled and linked system.

These apply to mobile object systems. Traversing through heap space is needed to swizzle state and marshal the object graph into a format suitable for transmission. Traversing through the run-time stack captures the current point at which a series of method invocations has been interrupted; this is transmitted to the new site as the computation's continuation point. Templ demonstrated that the effort required to develop sophisticated debugging and shell tools was greatly reduced when these meta-programming facilities were used. A similar benefit is obtainable for mobile object systems.

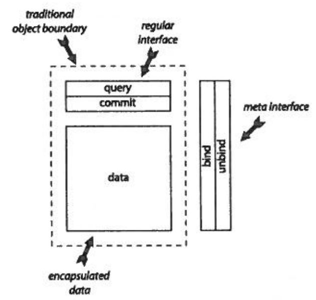
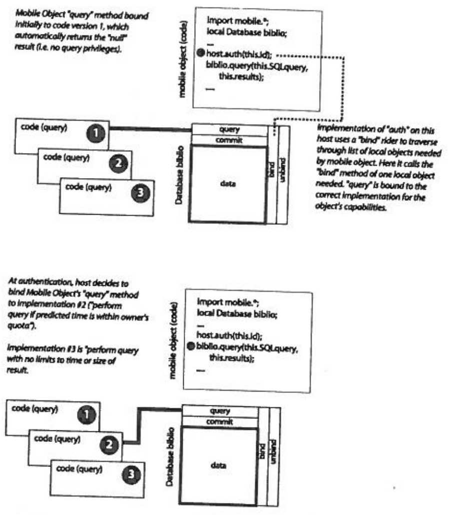


Fig. 3. Schematic of Database object showing interfaces

My current research asks "what additional meta-programming facilities are required to support an open mobile object system (OMOS)?" I propose the following mechanisms:

- *Network protocol riders* modify how data is transferred to and from the executing platform through the network layer. Optimizations are placed here by system developers, such as communicating through the network to determine the presence (or absence) of objects at the remote site that are needed by the travelling object. Access to the network interface also allows insertion of custom security code at this point where the mobile object execution platform meets the network.
- *Method binding riders* can be used with security and/or resource monitoring mechanisms. For instance, a recently arrived object would have methods bound immediately to safe (and virtually powerless) implementations until authentication succeeds. At that point, methods would be bound with versions having more functionality; several levels of security are possible. See Figures 4 and 3. The advantage is given to the OMOS developer, who decides *when* binding occurs (equivalently, choosing when bindings change).
- *Security riders* in conjunction with heap space riders, used to implement security models that "watch" the computations and react to suspicious results.
- *Ontology filters* insert conversions from one knowledge representation to another. This is useful when a mobile object from one organization with a vocabulary arrives at another with an equally entrenched (and disjoint) vocabulary but related concepts. What those conversions are, and how they are represented, are determined with the users and the developer. This filters hook into the method invocations at points deemed critical.

Referring to Figure 4, the authentication/increased functionality example in the figure is one of a method binding rider. Another is for a coarse-grained adaptation:


Mobile Object "query" method bound initially to code version 1, which automatically returns the "null" result (i.e. no query privileges).
mobile object (code)
code (query) 1
code (query) 2
code (query) 3
Database biblio
query
commit
data
bind
unbind
Implementation of "auth" on this host uses a "bind" rider to traverse through list of local objects needed by mobile object. Here it calls the "bind" method of one local object needed. "query" is bound to the correct implementation for the object's capabilities.
At authentication, host decides to bind Mobile Object's "query" method to implementation #2 ("perform query if predicted time is within owner's quota").
Implementation #3 is "perform query with no limits to time or size of result.
mobile object (code)
code (query) 1
code (query) 2
code (query) 3
Database biblio
query
commit
data
bind
unbind

```
import mobile.*;
local Database biblio;
...
host.auth(this.id);
biblio.query(this.SQLquery,
    this.results);
...
```

```
import mobile.*;
local Database biblio;
...
host.auth(this.id);
biblio.query(this.SQLquery,
    this.results);
...
```

Fig. 4. Per-Object Bindings of Mobile Object to Database Methods

- lightly loaded server offers a powerful method implementation to object;
- server retracts implementation, substituting less CPU intensive version as server load increases over some threshold.

Another aspect of meta-programming is the development of meta-objects protocols (MOPs) for mobile object systems, which have been shown to aid users as they develop code [5]. I propose to examine characteristics of mobile object system MOPs by investigating their use in the following type of mobile object systems:

- data-centric workflow
- process-centric workflow
- distributed database query frameworks
- industrial information exchange protocols for process management (e.g. STEP).

Protocol characteristics will be analyzed, and a common set of operations identified in order to be added to the original list of mechanisms given above.

Many associate metaprogramming with computational reflection [13], which focuses on object's modification and adjustment of its own data structures and algorithms given the results of computation (which itself will depend on the data collected so far in the current and previous environments). It is not the

focus of this research, but rather an important topic in the distributed artificial intelligence community.

4 In-the-large: Future Work

Mobile objects promises to aid in and reduce the development effort required for distributed system deployment. This leads to two observations:

- At present there is no technique which can demonstrate a large system will achieve its specified aim.
- No tools exist for decomposing a large distributed system specification into mobile and stationary portions. Note that this is different from determining what processes may migrate for load balancing purposes.

The problem with the first item is not that "satisfying a specification" is inexpressible. For instance, I have performed experiments with casting interactions between objects into CCS (Calculus of Communication Systems) [14] and the temporal properties of the system into a temporal logic (μ-calculus [4]). Unfortunately, freely available concurrency workbenches with built in model checkers have difficulty testing all but the most trivial properties of systems such as those in Figure 2 I propose that patterns of mobile object communication can be exploited and thereby leading to model generators tuned to these characteristics, hence eliminating the state explosion problem. I feel that another possible solution is to use the results of object-oriented process modeling along with emerging theories of object interaction [17]. My goal in this phase of the research is to develop property verification techniques for very large (dozens to hundreds) of cooperating mobile objects. This must be implementable and eventually formed into a tool.

As for the second item, any success in developing a decomposition tool will depend on the existence of domain-specific packages. For instance, success has been reported for using the OO approach to taking a DSP system from specification to an optimized implementation using frameworks and synthetic benchmarks [7, 6]. In a similar spirit, I suggest that this is applicable to mobile object system design and implementation. As an example, given the details of a paper-based workflow process, such as:

- dependencies between documents and forms;
- sets of personnel able to perform the human work;
- locations of those people and their machines; and
- distribution of the data amongst a set of servers

an analysis tool will generate a list of mobile and stationary processes. After verifying that the system as specified has desirable properties (e.g. it is possible that all parts of the document can be completed and delivered to a specific department) and doesn't have bad ones (e.g. deadlock, starvation), then code and data needed by the mobile objects is laid down.

I maintain that the goal of verification and synthesis from specification are not only desirable, but may be necessitated by the ubiquity of mobile objects and the size of the end systems.

References

1. Mansoor Ansari, Linda Ness, Marek Rusinkiewicz, and Amit Sheth. Using Flexible Transactions to Support Multi-system Telecommunication Applications. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia*, 1992.
2. Jeff Ullman Avi Silberschatz, Mike Stonebraker. Database Research: Achievements and Opportunities into the 21st Century. *SIGMOD Record*, pages 52–64, March 1996.
3. Krishna A. Bharat and Luca Cardelli. Migratory applications. Technical Report SRC 138, Digital SRC, February 1996.
4. Julian Charles Bradfield. *Verifying Temporal Properties of Systems*. Birkäuser, 1992.
5. Paul Dourish. Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 2(1):40–63, March 1995.
6. Nayeem Islam. *Distributed Objects: Methodologies for Customizing Systems Software*. IEEE CS Press, Los Alamitos, CA, 1996.
7. Nayeem Islam. Customizing Systems Software Uusing OO Frameworks. *IEEE Computer*, 30(2):69–78, Feburary 1997.
8. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
9. Gregor Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Meta-object Protocol*. MIT Press, 1993.
10. Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.
11. Danny B. Lange and Daniel T. Chang. Ibm aglets workbench: Programming mobile agents in java. Technical Report http://www.ibm.co.jp/trl/aglets, IBM Corporation, September 1996.
12. Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL:System support for Distributed Programming. *Communications of the ACM*, 36(9):37–46, September 1993.
13. P. Maes. Computational reflection. Technical Report Tech. Rep. 87.2, Vrije Universitaet, Brussels, Belgium, 1987.
14. A.J.R.G. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
15. Josef Templ. *Metaprogramming in Oberon*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1994.
16. Jan Vitek. Secure Object Spaces. In *2nd International Workshop on Mobile Object Systems, ECOOP '96*, 1996.
17. Peter Wegner. Interaction as a Basis for Empirical Computer Science. *ACM Computing Surveys*, 27(1):45–48, March 1995.
18. J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace - A White Paper. Technical report, General Magic, Inc., 1994.

This article was processed using the LaTeX macro package with LLNCS style