Development of Distributed Applications with Separation of Concerns

António Rito Silva, Pedro Sousa and José Alves Marques
IST/INESC, R. Alves Redol n°9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287 – Fax: +351-1-525843
email: Rito.Silva@inesc.pt

7th ERCIM DBRG Workshop on Object-Oriented Databases.
INESC - Lisbon, Portugal.
May, 15-16, 1995.

Abstract

The development of distributed applications is an open area involving researchers from different communities. We propose an object-oriented approach to the development of distributed applications emphasizing separation of concerns. Our approach combines the needs of transparency, encapsulation of distribution issues, and support of non-traditional models, where cooperation and sharing are needed. The development process is constructive, thus allowing partial verification of results. We recognize seven concerns: fragmentation, replication, naming, concurrency, failure, configuration, and communication. Each concern is perceived in three levels of abstraction: models, policies and mechanisms. Besides a development centered on separation of concerns we propose another process centered on development stages. Both, concerns and stage perspectives, are part of an integrated and flexible development process.

Keywords: Distributed Systems and Architectures, Object-Oriented Analysis and Design.

1 Introduction

Sequential applications in a centralized system use traditional models where activities and data are isolated. In distributed systems activities and data are intrinsically distributed and shared. Distribution yields a model inherently more complex, thus making application development a difficult task. Transparency, offering in a distributed system the same model that centralized systems offer, has been the goal of researchers.

Distributed operating systems and platforms [29, 6] have been proposed offering several degrees of transparency: failure, access, location, concurrency, replication, migration, performance, and scaling [11]. These researchers do not agree on the need of full transparency, raising two questions: Is it possible to achieve? Is it worthwhile? Nevertheless, with transparency, developers needn't concern themselves about distribution issues, since the platform encapsulates them.

The obvious advantages of transparency have however some drawbacks: absence of detail and lack of support for non traditional models [32]. In most systems, transparency does not allow distributed issues to be handled at different levels of abstraction, according to the particular application needs. Moreover, only traditional application models are supported by such systems. It is hard to support new emerging areas, as cooperative work, where those models are relaxed.

Our approach combines the advantages of transparency and the support of non-traditional models. Furthermore, development can be done at different levels of abstraction.

We propose a constructive development with separation of concerns. Applications are developed in a stepwise manner, handling one concern at a time. One of our environment's key features is the independence between solutions for each concern, allowing transparency in all issues but the one being solved. Solutions of concerns do not interfere[1], although some combinations may not be possible. Furthermore, a functionality-centered process is also described and consistently integrated with the concerns perspective. Verification and debug are also stepwise processes. Default solutions are offered to achieve a rapid development and support for traditional models. We offer

[1] In section 4 we explain what is meant by non-interference.

tools which incorporate default solutions, allowing a transparent development.

Concerns are perceived in three levels of abstraction: models, policies and mechanisms. Solutions for each one of these levels are considered in the construction of each concerns's solution. This approach allows several independence levels and, so, an uniform development of heterogeneous applications: heterogeneity of policies and mechanisms. The mechanism level of abstraction can be implemented over heterogeneous languages and machines.

Our approach follows the object-oriented analysis and design paradigm. Other object-oriented approaches, e.g. [33, 18, 4, 9], have as yet not addressed distribution issues in depth.

The rest of this paper is structured as follows. Distribution concerns are described in section 2. In section 3 abstraction levels are introduced. The development process either centered on concerns or on stages is described in sections 4 and 5, respectively. Section 6 describes the concurrency concern in depth. Related work is presented in section 7 and conclusions in section 8.

2 Concerns

Designing distributed applications involves the solution to particular problems. Each one of these problems is a development process concern. We have recognized seven main concerns which we believe cover most distribution issues: fragmentation, replication, naming, concurrency, failure, configuration, and communication.

- **Fragmentation**. A distributed application schema is the composition of several schemas that together constitute the application schema[2]. When designing distributed applications using our approach, application functionalities are isolated in different logical distribution units, referred to as worlds. The fragmentation concern is about the definition of worlds and their schemas such that application semantics results from the semantics of the different worlds. For instance, in a bank's distributed database system, client accounts are stored in the local database branch where the account was open. In the example, each local database branch is a world and the client account is fragmented in the different worlds.

[2] We use schema to denote both static and dynamic structure of applications.

- **Replication**. Some distributed applications need to replicate information because of both availability and reliability. Availability permits access to replicated information without remote accesses. Reliability hides failures of remote nodes from the local computation. The replication concern defines replica objects that are distributed between worlds and together behave consistently as a unique object. The replication concern should preserve some kind of object consistency.
- **Naming**. Distributed applications use different name spaces and so the meaning of a name dependents on a particular name space context. Furthermore, names can have different properties depending on the name space they belong to, e.g. location transparency, where the name does not restrict the location of the named object to a particular node. The naming concern defines the name spaces needed for the distributed application such that names coexist consistently in the application, e.g. different name spaces sharing a name.
- **Concurrency**. Shared access to resources is a feature of distributed applications. Applications must generate and control the concurrency within a resource. Concurrency control has several properties, e.g. it may be transparent in the sense that users do not cooperate when accessing the resource. The concurrency concern should define concurrency generation and control. The former creates and destroys activities. Concurrency control constrains undesirable interactions.
- **Failure**. In distributed systems, applications are likely to fail and the robustness degree depends on application semantics. Failures can be masked from the user. The failure concern defines the desired degree of robustness and guarantees that it is achieved.
- **Configuration**. During an application's lifetime worlds can be created and destroyed. Upon creation, worlds and their objects, need to be integrated within the application, and, after a world is destroyed, the application should adapt itself to the new configuration. Several kinds of configurations are possible: static configuration, where inter-world relationships are hard-coded; or dynamic configuration, where inter-world relationships can change at run-time. The configuration concern defines how worlds and their objects react upon creation and destruction of other worlds. For instance, when a server is shut down, client nodes must send requests to another server.

- **Communication**. Worlds in a distributed application can communicate using different communication paradigms as, for instance, procedure call or mailboxes. These paradigms have different communication models and different invocation syntax. The communication concern defines the communications paradigms the application uses and integrates them within the application, e.g., asynchronous communication is related with the concurrency concern.

The degrees of transparency referred to in section 1 [11] are obviously related to distribution concerns since they intend to make development easier by encapsulating the distribution issues in the platform. Failure, concurrency and replication concerns handle the same problems as failure, concurrency and replication transparency. Access transparency, in which identical operations are used to access local and remote objects, is handled by the communication concern. Location transparency, in which the application is not aware of the resource location, is handled by the naming concern. Migration allows for an object to move from a source world to a destination world. This is a combination of the replication concern, since it corresponds to the creation of a replica in a destination world and deletion of the replica in the source world, and of the configuration concern, since the new replica should be accessible to clients. Performance transparency, reconfiguration of the system to improve performance, is handled by the configuration (dynamic) concern. Scaling transparency, expanding in scale without changing the application structure, is not handled by a particular concern but is achieved if all the concerns consider scalable solutions.

3 Abstractions

We consider that concerns are divided into three levels of abstraction: model, policy and mechanism (figure 1). Models describe the expectation of users about the application or system behavior. Policies define algorithms which support application models. Mechanisms define functionality which is used by policies to implement their algorithms.

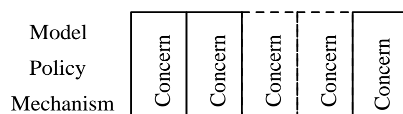

Figure 1: Abstractions and concerns.

We distinguish between abstract and concrete mechanisms. Abstract mechanisms fulfill the needs of models and policies and are platform independent. Concrete mechanisms are the actual platform mechanisms and are used to implement abstract mechanisms. Abstract mechanisms should be expressive enough to support policies and simple enough to be implemented by platform mechanisms. Due to their proximity to platform mechanisms, the same abstract mechanism can be used in the implementation of different concerns policies.

Consider, as an example, the concurrency concern which is divided in concurrency generation and concurrency control. A possible model of concurrency generation is one in which the user is not aware of the existence of concurrent activities. A model of concurrency control is a serial model and establishes that concurrent access to resources do not interfere, which means that the effects of concurrent execution of activities should be equivalent to their serial execution. From the user's point of view activities are independent since other activities (users) currently executing are not relevant to the results. This model is supported by database systems, either using optimistic or pessimistic policies. Each one of these policies can use abstract mechanisms to generate concurrency, e.g. *Activity* which abstracts *thread* mechanisms, and to control it, e.g. *MutualExclusion* which abstracts *mutex* mechanism.

Perceiving concerns in three levels of abstraction offers independence, heterogeneity and transparency while allowing concerns to be handled at different levels of detail. The separation of models and policies allows a model to be independent of a particular algorithm which supports it. Separation of policies and concrete mechanisms permits policies to be independent of implementation platforms, e.g. *Activity* can be implemented in a given platform with operating system threads while *MutualExclusion* can be supported by synchronization primitives *Mutex* and *Condition*. Policy independence allows the same model to be simultaneously supported by different policies, so that heterogeneous policies, offering the same model, can coexist in the same application. Moreover, independence between policies and concrete mechanisms allows the application to be supported by heterogeneous platforms.

Sequential centralized systems offer a set of traditional models, which we call strict models, where activities and data are isolated. In these models users expects system reaction considering only its inputs. Degrees of transparency [11] and ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions [44] offer the strict models of sequential and centralized systems on top of distributed systems[3].

[3] Quite often strict models are supported by computer mecha-

The strict models expected by users are built upon computation non-strict models inherent to distributed systems, e.g. a communication delay can generate replicas with different values (figure 2). However, these models do not correspond to application needs being a consequence of computational behavior and can be considered for low level efficient implementation [25].

Application Non-Strict Models
Strict Models
Computation Non-Strict Models

Figure 2: Strict and non-strict models.

More recently, developers are requesting particular non-strict models for cooperative work [32] in which user control and synergistic cooperation [3, 13] are needed. These models fulfill application semantics and we refer to them as application non-strict models (figure 2).

Since application non-strict models are still an open research area, no general solutions are as yet well established. Therefore our approach permits application non-strict models to be built using policies and mechanisms that do not encapsulated distribution issues, instead of offering closed, not open, solutions for this kind of models.

Strict models are well established and propose a virtual interface where distribution issues are masked. Strict models for the fragmentation concern consider that all objects have the same interface, the designer is not aware of logical distribution, and the application schema is included in only one world. In strict models for the replication concern all replicas have the same state, in the sense that any replica answers identically to requests. Strict models for the naming concern consider all names to be recognizable and identifying the same object in any world (names are universal and absolute). Isolation, where activities do not interfere, is the strict model for the concurrency concern. The strict model for the failure concern masks failures allowing designers to ignore faults. In the strict model for the configuration concern worlds and objects are statically connected with others. Finally, the strict model for the communication concern considers that distributed communications are equivalent to procedure calls, synchronous and without failures[4].

The development of distributed applications is usually centered in the concrete mechanisms offered by the platforms. For instance, some mechanisms have been proposed to support particular models, e.g. remote procedure call for a strict communication model.

nisms that hide distribution issues.

[4] Remote procedure call is a mechanism that implements this model.

4 Activities

The development process assigns an activity to each concern. Activities are responsible for choosing the right solution to concerns in the application context. Furthermore, activities should integrate the solutions in the application.

4.1 Representations

Activities can manipulate two kinds of representations: specification and code. Specifications describe solutions in a language and platform independent manner. Activities can generate code once target platforms and languages have been chosen. There is a close relationship between specification and code. Code implements the specification and can be automatically generated if tools are available. It would be desirable that both, specifications and code related to a given concern, were developed and tested independently of other concerns, thus allowing stepwise verification and testing (figure 3).

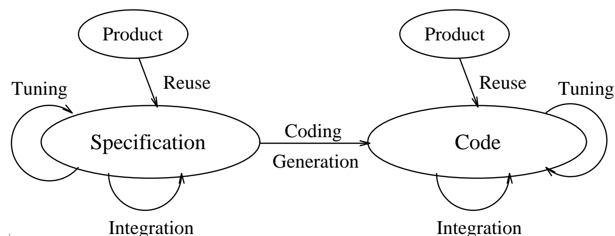

Figure 3: Activities progress.

Each activity can reuse products in specification and code. These products assist in concern solution since they hold some of the domain semantics.

Integration of solutions is twofold: integration of concerns and integration in the application. The former should consider solutions adopted for other concerns because they can interfere, in the sense that some solutions for a concern can restrict the set of possible solutions for another, e.g. dynamic configuration implies the existence of name spaces with special properties. Moreover, some tuning can be done between solutions by using a solution that integrates and optimizes several solutions. The latter integrates solutions such that changes to existing representations are minimal. These controlled changes allow representations to be traceable, i.e., it is possible to trace objects in

one representation to objects in another [18]. Furthermore, minimal changes allow for stepwise verification and testing: new inconsistencies and bugs can only be caused by the solution for the current concern.

4.2 Assistant Products

During development, predefined components, defined in specification or code, can be used as assistant products. These products can be concrete, ready to use solutions, or abstract, general solutions that need to be specialized, e.g. an object-oriented library and its specialization.

The existence of a set of concrete products with the help of integration tools permits a selective transparency of concerns to the designer. The designer only needs to choose which concerns he wants to consider and which he wants solved using default solutions of traditional models, policies and mechanisms.

4.2.1 Frameworks

A framework is an aggregation of collaborating classes with a common purpose, it constitutes a skeleton which should be fleshed out to build complete applications [45].

Concerns can be seen as domain problems and analyzed using domain analysis [42]. Frameworks assisting particular concerns were developed as result of concern domain analysis, e.g. study research done on replication consistency allows the design of a framework which abstracts several strict model policies and allows specializations of these policies to be integrated in the application. The framework defines an abstract interface that concrete specializations preserve thus allowing concrete solutions to be interchangeable.

Frameworks can be defined both in specifications and in code. Obviously, specification frameworks are language and platform independent. Frameworks use other assistant products in their construction as patterns and abstract mechanisms.

4.2.2 Patterns

A pattern is a grouping of classes and objects which can be applied in more than one problem domain [8]. Patterns are more than a solution, they describe when to apply themselves, the elements that make up the design solution and the trade-offs of applying the pattern [17].

Patterns are identified when developing frameworks. They constitute some of the frameworks building blocks. Patterns differ from frameworks because they are domain independent, e.g. patterns for recovery control could be used in replication, concurrency and failure concerns.

4.2.3 Abstract Mechanisms

Abstract mechanisms can be represented as classes that define a generic interface providing mechanism functionality. Activities use abstract mechanisms and an implementation with concrete mechanisms. Default implementations of abstract mechanisms permit rapid development and testing in the most convenient development platform, e.g. testing of concurrency control is done in a single address space. Abstract mechanisms support mechanism independence in the development process and permit deferring implementation decisions when finding solutions to concerns.

5 Process

The development process of distributed applications is described using the concepts and nomenclature defined in [27].

The development process can be divided into stages. A stage has a purpose, the stage goals, and describes a set of activities to carry out in order to achieve it. Stages use concepts in the generation of deliverables. Deliverables are representations of the application. Concepts can be either particular to one stage or common to several.

The stages we consider as part of the development of distributed applications are: analysis, logical distribution, durability, concurrency, robustness and physical distribution. Each one of these stages intends to enrich application functionality.

- **Analysis**. During analysis user requirements are defined. Requirements should define the models the user is expecting.
- **Logical distribution**. A distributed computing system is defined in [1] as a system of multiple autonomous (logical) processors that cooperate only by sending messages over a communication channel. Note that this definition does not distinguish between physically separated components and logically autonomous modules communicating via messages. The logical distribution stage designs the application as a set of logically distributed worlds that can communicate

only through messages [5]. Several concerns are involved in this stage: fragmentation, naming, replication, and configuration. Fragmentation ensures that the global application schema is the composition of several local schemas. Naming allows each world to have its own name space. Replication deals with the consistency of local replicas generated by fragmentation. Configuration handles world management.

- **Durability**. The durability stage provides the application with persistence support. It is necessary to consider the replication concern because transient and persistent replicas should be kept consistent. Moreover, the naming concern is relevant since storage offers a new naming space that must be integrated with other naming spaces.

- **Concurrency**. The concurrency stage provides the application with concurrency management and concurrency control. This stage considers concurrency and communication concerns. The whole application can be seen as a resource shared by users and solutions of the concurrency concern extended to the whole application. Some communication models are strongly related to concurrency, e.g. the mailbox paradigm requires an activity in the server.

- **Robustness**. After this stage, applications are able to handle and recover from failures. This stage considers failure, replication, naming and configuration concerns. Replication is used to support reliability. Naming offers identity to the set of replicas while configuration is needed when partitions occur.

- **Physical distribution**. Physical distribution turns logical distribution into a physical one. Mainly, abstract mechanisms should be implemented in each one of the platforms. All activities are involved in this stage since all the mechanisms they used need to be implemented.

The relationship between stages and concerns is depicted in figure 4.

Analysis utilize use case representations [18], describing user requirements, enriched with model descriptions, e.g. the degree of interference between use cases [39]. All other stages use an object-oriented notation for representations as in [9].

The designer can optionally skip some of the stages, e.g. some distributed applications do not need to be robust if they are not critical and can fail.

[5] In this stage the strict model, procedure call, is used

stages / concerns	logical distribution	durability	concurrency	robustness	physical distribution
fragmentation	X				X
replication	X	X		X	X
naming	X	X		X	X
concurrency			X		X
failure				X	X
configuration	X			X	X
communication			X		X

Figure 4: Stages and concerns.

Our approach proposes a flexible development, adjustable to designer needs. The development can be centered in the sequence of stages, where solutions for a concern are revisited in each stage being extended and integrated, or done concern by concern, where the activity takes into account all the functionality the application requires.

6 Concurrency Concern

To illustrate our proposal, the concurrency concern is chosen. This is an ongoing work [38].

Concurrency concern generates and controls access to shared resources. If the resource allows simultaneous activities, it is necessary to prevent undesirable interferences. This task is usually referred to as concurrency control. Our main work was centered in concurrency control.

6.1 Abstractions

There are several models of concurrency control where isolation is the strict model. Isolation ensures sequential execution in face of concurrency. Interleaving of actions is allowed if equivalent to their sequential execution. New requirements for non-strict models have been proposed in [3]: supporting long transactions, supporting user control, and supporting synergistic cooperation.

The strict model is supported by several policies. Concurrency control policies use two different perspectives [28]: pessimistic [19], when the application is expected to have high contention, and optimistic, when the level of contention is supposed to be low. Moreover, policies are distinguished by their synchronization primitives: locking [15] and timestamping [41]. Policies are divided [43] in dynamic, in which serialization order is determined from the order in which they access the objects, and static, in which serialization order is based on a predefined total order. The

combination of this three aspects generates well know policies as, for instance, two-phase locking, which is pessimistic, dynamic and locking.

Concurrency control includes mechanisms that represent activities and which allow them to communicate and synchronize.

Besides the levels of independence between models, policies and mechanisms it is possible for different policies to coexist within the same applications (policy heterogeneity) and different implementations for abstract mechanisms (platform heterogeneity). The coexistence of heterogeneous policies is feasible [30].

6.2 Activities

The models and assistant products used in this activity have been described in [38]. A domain analysis was done of the concurrency control domain and consequently a framework, patterns and abstract mechanisms were defined.

6.2.1 Assistant Products

The framework establishes a uniform interface to concurrency control policies for the serial model (serializability policies). Particular policies are built by framework-derivation. Policies are constructed redefining three characteristics: transactions execution order definition time, object accesses consistency check time, and synchronization primitives to be used (locking or timestamping).

The set of objects that belong to the same world share the same policy but different worlds can use different policies. The framework permits the coexistence of several concurrency control policies within the same application. An inter-world invocation creates a transaction in the invoked world, sub-transaction of the invoker transaction, thus generating a set of nested transactions. To commit a transaction, a two-phase commit protocol verifies if local orders of execution are compatible, aborting the transaction otherwise.

In order to control concurrent accesses to objects, accesses must be trapped. An object manager is defined for each object and all invocations are done through this new object.

The framework uses patterns for recovery control, either based on versions or replicas. However, patterns are not independent of the particular concurrency control policy: optimistic policies require recovery control using replicas, while pessimistic policies require versions. The designer must choose the correct pattern for the policy under going implementation.

The abstract mechanisms we needed were *Activity* and *MutualExclusion*. At the physical distribution stage we used *Thread* or *RPC* to implement *Activity* if the prototype was centralized or distributed respectively.

6.2.2 Representations

Concurrency control specifications and code are constructively integrated with previously developed ones such that traceability [18] is achieved in both code and specifications. The integration does not interfere with results of previous activities. This is achieved if object managers and transactional invocations are carefully defined.

Figure 5 depicts a transactional invocation, we enriched the Fusion [9] notation with double arrows for distributed invocations, and its counterpart after the concurrency control stage.

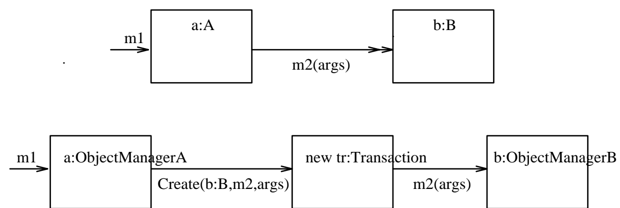

Figure 5: Transactional invocation.

Figure 6 shows the code after the logical distribution stage. All transactional invocations are traced in the code by new operations, *ADistCallB* in this example. Introducing code for effective transaction generation requires only redefinition of these new operations.

```
void A::m1()
{
    ...
    ADistCallB(args);
    ...
}

void A::ADistCallB(args)
{
    b->m2(args);
}

void B::m2(args)
{
    ...
}
```

Figure 6: Code generated after logical distribution.

In order to intercept invocations, object managers must have the same interface of the objects they encapsulate. We define object managers without changing the client class using the envelope/letter idiom

described in [10]. Object manager classes use the original names and redirect invocations to instances of the original classes (now renamed).

A set of default policies are available and can be automatically introduced into specifications and code. Afterwards, policies can be replaced by new ones. This approach allows rapid application development and future tuning by a programmer which has specific knowledge of application semantics. Tools are being constructed that automatically generate the envelopes and include concurrency control code [24].

An implementation was done in C++[40] on UNIX and using DCE threads package. The implementation allows the construction of centralized prototypes with logical distribution which can be debugged without interference of communication faults, errors or delays.

7 Related Work

Recently, researchers have raised the need for the support of non-strict models. In [32] it is stated that computer support for cooperative working (CSCW) needs open distributed systems with a clean separation of mechanisms and policies, tailored mechanisms and policies. In [3] the new requirements for concurrency control are described: supporting long transactions, supporting user control, and supporting synergistic cooperation. [3, 13] describe several transactional systems and models that support non-strict models. Multidatabases need to maintain consistency of related data; [34] proposes the specification of constraints among multiple databases defined in terms of time, data state, and operation occurrence.

Pu describes the problem of heterogeneity of policies in superdatabases and proposes a solution for the integration of different concurrency control policies for the serial model [30].

The degrees of transparency [11] that have been proposed are supported by system mechanisms [29, 6] implementing the strict models. The system is fully transparent if traditional mechanisms are completely hidden by the new mechanisms.

Several systems offer support for the development of distributed applications [7]. In particular, systems like Argus [20], Avalon [14], Arjuna [37] and Hermes [16] offer different architectures to support distribution issues. Argus offers a modular concept and encapsulation. Avalon and Arjuna offer a set of object-oriented abstract classes which provide distribution functionality and, in Hermes, the functionality is set by invocation parameters.

From the Open Distributed Processing (ODP) reference model [31] point of view applications should be developed using several perspectives: enterprise, information, computational, engineering, and technology. The engineering perspective is in charge of distribution policies while technology uses mechanisms. During the development process, entities in a perspective are re-implemented using lower level perspectives. Since the distribution is transparent for the other perspectives it is not clear where models of distribution concerns should be specified.

In [21], pre- and post-conditions are used to define user-oriented specifications of distributed applications free from implementation details but with performance requirements. This approach is centered in the specification of models and the specification of non-strict models results from relaxing strict models which are the expected.

"Pictures that play" [5] are design diagrams to explore alternative designs without the need of complete and consistent models. The development process is centered on the system functionalities and the emphasis is put on the concerns of concurrency and communication.

DOCASE [26] abstracts common design elements and separates distribution aspects from application algorithms using superimpositions. Design elements include the concept of configuration element and in particular the concept of logical node. Relation types specify complex semantic relationships as interaction, collocation and cooperation. A repository, holding design artifacts, is used to the reuse design elements. Superimpositions allow a constructive development. Reflection is used to deal with dynamic configuration [46].

The Conic environment [22] separates the configuration facilities from the programming language, offering dynamic configuration capabilities. Logical nodes are configured by the configuration language and represent autonomous and reusable entities. The development process is constructive and dynamic configuration is the solution for modification and evolution. Heterogeneity is supported by an abstract runtime which is defined using Conic language.

The fragmented object model [23, 12] offers two different levels of abstraction: transparency of distribution issues for clients, and relevance of distribution issues for designers. Objects, when split in several address spaces, have a fragment in each one of them. The designer can define policies and mechanisms by programming fragments. Default fragments are offered for common policies and mechanisms.

PROTOB [2] uses high-level Petri nets and

dataflows to model objects of an event-driven system. To develop an application, an executable model of the system, the set of its objects, is built to allow interactive validation of system's behavior. Afterwards, the model is emulated taking into account implementation details, and finally application code is generated for the real environment.

Schmidt proposes a framework [36] which simplifies the development, configuration and reconfiguration of applications. It provides a collection of reusable components for communication, service configuration, and concurrency. In particular a pattern, called reactor [35] for demultiplexing and dispatching of multiple event handlers, is extensively used.

Object-oriented analysis and design methods, e.g. [33, 18, 9], are becoming widely used in the development of applications. Nevertheless, developing applications for distributed object-oriented systems involves issues that suggest deep changes in current methods [9]. The method described in [4] treats design in depth and has a notation capable of representing several kinds of concurrency semantics and of object invocation.

8 Conclusions

Our development process allows constructive development of applications and the traceability of development from specifications to code. It integrates with a global development process, from analysis to implementation, and consistently relates the levels of abstraction of distribution issues to the stages of the development process. Development is flexible, because it can be either centered on concerns or on stages.

The requirements of CSCW presented in [32] are fulfilled by our development process. Moreover, we introduce the model abstraction independently from policy abstraction, thus allowing application development using heterogeneous policies [30].

Our approach is not formal, but the three-level distinction of abstractions allows a consistent development. During analysis the models are defined in a user-oriented perspective free of implementation details [21].

The global architecture we propose is centered on the design of generic solution components to solve specific concerns. Each of these generic solutions is described by object-oriented frameworks from which concrete solutions are obtained by specialization.

Acknowledgments. We would like to thank João Pereira for the work done on concurrency control and Ellen Siegel for preliminary discussions on this subject. We also thank David Matos for his careful proofreading.

References

[1] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[2] Marco Baldassari, Giorgio Bruno, and Andrea Castela. PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems. *Software - Practice and Experience*, 21(8):823–844, August 1991.

[3] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[4] Grady Booch. *Object-Oriented Analyis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[5] R. J. A. Buhr. Pictures That Play: Design Notations for Real-time and Distributed Systems. *Software - Practice and Experience*, 23(8):895–931, August 1993.

[6] D. R. Cheriton. The V Kernel: A software base for distributed systems. *Comm. of the ACM*, 31(3):314–333, March 1988.

[7] Roger S. Chin and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–123, March 1991.

[8] Peter Coad. Object-Oriented Patterns. *Communications of the ACM*, 35(9):152–159, Spetember 1992.

[9] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[10] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Publishing Company, 1992.

[11] G. F. Coulouris and J. Dollimore. *Distributed Systems: Concepts and Design*. Reading, Addison-Wesley, 1988.

[12] Peter Dickman and Mesaac Makpangou. A Refinement of the Fragmented Object Model. In *Second International Workshop on Object-Orientation in Operating Systems*, Dourdan, France, September 1992.

[13] Ahmed K. Elmagarmid. *Database Transaction Models*. Morgan Kaufmann, 1992.

[14] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[15] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Comunications of the ACM*, 19(11):624–633, November 1976.

[16] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Concurrency control for distributed nested transactions in hermes. *International Conference for Concurrent and Distributed Systems*, 1993.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[18] Ivar Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[19] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions Database Systems*, 6(2):213–226, June 1981.

[20] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[21] Barbara Liskov and William Weihl. Specifications of Distributed Programs. *Distributed Computing*, 1(118):102–118, 1986.

[22] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[23] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented Objects for Distributed Abstractions. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1992.

[24] David M. M. Matos. Técnicas para a Programação de Aplicações Distribuídas e Persistentes num Modelo de Objectos Uniforme e Transparente. Tese de mestrado, Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa, Portugal, Setembro 1994.

[25] David Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, January 1993.

[26] Max Muhlhauser, Wolfgang Gerteis, and Lutz Heuser. DOCASE: A Methodic Approach to Distributed Programming. *Communications of the ACM*, 36(9):127–138, September 1993.

[27] OMG. *Object Analysis and Design: Comparation of Methods*. John Wiley, 1994.

[28] M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[29] G. Popek and B. J. Walker. The Locus Distributed System Architecture. Technical report, Cambridge. MIT Press, 1985.

[30] Calton Pu. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the 4th Data Engineering Conference*, pages 548–555, Los Angeles, USA, February 1988. IEEE.

[31] K. A. Raymond. Reference Model of Open Distributed Processing: a Tutorial. In *International Conference on Open Distributed Processing*, pages 3–14, Berlin, Germany, 1993.

[32] Tom Rodden and Gordon Blair. CSCW and Distributed Systems: The Problem of Control. In *Proceeding of the Second European Conference on Computer-Supported Cooperative Work*, pages 49–63, Amsterdam, The Netherlands, September 1991.

[33] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[34] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *Computer*, 24(12):46–53, December 1991.

[35] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Jim Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[36] Douglas C. Schmidt and Tatsuya Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Systems. *BCS/IEE Distributed Systems Engineering Journal*, 1995.

[37] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An overview of the arjuna distributed programming system. *IEEE Software*, January 1991.

[38] António Rito Silva, Jo ao Pereira, and José Alves Marques. A Framework for Heterogeneous Concurrency Control Policies in Distributed Applications, July 1995. Submitted for publication.

[39] António Rito Silva and Ellen Siegel. Specifying Concurrency and Replication for Application Development. Technical Report RT/066/94-CDIL, INESC, 1994.

[40] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991.

[41] Robert H. Thomas. A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(24):180–209, June 1979.

[42] Steven Wartik and Ruben Prieto-Diaz. Criteria for Comparing Reuse-Oriented Domain Analysis Approaches. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):403–431, 1992.

[43] William Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[44] William Weihl. Transaction-Processing Techniques. In Sape Mullender, editor, *Distributed Systems - Second Edition*, pages 329–352. Addison-Wesley, 1993.

[45] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[46] Christian Zeidler and Bernhard Frank. Integrating Structural and Operational Programming to Manage Distributed Systems. In *Object-Based Distributed Programming*, pages 55–72, Kaiserslautern, Germany, July 1994. Springer-Verlag.