Dagstuhl Seminar
on
Object Orientation with Parallelism and Persistence

Organized by

Burkhard Freitag (Universität Passau)
Clifford B. Jones (University of Manchester)
Christian Lengauer (Universität Passau)
Hans-Jörg Schek (ETH Zürich)

Schloß Dagstuhl 3. – 7.4.1995

Contents

1 Preface

The seminar was devoted to the study of the object-oriented programming paradigm with a focus on parallelism. This topic is located in the intersection of two research areas: programming languages and databases. The next paragraphs sketch the rôle of object-orientation in both areas.

In *programming languages*, object orientation extends the concept of *abstract datatypes* as in the class concept of Simula. An abstract datatype joins a value domain with a set of operations that can be applied (exclusively) to values of the domain. Abstract data objects are declared statically and are protected from direct access by other program parts (encapsulation). The principle of object orientation is rooted in the desire to modify datatypes (through restriction or extension), or to construct new datatypes from existing ones. It suggests concepts like *inheritance* of properties (attributes) or operations (methods) and *dynamic management*, i.e., creation and destruction of objects at run time.

In *databases*, the *persistence* of objects (i.e., their existence across program runs) and the *concurrent access* of objects are of prime importance. Also in databases, entities of the real world are often modelled directly as objects. Since the identifiability of fixed objects by means of their state cannot be guaranteed for long time intervals, the concepts of *object identity* and *object evolution* play a more central rôle than in programming languages.

Inter- and intra-object *parallelism* have received an increasing amount of attention in the last years by researchers in the area of object-oriented programming; concurrency is also of great relevance to databases. At first glance, an object is very similar to a *process* which offers services to other processes and demands services from them. It has turned out, though, that object-oriented concepts cause problems when combined with parallelism. In databases, the combination of object orientation and parallelism requires a generalization of the transaction model.

The main goal of the seminar was to put the new research area "object orientation with parallelism" on an interdisciplinary basis from the outset. One prerequisite for a precise exchange of ideas is the use of formal models and methods.

The seminar provided a forum for the discussion of questions like:

- How can the concepts of object in programming languages and databases be unified? In what cases is a unification desirable?
- Several new models of dynamic parallelism have emerged in the area of programming languages (actors, concurrent rewriting, π-calculus, etc.). What is their relevance for relational and object-oriented databases?

- What is the relevance of newly developed correctness criteria in databases, like semantic or abstract serializability, for parallel programming and which logical concepts used in object bases can be incorporated into object-oriented programming languages?
- What transaction model is suitable for object-orientation with parallelism? How can the properties of a transaction be adapted flexibly to the semantics of the object types involved?
- What kind of technical problems arise when combining object orientation and parallelism and how can they be solved?
- What requirements do applications have on both areas?

The 32 participants of the workshop came from 8 countries: 13 from Germany, 15 from other European countries and 4 from the United States. The organizers would like to thank everyone who has helped to make this workshop a success.

Burkhard Freitag Clifford B. Jones Christian Lengauer Hans-Jörg Schek

2 Abstracts

There's Nothing Like Shared Nothing

Peter Thanisch
Edinburgh University, UK

An unholy alliance of DBMS vendors' marketing departments has given undue prominence to a parallel database architecture called "shared nothing". In this architecture, each processor has its own memory and its own disks; interprocessor communication is via message passing in the interconnect. Although originally intended for parallel relational databases, shared nothing has already started to impact on parallel object-oriented DBMSs. To make matters worse, the dominance of relational DBMS in the market place means that parallel OODBMS projects might have to adapt to this architecture. Fortunately, hardware vendors are realising that many of the claimed benefits of shared nothing are illusory. Some recent products have shifted away from this idea. Perhaps the academics and DBMS vendors will come to this realisation before long.

A Composition Language for Open Applications

Oscar Nierstrasz
University of Berne, Switzerland

A *composition language* supports the development of open, evolving applications from standard, generic software architectures and open, plug-compatible software components. By making application architectures explicit and manipulable, disciplined evolution according to the composition model of a generic software architecture is facilitated. A software entity is a *component* if it is designed to be used in multiple contexts according to a particular composition model. Although objects may be components, in general components may be of either coarser or finer granularity than objects. Whereas objects can be viewed as message-passing processes, components are abstractions over the object space that can be composed using a variety of different composition mechanisms as defined by a generic software architecture. We are developing a general framework and a corresponding language

for specifying composition models, generic components and software compositions.

Generating Application-Specific Database Systems

Don Batory
The University of Texas at Austin, USA

Years from now, the subjects of object-oriented software design, persistence, concurrency, parallelism, and their relationships to programming languages will be so well understood that it will be possible to automate the construction of software with these features for families of applications. This will be accomplished through the use of software system generators.
¿From our experience, such understandings require paradigm shifts. For example, units of software larger than that of classes are the preferred system building blocks, generators rely on a minimal use of inheritance and make extensive use of parameterization, and the programming paradigm of generators is based on program transformations, not object-orientation.
We discuss the role and importance of generating lightweight database systems, and present experimental results illustrating the potentials of software generators. Unifying the themes of this seminar may require a similar shift in paradigms. We offer our work on generators as an example.

Exception-Handling with Future Evaluators

Trevor Hopkins
IBM UK Ltd., UK

Many commercial client-server systems use a hybrid approach, using relational DBMS technology on their server(s) and object technology (using, for example, Smalltalk as a programming language) on the clients. Such systems are frequently slow, as perceived by the user, since the round-trip time to the server may be several seconds.
One approach is to use *concurrency* on the client, to allow an interactive request to the server to be overlapped with local operations, such as the creation of a window to display the results of the request. A convenient way of expressing this concurrency is to use *future* evaluation, since the complexity of resynchonization can be (longly) hidden from the programmer. It is well

known how to implement future evaluators in Smalltalk, using the reflective capabilities of the language.
Unfortunately, client frameworks which support communication frequently raise *exceptions*; for example, a network failure. For robot systems, these exceptions must be handed correctly. Some exceptions can be entirely handled in the future thread, but others will have to be passed to the original thread, at the point where the future result is required. This assists in preserving the illusion that only a serial system is in use, and thus simplifies the client programming tasks.

Query Optimization in Parallel Databases

Johann-Christoph Freytag
Humboldt-Universität Berlin, Germany

With the increasing use of parallel processors we have to adjust and extend current database management systems to new requirements. This is especially true for the query optimizer in a DBMS which must produce an optimal (or at least a very good) query evaluation plan (QEP) for the parallel execution environment.
In our presentation we first discuss the traditional approach to query optimization and how it has been extended to the parallel environment by simply adding two additional processing steps: the first one takes a (sequential) QEP and divides it up into several (partial) QEPs, thus creating a parallel QEP (pQEP) which is allocated to a set of processors in the second step. It is well known that this approach does not produce optimal plans for parallel execution. We then identify important problems in query optimization that must also be addressed namely the problems of considering many more QEP alternatives than before, different plans and more operators for getting finer granularity for parallel execution and of considering alternative solutions to static (compile-time) optimization. We describe several possible solutions to the problems mentioned: dynamic (run-time) optimization, parallel optimization, dynamic index selection, and sampling. Finally, we argue that the approach taken in database systems for taking advantage of parallelism is preferable to many others. Providing a declarative language is a good (the best?) way of hiding the complexity of parallel execution environments from the applicaton programmer. It is the responsibility of the underlying database system to generate an efficient parallel program.

Concurrent Object-Oriented Programming in Oz

Christian Schulte
DFKI Saarbrücken, Germany

Oz is a higher-order concurrent constraint programming system developed at DFKI. It combines ideas from logic and concurrent programming in a simple yet expressive language. From logic programming Oz inherits logic variables and logic data structures, which provide for a programming style where partial information about the values of variables is imposed concurrently and incrementally. A novel feature of Oz is the support of higher-order programming without sacrificing that denotation and equality of variables are captured by first-order logic. Another new feature of Oz are cells, a concurrent construct providing a minimal form of state fully compatible with logic data structures. These two features allow to express objects as procedures with state, avoiding the problems of stream communication, the conventional communication mechanism employed in concurrent logic programming.
Based on cells and higher-order programming, Oz readily supports concurrent object-oriented programming including object identity, late method binding, multiple inheritance, "self", "super", batches, synchronous and asynchronous communication.
A paper on the work reported in the talk appears as: Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, Chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995.

Concurrent Objects in Rewriting Logic

José Meseguer
SRI International, USA

Rewriting logic is a logic of action in which states are axiomatized as elements of an algebraically specified data type. Therefore, each state can be viewed as an equivalence class $[t]_{\Vdash}$, where t is a term, and $\in$ is a set of equations.
In particular, concurrent object-oriented systems can be so specified and programmed. Typically the distributed state of such a system has the structure of a multiset of objects and messages. Therefore $\in$ includes in this case associativity and commutativity of multiset union.
Concurrent computation coincides with logical deduction in rewriting logic. This led us to:

1. design a specification language called Maude, based on rewriting logic,
2. design a subset called Simple Maude that can be efficiently compiled on MIMD, SIMD and MIMD/SIMD machines.

Coordination Abstractions for Concurrent Object-Oriented Programming

Gul Agha
University of Illinois at Urbana-Champaign, USA

The problem of coordination collections of objects is fundamental to concurrent systems. Current generations of concurrent programming languages are very low level in their ability to abstract over interaction patterns. This limits reusability, makes reasoning unnecessarily complex and puts an unreasonable burden for optimizing code on the programmer.
We define a primitive model of concurrency based on Actors. We then discuss local synchronization constraints. Synchronization constraints separate "how/what" from "when". When some activity happens is expressed in terms of what but is not part of the representation. We then describe multi-object synchronisation abstraction in terms of synchronizers which encapsulate declarative constraints. We also discuss ways of defining groups of actors abstractly. Finally, we discuss how explicit manipulation of a meta-level architecture allows reusability of fault-tolerant protocols which otherwise need reimplementation.

Advanced Models of Database Transactions

Gottfried Vossen
Universität Münster, Germany

Transactions are a central paradigm for the synchronization and fault tolerance of activities in database systems. In this area, theory and practice influence each other a lot. The program (or methodology) for designing a transaction model and correctness criteria for concurrent transaction execution (schedules) is vastly uniform even in distinct contexts. We descibe this methodology and illustrate it using various concrete examples, ranging from simple reads and writes on pages to semantically rich operations. We also survey recent transaction models and their correctness criteria, whose goal is

to adequately support a variety of requirements arising in modern database applications. We conclude with a brief discussion of the emerging issue of workflow management.

Unified Theory for Classical and Advanced Transaction Models

Haiyan Hasse, Y. Breitbart, H.-J. Schek, R. Vingralek, G. Weikum
ETH Zürich, Switzerland

The classical theory of transaction management is based on two different and independent criteria for the correct execution of transactions. The first criterion, serializability, ensures correct execution of parallel transactions under the assumption that no failures occur. The second criterion, strictness, ensures correct recovery from failures.
In this talk we present a unified model that allows reasoning about the correctness of concurrency control and recovery within the same framework. We introduce the correctness criteria of (prefix-)reducibility and (prefix-)expanded serializability and show their relationship to the classical criteria. We investigate further the exact characterization of the class of prefix reducible schedules and show that the exact characterization of the class of prefix reducible schedules for the simple read/write model is serializable with ordered termination (SOT). However, SOT is not feasible anymore if we consider schedules with semantically rich operations. Thus, we study this problem by following the two approaches: restricting the class of prefix reducible schedules and imposing some restrictions on a commutativity relation. We investigate the first approach and propose here two subclasses of prefix reducible schedules, forward safe and backward safe schedules, and argue that serializability and atomicity can be unified by considering schedules from these classes. We introduce further special properties of commutativity relations by following the second approach.

Concurrency Control and Recoverability in Active Database Management Systems

Andreas Geppert
Universität Zürich, Switzerland

Active database management systems (ADBMSs) support reactive behavior in the form of event-condition-action rules (ECA-rules). If an event occurs

(in the database or its environment) and the condition (a predicate on the database state) is true, the corresponding action must be executed. Advanced ADBMSs such as SAMOS support composite events, which occur when their component events have occurred within some time interval. Since component events can span several transactions or even sessions, the history of component events must be made persistent. The persistence of the event history leads to several problems with respect to transaction management. First, concurrency control for the event history must be provided, since concurrent transaction may raise events, resulting in possibly conflicting updates of the event history. Secondly, upon transaction abort, it must be ensured that the event history is in a consistent state, i.e., does not contain event occurrences that have been raised by aborted transactions. We give an introduction of our object-oriented ADBMS SAMOS and illustrate three possible solutions of the aforementioned problems in the talk. The first one is currently implemented in SAMOS and uses the transaction model of the underlying (passive) object-oriented DBMS: the strict two-phase locking protocol for closed nested transactions. It is not yet optimal with respect to the potential degree of parallelism of triggering transactions. The second solution restricts the ECA-rule expressiveness; it permits a higher degree of parallelism and allows a simple recovery algorithm. The final solution exploits the semantics of the event detectors and proposes semantic concurrency control together with multi-level transactions. In comparison to the first two approaches, it allows the highest degree of parallelism, but also requires the most complex recovery algorithm.

Concurrent Transaction Logic

Michael Kifer
SUNY at Stony Brook, USA

Sequential Transaction Logic provides a framework for tasks ranging from transaction specification and execution in databases to view updates, to triggers in active databases, to discrete system simulation, to robot planning and procedural knowledge in AI.
In this talk, we describe Concurrent Transaction Logic, which extends Transaction Logic with connectives for modeling the concurrent execution of complex actions. The concurrent actions execute in an interleaved fashion and can also communicate and synchronize themselves. All this is provided in a completely logical framework, including a model theory and a proof theory. Moreover, the framework is flexible in that it can accommodate many different semantics for updates and for databases. For instance, not only can

updates insert and delete tuples, they can also insert and delete null values, or rules, or arbitrary logical formulas. Likewise, not only can databases have a classical semantics, they can also have a well-founded semantics, a stable-model semantics, etc. Finally, the proof theory for Concurrent Transaction Logic has an efficient SLD-style proof procedure, i.e., a Prolog-style inference system based on unification. As in the sequential version of the logic, this proof procedure not only finds proofs, it also executes concurrent transactions, finds their execution schedules, and updates the database.

A Programmer's Introduction to the ß-Calculus

Benjamin C. Pierce
University of Cambridge, UK

Milner, Parrow and Walker's π-calculus is seeing increasing use of a formal foundation for high-level concurrent programming idioms such as objects. We introduce the syntax, operational semantics, and some simple examples showing how to "program" in this setting.

Concurrent Objects in a Process Calculus

Benjamin C. Pierce and David Turner
University of Cambridge and University of Glasgow, UK

A programming style based on concurrent objects arises almost inevitably in languages where processes communicate by exchanging messages. Using the PICT language as an experimental testbed, we introduce a simple object-bases programming style and compare three techniques for controlling concurrency between methods in this setting: explicit locking, a standard choice operator, and a more refined "duplicated choice" operator.

Behavioural Equivalences: From Testing to Bisimulation

Davide Sangiorgi
INRIA-Sophie Antipolis, France

Two forms of behavioural equivalences for concurrent processes have emerged and become predominant: testing-based equivalences and bisimulation-based equivalences. We present and compare these two groups, discussing major advantages and disadvantages.

A Model of Concurrent Objects and its Semantics

Matthias Radestock
Imperial College of Science, Technology and Medicine, London, UK

The appeal of the object-oriented paradigm lies in its apparent simplicity. The major motivation for designing concurrent object-oriented languages is the advantage that is achieved by conceptually unifying the abstraction for processor and memory. While the object-oriented paradigm essentially is a model for *data abstraction*, processes are models for *control abstraction*. Concurrent object-oriented languages have to be based on object models that deal with issues of concurrency and distribution in a natural way. Viewing objects in a very abstract sense, it can be observed that these notions are implicitly embedded - objects exist simultaneously, are potentially distributed and act in parallel. However, a refinement of this very abstract model is necessary in order to actually derive design methods and implementation strategies. Traditionally object-oriented systems were sequential systems. As a result aspects of concurrency and distribution were lost during the refinement process. Many universally applicable concepts and ideas of designing and implementing sequential object-oriented systems therefore conflict with those aspects and can no longer be applied in the context of concurrent systems.
The aim of our research was to devise an object model that would become a basis for the implementation of concurrent object-oriented systems and the development of design methods. Starting from the abstract object model a refinement was carried out that preserved the inherent aspects of concurrency and distribution in the original model. Objects in the model are viewed as *nodes* in a decomposition hierarchy. This hierarchy serves as a *logical communication topology* – objects can only communicate with their sub-objects and container object. Objects act as *routers* for other objects and forward their messages. The route to an object uniquely identifies the object and can thus be used as an *object identifier*. Absolute object identifiers correspond to routes from the root of the decomposition hierarchy whereas relative object identifiers correspond to communication paths between objects. In many applications most communication between objects is local with respect to the decomposition hierarchy. In our model this locality is preserved – the routes

to such objects are shorter than others. Objects have an associated *behaviour script* that can be set, retrieved and evaluated. Evaluation takes place within the context of an object. Attributes and methods are both viewed as special subobjects. The local variables of methods and their formal parameters are all subobjects of the method object. Objects are *cloned* for evaluation to enable concurrent method invocations and recursion. Inheritance between objects can be modelled using the *recipe-query scheme* in which the script of a method is fetched from the parent object and evaluated locally.
The semantics of our object model has been described in the π-calculus. The calculus provides us with means of controlling the fine-grain (i.e., intra-object) and large-grain (i.e., inter-object) concurrency. Messages to an object are processed in parallel if this does not change the semantics of *in-sequence processing*. Based on the model a small language was designed and its syntax and semantics were defined. It can be viewed as a kind of *object-oriented assembler language*. Features of high-level object-oriented languages can be translated easily into the basic language.

Two Ways of Defining the Semantics of an Object-Based Language

Clifford B. Jones
University of Manchester, UK

I argue that concurrent object-based (or object-oriented) languages are a suitable target for a compositional design method that copes with the interference inherent with concurrency. They are more tractable than full shared variable languages and more realistic than process algebras. The development method that I have proposed elsewhere uses equivalences (and rely/guarantee-conditions for more complex cases). These equivalences need to be justified with respect to a semantics for the language used in the design process (known as $\pi o \beta \lambda$). An structured operational semantics and a semantics given by mapping to the π-calculus are both outlined and their usefulness is discussed.

Concurrent Object-Oriented Design Specification in SPECTRUM

Friederike Nickl, Martin Wirsing and Ulrike Lechner
Universität München, Germany

An algebraic approach to formal object-oriented design specification is presented where the static and functional part of a software system is described by classical algebraic specification whereas the dynamic behavior is modeled by a transition relation. The approach is inspired by Astesiano's SMoLCS formalism and is based on Meseguer's rewriting logic; but it has two additional features, which are motivated by pragmatic object-oriented software development techniques: it supports the construction of subsystems, and the flow of messages can be controlled by use of a simple but powerful concurrent language. Liveness and safety properties of design specifications are formulated with the help of structured message expressions; methods for proving such properties are briefly discussed. As underlying specification language Broy's SPECTRUM is extended by features for concurrent object-oriented specification.

From Actions to Transactions – Refinement in Object-Oriented Specification

Grit Denker and Hans-Dieter Ehrich
TU Braunschweig, Germany

The focus of our work is the stepwise design of information systems, i.e., databases with application programs, in the object-oriented framework. Here, three keywords show up which make our work settled in the intersection of different fields: object-orientation, design process, and database.
The notion of object comprises structure and behavior. Therefore, applying refinement techniques during the design process implies that we have to deal with data refinement and action refinement. Action refinement means that an action which is atomic from the abstract point of view becomes compound from the refined point of view. Changing the level of granularity leads to well-known problems which have been thoroughly investigated in database concurrency control theory.
We propose a logic which serves as a domain for translating object-oriented specifications and interpret this logic in a model for families of concurrent objects. This model is based on the notion of labelled event structures and provides full concurrency though we use linear temporal logic for describing object systems. This is achieved by a principle called local sequentiality.
Especially, we introduce the concept of transaction in the logic and give it a specific semantics in the event-based model. This way, we treat transactions accordingly to the notion in database theory and leave freedom for possible interleavings, i.e., a sequence of abstract actions is refined to a interleaving of the corresponding transactions.

Finally, we come up with correctness criteria for refinement of object-oriented specification.

Behavioral Refinement of Object-Oriented Specifications Using the Modal ¯-Calculus

Ulrike Lechner
University of Passau, Germany

The specification language Maude provides powerful mechanisms for specifying synchronization and communication in a collection of objects. The modal μ-calculus enables us to reason about the properties of the behavior of a collection of objects in a property-oriented way. We develop ideas of behavioral refinement and abstraction of object-oriented specifications with respect to properties formulated in the μ-calculus. Results on the preservation of properties support certain design decisions made in Maude like the use of asynchronous message passing, the rewriting calculus and the inheritance relation.

Agent Based Coordination

J. M. Andreoli
Rank Xerox Research Center, Maylan, France

Coordination is concerned with environments consisting of possibly distributed, possibly heterogeneous software components. Heterogeneity is dealt with by the notion of "wrappers", which are pieces of software which encapsulate heterogenous components and present a uniform interface to the outside world. Coordination is then realized by interactions among these homogenized wrappers. The Linda model provides an interesting model for the interactions required by coordination (based on a blackboard-style communication). However, limitations of this model appear in complex coordination behaviours as found, for example, in transactional workflows. Traditional transaction systems are unfortunately also inappropriate for these situations, where not only data but also actions are distributed, both physically and conceptually. A framework based on the notion of "objects as resource handlers" is proposed as a first step in the direction of realizing transactional workflow coordination. A rule based scripting language is described.

Modelling General Relationships in Object-Oriented Databases

Jürgen Schlegelmilch
Universität Rostock, Germany

I promote the use of a mechanism to model general relationships in object-oriented systems by extending the approaches found in the literature by derived relationships, additional communication control features and for databases, a persistency concept based on roles (= attributes of relationships) as opposed to reference-based persistence. This persistency concept can, in conjunction with derived relationships and utilisation of the meta-schema, simulate all other persistency concepts found in commercial systems as well as research prototypes, including persistency by reachability, by class-membership and manually managed persistency. So, for databases, general relationships are a more powerful drop-in replacement for references.

Type Inference with Subtyping: State of the Art

Jens Palsberg
Aarhus University, Denmark

Recently, type inference with subtyping has been studied intensively. Many algorithms have been developed for typing such constructs as functions, records and objects. Moreover, lower bounds for several of the type inference problems have been obtained. In this talk, I will survey most known results, and I will compare them with results for type systems without subtyping.

Verifying Substitutability of Collaborating Objects

Else K. Nordhagen
University of Oslo, Norway

The talk presented an object-oriented calculus which directly models object-oriented concepts: object identity, instance variables, methods, object creation, encapsulation, inheritance and messages sent to objects, as opposed to functional concepts of values and function applications and process models of signals on channels and spawning of processes.

The talk presented results from applying the calculus to define and reason about congruence relations between objects and components built from objects. The results include a study of properties of versions of such relations in relation to compositionality of objects and components. This defines requirements on component specification and implementation to assure that component based systems function as anticipated.

Confluent Processes for Transformation Correctness

Uwe Nestmann and Martin Steffen
Universität Erlangen, Germany

Program transformations are central for the correct development of parallel object-based programs in the language $\pi o \beta \lambda$. A π-translation provides a formal semantics for $\pi o \beta \lambda$. In order to prove the correctness of program transformations for the example program of a symbol table, a theory of confluent mobile processes is established and applied.

3 List of Participants