CONCURRENCY MADE EASY

Bertrand Meyer

ERC Advanced Investigator Grant project, 2012-2017

(Submitted February 2011)

Project Description

This document contains the essential elements of the 2.5-million Euro Advanced Investigator Grant proposal successfully submitted to the European Research Council.

It includes both the Extended Synopsis (pages 5 to 9) and the full scientific proposal (pages 10 to 25). The 5-page description of the Principal Investigator's background (CV, awards etc.) has been removed, as well as the detailed budget table. The contribution of Sebastian Nanz to the preparation of the project is gratefully acknowledged, as well as the help of Carlo Furia and Claudia Günthart.

The information presented here is public and is copyright Bertrand Meyer, 2011. If you have any question on the project please feel free to contact Prof. Meyer.

SEVENTH FRAMEWORK PROGRAMME

"Ideas" Specific programme

European Research Council

Project acronym: **CME**

Project full title: **Concurrency Made Easy**

Duration: **60 months**

Principal Investigator: Prof. Bertrand Meyer

Host Institution: ETH Zurich

European Research Council

ERC Advanced Grant 2011
Annex I – Description of Work (DoW)

Concurrency Made Easy

(CME)

The "Concurrency Made Easy" project is an attempt to achieve a conceptual breakthrough on the most daunting challenge in information technology today: mastering concurrency.

Concurrency, once a specialized technique for experts, is forcing itself onto the entire IT community because of a disruptive phenomenon: the "*end of Moore's law as we know it*". Increases in performance can no longer happen through raw hardware speed, but only through concurrency, as in multicore architectures. Concurrency is also critical for networking, cloud computing and the progress of natural sciences. Software support for these advances lags, mired in concepts from the 1960s such as semaphores. Existing formal models are hard to apply in practice.

Incremental progress is not sufficient; neither are techniques that place the burden on programmers, who cannot all be expected to become concurrency experts. The CME project attempts a major shift on the side of the supporting technology: languages, formal models, verification techniques.

The core idea of the CME project is to make concurrency easy for programmers, by building on established ideas of modern programming methodology (object technology, Design by Contract) shifting the concurrency difficulties to the internals of the model and implementation.

The project includes the following elements.

1. Sound conceptual model for concurrency. The starting point is the influential previous work of the PI: concepts of object-oriented design, particularly Design by Contract, and the SCOOP concurrency model.
2. Reference implementation, integrated into an IDE.
3. Performance analysis.
4. Theory and formal basis, including full semantics.
5. Proof techniques, compatible with proof techniques for the sequential part.
6. Complementary verification techniques such as concurrent testing.
7. Library of concurrency components and examples.
8. Publication, including a major textbook on concurrency.

Section 1: ***The Principal Investigator***

1(a) Scientific Leadership Profile

[This five-page section, which described the Principal Investigator's background as applied to this project — CV, publications, awards, memberships, previous grants etc. — has been removed from the present scientific description.]

Section 1(d) ***Extended Synopsis of the project proposal***

D1 Overview

D1.1 Against the accepted view

Two widely held views of the state of Information Technology (IT) are that: (1) the future requires a mass move to concurrent architectures and, as a result, concurrent programming; (2) concurrent programming is very hard and requires a fundamental change in programming patterns. The Concurrency Made Easy project (CME) embraces the first assumption and categorically rejects the second one. Its aim is:

> **To achieve a breakthrough in computing by making it just as natural for programmers to write concurrently as sequentially, without massive retraining, and retaining principles and techniques that have proved scientifically sound and practically successful in sequential programming.**

The disagreement with the accepted view is not about the intrinsic difficulty of concurrency: concurrent programming clearly raises major scientific and engineering challenges, which have eluded some of the best minds. The disagreement is about who should field that difficulty. Existing approaches unfairly and unrealistically pass on far too much of it to programmers. The CME project assigns the most difficult tasks to the theory and supporting software, presenting programmers with a simple picture that enables them to use proven patterns of reasoning to write demonstrably correct and efficient concurrent programs.

D1.2 CME in a nutshell

The CME project pursues the following goals:

1. Develop a comprehensive model of programming that retains the benefits of proven ideas such as object-oriented programming, Design by Contract (the seminal approach to high-quality programming which we have introduced and which has proved widely influential), formal reasoning, seamless development and other core principles of modern software engineering.
2. Provide a quality implementation of the model, available for large-scale experimentation and integrate it in a full IDE (Integrated Development Environment).
3. Analyze performance and ensure high performance for practical applications of large-scale concurrency.
4. Develop a comprehensive theory of the approach and integrate the results as verification techniques in a verification environment, including both proofs and tests.
5. Develop a comprehensive library of common concurrency patterns and a large set of extensively documented examples, as well as a textbook explaining the approach for a large audience.
6. Perform extensive empirical evaluation of the approach in both academic and industrial settings. Feed the results of the evaluation into improvements of the model and the implementation.

This highly ambitious program entails risks, assessed in part B2. These risks should be balanced with the risks of *not* solving the concurrency problem. If computer science cannot provide the tools that will enable masses of mainstream programmers to master concurrency, IT will fail to deliver the powerful concurrent applications in health care, the environment, natural sciences and many other fields from which society critically expects major progress over the next decades.

If successful, the project has the potential of radically affecting Information Technology by breaking the current barrier to improved performance, and by putting concurrency in the hands of those who hold the key to the future of the field and of its contribution to society: professional programmers.

Section D2 describes why the concurrency challenge is critical for the progress of IT and society in general. D3 analyzes the programming challenge and D4 the basic CME approach to tackle it. Section D5 presents the starting point, D6 the project tasks, and D7 a summary.

D2 The concurrency challenge

It is not exaggerated to state that the focus of the CME project is the most important technical challenge facing the entire field of information technology and computer science today. The US National Academy of Science, in a forthcoming report [NAS 11], describes it in dramatic terms:

> *The only foreseeable way to continue advancing performance is to match parallel hardware with parallel software.* [...] *Heroic programmers can exploit vast amounts of parallelism.* [...] *However, none of those developments comes close to the ubiquitous support for programming parallel hardware that is required to ensure that IT's effect on society over the next two decades will be as stunning as it has been over the last half-century.*

The reference to "Heroic Programmers" is telling: one can train a few programmers to master the intricacies of thread libraries and hope that they will avoid data races; or, at the other extreme, one can hope that enough programmers will learn calculi such as CSP or the Pi-calculus. But these solutions do not scale up. The aim of the CME project is to make ordinary professional programmers use concurrency as naturally as they use loops and hash tables.

Before the National Academy, leaders from both industry and academia started sounding the alarm:

- Intel Corporation: "*Multi-core processing is taking the industry on a fast-moving and exciting ride into profoundly new territory. The defining paradigm in computing performance has shifted inexorably from raw clock speed to parallel operations and energy efficiency.*" [Intel 06].
- Rick Rashid, head of Microsoft Research: "*Multicore processors represent one of the largest technology transitions in the computing industry today, with deep implications for how we develop software*." [Microsoft 07].
- Bill Gates: "*Multicore: This is the one which will have the biggest impact on us. We have never had a problem to solve like this. A breakthrough is needed in how applications are done on multicore devices.*"

The matter even reached the front page of the New York Times [Markoff 07] with an article on how the progress of hardware can no longer meet the increasing demands of software, how concurrency, the only hope in sight to address this issue, is the new frontier of computing, and how the central issue is the difficulty of *programming* concurrent hardware architectures. Industry luminaries cited in the article state that they simply do not know of any programming technique to overcome this problem — which one of them, Andrew Grove (founder of Intel), describes as the biggest roadblock that the IT field has ever faced, and entirely unprecedented.

The background is what may be called the **end of Moore's Law as we know it**. For five decades the IT industry has been used to improvements in basic computing power with no precedent in any field of technology: roughly, doubling of power at constant price every year and a half. The underlying phenomenon is known as "Moore's law": the regular doubling of transistor density. It resulted in an exponential growth of CPU speed and many other performance indicators. Sometime at the beginning of this century, the free ride came to an end: while density continues to increase, CPU clock speed for a single processor has reached a plateau (fig. 1).

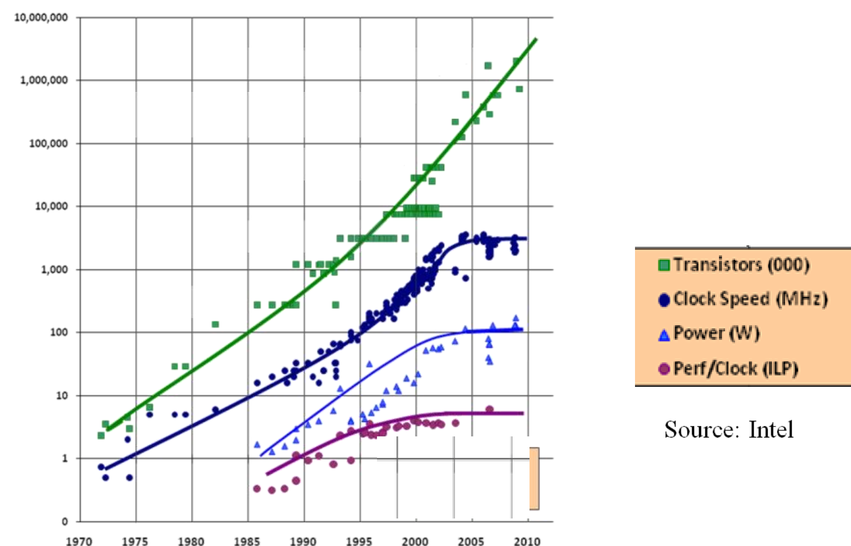

Figure 1: The "end of Moore's Law as we know it"

As advances in electronics are reaching their natural limits, the only way to continue providing growth is through systems that perform many computations *concurrently*; hence the introduction of multicore architectures (comprising several processors or "cores") as referred to in the earlier quotations, and network-based solutions (Web services, grid computing, cloud computing).

Simply doubling the number of processors, or multiplying it by a large factor as in the next generation of hardware architectures, does not by itself achieve the desired performance increases. Even for the parts of an application that can conceptually be parallelized (not all can), the program must have been written accordingly, or the compiler must produce parallel code. As noted by Grove and many others, we do *not* today have the corresponding programming techniques for use by mainstream software development.

Besides the quest for speed, the critical role of concurrency also arises from user convenience (which often suggests multithreading) and from the inherently concurrent nature of networks, Web services and cloud computing. Information technology has, over the past decades, profoundly transformed almost every aspect of society. The problems already solved pale against the problems that await us; three examples among many are: environmental challenges, where IT is essential; health care, where progress requires ever finer analyses; and the needs of the natural sciences, which are increasingly dependent on the treatment of data sources (such as telescope or satellite feeds) of a volume orders of magnitude higher than what was considered advanced just a while ago. None of these goals can succeed without a comprehensive and widely usable approach to concurrency, which the CME proposes to develop.

D3 The programming challenge

Whatever the goal — raw performance, convenience, network architectures — taking advantage of concurrency raises major programming difficulties. While software construction has made considerable advances in the past decades through structured programming, safer and more expressive programming languages, object technology, design patterns and other seminal ideas, the focus has overwhelmingly been on *sequential*, not concurrent programming. Concurrency remained until recently a somewhat arcane field of interest to an elite set of programmers writing operating systems and networking infrastructures.

There has been considerable research to propose better concurrent programming techniques (even though the US National Science Foundation stopped funding concurrency research on a significant scale in the nineties); in particular, concurrency calculi have attracted considerable attention. Very little of this work has influenced industry practice (two exceptions are the Web Services language BPEL, strongly influenced by the Pi-calculus, and the notion of lock-free data structure). The dominant concurrent programming techniques in industry remain shockingly the same as invented in the 1960s (prior even to the development of "structured programming"), in particular *semaphores* — a powerful invention in its time, but today a low-level programming construct removed by several levels of abstraction from the powerful and elegant mechanisms of modern programming.

The result is that it is extremely difficult to develop concurrent applications, in particular multithreaded ones, in a way that guarantees their correctness. The problem is compounded by the inherently *non-deterministic* nature of concurrent programs: one execution in a million may exhibit completely different behavior as a result of minor timing variations. Such abnormal behavior includes (in addition to wrong results) special and tricky forms of concurrency faults:

- Data races: even though two concurrent threads handle a shared data item in a way that is correct from each thread's perspective, a particular run-time interleaving produces inconsistent results.
- Deadlocks: in a situation where some threads still have computations to execute, none of them can proceed as each is waiting for another one to proceed.
- Improper scheduling of threads or processes, "priority inversions", "starvation" (some processes do not proceed because the resources they need are unduly taken away by others).

The traditional ad hoc techniques, used to get sequential programs to an acceptable degree of quality through testing and debugging, are largely ineffective here, because concurrency faults are often intermittent and difficult to reproduce, making debugging, in particular, a hazardous proposition.

This particular situation suggests that *formal verification techniques* have a natural place for concurrent programming. In spite of a number of resounding successes, the standard scheme in industry is still to deliver a product with no obvious fault, then if there is a bug report from the field reproduce it and correct it. With concurrency this is unrealistic: the faults may be damaging (the program or the machine gets stuck) and intermittent; reproducing them on a machine with different timing conditions may be hard or impossible. There may simply be no other way than more systematic verification techniques with a rigorous basis. The CME project includes a considerable formal semantics and verification side. Without verification, no approach to concurrency can be successful.

The chasm between theory and practice is much wider in concurrency than in traditional programming: elegant theories cover the concurrency aspect only and do not directly relate to the rest of a program's functionality and behaviour; low-level practice still uses pre-structured-programming techniques making it very hard to reason at all about programs.

The goal of the CME project is to remove this chasm, and provide a basis for building high-quality concurrent programs, correct by construction. The basis includes advanced theory, language constructs, a programming methodology, compiler technology, supporting tools, and a vast repertoire of examples.

D4 Basic approach

CME introduces a new programming model for concurrent programming. The following analysis explains why we believe this project is the best hope for addressing the concurrency challenge.

1. **Programmers cannot ignore concurrency**. One might hope that compilers will parallelize sequential code. Parallelizing compilers have achieved successes; further advances will occur, but will not suffice. The problems get more difficult as research moves away from the “low-hanging fruit”. In many cases only programmers can provide the necessary information; for example synchronization may be involved. The challenge, addressed by CME, is to make programmers specify the *minimal* concurrency needs and let powerful compilers do the rest.
2. Other than performance, **the key criterion is ability to reason about programs**. The main deficiency of current concurrency techniques is that, unlike with modern sequential programming, it is hard to understand what programs will do without taking an *operational* view. Much of the progress in programming has come from the programmers ability to understand their programs’ behavior abstractly, from the properties of the program text, without having to understand what sequences of actions the computer will take. This is the benefit of all programming advances noted above (structured programming, OO programming, design patterns), each with its own abstractions. In concurrency, with semaphores and similar techniques we are back to the operational mode. We need the appropriate abstractions.
3. For programmers, **the change from sequential programming mechanisms must be minimal**. Humans reason sequentially much better than concurrently; we cannot expect programmers to master the intricacies of concurrency, as they must do today to use threads and semaphores); they must continue to program for the most part as in the sequential world, and handle the concurrent aspects through simple mechanisms that they fully understand. Combined with goal 1, this requirement means that the mechanism must both let programmers express key concurrent aspects in high-level terms, and relieve them from the details of the concurrency machinery.
4. **Object-oriented programming provides the right basis**. Object technology has become the accepted framework for much of today’s developments thanks to its successes in providing the right abstractions. In addition, if one looks at concurrency in the appropriate way, as discussed in Section B2, then it can be added to the basis of an OO language so that the result is simple, can be taught to ordinary programmers, and delivers the expected benefits.

The breakthrough nature of the project results in part from goal 3. Research in concurrency theory has implicitly assumed that programmers would radically change their modes of programming. In other words, *researchers* have passed on the responsibility for a breakthrough to *programmers*. This is unfair, and is not going to happen. The CME project attempts breakthroughs on the side of programming support, theory, compilers, tools and libraries so that programmers can handle concurrency while keeping the necessary changes (goal 1) to a minimum. The only way to make concurrency succeed is, as reflected in the project name, to make it seem *easy* so that concurrent programs, including large ones, can be made demonstrably correct and efficient, by normal programmers and with normal levels of effort.

D5 Starting model

The starting point for the CME programming model is the SCOOP approach (Simple Concurrent Object-Oriented Programming) developed by our group, which extends to the concurrent world the techniques that have proved effective in mastering sequential computation. SCOOP addresses a number of problems but leaves others unanswered; the project will have to solve these open issues.

The details of SCOOP appear in part B2. Perhaps the main advantage of SCOOP is the simplicity for the programmer, addressing goal 3 above. As a typical example, the essential part of the solution to a problem such as the “dining philosophers” (an academic example, but typical of difficult resource-sharing problems), ensuring fairness, is just

```
class PHILOSOPHER inherit PROCESS redefine step end
feature
        left, right: separate FORK              -- The two available forks
        think do … end                          -- Think for some time
        eat (l, r: separate FORK) do … end      -- Grab the forks and eat
        step do eat (left, right) ; think end   -- Basic operation of a philosopher’s life
end
```

Most of this program extract is what any competent programmer would have written in a *sequential* context to describe the Dining Philosophers setup. Note in particular the use of inheritance to rely on a *PROCESS* library class that describes the overall life of any "process".

The only truly concurrent part is the declaration of the forks *left* and *right*, and correspondingly the arguments to *eat*, as **separate**, meaning simply that they run on separate "processors" (sequential execution mechanisms, for example threads). The other highlighted part is the passing of *left* and *right* as arguments; because they are separate, the effect is to wait until the corresponding processors are available, and then to reserve (lock) them for the duration of the call. This is all there is to do to obtain a correct solution to the problem; it will not deadlock, and guaranteed to be fair: no philosopher will starve.

There are of course numerous solutions to the Dining Philosophers problem in the literature, but avoiding deadlock requires the programmer to tweak the problem (numbering the philosophers and changing the order in which they pick their forks, or admitting one fewer than the number of philosophers) and use low-level synchronization calls (semaphore "signal" and "wait"). Getting fairness is particularly delicate. In this and many other examples, such as those in a survey of classic concurrency problems [Downey 05], the SCOOP version contains a minimum of synchronization code, enabling the programmer to express the key needs at a high-level of abstraction, in terms directly related to the statement of the problem.

SCOOP has attracted significant attention. A prototype, used in courses, has shown the feasibility of the basic model; empirical evaluation shows that students understand it better than traditional approaches. This experience confirms that it is a suitable starting point; the critical challenges, however, remain open.

D6 Project tasks

The goal of the CME project is to achieve a fundamental research advance. Since scalability and performance will determine practical applicability, the focus includes not only concepts and theories but a high-quality implementation and a rich set of examples. The project includes the following tasks.

- *Model development*: building on the successful abstractions of object-oriented programming, develop abstractions enabling programmers to write efficient concurrent applications and reason on them.
- *Reference implementation*: develop a high-quality open-source implementation; integrate it in a research IDE to enable development of practical systems.
- *Performance analysis*: measure performance and develop techniques to take full advantage of physical concurrency, comparable to the performance of low-level approaches.
- *Formal basis*: define a full formal semantics for concurrent OO computation, covering the entire programming language with sequential as well as concurrent aspects; in addition the semantics must not only be theoretically sound but also provide the basis for proofs.
- *Proof system*: turn the results of the semantic specification into a directly usable set of mechanisms for proving the correctness of concurrent programs, relying on solid existing proof technology and embedded in an existing verification environment (EVE, developed by our group).
- *Other verification techniques*: to complement proofs, develop other approaches such as tests, building on our previous work (AutoTest framework for automatic testing, included in EVE).
- *Examples, libraries and textbook*: develop a library of components covering fundamental concurrency patterns, and a rich set of examples across many applications of concurrency. Write a textbook explaining to a broad audience of professionals that concurrency can be effective, and easy.

D7 Summary

The Concurrency Made Easy project addresses a central issue of today's IT and computer scene, rich with research and industrial applications. CME breaks away from conventional wisdom by asserting that concurrency can be made as easy and reliable as the best of today's sequential programming. This is a high-risk project, but holds the promise of a breakthrough in modern information technology.

Section 2: *The Project proposal*

a. State-of-the-art and objectives

The aim of the Concurrency Made Easy (CME) project, as previewed in the Extended Synopsis, is to solve the major open issue in the science and practice of computer programming:

> **To achieve a breakthrough in computing by making it just as natural for programmers to write concurrently as sequentially, without massive retraining, and retaining principles and techniques that have proved scientifically sound and practically successful in sequential programming.**

The target is possibly the toughest challenge in IT today: enabling society to take advantage of the enormous possibilities opened by concurrent hardware architectures. The main obstacle is our current inability to *program* them in a clear and safe way, guaranteeing the correct functioning of applications. The CME project proposes to develop the corresponding model, theory, proof techniques, implementation and libraries.

The core idea, going away from all the approaches tried so far, is that concurrency should be *easy* for programmers. Since concurrency implies daunting conceptual challenges, this can only mean that the underlying machinery handles all the complexity. We feel that this is the only scalable approach; the many elegant concurrency models that have been proposed (section E2) assume that programmers will fundamentally change their mode of working to "think concurrent" throughout; this is unrealistic. CME strives in contrast to retain the most successful benefits of modern programming, in particular the fundamental abstractions of object technology and systematic reasoning on programs as embodied in the Design by Contract approach. This radical departure from accepted views implies risk, as discussed in section E7, but we feel that these risks can be addressed and that the approach, however unconventional, is the requisite path to taming concurrency and enabling IT to continue deliver on its promises to society.

E1 Concurrency and the unending quest for performance

Appetite for computing power is insatiable, fuelled by multimedia, the Internet, cloud computing and constantly appearing new applications. The extended synopsis in part B1 described this background. Most experts will concede that in the practice of software development we do not know how to program concurrent applications correctly and efficiently. The programming models are complex and error-prone; concurrent systems hang, enter "data races" and deadlocks, and fail to deliver the performance improvements to which concurrent hardware should entitle us. This leads to widespread industry disappointment with multicore architectures, as they fail to deliver the speedups promised by the hardware.

The problem of finding a satisfactory framework for concurrent and distributed development has eluded researchers and practitioners for decades; it has become critical today, as many applications traditionally thought of as sequential now require concurrency. Multicore architectures and performance concerns are not the only drivers behind the push for concurrency:

- The meteoric rise of the Web turns many applications into network programs, with a high degree of concurrency and new challenges.
- Ever more applications are *multi-threaded* — able to perform several applications at once — for reasons of user convenience as much as performance.
- Physics, biology and other natural sciences are faced with the need to process unprecedented amounts of data, requiring massive concurrency.

Industry leaders as well as research experts have conceded both that concurrency is the next IT frontier and that we have no clear path to a solution today (see the citations in B1(d)). The aim of the CME project is to chart such a solution, with an impeccable theoretical foundation, ready to use by competent programmers without massive retraining and without losing the benefit of decades of progress in software engineering.

E2 State of the art: earlier work and literature survey

Surveys of traditional concurrency techniques include [Andrews 00], [Ben Ari 06], and [Herlihy 08]. [Alonso 04] is a good starting point about the state of the art in distribution and Web services.

In today's practice of concurrent programming, developers overwhelmingly use "thread libraries" callable from languages such as C, Java, C# or Eiffel. Well-known libraries include POSIX threads, Windows threads, Java Threads, and .NET threads. While differing in the details, they share the same conceptual basis, going back to the sixties and seventies: semaphores [Dijkstra 68] and (less commonly) monitors [Hoare 74].

At the other end of the spectrum, *process calculi* are mathematical models to describe systems of processes algebraically. One of the most influential has been CSP [Hoare 85], [Roscoe 97-05]. Another is CCS [Milner 89], the source of the more recent Pi-Calculus [Milner 99], which integrates a notion of mobility by allowing

processes to change their communication links. The Pi-Calculus has spawned a large family of related calculi, including Mobile Ambients [Cardelli 00], which handles moving processes through administrative domains, and the Join Calculus [Fournet 02], supporting the specification of inter-process communication by pattern-matching of calls and messages. Another classic model is the *Petri net* [Petri 62], a type of transition system in which the occurrence of an event does not affect the global state, but only local conditions. Events can occur in parallel if their local conditions do not intersect, leading to a "true concurrency" semantics, in contrast to the otherwise predominant interleaving semantics. A further theoretical approach, the *Actor model* [Hewitt 73], [Agha 90], holds the view that the universal primitives of concurrent computation are actors, entities that can react upon receiving messages by local computations, message sending, and replication.

CME adopts an object-oriented (OO) approach. Papathomas [Papathomas 92] gave a first classification of concurrent OO languages (COOLs). A special issue of the *Comm. of the ACM* edited by the PI [Meyer 93] presented a number of important approaches to concurrent OO programming. An earlier collective book [Yonezawa 87] served as catalyst for much of the initial work. A large survey on COOLs is due to Philippsen [Philippsen 00]. One of the earliest parallel OO languages was POOL [America 89], using a notion of active object, which was found to raise problems when combined with inheritance. The term "inheritance anomaly" was introduced by Matsuoka and Yonezawa [Matsuoka 93]. Also influential has been the work around the Hybrid language [Nierstrasz 92], [Papathomas 92], which does not distinguish between active and passive objects but relies instead on the notion of thread of control, called *activity*. The basic communication mechanism is remote procedure call (RPC), synchronous or asynchronous.

Recent concurrent languages are often influenced by the above models. The Actor model underlies the OO language Axum [Axum 10]. Axum offers several operators to construct explicit dataflow networks for propagating messages. Besides message passing, Axum also provides shared state for efficiency, using so-called domains to encapsulate a state shared between groups of actors. Other languages based on Actors are Erlang [Armstrong 10], which has a functional core, and Scala [Odersky 08], with both functional and OO elements. Process calculi have also influenced the design of OO languages. For example Cω (an extension of C# previously known as Polyphonic C# [Benton 04]) integrates elements of the Join Calculus. Cω allows computations to be spawned off into different threads using asynchronous methods: while for synchronous methods the caller must wait until a routine completes, asynchronous methods return immediately while their body is scheduled for execution in another thread. Cω supports so-called *chords*, which associate the body of a routine with more than one method; only if all methods have been called will the body be executed. Another influential language is Cilk [Blumofe 95], which extends C with the keywords **cilk**, **spawn** and **sync**. A method marked **cilk** can be asynchronously spawned off with the **spawn** keyword. **sync** requires the current method to wait for all previously spawned asynchronous tasks to complete. Cilk also implements a work stealing mechanism to achieve high performance by dividing method execution efficiently among processors. Thread libraries such as Intel's TBB implement load balancing in a similar way. Programming models such as *Message Passing Interface* (MPI) and *Partitioned Global Address Space* (PGAS) target high-performance parallel computing. While MPI is based on message-passing, PGAS uses global shared memory. PGAS associates portions of shared memory to specific processors to improve performance and scalability. A number of recent languages rely on the PGAS model, for example Fortress [Allen 04], X10 [Charles 05], and Chapel [Joyner 06]. *Coordination languages* rely on inter-process communication. Tuple spaces, introduced with Linda [Gelernter 85], are multisets of messages composed of multiple fields; messages can be put into tuple spaces and retrieved again via pattern matching using the **out**, **in**, and **read** language primitives. The coordination approach has been revived in languages such as Java (JavaSpaces).

Classical work on *distributed* systems includes the CHORUS operating system, of which [Lea 93] describes an OO extension; the GUIDE language and system [Balter 91]; and the SOS system [Shapiro 89]. The EPEE system [Jézéquel 96] used OO concepts and specifically Eiffel.

Much recent work on concurrency has revolved around *software transactional memory* (STM) [Herlihy 93]. The basic idea is to avoid unnecessary a priori locking, and instead correct any actual access conflicts a posteriori. Although such models for non-blocking concurrency remain active research topics, the expectation of a breakthrough has not been realized. The bookkeeping of operations and states, necessary to perform a rollback in case a conflict is detected, appears to cause overheads of its own, which may be as bad as locking. A number of Java extensions support STM, such as [Harris 03] and [Welc 03]. The ideas from STM have also been re-expressed in Haskell [Harris 05], extending the approach with a composable form of blocking, and allowing for transactions to be composed as alternatives. The SCOOP-based approach, described in the following as the starting point of the CME project, could also benefit from some of the ideas of STM: one may envision optimizations that give the programmer the appearance of locking, and the

associated techniques for reasoning safely about programs, but actually delay locking until when and if it is needed, in STM style.

A more general issue is how to guarantee the correctness of concurrent programs. Transactions were originally proposed as a synchronization mechanism in work on *abstract data types* [Liskov 94], where constraints on subtype conformance result from *behavioral subtyping*, directly related to the rules of Design by Contract (precondition weakening, postcondition strengthening). In spite of the strong interest in transactions, most recent work on concurrency for contract-enabled languages has focused on refinements of the *axiomatic semantics* foundation or Hoare logic [Hoare 69] and explicit locking. Seminal efforts to tackle concurrency in axiomatic semantics go back to the 70's. The Owicki-Gries approach [Owicki 76] provided a framework for determining under what conditions two blocks of code can be executed in parallel without producing data-races. Owicki-Gries basically evaluates, for *every* single instruction in a first code block, whether it conflicts with any of the instructions of the second block, requiring O (*nm*) single verifications for two code blocks of *n* and *m* elementary instructions respectively. An OO setting complicates the situation through the possibility of nested method calls and aliasing. Lipton's early work [Lipton 75] aims at identifying atomic regions in programs: regions that can be considered to execute atomically, and hence may reduce the complexity of applying Owicki-Gries.

The *rely-guarantee* approach [Jones 83] was introduced as a computationally less expensive alternative to Owicki-Gries; it is compositional whereas in Owicki-Gries a change of any component may affect the proof of any other. Rely-guarantee augments Hoare triples with *rely* and *guar* predicates. A code block executes as soon as its precondition holds; only *rely* is subsequently required to remain valid, and *guar* is ensured, before the end of the code at which only the (full) postcondition must hold. The resulting composition rule is slightly more complicated but compositional, with the downside that it may require more input from the programmer. These ideas have interesting potential applications to CME.

Concurrent separation logic [O'Hearn 07] is a recent and promising extension of Hoare logic for reasoning about concurrent programs in the presence of aliasing. It uses separation logic's *-connective and Frame Rule to achieve modularity and local reasoning [O'Hearn 01]. The predicate P * Q means that both P and Q hold in the execution state (stack and heap), and that they rely on disjoint portions of the heap. This yields a simple rule for disjoint concurrency, where || is the parallel composition operator: {P} C {Q} and {P'} C' {Q'} together imply {P * P'} C || C' {Q * Q'}, provided C does not modify any variables free in P', C', Q' and conversely. Although simple, this rule is sufficiently powerful to prove algorithms such as parallel Mergesort correct. Other logical concepts such as resource invariants are used to reason about semaphores and inter-process ownership transfer of state at synchronization points [O'Hearn 07].

A common concurrent programming pattern, improving performance and avoiding data races, gives processes shared read access to a global state and prohibits changes to the state if any process is reading or writing it. Fractional permissions [Bornat 05] offer an elegant lightweight way to reason about such programs. They support permission transfer, subdivision and combination in separation logic, and have been used outside of separation logic, for example in the Z3-based Chalice tool and methodology [Leino 09].

The rely-guarantee approach makes reasoning about process interference tractable, yet its specifications must describe the entire state. On the other hand, separation logic struggles with interference, but copes well with process independence and avoids global state descriptions. Vafeiadis and Parkinson [Vafeiadis 07] proposed a marriage between the two approaches, describing interference in rely-guarantee style and using local reasoning in the absence of interference; like separation logic and rely-guarantee, the approach uses parallel composition. The observation that practical concurrency often uses threads led to *deny-guarantee* reasoning [Dodds 09], where the lifetime of a thread can be scoped dynamically. Correspondingly, interference can be split and combined dynamically, and deny and guarantee permissions are used: deny permissions specify that the environment cannot perform certain actions, and guarantee permissions allow a process to perform certain actions. The fact that deny-guarantee facilitates dynamic thread manipulation in the presence of aliasing makes it a promising research direction for the verification of concurrent OO systems.

Along this work on axiomatic semantics, most *behavioral* frameworks for multithreaded OO programming seek to increase parallelism by allowing simultaneous execution of code blocks or methods. The goal is to prove *non-interference*; one possibility is to identify code blocks that can be (apparently) executed atomically. The SCOOP model [Meyer 93a, 97], [Nienaltowski 07, 09], used as a starting point for CME, makes strong restrictions on access to "separate" objects to allow application of standard axiomatic techniques. It includes a "wait by necessity" mechanism, adapted from [Caromel 93]. Other recent work includes [Rodriguez 05], on concurrency specifications for the Java Modeling Language (JML). [Burdy 03] and [Leavens 05] add syntax for defining locks and which objects they control. Sources behind these developments include [Flanagan 04], [Wang 06], [Robby 06]. Spec# [Barnett 04], like Eiffel [Meyer 91],

integrates contracts into the programming language and, like JML, has an extensive tool set, including a verification engine. [Jacobs 04] emphasizes two keys concepts for multithreaded Spec# programs, aliasing and locking, addressed through keywords *pack/unpack* and *acquire/release*. The authors' main focus is on the preservation of invariants in the presence of multithreading. The techniques provide an interesting alternative to those of JML. Sing# [Fähndrich 06] is a superset of Spec# which adds capabilities for concurrent programming using message-based communication. The language has been used to efficiently implement Singularity [Fähndrich 06], an experimental operating system built for high dependability.

It remains difficult to obtain satisfactory axioms in the presence of multithreading, given the intertwining of application logic and synchronization. Many hopes have been put in *temporal logic* [Pnueli 77] to specify synchronization separately from operations. For SCOOP, [Ostroff 09] showed that temporal logic can specify liveness properties beyond the scope of axiomatic semantics. The question of efficiency remains. *Model checking* [Clarke 99] of temporal logic specifications provides an effective verification method for abstract concurrency models [Jhala 09] and has been applied to concurrent programs, for example in the Java PathFinder [Brat 00]. State space explosion, particularly acute with concurrent interleaving, currently prevents model checking from scaling to large concurrent programs; initial steps have been taken to address this problem [Basler 09]. *Abstract interpretation* [Cousot 77] and other *static analysis* techniques [Nielson 04] are more resistant to large state spaces because they deal with sound abstractions, and can detect data races [Kahlon 07]. They suffer, however, from false positives. *Testing* is dominant industry technique for finding errors. In applying testing to concurrent programs, the main concern is to drive the program systematically through different scheduling orders without causing interference that removes evidence of the error ("*Heisenbugs*"). Promising approaches combine testing with model checking and static analysis. [Musuvathi 08] uses techniques from model checking to guide the exploration of the state space and bound the number of context switches per execution as a heuristics to find errors quickly. Sen and Agha [Sen 06] use a combination of concrete and symbolic execution, termed *concolic execution*, to test multithreaded Java programs. Kundu, Ganai, and Wang [Kundu 10] present a framework that combines conventional testing with symbolic analysis. For symbolic verification, concrete traces are relaxed into concurrent trace programs, which capture all linearizations of events that respect the control flow of the program.

E3 Current and potential competition

The currently dominant techniques, multithreaded mechanisms and libraries, do not scale up; the bigger and more complex the problems, the more hopeless they are at helping programmers reason about their programs. It is clear to many experts that something needs to be done.

Current mainstream development platforms such as Sun's Java-based J2EE are not suited for highly concurrent programming, and can only evolve slowly because of the huge existing programmers and program base. This applies only slightly less to the more recent Microsoft .NET framework and its supporting languages (C#, Visual Basic .NET). For that reason concurrency developments have mostly been implemented through libraries or, as in the JML case, with a preprocessor.

A number of approaches have attempted programming language integration:

- Sun's *Fortress* [Allen 04] is a brushed-up Fortran with Java-like syntax extended with mathematical operators. For concurrency, Fortress offers implicit parallelization of loops and operations on data structures. So far, only a limited interpreter is available for the core features of the language.
- Microsoft's *Cω*, noted above, is innovative but requires intensive programmer training.
- IBM's *X10* [Charles 05] is built on the PERCS programming model and targeted at multi-core SMP chips with non-uniform memory hierarchies interconnected in scalable cluster configurations. X10 provides important abstractions (places, asynchronous methods, future invocations, barriers) but places a considerable burden on programmers. Only partial results have been published.
- Intel proposes *Pillar* and *Ct* [Anderson 07], C variants with explicit parallelism annotations[Ramsey 00]. For today's software engineer used to OO languages and methodologies, such a model is too low-level.
- Cray's *Chapel* [Joyner 06] provides a higher-level multithreaded parallel programming model with abstractions for data parallelism, task parallelism, and nested parallelism. Its parallel concepts are most closely based on ideas from various extensions of Fortran.
- *Erlang* [Armstrong 10] offers an elegant model of functional concurrent programming. The premises, however, are too far away from programmers' practices, in particular OO technology which industry has widely adopted for its quality benefits such as extendibility and reusability, and is unlikely to renounce.

The languages cited all provide specific syntax for parallelism, requiring specifically trained programmers. The CME project chooses to retain traditional programming abstractions and their benefits; it includes only a small syntax extension, and provides semantic extensions to support the needs of concurrent programming.

Existing solutions fall into two categories: low-level approaches, generally library-based, relatively easy for programmers but not supporting precise reasoning about programs, and opening the way to errors; and attractive theoretical models, which would generally require an overhaul of programmers' thinking patterns. In addition, their proof mechanisms often address concurrency aspects only, whereas programmers need to prove *all* properties of a program's behavior. CME's aim is to get away from this dilemma.

b. Methodology

At the core of the concurrency challenge lies the difficulty of *reasoning* about programs. With sequential computation, even programmers not familiar with formal techniques can understand the effect of programs in an abstract way, not in operational terms. With concurrency and all the possible interleavings of operations, such reasoning is no longer possible in the dominant low-level approaches. One of the principal goals of the CME project is to bring back the possibility for programmers to reason simply and effectively about the behavior of concurrent programs in the same way they can reason about sequential programs.

E4 Not offloading responsibility to programmers

Incremental improvements are not sufficient; a breakthrough is needed, but it cannot be discharged on individual developers. Some theoretically attractive approaches require a major departure from traditional practices; sequential programmers must reinvent themselves as concurrent programmers. This is unrealistic.

Experience has shown that most developers will not become concurrency experts; instead they need to retain a mode of programming closely related to the classical sequential model. The CME project attempts to gives programmers the *illusion* of an incremental change. To make this appearance of compatibility possible requires a major scientific shift on the side of the supporting technology: models, theory, compilers, tools.

Improvements are needed in: programming language constructs; compiler technology; supporting tools; and — a particular focus of CME — verification techniques. Verification advances in turn require a number of developments: new semantic models; axiom systems adapted to concurrent programs but upward-compatible with the traditional sequential models; integration of proof techniques with a standard development environment, in a form understandable by ordinary developers. Because proofs do not always succeed, CME also includes a *testing* component, with the intent of developing testing techniques adapted to concurrent programs. This last goal is also a major challenge, requiring breakthrough advances: the conventional wisdom, backed by strong conceptual arguments as well as by current practice, holds that testing concurrent applications is essentially hopeless. ("*Nondeterministic systems are in principle untestable*" [Roscoe 97-05].) We have performed some exploratory work [West 11] on integrating processor scheduling into an automatic testing framework (AutoTest [Meyer09]), but the results are only tentative so far.

E4.1 Project scope

The CME project proposes to bridge the gap between the available hardware potential for concurrent computing and the current support for devising concurrent software exploiting that potential by integrating a simple yet powerful model for concurrency in a modern OO programming language.

More generally, it is a major goal of CME to ensure that programmers familiar with modern object-oriented development can transition smoothly to the concurrent world, retaining many of the successful reasoning patterns they have learned to master. Some conceptually elegant approaches to concurrent programming assume that programmers must be retrained from scratch to "think parallel", and that sequential programming becomes just a special case. We do not adopt this approach, as it is unrealistic, and ignores the fact that even in the most parallel of programs a large part remains sequential and continues to raise the traditional problems of software architecture. For this reason, the CME approach transfers a considerable part of the burden to the implementation. The model supports this approach by defining only a small language extension to the framework of a modern OO language.

E4.2 The starting model: SCOOP

An initial attack on the issues has been SCOOP (Simple Concurrent Object-Oriented Programming) [Meyer 93a, 97] [Nienaltowski 07, 09] [Morandi 08, 11]. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract as initially introduced by the PI — which have proved widely successful in improving the dependability of sequential programs — and extend them to cover concurrency, in a way that appears minimal to the programmer but with considerable semantic adaptation, compiler technology, proof technology and tool support behind the scenes.

An empirical study with undergraduate computer-science students [Nanz 11] has shown an advantage for SCOOP in teachability and usability when compared to multithreaded Java programming.

SCOOP has attracted considerable attention but has only had prototype implementations so far, and leaves many issues open, such as automatic deadlock avoidance, with an initial, partial attempt requiring additional annotations [West 10]. CME will retain the most successful ideas of SCOOP but needs to achieve major advances to reach the goal of bringing safe concurrency to the masses.

E4.3 Object-orientation and concurrency

Over the last fifteen years, OO programming and languages have grown from marginal influence to widespread acceptance; their attraction comes in particular from their ability to model real-life abstractions simply, and from resulting improvements in software quality. Through his books (in particular the best-seller Object-Oriented Software Construction), articles and lectures, the PI has played a major role in this transformation of object technology from arcane specialty to dominant paradigm.

The standard OO model is sequential. Addressing concurrency requires extending it. To many people, the initial path seems easy, since some analogies spring to mind, in particular a common emphasis on modularity. Going beyond these similarities is, however, unexpectedly hard.

An example of a seemingly natural idea that turns out to be problematic is the concept of *active object* used in many approaches (e.g. [Nierstrasz 92], [Petitpierre 98]). An active object is also a process, with its own program to execute. This idea clashes with the definition of objects: an object does not perform just one sequence of operations; it is a repository of services and just waits for the next client to solicit one of those services. Making an object active means asking it to schedule its operations and creates a conflict with the clients's view: each party expects that the other will be ready to provide its requested services immediately. A consequence of this incompatibility is the so-called "inheritance anomaly" [Matsuoka 93].

Another reason for weak acceptance of concurrent OO languages is that the existing literature devotes little attention to issues of correctness and of helping programmers reason about concurrent OO programs.

The idea behind CME is to take OO programming, in a simple and pure form based on the concepts of Design by Contract, which have proved highly successful in improving the dependability of sequential programs, and extend them minimally to cover concurrency and distribution.

E4.4 Minimality of mechanism

Object technology is a rich and powerful paradigm which intuitively seems ready to support concurrency. It is essential to aim for the smallest possible extension. Such minimalism is not just good language design. If the concurrent extension is not minimal, some concurrency constructs will be redundant or contradictory with OO constructs, making the developer's task hard or impossible. The basic language extension in CME is the addition of a single keyword (**separate**) originally introduced by SCOOP.

E4.5 Objects and processors

For every object, the CME model assigns a *processor* to handle calls on the object. A processor can be a CPU, or an operating system task, or a thread. The result (fig. 1) is to partition the set of objects into a number of "regions" ("kens" in Schmidt's terminology, e.g. [Ling 03]); each region is associated with a processor, which handles all calls on the corresponding objects.

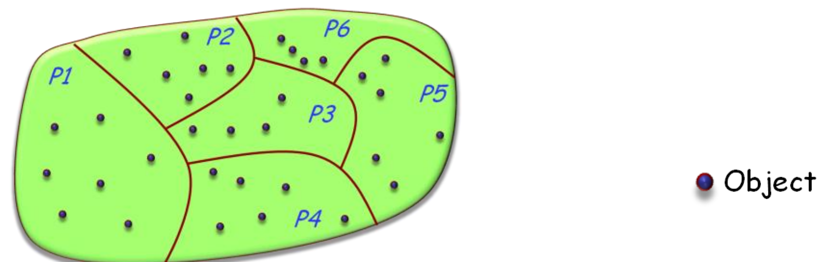

Figure 1: Partitioning the set of objects among regions (each assigned to a processor)

In the usual approaches to concurrent programming in an OO language, such as JavaThreads, a thread library is superimposed on an OO program, without regard to its OO architecture. The constant risk of errors such as data races is a direct consequence of this clash of paradigms. In contrast, CME's partitioning into regions handled by processors establishes a close correspondence between the OO structure and the concurrency structure. Data races are excluded by construction.

In line with the principle of "Concurrency Made Easy", the CME programmer does not need to know the explicit allocation of objects to processors; what counts is to know, when an object executes a call on another object, whether the two objects are in the same region (handled by the same processor) or not. This property

changes the semantics of the call: *x.r* (*a*) is a synchronous call if the object for *x* is in the current region; if not, the call does not need to wait (this is the main benefit of concurrency) and hence is asynchronous. This difference of semantics must be reflected in the syntax: the declaration *x*: **separate** *T* (as opposed to just *x*: *T*) indicates that calls *x.r* (*a*) using *x* as their target will have asynchronous semantics:

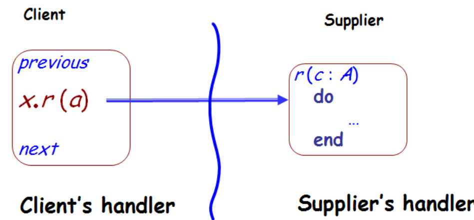

Figure 2: Asynchronous call

The "separate" notion also gives the basic synchronization mechanism without the need for any other language construct. The idea is simply that in a routine call *r* (*a, b, …*) (or *x.r* (*a, b, …*)) where the formal arguments are declared **separate**, the call needs exclusive access to all the needed separate objects *a, b, …* and will lock their processors before proceeding. The mechanism guarantees fairness, a concern that otherwise requires complicated algorithms that pollute application programs.

Assigning the task of multiple object acquisition to the implementation also decreases the possibility of deadlocks, which arises when programmers attempt to do the job manually. The general issue of deadlock avoidance remains open, however, and is one of the problems that the CME project intends to solve.

The well-known "Dining philosophers" example [Dijkstra 68], previewed in part B1, illustrates the model's expressive power, simplicity and safety (fig. 3; the use of inheritance is explained below.) The key step is the call *eat* (*left, right*) which, because the corresponding formals *l* and *r* are separate, locks both fork objects.

```
class PHILOSOPHER inherit
    PROCESS
        rename setup as getup
        redefine step end
feature {BUTLER}
    left, right : separate FORK
    step
        do
            think ; eat (left, right)
        end

    eat (l, r : separate FORK)
            -- Eat, having grabbed l and r.
        do ... end
end
```

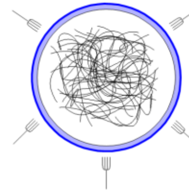

Figure 3: Dining philosophers

This classic example (an abstraction of resource allocation problems) has given rise to hundreds of published solutions, many of which involve delicate handling of synchronization and fairness issues; we do not know of any that is as simple for programmers to write. In particular, locking *several* resources is a delicate problem: if the client succeeds in reserving one but the next one turns out to be available it must release the first and try again; the risk of deadlock looms at every step. The problem is even harder in a distributed context. Here the difficulty is all handled by the implementation; the solution is deadlock-free and fair.

The other defining characteristic of this example, symbolic of the aim of CME, is that the concurrency baggage for the programmer is particularly light. Almost all of the code is what a programmer would write to describe the problem without regard to concurrency; the only concurrency-specific parts are the **separate** qualifiers. The details of synchronization are handled by the implementation in accordance with the programmer's intuitive expectation.

E4.6 Contracts and synchronization

The last observation can be formulated as a "Principle of Least Surprise", a defining characteristic of CME. Instead of forcing programmers to renounce patterns of thought that have been proved fruitful both by extensive practical application and by sound theoretical justification (abstract data types, axiomatic semantics, Design by Contract), the CME mechanisms transpose to the concurrent world the results that the programmers would expect.

As an another example, this time involving contracts and synchronization conditions, consider a class to describe bank accounts, defining two routines *deposit* and *withdraw* with preconditions (**require**), postconditions (**ensure**) and a class invariant. In sequential programming it could be written as follows:

```
class ACCOUNT feature
        balance: INTEGER
        deposit (sum: INTEGER)
                require  sum >= 0 do
                         balance := balance + sum           -- Possibly other operations as well
                ensure   balance = old balance + sum end
        withdraw (sum: INTEGER)
                require
                         sum >= 0
                         balance >= sum
                do
                         balance := balance – sum           -- Possibly other operations as well
                ensure   balance = old balance – sum end
invariant
        balance >= 0
end
```

The contract expressed by the precondition (**require**) and the postcondition (**ensure**) expresses the expectations and results of deposit and withdrawal operations. The implementations (**do**) have been shown, but client classes can use the operations on the sole basis of the contracts.

A typical use in a client class is a routine to transfer money from an account to another:

```
transfer (source, target: ACCOUNT; value: INTEGER)       -- Declarations changed to separate below
        require
                val >= 0
                source.balance >= value
        do
                source.withdraw (value) ; target.deposit (value)
        ensure
                source.balance = old source.balance – value
                target.balance = old target.balance + value
        end
```

In a sequential context, the properties advertised by the postcondition will hold on exit (if the precondition holds on entry), as a result of the contracts of class *ACCOUNT*. A typical use would be

```
if acc1.balance >= 800 then                              /1/
                -- Possible interference here! See below
        transfer (acc1, acc2, 800)
end
```

Now consider a concurrent context. If two clients execute code such as **/1/** concurrently, the second one using an account *acc3* rather than *acc2* in the call to *transfer*, but the same *acc1* as first argument, interference is possible: both callers might find that enough money is available and perform the transfers, whereas in fact there is not enough for both operations. The invariant will be violated and the behavior is incorrect. Handling such cases is at the core of concurrency mechanisms; the solutions may be very complicated as they need to synchronize and wait on two objects, and to roll back if the second object is not available. As noted for the previous example, things are even harder in a distributed context.

The CME solution is dramatically simple. The programmer needs to know that the source and target accounts are on different processors, i.e. "separate"; but that is *all* that he or she needs to know. Accounts *acc1* and *acc1* are declared not just as *ACCOUNT* but as **separate** *ACCOUNT*. The code, including class *ACCOUNT*, remains unchanged except for the declaration of arguments of *transfer*, whose header becomes

```
transfer (source, target: separate ACCOUNT; value: INTEGER)
```

The semantic rules imply that for separate arguments call *transfer (acc1, acc2, 800)* waits until not only the processors for *acc1* and *acc2* are available but also the second precondition clause (positive balance) is satisfied. Then it spawns asynchronous calls to *withdraw* and *deposit*. During the execution of *transfer*, the processors for *acc1* and *acc2* are locked, avoiding any data race or other conflict. (The first precondition clause, on *val*, not separate, remains a correctness condition.)

Following the principle of Least Surprise, the programmer gets the semantics expected in the sequential case (atomic operations, guaranteed preconditions), now safely transposed to a concurrent context. There is no need for complicated manual handling of synchronization, waiting and rollback, and no data races.

The use of preconditions as wait conditions, surprising at first but conceptually inevitable as the sequential semantics is not applicable any more [Meyer 97, Nienaltowski 09], subsumes many traditional synchronization mechanisms such as conditional critical regions and monitors into a single and general concept. It has proved adept at addressing a wide diversity of concurrency schemes [Alavi 10].

Calls on separate targets, such as the calls to *withdraw* and *deposit* in this example, are asynchronous if the routines are *procedures* (not returning a result). Calls to *queries* (functions or attributes, returning a result) will use "wait by necessity" [Caromel 93], completing the basic set of synchronization mechanisms. Complementary mechanisms, exceptions and "duels" [Nienaltowski 07] are needed to break holds in a safe way, for example to implement a timeout, requiring new developments as part of the project.

E4.7 Full use of inheritance and other object-oriented techniques

It would be unacceptable to have a concurrent OO mechanism that renounces key OO techniques, in particular inheritance. The "inheritance anomaly" and other apparent conflicts are not inherent but follow from specific choices of mechanisms, in particular active objects, which should therefore be rejected.

Instead of treating inheritance as an obstacle, we use the mechanism as an ally in the search for elegant architectures, concurrent as well as sequential. For example the notion of process is not a language concept but a library class *PROCESS*. Class *PHILOSOPHER*, in fig. 3, inherits it. *PROCESS* defines the properties common to all processes, such as a loop that repeatedly executes *step*. *PHILOSOPHER* simply provides its own version of *step*. The OO method's power resides in its abstraction facilities, which allow describing important schemes through libraries, a technique as useful for concurrency as for sequential programming.

E4.8 Initial proof rule

The CME model enables programmers to reason about programs in a way very similar to sequential reasoning. The basis is the rule that restrict application of separate calls to targets that are separate formal arguments, and hence are locked by the caller. This informal property has a formal expression. The Hoare rule for a sequential procedure r, simplified (no callbacks) but adapted to OO programming, reads

$$\frac{\{Pre_r \textbf{ and } INV\}\ Body_r\ \{Post_r \textbf{ and } INV\}}{\{Pre'_r\}\ a.r\,(x)\ \{Post'_r\}}$$

where priming represents substitution of actual arguments (and the target *a*) for formals, and INV is the class invariant. This is the key to the simplicity and power of OO programming, which would go away in the presence of interleaving. The tentative rule for concurrency [Nienaltowski 07, 09] reads

$$\frac{\{Pre_r \textbf{ and } INV\}\ Body_r\ \{Post_r \textbf{ and } INV\}}{\{Pre'^{CONT}_r\}\ a.r\,(x)\ \{Post'^{CONT}_r\}}$$

where CONT denotes the restriction to "controlled" variables: those which are non-separate, and the separate ones that have been locked. About others, no guarantee is made. This is only a partial rule, which needs to be refined in the CME project to handle the full scope of language mechanisms. It does confirm, however, our contention that the successful reasoning patterns of OO programming can be extended to concurrency.

E4.9 Applicability to many forms of concurrency and distributed programming

A general criterion for the design of a concurrent mechanism is that it should support many different forms of concurrency: shared memory, multitasking, real-time, client-server computing, peer-to-peer applications, network and distributed programming. With such a broad set of application areas, a language mechanism cannot be expected to provide all the answers, but it should lend itself to adaptation to all the intended forms of concurrency. CME achieves this goal by using the abstract notion of processor, and relying on a distinct facility to adapt the solution to any particular hardware architecture that you may have available.

SCOOP enabled us to gain preliminary evidence that the general ideas can work. [Alavi 10] describes solutions of a large set of classic concurrent problems, showing the simplicity of the model. We have also built and programmed a prototype *hexapod* robot (fig. 4); concurrency enables us [Ramanathan 10] to follow biological models of motion where different groups of legs are simultaneously active, enhancing the precision and effectiveness of motion. Using such concurrency is tricky with usual multi-threaded facilities. The techniques outlined above yield simple and clear solutions.

These applications remain small, however, and scaling up to realistic applications will require addressing a number of difficult issues which are the focus of the CME project.

Figure 4: Hexapod robot programmed with SCOOP

E5 Major research problems

The SCOOP project provided a first step towards solving the problems of concurrency and showed the feasibility in principle of an approach focusing on simplicity and leveraging on OO technology and Design by Contract. It is only an initial exploration. CME expands on these ideas and revises them as needed to provide a seminal solution, ready for wide use for large and demanding applications of major societal interest, e.g. health care, the environment, and the needs of natural sciences (faced with the processing of data flows on an unprecedented scale, requiring massive concurrency).

A major issue is **scalability**. SCOOP solutions have not been validated in the large. CME will address in particular the following concerns, all essential for scalability.

Fault tolerance is a prime component of scalability. Good fault tolerance requires in particular a sophisticated *exception* mechanism. The PI's pioneering work on exception handling [Meyer 97] established a sound theoretical basis, defining exceptions as contract violations. Concurrent programming raises difficult new problems, such as what to do with an exception when the context that initially raised it is gone.

Performance is a central goal of concurrency. SCOOP concentrated on conceptual correctness. Its prototype implementation uses a centralized scheduler; this is not a scalable solution. We have started to investigate techniques for *distributed scheduling*. The challenge is to provide programmers with the illusion (the semantic effect) of locking (guaranteeing safe behavior as in sequential programming, with no unwanted interference according to the Least Surprise principle) while physically locking as little as possible. In a call *transfer* (*acc1, acc2, 800*) where *acc1* and *acc2* are separate, both are in principle locked throughout the call; but closer examination reveals that all the programmer really needs is a guarantee of correct execution order ("linearization") of any calls *source.f* or *target.f* (where *source* and *target* are the formal arguments) in the body of *transfer*. The actual locking can be deferred until the first such operation actually occurs. Such optimizations can make a dramatic difference in actual performance. Integrating techniques from Software Transactional Memory techniques may also be fruitful; [Nienaltowski 07, 09] presents preliminary ideas.

Although not detailed in the above presentation, **lock passing** is an important property of the model, requiring carefully defined semantics to avoid deadlock. Extensive work is needed on this topic.

More generally, while data races are impossible by construction in the CME model, **deadlock** remains a risk. Ordinary deadlock does not arise because of the power of the object reservation mechanism, locking several processors atomically as in the example calls, *eat* (*left, right*) for philosophers and *transfer* (*acc1, acc2, 800*) for accounts. But it is still possible to create deadlocks in convoluted situations [West 10]. We will address this issue through both static (prevention) and dynamic (detection) techniques. Preliminary *prevention* work has explored order annotations [West 10]. Another approach would take advantage of the PI's recent work on the *alias calculus* [Meyer 11], which defines practical techniques for detecting reference aliasings in an OO program; Coffman-style deadlock can be expressed in the CME model as an aliasing problem. Only initial investigations have been performed in this direction. For deadlock *detection*, the processor-region model seems to allow integrating dynamic deadlock detection as part of the scheduling mechanism.

The **type system** is a prime focus. Work by Nienaltowski and others [Nienaltowski 06] established a strong foundation, relying in part on the concepts of *attached types* [Meyer 05, 10], which guarantee the absence of null pointer dereferencing. Our investigations on the type system will benefit from work on ownership in OO programming (e.g., [Boyapati 03]). The transposition is not immediate: in CME, objects are owned by processors, whereas the traditional ownership models focus on relationships between objects only, on one level. In addition, current ownership models are only poorly integrated with contracts [Dietl 05].

Proof techniques are a major part of the CME effort. The basic ideas were outlined in E4.8. Unlike formal approaches entirely focused on concurrency, the challenge in CME is to integrate proofs of concurrency aspects as part of a more general verification effort applying to all aspects of a program. Our EVE verification environment, under development at ETH, will provide the framework. EVE does not only create new proof tools but also integrates a variety of existing provers such as Boogie from Microsoft Research.

EVE also integrates **automatic testing** techniques such as AutoTest, developed in our group [Meyer 09]. CME will, as noted, continue to explore the application of these techniques to concurrent programming.

Implementation, libraries and libraries are an essential part of the project. The aim is not to produce a concurrent programming model that is elegant on paper, but to solve the major challenge of programming today. Reaching this goal requires a quality implementation. It will be integrated in the EVE Integrated Development Environment, providing the tools that programmers expect today (browsing, debugging etc.) and complemented by a library of reusable components and an extensive set of small and large examples.

A major **textbook** is also part of the project, playing for concurrent programming the role that the PI's earlier books, in particular *Object-Oriented Software Construction*, played for object-oriented programming: enabling a new generation of developers to discover today's most fascinating development in software.

E6 Tasks and deliverables

The overall goal of CME is to achieve the breakthrough needed by the IT industry to take advantage of the concurrent programming revolution, by making safe, simple, clear, efficient concurrent programming a reality. To achieve this goal, CME includes the following major tasks.

1. **Model definition**. Starting from the ideas outlined above, this task defines a comprehensive programming model supporting the construction of concurrent programs and reasoning about them. The model must appear to programmers as an incremental addition to the OO model and make full use of OO techniques and proven software engineering practices. It must be easy to explain, and produce programs with clear semantics.
2. **Reference implementation**. This tasks produces an efficient implementation of the model, integrated into the EVE IDE. The implementation includes both compiler support and a distributed scheduler.
3. **Performance analysis**. This task exercises the implementation extensively to find and remove performance bottlenecks, using techniques to minimize locking as outlined in the previous section, but also lock-free data structures and other modern concurrency algorithms.
4. **Theory and formal basis**. This task defines operational and axiomatic semantics for the model, integrated with a description of the sequential parts and serving as a basis for the next task.
5. **Proof techniques and tools**. This task builds on our group's current work, so far applied to sequential programming only, to integrate a variety of provers in the EVE verification environment. EVE uses a "Verification As a Matter Of Course" approach where verification techniques do not imply a heavy new process but are integrated through feedback to the developer during program construction, using a "blackboard" architecture that arbitrates between the suggestions of diverse verification tools such as automatic and interactive provers, invariant inference, model checkers.
6. **Other verification techniques**. This task complements the previous one by providing support for automatic testing, leveraging on our previous work on the AutoTest framework [Meyer 09].
7. **Library of concurrency components and examples**. This task builds a rich repository of components to support the model for various languages (such as Eiffel, Java, C#) and numerous examples.
8. **Publication**: Scholarly articles, but also a textbook on concurrency, which we hope to make a classic for introducing future generations to concurrency viewed as an essential component of software development.

Deliverables

The CME project has a simple delivery scheme with five full releases, at one-year intervals starting one year after the inception of the project. Each delivery will include a new version of all the items listed above:

- Progress report on each item: concepts proposed, issues encountered, solutions retained.
- New software releases for the items that result in software artifacts.
- A general progress report (publications, seminars, courses etc.)

E7 Risks

CME is a major attempt at changing the way we produce software. A number of factors could make it fail. We identify four major risk factors and planned ways of dealing with them.

- The project's very ambition could scare off potential users.
  This is a risk, but the problems are so critical and the need so crying that there will be a fundamental advantage to the first good solution that becomes available.
- The use of Eiffel could cause concerns.
  Eiffel is only a presentation vehicle. The CME results will be applicable to any modern OO language.
- The performance could be insufficient.
  Getting the highest performance for concurrent applications is at the core of the project. The novel distributed scheduling techniques that we have started to investigate and intend to develop should ensure that only a fraction of the "virtual" locking guaranteed to programmers results in physical locking.
- We could fail to develop the proof theory or tools.
  This is a challenging research area. The PI's record (including a development environment used by mission-critical applications of millions of lines each and, on the academic side, a long track record described in part B1) is solid evidence that we are equipped to tackle it.

The PI's earlier work on object technology and Design by Contract turned arcane subjects, until then largely understood as collections of tricks, into clearly defined disciplines with a sound scientific foundation, directly usable for both teaching and massive industry use. The goal of the CME project is to achieve the same effect for the most important technology for the future of software: concurrent programming.

c. Resources (incl. project costs)

[This section giving the detailed budget of the project has been removed from the present document. The total budget is approximately 2,485,000 EUR, i.e. around 3 M CHF (at Nov. 2011 rates), over a period of five years starting 1 April 2012.]

Bibliography

[Agha 90] Agha G., *Concurrent Object Oriented Programming*, in Comm. of the ACM, 1990, 33(9):125-141
[Agha 93] Agha G., Wenger P., Yonezawa A. (eds.), *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge (Mass.), 1993
[Alavi 10] Alavi M., *SCOOP in Practice*, Technical Report 717, ETH Zurich, January 2010.
[Allen 04] Allen E. et al., *Object-oriented units of measurement*. Proceedings of OOPSLA, 2004, 384-403.
[Alonso 04] Alonso G. et al., *Web Services: Concepts, Architectures and Applications*, Springer, 2004.
[America 89] America P., de Bakker J., Kok J.N., Rutten J.J.M.M, *Denotational Semantics of a Parallel Object-Oriented Language*, Information and Compution, 1989, 83(2): 152-205
[Anderson 07] Anderson T. et al., *Pillar: A Parallel Implementation Language,* In 20th International Workshop on Languages and Compilers for Parallel Computing , 2007
[Andrews 00] Andrews G.R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
[Andrews 83] Andrews G.R., Schneider F.B., *Concepts and notations for concurrent programming*, ACM Computing Surveys, 1983, 15(1): 3-43
[Armstrong 10] Armstrong J., *Erlang*. Communications of the ACM, 2010, 53(9):68-75.
[Axum 10] *The Axum Programming Language*, Microsoft Corporation, http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx, retrieved January 2010.
[Bal 89] Bal H.E., Steiner J.S., Tanenbaum A.S., *Programming languages for distributed computing systems*, ACM Computing Surveys, 1989, 21(3): 261-322
[Balter 91] Balter R. et al., *Architecture and Implementation of Guide, an Object-Oriented Distributed System*, in Computing Systems, 1991, vol. 4
[Barnett 04] Barnett. M., Leino K.R.M., Schulte W., *The Spec# Programming System: An Overview*. In CASSIS 2004, volume 3362 of Lecture Notes in Computer Science, 2004, Springer
[Barnett 05] Barnett B., Chang B.-Y. E., DeLine R., Bart Jacobs, Leino K.R.M.: *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. FMCO 2005: 364-387
[Basler 09] Basler G., Mazzucchi M., Wahl T., Kroening D., *Symbolic Counter Abstraction for Concurrent Software*, In Proceedings of CAV 2009, LNCS, vol. 5643, Springer-Verlag, 2009, 64-78.
[Bay 08] Bay T.:, *Hosting Distributed Software Projects: Concepts, Framework and the Origo Experience*, PhD thesis, ETH Zurich, January 2008
[Ben Ari 06] Ben-Ari M., *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006.
[Benton 04] Benton N., Cardelli L., Fournet C., *Modern Concurrency Abstractions for C#*, ACM Transactions on Programming Languages and Systems (TOPLAS), 2004, 26(5), 269-804.
[Blumofe 95] Blumofe R.D., Joerg C.F., Kuszmaul B.C., Leiserson C.E., Randall K.H., Zhou Y., *Cilk: an efficient multithreaded runtime system*. ACM SIGPLAN Notices, 1995, 30(8):207-216.
[Bornat 05] Bornat R., Calcagno C., O'Hearn P.W., Parkinson M.J., *Permission Accounting in Separation Logic*, In POPL, 2005, 259-270.
[Boyapati 03] Boyapati C., Liskov B., Shrira L. *Ownership types for object encapsulation*. SIGPLAN Not. 2003, 38(1): 213-223.
[Brat 00] Brat G., Havelund K., Park S., Visser W., *Model checking programs*. In ASE, 2000, 3-12.
[Burdy 03] Burdy L. et al., *An Overview of JML Tools and Applications.* ENTCS, 2003, 80
[Cardelli 00] Cardelli L., Gordon A.D., *Mobile Ambients*, Theo. Comp. Sci., 2000, 240(1):177-213.
[Caromel 93] Caromel D., Towards a Method of OO concurrent programming, CACM 36 (9), 1993, 90-102
[Charles 05] Charles P., Grothoff C., Saraswat V.A., Donawa C., Kielstra A., Ebcioglu K., von Praun C., Sarkar V., *X10: An object-oriented approach to non-uniform cluster computing*. OOPSLA, 2005, 519-538.
[Ciupa 08] Ciupa I., Leitner A., Meyer B., Oriol M., Pretschner A., F*inding Faults: Manual Testing vs. Random Testing.* International Symposium on Software Reliability Engineering, 2008, 157-166.
[Clarke 99] Clarke E.M., Grumberg O., Peled D.A., *Model Checking*, MIT Press, 1999.
[Cook 90] Cook W.R., Hill W.L., Canning P.L., *Inheritance is not Subtyping*, In POPL, 1990, 125-135
[Cousot 77] Cousot P., Cousot R., *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In POPL, 1977, 238-252.
[Crnogorac 98] Crnogorac L., Rao A. S., Ramamohanarao K., *Classifying Inheritance Mechanisms in Concurrent Object Oriented Programming*. In ECOOP'98. pages 571–600, 1998.
[Dhara 95] Dhara K., Leavens G., *Weak Behavioral Subtyping for Types with Mutable Objects*. Electronic Notes on Theoretical Computer Science, 1995, 1.
[Dietl 05] Dietl W., Müller P., *Lightweight Ownership for JML,* Journal of Object Technology, 2005
[Dijkstra 68] Dijkstra E., *Co-operating Sequential Processes*, in Prog. Lang., Academic Press, 1968, 43-112
[Dodds 09] Dodds M. et al., *Deny-Guarantee Reasoning*, In ESOP 2009, 363-377.
[Downey 05] Downey A., *The Little Book of Semaphores*, Green Tea Press, 2005.
[ECMA 367] ECMA, Eiffel: Analysis, Design and Programming Language, ECMA Standard 367, 2nd revision, 2006; also ISO/IEC 25436 of the International Standards Organization, 2006.
[Fähndrich 06] Fähndrich M. et al., *Language Support for Fast and Reliable Message-based Communication in Singularity OS,* ACM SIGOPS Operating Systems Review*,* 2006, 40(4):177-190.

[Flanagan 04] Flanagan C., Freund S., *Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs*. In Proceedings of POPL, 2004, pages 256–267
[Fournet 00] Fournet C., Gonthier G., *The Join Calculus: A Language for Distributed Mobile Programming*, Lecture Notes in Computer Science, vol. 2395, Springer-Verlag, 2002, 268-332.
[Gelernter 85] Gelernter D., *Generative Communication in Linda.* ACM TOPLAS, 1985, 7, 80-112.
[Gribomont 90] Gribomont E., *Development of Concurrent Systems by Incremental Transformations*. In 3rd European Symposium on Programming (ESOP ’90), 1990, pages 161–176
[Harris 03] Harris T., Fraser, K., *Language support for lightweight transactions*, In OOPSLA, 2003, 388-402
[Harris 05] Harris T., Marlow S., Peyton-Jones S., and Herlihy M., *Composable memory transactions*, in ACM Conference on Principles and Practice of Parallel Programming 2005 (PPoPP'05), ACM, 2005.
[Herlihy 08] Herlihy M., Shavit N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
[Herlihy 93] Herlihy M., Moss E.B., *Transactional Memory: Architectural Support for Lock-Free Data Structures*, In 20th Annual International Symposium on Computer Architecture, 1993, 289-300
[Hewitt 73] Hewitt C., Bishop P., Steiger R., *A Universal Modular Actor Formalism for Artificial Intelligence*, In Proceedings of the 3rd International Joint Conference on AI, 1973, 235-245.
[Hoare 69] Hoare, C., *An Axiomatic Basis for Computer Programming*. Comm. ACM, 1969, 12(10):576–585
[Hoare 74] Hoare C.A.R., *Monitors: An Operating System Structuring Concept*,CACM,1974,17(10):549-557
[Hoare 85] Hoare C.A.R., *Communicating Sequential Processes*, Prentice Hall International, 1985
[Intel 06] Intel Corporation, Parallelism Drives Performance: A Perspective on the Future of Processing, Intel Software Insight Magazine, August 2006.
[Jacobs 04] Jacobs B., Leino K., Schulte W., *Verification of Multithreaded Object-Oriented Programs with Invariants*. In ACM Workshop on Specification and Verification of Component Based Systems, 2004
[Jézéquel 96] Jézéquel J.-M., *Object-Oriented Software Engineering with Eiffel*, Addison-Wesley, 1996
[Jones 83] Jones C. B., *Tentative Steps Toward a Development Method for Interfering Programs*. ACM Transactions on Programming Languages and Systems, 1983, 5(4):596–619
[Joyner 06] Joyner M., Chamberlain, B.L., Deitz, S.J., *Iterators in Chapel,* In IPDPS 2006, 2006
[Kahlon 07] Kahlon V., Yang Y., Sankaranarayanan S., Gupta A., *Fast and accurate static data-race detection for concurrent programs*. In CAV, 2007, 226-239.
[Kleuker 97] Kleuker S.*, Incremental Development of Deadlock-Free Communicating Systems*. TACAS. 1997, 306–321
[Kundu 07] Kundu A., Eugster P., *Practical Strengthening of Preconditions,* Tech. Rep., Purdue Univ, 2007.
[Kundu 10] Kundu S., Ganai M. K., and Wang C., Contessa: Concurrency testing augmented with symbolic analysis, in CAV'10, volume 6174 of Lecture Notes in Computer Science, Springer, 2010, 127-131.
[Lea 93] Lea R. et al., *COOL: System Support for distributed Programming*, in [Meyer 93], 37-46
[Leavens 05] Leavens G. et al.*, JML Reference Manual*, 2005.
[Leino 04] Leino, K., Müller P., *Object Invariants in Dynamic Contexts*. In ECOOP, 2004
[Leino 09] Leino K.R., Müller P., *A basis for verifying multi-threaded programs*. In ESOP, 2009, 378-393.
[Leitner 07] Leitner A., Ciupa I., Oriol M., Fiva A. and Meyer B., *Contract-Driven Development = Test Driven Development - Writing Test Cases*, in *ESEC/FSE*, 2007.
[Ling 03] Ling S., Poernomo, Schmidt Heinz, *Describing Web Service Architectures through Design-by-Contract*, in ISCIS 2003, eds. A. Yazici and C. Sener, Springer LNCS 2869, 1008-1018
[Lipton 75] Lipton R., *Reduction: a Method of Proving Properties of Parallel Programs*. Comm. ACM, 1975, 18(12):717–721
[Liskov 94] Liskov B., Wing J., *A Behavioral Notion of Subtyping*. ACM TOPLAS, 1994, 16(6):1811–1841
[Matsuoka 93] Matsuoka S., Yonezawa A., *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in [Agha 93], 107-150
[Markoff 07] Markoff J., *Faster Chips Are Leaving Programmers in Their Dust*, New York Times, 2007
[Meyer 91] Meyer B., *Eiffel: The Language*, Prentice Hall,1991.
[Meyer 93] Meyer B. (ed.), Special issue on Concurrent OO Programming, in Comm. ACM, 1993, 36(9)
[Meyer 93a] Meyer, B. *Systematic Concurrent Object-Oriented Programming*, in *Comm. ACM,* 36(9):56-80.
[Meyer 97] Meyer B., *Object-Oriented Software* Construction, Second Edition, Prentice-Hall, 1997
[Meyer 03] Meyer B., *The Grand Challenge of Trusted Components*, in *ICSE*, May 2003, pages 660-667.
[Meyer 03a] Meyer B., *The Outside-In Method of Teaching Introductory Programming*, in *Perspective of System Informatics*, Lecture Notes in Computer Science 2890, Springer-Verlag, 2003, 66-78.
[Meyer 05] Meyer B., *Attached Types and Their Application to Three Open Problems of Object-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP 2005), 2005, 1-32
[Meyer 06] Meyer B., Arnout K., *Componentization: the Visitor Example*, in *IEEE Computer*, 39(7):23-30.
[Meyer 06a] Meyer B. *Teachable, Reusable Units of Cognition*, in *IEEE Computer*, 39(4): 20-24.
[Meyer 07] Meyer B., Ciupa I., Leitner A., Liu L., *Automatic testing of object-oriented software*, in *SOFSEM 2007* (Current Trends in Theory and Practice of Computer Science), January 20-26, 2007, LNCS
[Meyer 09] Meyer B. et al.: *Programs that Test Themselves*, IEEE *Computer*, Sept. 2009, 46-55
[Meyer 10] Meyer B. et al., *Avoid a Void: the Eradication of Null Pointer Dereferencing*, in Reflections on the work of C.A.R. Hoare, eds. C.B. Jones, A.W. Roscoe and K.R. Wood, Springer, 2010, 189-211.

[Meyer 11] Meyer B., *Steps Towards a Theory and Calculus of Aliasing*, in Broy Festschrift, ed. M. Wirsing, Springer-Verlag LNCS, 2011, to appear.
[Microsoft 07] Microsoft Corporation, *Supercomputing Expert Daniel Reed Joins Microsoft Research*, Press Release, at http://www.pressmediawire.com/article.cfm?articleID=3639
[Milner 89] Milner R., *Communication and Concurrency*, Prentice Hall International, 1989
[Milner 99] Milner R., *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge Univ. Press, 1999
[Morandi 08] Morandi B., Bauer S., Meyer B., *SCOOP – A Contract-Based Concurrent Object-Oriented Programming Model*, in LNCS 6029, LASER Summer School 2008: 41-90.
[Morandi 11] Morandi B., Nanz S., Meyer B., *A Comprehensive Operational Semantics of the SCOOP Programming Model*, Technical Report, http://arxiv.org/abs/1101.1038, 2011.
[Nanz 11] Nanz S., Torshizi F., Pedroni M., Meyer B., *Empirical Assessment of Languages for Teaching Concurrency: Methodology and Application*, in CSEE&T 2011, IEEE Computer Society, 2011. To appear.
[NAS 11] US National Academy of Science: *The Future of Computing Performance: Game Over or Next Level?*, 2011 report, (http://www.nap.edu/openbook.php?record_id=12980&page=14#) .
[Musuvathi 08] Musuvathi M. et al., , *Finding and Reproducing Heisenbugs in Concurrent Programs*, In OSDI, 2008, 267-280.
[Nielson 04] Nielson F., Nielson H.R., Hankin C., *Principles of Program Analysis*, Springer-Verlag, 2004.
[Nienaltowski 06] Nienaltowski P., Meyer B., *Contracts for Concurrency*. In First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), 2006
[Nienaltowski 07] Nienaltowski P.: *Practical framework for contract-based concurrent object-oriented programming*, PhD dissertation 17061, Department of Computer Science, ETH Zurich, February 2007.
[Nienaltowski 09] Nienaltowski P., Meyer B., Ostroff J.S., *Contracts for concurrency*, in Formal Aspects of Computing Journal 21(4): 305-318, 2009.
[Nierstrasz 92] Nierstrasz O., *A Tour of Hybrid: A Language for programming with Active Objects*, in Advances on Object-Oriented Software Engineering, Prentice-Hall, 1992, 167-182
[Nordio 08] Nordio M., Müller P., Meyer B.: *Proof-Transforming Compilation of Eiffel Programs*. TOOLS 2008, 316-335
[Nordio 09] Nordio M., Proof-Transforming Compilation, PhD Thesis, ETH Zurich, Nov. 2009
[O'Hearn 01] O'Hearn P.W., Reynolds J.C., Yang H., *Local Reasoning about Programs that Alter Data Structures*, Computer Science Logic, 2001, 1-19.
[O'Hearn 07] O'Hearn P.W., *Resources, Concurrency and Local Reasoning*, Theoretical Computer Science (Reynolds Festschrift), 2007, 375(1-3):271-307.
[Odersky 08] Odersky M., Spoon L., Venners B., *Programming in Scala*, Artima, 2008.
[Ostroff 09] Ostroff J.S., Torshizi F.A., Huang H.F., Schoeller B.: *Beyond contracts for concurrency*. Formal Aspects of Computing, 2009, 21(4):319-346
[Owicki 76] Owicki S., Gries D., *Verifying Properties of Parallel Programs: An Axiomatic Approach*. Communications of the ACM, 1976, 19(5):279–285
[Papathomas 92] Papathomas M., *Language design rationale and semantic framework for concurrent object-oriented programming*, PhD Thesis, University of Geneva, Switzerland, 1992
[Petitpierre 98] Petitpierre C., *sC++: Concurrent Object Oriented Programming, Verification, Modeling*, Presses Polytechniques et Universitaires Romandes, Fribourg, Switzerland, 1998
[Petri 62] Petri C.A., *Kommunikation mit Automaten*, Ph.D. Thesis, University of Bonn, 1962.
[Philippsen 00] Philippsen M., *A survey of concurrent object-oriented languages*, Concurrency: Practice and Experience, 2000, 12:917-980
[Pnueli 77] Pnueli A., *The Temporal Logic of Programs,* In FOCS, 1977, 46-57
[Ramanathan 10] Ramanathan F., Morandi B., West S., Nanz S., Meyer B., *Deriving Concurrent Control Software from Behavioral Specifications*, in Intelligent Robots and Systems (IROS'10), IEEE, 2010.
[Ramsey 00] Ramsey N., Jones S. P., *A single intermediate language that supports multiple implementations of exceptions*. In PLDI, 2000, 285-298
[Robby 06] Robby, Rodriguez E., Dwyer M., Hatcliff J., *Checking JML Specifications using an Extensible Software Model Checking Framework*. Software Tools for Technolgy Transfer, 2006, 8(3):280–299
[Rodriguez 05] Rodriguez E., Dwyer M., Flanagan C., Hatcliff J., Leavens G., Robby, *Extending JML for Modular Specification and Verification of Multi-threaded Programs*. In ECOOP, 2005, pages 551–576
[Roscoe 97-05] Roscoe A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1997 (revised 2005).
[Schoeller 06] Schoeller B., Widmer T., Meyer B., *Making specifications complete through models*, in *Architecting Systems with Trustworthy Components*, LNCS, vol. 3938, 2006.
[Schoeller 08] Schoeller B., *Making classes provable through contracts, models and frames*, PhD thesis, ETH Zurich, January 2008.
[Sen 06] Sen K., Agha G., *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools*, In CAV, Lecture Notes in Computer Science, vol. 4144, 2006, 419-423.
[Torkington 07] Nat Torkington, *The Faint Signals of Concurrency*, O'Reilly Radar, June 2006
[Vafeiadis 06] Vafeiadis V., Herlihy M., Hoare T., and Shapiro M, *Proving correctness of highly-concurrent linearisable objects*, In PPoPP, 2006, 129-136

[Vafeiadis 07] Vafeiadis V., Parkinson M.J., *A Marriage of Rely/Guarantee and Separation Logic*. In CONCUR 2007, 256-271.
[Wang 06] Wang, L. and Stoller, S. D., *Runtime Analysis of Atomicity for Multithreaded Programs,* IEEE Transactions on Software Engineering, 2006, 32(2):93–110
[Welc 04] Welc A., Jagannathan S., Hosking A.L., *Transactional Monitors for Concurrent Objects*, European Conference on Object-Oriented Programming (ECOOP '04), 2004, 519-542.
[West 10] West S., Nanz S., Meyer B., *A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model*, in Formal Engineering Methods (ICFEM'10), LNCS 6447, Springer, 2010, 597–612.
[West 11] West S., Nanz S., Meyer B., *Demonic Testing of Concurrent Programs*, submitted, 2011.
[Xu 97] Xu Q.-W., de Roever W.-P., He J.-F., *The rely-guarantee method for verifying shared variable concurrent programs*, In Formal Aspects of Computing, 1997, 9(2):149--174
[Yonezawa 87] Yonezawa A., Tokoro M., *Object-Oriented Concurrent Programming*, MIT Press, 1987

[The final section of the proposal covers administrative aspects and has been removed.]