Practical framework for contract-based concurrent object-oriented programming

PhD research plan

Candidate: Piotr Nienaltowski
Chair of Software Engineering, ETH Zurich
CH-8092 Zurich, Switzerland

Supervisor: Prof. Dr. Bertrand Meyer

Start date: 15 April 2002

Expected end date: 28 August 2006

1. THESIS STATEMENT

The main goal of this PhD thesis is to propose and implement a practical methodology for the construction of concurrent object-oriented programs and for reasoning about their properties. We claim that:

- It is possible to carry through the advantages of object technology, such as ease of reuse, extendibility, maintainability of software, to the concurrent context by providing a full support for object-oriented mechanisms and techniques, including inheritance, genericity, polymorphism, dynamic binding, and agents.
- The object-oriented techniques coupled with the Design by Contract **[62]** approach constitute the right basis for the modular development of concurrent systems and allow the programmers to efficiently develop correct, reusable concurrent software with little more effort than sequential one. Thanks to the proposed methodology, concurrent software can be understood, analysed, written, and reused in a much simpler manner than with other state-of-the art techniques.
- Static analysis of object-oriented code can be precise enough to detect and eliminate synchronisation defects such as atomicity violations. A proper type system can statically ensure interesting safety properties of concurrent programs.

2. OVERVIEW

We consider the SCOOP model (Simple Concurrent Object-Oriented Programming) introduced by Bertrand Meyer **[64]** to be a good starting point for our study. SCOOP is a good candidate for modelling object-oriented concurrent applications because it takes advantage of the existing synergies between the O-O concepts and concurrency. Also, it relies on very powerful O-O principles, including Design by Contract. Nevertheless, the insufficient study of the semantics of SCOOP, as well as the lack of a production-quality implementation make it difficult to assess the model with respect to other existing approaches and show that it indeed fulfils its purpose.

This thesis should fill in the gap by providing a working implementation of SCOOP, carrying out an in-depth analysis of the model, formalising it, identifying the inconsistencies, and proposing adequate solutions to the encountered problems:

- We study the relationship between object-orientation and concurrency with a particular focus on the role of assertions (preconditions, postconditions, invariants, modifies clauses) in the concurrent context.
- We provide an operational semantics for the. This allows us to formalise the interesting properties of concurrent systems, such as the absence of atomicity violations and deadlocks.
- We refine the model by relaxing the access control policy to allow shared locking and lock passing. This results in an increased expressivity of the model and makes concurrent applications less prone to deadlocks.
- On the language side, we follow the SCOOP approach and propose a simple extension of Eiffel with a minimal number of additional keywords.

- We introduce a tagged type system to statically eliminate potential atomicity violations, and prove its soundness with respect to the operational semantics.
- We integrate the *agent* mechanism and the *expanded types* with the model.
- We discuss the support for advanced object-oriented mechanisms in SCOOP.
- Finally, we provide a library implementation of the model (SCOOPLI) and a compiler that type-checks SCOOP code and translates it into pure Eiffel code with embedded library calls. A deadlock detection mechanism is implemented as part of SCOOPLI.

We expect that the theoretical and practical results obtained during this research work will constitute a major contribution to the state-of-the-art in concurrent object-oriented programming.

3. KEY QUESTIONS AND EXPECTED RESULTS

3.1. Relationship between object-orientation and concurrency

The relationship between the object-oriented techniques and concurrency has been often discussed in the literature **[4][21][76]**. After the initial enthusiasm ("all objects should be concurrent"), some important problems were identified. The most widely discussed problem is the apparent mismatch between the inheritance mechanism and concurrency **[76]**. Several kinds of *inheritance anomalies* have been identified **[60]**. The authors often conclude that the full integration of object-orientation and concurrency is not possible.

These problems are principally caused by the approach taken by most authors: the *extension* of an existing model or programming language. We agree that it is very difficult to extend an object-oriented language with concurrent capabilities without sacrificing some of its expressiveness **[76]**. Similarly, adding object-orientation on top of a concurrent language usually results in a hybrid language with a restricted support for object-oriented programming.

Therefore, we have decided to look much deeper into the relationship between object-orientation and concurrency, and search for a solution that relies on the *fusion* of both worlds. We take the following approach:

- We observe that all object-oriented computation is concurrent by nature.
- Sequential systems are a (small) subset of of all potential software systems. Certain properties of objects (such as their locality with respect to other objects, etc.) are less important in the sequential context but they must not be simply ignored or forgotten when no concurrency is involved.
- Following that second observation, we build a general model for object-oriented concurrency such that all the mechanisms known from sequential object-oriented programming are duly represented. Usually, this involves the need to abstract a given mechanism by giving it a more general semantics. In particular, we propose a new semantics for feature calls, argument passing, preconditions, postconditions, and invariants (see sections 3.2 and 3.3). We show that such a

semantics naturally reduces to the "obvious" sequential semantics in a non-concurrent context.

This is a novel approach: rather than looking for a smallest step from the world of sequential object-oriented programming into the world of concurrency (it is also the approach taken by the original SCOOP), we are looking for the smallest step that takes us from the concurrent object-oriented world into the sequential world. Under way we realise that, in fact, these two worlds are not separate, and that the well-understood object-oriented mechanisms and techniques have a more general semantics. It has a very positive impact on the reusability of software (see section 3.9).

3.2. Design by Contract in the concurrent context

Design by Contract **[62]** allows the programmer to enrich class interfaces with *contracts.* Contracts express the mutual obligations of *clients* and *suppliers* through the use of assertions. Routine *pre-conditions* specify the obligations on the routine client and the guarantee given to the routine supplier. Conversely, routine *post-conditions* express the obligation on the routine supplier and the guarantee given to the routine client. Class *invariants* express the correctness criteria of a given class – an instance of a class is in a consistent state if and only if the corresponding invariant holds in every observable state.

The modular design fostered by encapsulation and DbC reduces the complexity of software – correctness considerations can be confined to the boundaries of modules (classes) that can be proved and tested in isolation. The principles of Design by Contract can be used for proving correctness of a class. A class C is *locally correct* if:

- After creating a new instance of C, the class invariant Inv_C holds, and
- After execution of a routine r of class C, both the class invariant and the post-condition $Post_r$ of that routine hold, provided that both the invariant and the precondition Pre_r were fulfilled at the time of the invocation.

Clients can rely on the interface of a class without the need to know about the implementation details.

We aim at applying the local correctness to our model. In a more general (concurrent) setting, the semantics of assertions has to be re-considered, so that:

- it is possible to use similar rules as in the sequential setting to prove the correctness of classes,
- the generalised semantics nicely reduces to the "usual" sequential semantics in a non-concurrent setting.

We think that it is possible to achieve these two goals. Obviously, we do not claim that the proposed rules will be as simple as the sequential ones – since the concurrency is achieved through the interplay (interference) of several processors, this interference must be taken into account by the proof rules.

The focus of this work is not to provide a proof system for concurrent object-oriented programs but rather to lay a solid basis for the future development of such a system by providing a clear generalised semantics of assertions. We focus on pre-conditions, post-conditions, and invariants. We also consider the *modifies clauses* (expressed in Eiffel through the *ensure only* construct). Other assertions, such as checks, loop variants and

loop invariants, are only discussed shortly because their generalised semantics is very close to the sequential semantics as defined by the DbC.

The most interesting result obtained so far is the new semantics for pre-conditions - they can be seen as *wait-conditions*. The concept of wait-conditions was already introduced in the original SCOOP design but the model treated them as a "hijacked" pre-conditions, much closer to the concept of *guards* (this also raised the problem of wait-condition weakening vs. guard strenghtening). Our methodology does not discriminate between (sequential) pre-conditions and (separate) wait-conditions; we give a simple semantics to pre-conditions that caters for the needs of concurrency and nicely reduces to the sequential semantics. A (surprising at first) by-product of this research is the formalisation of a *pre-condition violation* as a kind of *deadlock*. This also leads to a deeper understanding of the rôle of pre-conditions in a program: we claim that pre-conditions are an integral part of software and they must not be ignored at execution. It seems to contradict the common approach to software construction where assertion checking is turned off before releasing the finalised version of a software system. In fact, there is no contradiction there: thanks to certain (very strong) assumptions that we can make in a sequential context, it is much easier to prove that the given property will hold immediately when required, thus making the pre-condition redundant. In a general (concurrent) setting, such situations are much less common, therefore assertions cannot be ignored at run-time.

DONE:

- Semantics of pre-conditions (informal)
- Semantics of argument passing (informal)

MISSING:

- Semantics of post-conditions and invariants
- Relation between contracts and assumption-commitment in SCOOP
- A sketch proof rule for SCOOP programs

NEXT STEPS:

- Read about assumption-commitment
- Define semantics of post-conditions and invariants

DEADLINE:

- 15/12/2005

3.3. Formalisation of the computational model

We provide an operational semantics for the model in the form of a labelled transition system (LTS). This allows us to give very precise meaning to all the constructs and mechanisms of the model, and express the interesting properties of concurrent systems, such as the absence of atomicity violations and deadlocks.

We follow the approach taken by G. Jalloul in her PhD thesis **[47]**. She proposed a simple LTS-based operational semantics for a concurrency model called CSS (Communicating Sequential Subsystems). CSS is interesting because it has several features that are similar

to the corresponding features of the SCOOP model. For example, CSS's subsystems resemble SCOOP's processors – both of them represent independent threads of execution and units of modelling. Also, an extension of Eiffel (CEE) was proposed for CSS – we are going to take advantage of that by reusing the part of the semantics related to the sequential subset of the language.

G. Jalloul made several assumptions that facilitated the development of the operational semantics. First of all, only a subset of CEE was considered. The following features were omitted:

- Expanded types
- Exceptions
- Assertions
- Input/output operations and *external* features
- *Once* and *unique* declarations
- Assignment attempt
- *holdif* statement (a CCR-like construct that plays a similar role as a feature call with separate arguments in SCOOP)

Also, the semantics did not discriminate between *deadlock* and *termination*.

We also make certain simplifying assumptions. In particular, we do not consider exceptions, input/output operations, *external, once*, and *unique* features, and *agents* (note that the integration of the latter with the model is discussed separately, see section 3.7). The modelling of these features is not essential to this study. In fact, the only non-trivial feature that we omit is the exception handling but this topic is treated exhaustively in the research work of another team member **[6]**. We also assume the absence of hardware and communication failures – this topic is closely related to distributed programming and treated separately in the research work of another team member **[86]**.

We do model assertions: preconditions, postconditions, and invariants. In fact, this is the most interesting and novel aspect of the proposed semantics. We demonstrate that the general semantics of assertions reduces to the "usual" sequential semantics in a non-concurrent context. A very interesting outcome of this work is the formalisation of a *precondition violation* as a kind of *deadlock*.

The operational semantics serves as a basis for the soundness proof of the proposed type system (see section 3.6).

DONE:

- First and second reading of Ghinwa's dissertation
- Modelling of pre-condition violation as deadlock

MISSING:

- Choice of language subset / intermediate language
- Operational semantics for the model
- Modelling of post-condition violation as deadlock
- Modelling of invariant violation as deadlock

NEXT STEPS:

- Sketch of intermediate language
- Sketch of operational semantics based on Ghinwa's and Arnaud's work

DEADLINE:

- 01/03/2006

3.4. Refinement of the access control policy

The locking policy of the SCOOP model is very restrictive: at most one client object may access the given supplier object at any time. If we consider the fact that *processors* are sequential, that is at most one feature is executed on a processor at any time, this ensures the absence of data races and gives the client a very strong guarantee of exclusive access to a processor. Therefore, one can easily decide which object is responsible for possible breaches in the contract (e.g. breaking the invariant of the class corresponding to the supplier object).

The biggest drawback of this approach is, of course, the loss of performance: very often, several clients try to access a shared data structure and only one of them is allowed to make a call at a time; the others have to wait, even if they do not try to modify the data structure. The solution seems to be obvious: we need to introduce some kind of "shared access". The idea itself is not new but the solution is not trivial since we want to increase the amount of concurrency without penalising the clients by weakening the guarantees they get.

We investigated the possibility of allowing the *intra-object concurrency* in our model, i.e. making the processors multi-threaded. We would only allow non-conflicting features to be executed in parallel on the same supplier. Unfortunately, the problem of non-interference of features is not decidable in the presence of aliasing and polymorphism. We proposed a very restricted solution **[67]** that would only allow so-called *pure functions* to be executed in parallel. Nevertheless, even that solution was not sound if we considered the standard notion of *purity* applied in the object-oriented languages (no side-effects, except for temporary ones). The main problem here was the impossibility of ensuring that the invariant of the class and the post-condition of a feature hold when the feature terminates its execution if another pure feature temporarily modifies the state of the supplier object. For example, a feature of class *LIST* that advances the cursor, performs a read operation, and moves the cursor back to its initial position, would be considered as pure in a sequential context but it may lead to an invariant violation if another feature has been executed and terminated before the cursor is moved back to its initial position.
In fact, in our initial approach, only attributes could be considered as pure enough to be evaluated in parallel (although in the presence of polymorphism, if the redefinition of an attribute into a routine is allowed, even attributes are not pure!).

We identified two problems with the initial solution. First of all, providing a "stronger" notion of purity in the concurrent context would contradict the overall approach we took – remember that we want to find a *more general* definition of purity that would reduce to the usual, sequential semantics. The second problem, a more practical one, was that the sequential code would not be reusable in a concurrent context, so programmers would

need to write their code so that it is “concurrency-friendly”. This would mean that our model does not really fulfil its purpose (a complete fusion of concurrency and object-orientation). Also, most of the existing code would be useless in a concurrent context (including all the data structures from EiffelBase).

We have decided to take a different approach that led us to the re-design of the access control policy of our model and to a proper definition of purity. We allow shared access to suppliers but *processors remain sequential*. This is a crucial difference with respect to the previous attempt. The parallel execution of several features is achieved through the *interleaving* rather than multi-threading. Only pure functions may be executed in the shared mode but this time the usual (weaker) notion of purity is applied. This allows us to preserve the possibility of assertional reasoning about software. Note that, from the clients’ point of view, the guarantees are as strong as in the SCOOP model. The interleaving satisfies the *serialisability* property. Therefore, a client can think of itself as a sole client of the given processor and does not even realise that other clients’ requests are interleaved with its own request. When a client requests a non-pure feature, the exclusive access to the given processor is required, just like in the original SCOOP model.

Another refinement of the access control policy is the introduction of *lock passing*. It increases the expressivity of the model – certain scenarios can be implemented now that were impossible to implement in SCOOP. It also allows programmers to better tailor their code. The mechanism makes concurrent programs less deadlock-prone. Lock passing relies on a new semantics for argument passing and attached types. Essentially, if a client *c* has exclusive access to its supplier *x* and some other object *y*, and it issues a separate call *x.f* (*y*) then, if the formal argument corresponding to *y* is declared as attached, the call will be executed synchronously, with *c* shedding all its locks to *x* and waiting until the execution of *f* terminates, then revoking all its locks from *x* and continuing its own execution.

To summarise: we refine the model by relaxing the access control policy to allow shared locking and lock passing. This results in an increased expressivity of the model and makes concurrent applications less prone to deadlocks. The ability to reason about the software in terms of pre-conditions, post-conditions, and invariants is preserved. The enhanced access control policy proves very useful, in particular in the applications where multiple clients access shared data and most operations are pure (“read”) features.

DONE:

- Locking based on attached/detachable
- Lock-passing based on argument passing
- Shared locking based on pure queries (draft)

MISSING:

- Formal description of lock-passing
- Precise definition of pure queries
- Precise locking rules for shared locks

NEXT STEPS:

- Get the definition of pure queries

- Decide about the best notation ("only" vs. "pure" keyword)
- Describe precisely the lock passing mechanism

DEADLINE:

- 20/12/2005

3.5. Language extensions

On the language side, we follow the SCOOP approach and propose a simple extension of Eiffel with a minimal number of additional keywords. Eiffel offers a full support for Design by Contract and object-oriented techniques, including certain advanced mechanisms that are (or were, until recently) not present in other languages that we could consider (Spec#, JML). At the same time, the presence of these advanced mechanisms (multiple inheritance, genericity, agents, expanded types) represents a real challenge for us - the design and implementation of the concurrency model must take all these mechanisms into account and cater for their needs. Also, the presence of rich, carefully designed libraries and software based on these libraries, rises the issue of backward-compatibility.

The careful design of the language helps us keep the need for syntactic extensions to a bare minimum. We decided to reuse the *separate* keyword introduced by SCOOP (although the semantics of that construct is slightly different); we also use two additional keywords – *domain* and *in* – for the extension of the type system.

As a result, the language extension is fully backward-compatible with the standard Eiffel language [ECMA]. Existing Eiffel classes can be used in programs based on our model.

DONE:

- Domains

MISSING:

- "like"-notation for domains
- Notation for pure queries

NEXT STEPS:

- "like"-notation for domains
- Notation for pure queries

DEADLINE:

- 01/04/2006

3.6. Extended type system for the elimination of potential synchronisation defects

We introduce an ownership-like type system to statically eliminate potential atomicity violations, and prove its soundness with respect to the operational semantics.

Let *TypeId* denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$TaggedType = OwnerId \times TypeId$$

where *OwnerId* is a set of owner tags declared in the given class. Each class implicitely declares two owner tags: • (current processor) and ⊥ (unknown).

The subtype relation $\prec$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where α is a tag, $S,T \in TypeId$, and $\prec_{Eiffel}$ denotes the conformance relation on TypeId:

$$(\alpha, T) \prec (\alpha, S) \Leftrightarrow T \prec_{Eiffel} S$$

$$(\alpha, T) \prec (\bot, T)$$

The extended type system allows us to reason about the *locality* of objects that are represented by entities. Typing rules are defined in such a way that all potential atomicity violations are eliminated (the intervleaving of pure functions is not regarded as an atomicity violation). This allows us to ensure interesting safety properties of concurrent systems. The use of detachable and attached types makes it possible to integrate seamlessly the proposed extension of the access control policy into the language.

The rules are straightforward – in fact, the proper definition of the notion of subtyping takes care of most problems. Particular rules are only required to accommodate the expanded types and the agent mechanism (see section 3.7). Here is an example rule (for assignment):

$$\text{[Assign]} \qquad \frac{\Gamma \vdash l :: (\alpha, T),\ \Gamma \vdash e :: (\beta, S),\ (\beta, S) \prec (\alpha, T)}{\Gamma \vdash l := e}$$

Note that our task is simplified by the fact that the model is free of data races by construction. The only synchronisation defects we have to consider here are atomicity violations. The problem of deadlocks is treated separately (see section 3.8).

DONE:

- Subtyping relation based on tagged types (draft)
- Basic type rules: assignment, procedure call, function call (draft)
- Rules for feature redefinition (draft)

MISSING:

- Simplified language definition for a subset of Eiffel
- Formalisation of domain notation
- Formalisation of rules
- Proof of soundness w.r.t. operational semantics

NEXT STEPS:

- Review the drafts (until 24/11/2005)
- Decide about the subset of language

DEADLINE:

- 01/05/2006

3.7. Support for advanced object-oriented mechanisms

Inheritance

One of the most powerful concepts of object-oriented programming is inheritance, especially when it is used for code reuse and refinement. Unfortunately, in the presence of concurrency coordination code, reuse problems are likely to occur. In general, there is a high interdependence between attributes of a class and coordination constraints of different routines. Concurrency coordination and functional code are interwoven. Because of this interdependence, features often cannot be redefined in subclasses without affecting other features. Very often, affected features must be redefined in the descendant and the ancestor, which degrades the maintainability and prevents the reuse of code. Even if the coordination code is isolated from the functional code, it is often necessary to redefine it completely for all inherited features instead of having local extensions of its parts. These and other difficulties in combining inheritance with concurrency are referred to as *inheritance anomalies* **[60][51]**.

We realised that, thanks to the application of the Design by Contract methodology (with a proper redefinition of the semantics of assertions), most inheritance anomalies are not a problem in the proposed model. For example, *partitioning of acceptable states* and *modification of acceptable states* anomalies are prevented by the appropriate rule for feature redefinition (more precisely by the pre-condition weakening rule). Similarly, we do not have any problems with the definition of sets of non-interfering features thanks to the imposed restrictions on interleaving of features (only pure functions might be interleaved) and a proper rule for redefinition of routines in descendant classes. Interestingly, the latter is not specialised for dealing with concurrency. Nevertheless, it proves useful in the concurrent context.

Agents

Another powerful mechanism of object-oriented programming, the *agents*, needs to be carefully analysed in order to be applicable in the general (concurrent) context. We identify the need for a special typing rule for agents and their actual arguments (tuples). Essentially, an agent may be declared as separate and treated as any other separate entity. The semantics of agent creation is refined so that the type (static and dynamic) of the agent object reflects the locality of the agent's target:

```
my_agent: separate PROCEDURE [X, TUPLE]
my_non_separate_agent: PROCEDURE [X, TUPLE]
x: separate X
non_separate_x: X
…
my_agent := agent x.f              -- OK
my_agent := agent non_separate_x.f          -- OK
```

```
my_non_separate_agent := agent non_separate_x.f  -- OK
my_ non_separate_agent := agent x.f              -- Invalid! Type error.
…
r (an_agent: separate PROCEDURE [X, TUPLE]) is
        do
                an_agent.call ([])      -- OK, because the target's processor
                                        -- has been locked.
        end
```

This solution cannot be applied to agents with an open target because the locality of the potential target cannot be considered at the time of agent construction. This is not an issue if the agent under consideration is declared as non-separate. We suggest that a run-time check at call time be performed for separate agents with an open target.

Expanded types

Expanded types seem to be quite tricky to model in a concurrent world without imposing certain restrictions on them. For example, in the original SCOOP model, only "fully expanded" objects are allowed to cross the boundary of a processor (fully expanded objects must not have fields that are non-separate references). This solution is not satisfactory because most expanded types are not fully expanded so their use in a concurrent context would be prohibited.

The problem with finding a satisfactory generalised semantics for the expanded types comes from the fact that such types serve two purposes:

- Objects of an expanded type have copy semantics, i.e. there exists only one "reference" to a given object, an attempt at getting a reference to the object results in the creation of a (shallow) copy of that object.
- Entities of an expanded type directly represent objects rather than references to objects.

If we were able to consider just the first point, i.e. the copy semantics of expanded classes, there would be no problem with the integration of expanded types into the our programming model. Expanded entities could be also declared as separate and treated in the same manner as any other separate entity. This approach is sound and theoretically beautiful but it is very cumbersome in practice – it is hard to imagine that programmers would accept to use such types as *separate INTEGER* or *separate BOOLEAN*.

We are convinced that more appropriate (and more practical) rules can be devised. We suggest that an entity of an expanded type should be seen as a non-separate entity in every typing context, independently of its owner tag. To be more precise, if *ET* is an expanded type, then all the entities below have the type *(•, ET)*:

e1: *ET*

e2: **separate** *ET*

e3: **separate** *ET* **in** *d1*

e3: **separate** *ET* **in** *d1*

When a separate object has to be passed across the boundary of a processor, a *deep import* operation is performed. Deep import is similar to a *deep clone* but it does not follow the separate references. Therefore it can be considered as "deeper" than the (shallow) *clone* operation but more "shallow" than the *deep clone*.

A corresponding type rule is introduced that allows all expanded entities to be seen as non-separate.

DONE:

- Integration of agents
- Integration of expanded types

MISSING:

- Agents with open target
- Implementation (depends on ES 5.7 for expanded types and attached/detachable for agents with open target)

NEXT STEPS:

- Figure out handling of agents with open target (might be based on detachable/attached) (by 23/11/2005)

DEADLINE:

- 01/05/2005

3.8. Deadlocks and their treatment in the object-oriented world

Absence of deadlocks is one of the most interesting properties of concurrent programs. In fact, the problem of deadlocks was one of the initial motivations of our work. We set off to devise a methodology for deadlock prevention, detection, and resolution in SCOOP programs. We quickly understood that, in a general case, it is impossible to provide a sound and complete modular proof of deadlock-freedom, especially for complex concurrent programs.

Additionally, the results of our work on the semantics of assertions showed that there is a close relation between pre-condition violations and the "usual" deadlocks due to the circular wait. Therefore, we changed our approach and focused on exploring the nature of deadlocks and their relation with other phenomena in the object-oriented world.

The result of our work is the analysis of the rôle of deadlocks in the concurrent object-oriented world, with a particular focus on the relation between assertions and deadlocks. This should constitute a firm basis for the future development of an adequate anti-deadlock methodology. The development of such a methodology is beyond the scope of this thesis. Nevertheless, we do propose and implement an efficient run-time mechanism for deadlock detection as part of the SCOOPLI library (see section 3.10). We expect the deadlock resolution mechanism to be based on the *duel* mechanism.

DONE:

- Run-time mechanism for deadlock detection (Daniel Moser's semester project)

NEXT STEPS:

- Technical report on deadlock detection in SCOOP (with Daniel) (January)

DEADLINE:

- 01/05/2006

3.9. Implementation and tools

The lack of a full implementation was the main difficulty in the assessment of the original SCOOP model. That is why we started our work by implementing the model. We decided to implement SCOOP with an Eiffel library rather than by extending the compiler. This provided several advantages, in particular the ability to 'play' with the model by trying out various refinements and extensions, and to implement it on several platforms without getting bogged down in compilation-related issues.

A first, partial implementation of SCOOP **[68]** targeted the .NET platform. Later on, we ported it to native Windows. In that initial approach we used multiple inheritance to implement separateness – separate types were represented by separate classes that had to inherit from *SEPARATE_SUPPLIER*. A declaration of the form *x*: **separate** *X* had to be transformed into *x*: *SEPARATE_X* where class *SEPARATE_X* was defined as

class *SEPARATE_X*

inherit

X

SEPARATE_SUPPLIER

end

Also, all separate clients (classes that declare and use separate entities) had to inherit from a special class *SEPARATE_CLIENT*.

The biggest problem with that initial, inheritance-based approach was that the conformance relation between classes, as expressed by the inheritance relation, was exactly the opposite of the intended subtype relation (*X* should conform to **separate** *X* and not vice-versa). This made it impossible to pass non-separate objects of a reference type across the boundary of a processor. Additionally, the code had to be instrumented by hand, which was a cumbersome and error-prone process. We developed a pre-processor to aleviate that problem but the tool was not very efficient – due to the abovementioned use of inheritance, the code of all the ancestors of a given class *X* had to be analysed in order to produce correct renaming and redefinition clauses in the wrapper class *SEPARATE_X*.

With the introduction of the conversion mechanism into Eiffel, we decided to change the approach and implement the separateness through the use of proxies. A *separate proxy* for a class *X* does not inherit from *X*. Instead, it delegates the calls to the appropriate features of the actual supplier object. No inheritance relation exists between class *X* and its proxy class *SCOOP_SEPARATE__X* but *X* conforms (is convertible) to *SCOOP_SEPARATE__X*. This captures much more precisely the intended semantics of separate types. Also, a pre-processing tool only needs to look at class *X* when generating proxy code for it – no analysis of proper ancestors is necessary.

We provide a library implementation of the model (SCOOPLI) [91] and a compiler that type-checks SCOOP code and translates it into pure Eiffel code with embedded library calls [92]. The compiler is also able to detect potential deadlocks and use this information as the input to a run-time deadlock detection mechanism implemented as part of SCOOPLI.

DONE:

- SCOOPLI library for POSIX and .NET
- scoop2scoopli

MISSING:

- extended type checker
- support for latest ES 5.6
- detachable/attached
- agents
- expanded types
- constrained genericity
- integration with ES

NEXT STEPS:

- implementation of the type checker (February - April)
- agents, expanded types, attached types (March - April)
- version for ES 5.7 (March – April)

DEADLINE:

- 01/05/2006 (dependent on ES 5.7 to be published in March 2006)

4. THESIS BACKGROUND

4.1. Object-oriented concurrent programming

In traditional sequential programming, the object-oriented model has gained wide acceptance. Concurrent and distributed programming remains, however, one of the last areas of software engineering where no single direction has been generally recognised as the preferred approach. Frameworks such as CSP, CCS, and π-calculi enjoy strong academic support, but they remain far from the techniques actually applied in the industry. Finding a satisfactory framework for concurrent and distributed development is an urgent issue for the industry. Concurrent programming has become a required component of ever more types of application, including some that were traditionally thought of as sequential in nature. Beyond mere concurrency, today's systems have become distributed over networks, including the network of networks — the Internet. The industry is in particularly dire need of simple, teachable techniques, directly supported by tools, which can guarantee the efficient production of correct and robust software providing a high Quality of Service.

Several proposals have been made to combine the advantages of object-oriented programming techniques with the increased power of parallel machines. However, combining both concepts turned out to be extremely difficult in that, depending on the approach, either the main characteristics of the O-O paradigm or key performance factors of parallelism are sacrificed, resulting in unsatisfactory languages **[76]**.

Philippsen **[76]** identifies several programming problems specific to concurrent object-oriented models. We can claim that most of these problems can be avoided by appropriate language design and programming style.

Parallel performance

Three aspects of the problem should be considered: fan-out, intra-object concurrency, and object locality.

Fan-Out

In general, the most inefficient way of spawning activities is to create only one new activity at a time. In languages that only support sequential creation, it takes *O(n)* steps to spawn *n* activities on *n* processors (the cost can be reduced to *O(log n)* using a binary creation tree; such code is, however, difficult to read and understand. Several approaches propose spawning constructs with high fan-out, i.e. they offer the possibility to create more than one activity at a time.

Intra-object concurrency

Most concurrent O-O languages allow at most one feature of a given supplier object to be executed at a time. Other invocations are delayed. The main reason for prohibiting the intra-object concurrency is that it makes it much easier to analyse the semantics of execution and reason about the correctness of a class implementation. Also, most types of inheritance anomalies can be avoided **[60]**, since the programmer does not have to implement any form of concurrency coordination to express which methods can be safely executed in parallel.

Locality

On distributed memory parallel machines, good performance can only be achieved if objects and activities are located on the same node, to avoid the overhead of remote accesses. Most concurrent O-O languages do not consider this problem at all.

Encapsulation

Encapsulation is a central paradigm of the O-O approach and a major software quality factor. The modular design fostered by encapsulation reduces the complexity of software – correctness considerations can be confined to the boundaries of modules (classes) that can be proved and tested in isolation.

The principle of Design by Contract introduced by Bertrand Meyer **[62]** can be used for proving correctness of a class. A class *C* is *locally correct* if:

- After creating a new instance of *C*, the class invariant Inv_C holds, and
- After execution of a routine r of class C, both the class invariant and the post-condition $Post_r$ of that routine hold, provided that both the invariant and the precondition Pre_r were fulfilled at the time of the invocation.

Clients can rely on the interface of a class without the need to know about the implementation details.

Encapsulation by callee-side coordination

The idea is to implement concurrency coordination on the supplier (callee) side. Language constructs that achieve callee-side coordination by design are called *boundary coordination mechanisms*. With such mechanisms, the class implementation can ensure that routines will only be executed concurrently if their interleaving does not affect correctness. Interfering feature invocations will be delayed.

The main advantage of callee-side coordination is that it is possible to reason about the correctness of classes based on local information since all coordination code is part of the class implementation.

Activity-centered coordination

Another approach is to allow for intra-object concurrency but make the *client* (caller) responsible for its coordination. This may break encapsulation for two reasons. First, the client has to know the implementation details of the invoked routine to be convinced that it can be executed concurrently without harmful interference with other routines. Secondly, changing the implementation of a routine in the supplier class requires careful analysis of all the code where that particular routine is called, since the new implementation might require coordination constraints not yet implemented in all the client classes. Both problems are particularly painful where dynamic binding and code reuse come into play.

Inheritance anomalies

One of the most powerful concepts of object-oriented programming is inheritance, especially when it is used for code reuse and refinement. Unfortunately, in the presence of concurrency coordination code, reuse problems are likely to occur. In general, there is a high interdependence between attributes (instance variables) of a class and coordination constraints of different routines. Concurrency coordination and functionality are intimately interwoven. Because of this interdependence, routines often cannot be redefined in subclasses without affecting other routines, due to modified coordination constraints. The other routines must be redefined in the descendant and the ancestor, which degrades maintainability and prevents reuse. Even if the coordination code is isolated from the functionality code, it is often necessary to redefine it completely for all inherited routines instead of allowing local extensions of its parts. These and other difficulties of combining inheritance with concurrency are referred to as *inheritance anomaly* **[60][51]**.

4.2. SCOOP

The **SCOOP** model (Simple Concurrent Object-Oriented Programming) **[61] [64]** offers a comprehensive approach to building high-quality concurrent and distributed systems. 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*, which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to

cover concurrency and distribution. The extension indeed consists of just one keyword **separate**; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. For application programmers, writing concurrent applications with SCOOP is extremely simple, and does not require the usual baggage of concurrent and multithreaded programming (semaphores, rendezvous, conditional critical regions etc.). The model takes advantage of the inherent concurrency implicit in object-oriented programming to provide programmers with a simple extension enabling them to produce concurrent applications with little more effort than sequential ones.

Processors

SCOOP uses the basic scheme of the object-oriented computation: the feature call, e.g. x.f(a), which should be understood in the following way: the client object calls feature f on the supplier object attached to x, with the argument a. In a sequential setting, such calls are synchronous, i.e. the client is blocked until the supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one processor to handle execution of features. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects[1]. If different processors are used for handling the client and the supplier objects, the feature call becomes asynchronous: the computation on the client object can move ahead without waiting for the call to terminate. Processors are the principal concept that SCOOP adds to the sequential object-oriented framework. Contrary to a sequential system, a concurrent system may have any number of processors, independently of the number of available CPUs.

Separate calls

A declaration of an entity or function, which normally appears as x: SOME_CLASS may now also be of the form x: **separate** SOME_CLASS. Keyword separate indicates that entity x is handled by a different processor, so that calls on x should be asynchronous and can proceed in parallel with the rest of computation. With such a declaration, x becomes a *separate entity*. If the target of a call is a separate expression, i.e. a separate entity or an expression involving at least one separate entity, such call is referred to as *separate call*.

Synchronisation

No special mechanism is required for a client to resynchronise with its supplier after a separate call x.f(a) has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in value := x.some_query. This automatic mechanism is known as *wait by necessity* **[21]**. SCOOP ensures that the separate calls made by the client to each supplier are executed in the correct order (FIFO).

[1] It can be implemented by a piece of hardware (CPU), a process, a single thread in a multithreaded environment, or an application domain in Microsoft .NET, etc.

Contracts

SCOOP relies largely on the principles of Design by Contract. In particular, it introduces a new semantics for preconditions. Invariants play an important role in ensuring the consistency of concurrent applications.

Preconditions

The semantics of *preconditions* is different in sequential and concurrent setting. In sequential programs, preconditions are assertions that have to be fulfilled by the client object before calling the routine of the supplier object. If one or more preconditions are not met, the contract is broken and an exception is raised in the client object. In a concurrent context, the preconditions which do not involve any separate entities (e.g. value_specified, see Example 1) keep their original semantics: they are *correctness conditions*. The preconditions involving calls on separate objects (e.g. buffer_not_full) change their semantics: they become *wait conditions*. If such precondition is not satisfied, it does not result in an exception raised in the client; it only causes the client to wait until the precondition is satisfied.

Example 1. Use of preconditions in SCOOP

```
store (buffer: separate BUFFER; value: INTEGER) is
    -- Store value into buffer.
  require
    value_specified: value /= Void
    buffer_specified: buffer /= Void
    buffer_not_full: not buffer.is_full
  do
    buffer.put (value)
  ensure
    buffer_not_empty: not buffer.is_empty
  end
```

Invariants

Another interesting problem is the relation between concurrent execution and *class invariants*. The class invariant is the most important part of a contract, since it ensures the consistency of the class instances (objects). An object-oriented software system can be consistent if and only if every object in the system is consistent with its specification (its base class).

At first, the connection between class invariants and concurrency does not seem so obvious. If one tries to reason about concurrent programs written in SCOOP, the use of invariants is indeed exactly the same as in a sequential context **[65]**. This is because only one routine can be called on the supplier object at any given time, so satisfying or violating the invariant of the supplier's base class depends only on the outcome of this routine call. On the other hand, if we allowed concurrent execution of several routines of the same supplier object, the invariant may be violated, even if the sequential execution of the same routines would not violate it (see 3.2 for more details).

Locking policy

The control of access to shared resources is the main problem in concurrent computation. In non-object-oriented settings the concept of *critical section* is used: it is simply a code

fragment in which a shared resource is accessed. At most one process[2] can be executing the critical section at any given time. Efficient solutions to conflict problems must be characterised by a synchronisation among processes, so that they have to wait for executing a critical section if another process is accessing the shared resource that critical section is for. This kind of synchronisation is referred to as *mutual exclusion* (from running the critical section at the same time). The following design guidelines should be followed:

- No two processes may be simultaneously in their critical sections related to the same shared resource.
- No process running outside its critical sections may block any other process from running its own ones.
- No process should have to wait (potentially) forever to enter its critical section.
- No assumptions can be made on the number of available CPUs.

The situation changes significantly when we deal with object-oriented computations. Explicit critical sections are not required any more, since they can be encapsulated in class routines, as it is done in SCOOP. The most important question is: how do we ensure that concurrent calls to the routines of the same object do not cause deadlock, and do not violate the integrity of the object (i.e. the invariant of its base class)? An appropriate locking policy should be applied in order to ensure these two conditions.

SCOOP does not use the concept of the critical section, instead it relies on the mechanism of argument passing. For a separate call to be valid, the target of the call must be a formal argument of the enclosing routine. Such "embedding" of separate calls in routines allows exclusive locking of objects. For instance, in order to obtain exclusive access to a separate object `buf`, it suffices to use it as an argument of the corresponding call, as in `store (buf, 10)`.

The locking policy of the SCOOP model, as defined in **[61]**, is very restrictive: at most one client may access any supplier object at any given time. This certainly simplifies the implementation and makes it easier to reason about concurrent programs. Since only one client object can hold a lock on the given supplier object at any time, interference between several client objects is impossible. Therefore, one can easily decide which object is responsible for possible breaches in the contract (e.g. breaking the routine's precondition, breaking the invariant of the class corresponding to the supplier object). The biggest drawback of this approach is, obviously, the lost of performance: very often, several clients try to access a shared data structure and only one of them is allowed to make a call at a time; the others have to wait, even if they do not try to modify the data structure.

4.3. Deadlocks

Deadlocks in SCOOP

The locking mechanism of SCOOP is based on the argument passing. Consider Example 2. We deal with the producer-consumer synchronisation. Assume that several producer

[2] Here *process* = *thread of execution*. It may be called *process*, *thread*, *processor*, etc.

objects are producing integer values and storing them into the shared buffer buf; several consumer objects are consuming elements from that buffer. From the point of view of both the producers and the consumers, buf is a separate object (that is why it is declared as separate in the source code of both classes). In order to perform a call to buf, a client object (be it producer or consumer) must obtain an exclusive lock on buf. Since SCOOP relies on the argument passing mechanism for this purpose, the target of a separate call must appear as an argument of the enclosing routine; that is why all the calls to buf are embedded into routines store and consume_one. Direct calls to buf.put, buf.item, and buf.remove are forbidden.

Example 2. Producer-consumer synchronisation[3]

```
class PRODUCER
feature
 store (buffer: separate BUFFER [INTEGER];
        value: INTEGER) is
     -- Store value into buffer.
  require
    buffer_specified: buffer /= Void
    buffer_not_full: not buffer.is_full
    value_specified: value /= Void
  do
    buffer.put (value)
  end
 random_gen: RANDOM_GENERATOR
 buf: separate BUFFER [INTEGER]
 produce_n (n: INTEGER) is
      -- Produce n integer values and store
      -- them into a buffer.
   local
     value: INTEGER
     i: INTEGER
   do
     from i := 1
     until i > n
     loop
       value := random_gen.next
       store (buf, value)
         -- buf.put (value) is forbidden
         -- here
       i := i + 1
     end
   end
end -- class PRODUCER

class CONSUMER
feature
 consume_one (buffer: separate
```

[3] To simplify the example, the postconditions have been omitted.

```
                BUFFER [INTEGER])
    is
      -- Consume one element from buffer.
    require
      buffer_specified: buffer /= Void
      buffer_not_empty: not buffer.is_empty
    do
      value := buffer.item
      buffer.remove
    end

  buf: separate BUFFER [INTEGER]

  consume_n (n: INTEGER) is
       -- Consume n elements from a buffer.
    local
      i: INTEGER
    do
      from i := 1
      until i > n
      loop
        consume_one (buf)
          -- buf.item and buf.remove are
          -- forbidden here
        i := i + 1
      end
    end

end -- class CONSUMER
```

Let us have a closer look at the locking mechanism. When a consumer object is making a call to consume_one inside routine consume_n, it passes buf as argument to that call. According to the SCOOP access control policy, when one or more arguments of a routine are separate objects, the client must obtain exclusive locks on all these objects before executing the routine. Therefore, the consumer object in our example must obtain an exclusive lock on buf before executing consume_one. If another object is currently holding the lock, the client has to wait until the lock has been released, and then try to acquire it. When the client has finally acquired the lock, the preconditions are checked. If all the preconditions hold, the routine is executed, and the lock is released after the routine has terminated. Should one or more preconditions involving separate objects (i.e. wait conditions) not hold, the client releases all the locks and restarts the whole process from the beginning: first acquiring the locks, then checking the preconditions[4]. This allows other clients to access the supplier object, hopefully changing its state, so that the wait conditions required by our client are eventually met.

Note that acquiring and releasing locks is done atomically: either all of them are granted (respectively: released) or none. This prevents, in most cases, one of the necessary conditions for deadlocks: *wait-and-hold*. Unfortunately, it is still possible to cause a deadlock if a routine executed by the supplier includes calls to other features which request locks on separate objects that have not been passed as arguments to the supplier

[4] We only consider wait conditions here. Preconditions that do not involve any separate entities are correctness conditions, so their violation is handled in the same way as in a sequential setting, i.e. by raising an exception in the client code.

routine. This corresponds to a hold-and-wait scenario: requesting a new resource while holding already acquired ones. Since the two other necessary conditions for deadlocks, i.e. *mutual exclusion* and *no preemption*, are satisfied by the locking policy, SCOOP-based programs may deadlock.

Deadlock prevention
The idea is to disallow one of the four necessary conditions for deadlock. As mentioned before, *mutual exclusion* and *no pre-emption* are satisfied by the locking mechanism of SCOOP. So, obvious candidates for elimination are *hold-and-wait* and *circular waiting*.

In fact, it is quite easy to design very restrictive (pessimistic) rules that would prevent *hold-and-wait*:

- Require the client object to request and be allocated all its resources before it begins calls the routine, or
- Allow the client to request resources only when it has none.

However, this would unnecessarily restrict the model, making it much less flexible. It may also lead to low resource utilisation. Another problem is *starvation*: the client may wait indefinitely for all its required resources.

Circular waiting is much more difficult to prevent, especially with statically-checkable rules:

- Impose a total ordering on all resource types.
- Require each client to request resources only in a strictly increasing order.
- Require that resources of the same type must be requested together.

Such rules are unacceptable for a general-purpose object-oriented programming model. Nevertheless, some run-time mechanism may be devised to detect an occurrence of circular waiting (see “Deadlock Detection”).

Deadlock avoidance
The idea is to grant the requested resource if and only if this allocation does not have the potential to lead to a deadlock. Obviously, this problem cannot be solved statically at compile-time. Instead, a run-time mechanism must be devised to ensure that granting a new lock does not cause deadlock.

How could such a mechanism for SCOOP look like? *Shared locks* **[67]** would eliminate, in some cases, the *mutual exclusion* condition, thus preventing deadlocks even in a *circular waiting* scenario. Similarly, the *no pre-emption* condition may be eliminated by the use of the duel mechanism of SCOOP.

Deadlock detection
The last approach is to always grant the requested resource when possible, and then periodically check for deadlocks. If a deadlock exists, recover from it.

There are two main problems with deadlock detection. First of all, how do we detect a deadlock? The mechanism should keep track of all current resources and resource requests, and use some algorithm to decide whether the system is in a deadlock state. Secondly, what recovery mechanism may be used? A generic recovery mechanism is difficult to devise. Should we pick one or more objects and raise an exception in the

routines they are currently executing, thus forcing them to release all the locks and retry? This would certainly be a dangerous solution: what if class invariants are broken? Here, once again, the new locking policy may be helpful. If the pre-empted routines are side-effect-free, there is no danger of breaking class invariants and the routines can simply get rescheduled.

5. OTHER TASKS

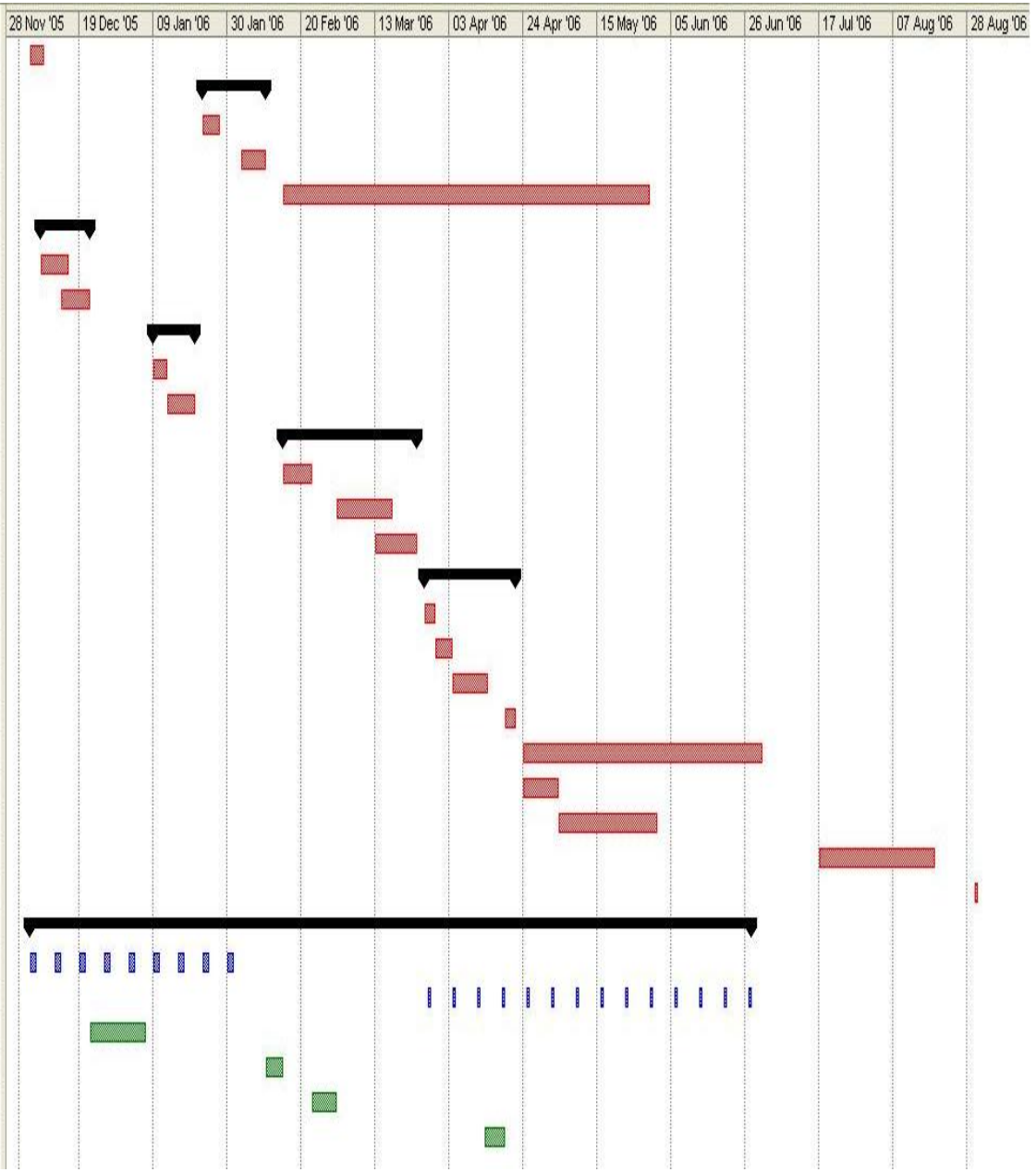
The proposed schedule (see section 6) takes into account the responsibilities that I have as a research assistant at the Chair of Software Engineering:

- **Teaching assignments**
  - Introduction to Programming (WS05/06),
  - Concurrent Object-Oriented Programming (SS06).
- **Organisation of scientific events**
  - LASER Summer School on Software Engineering 2006

- **Administrative tasks** within the Chair of Software Engineering.

6. TENTATIVE SCHEDULE

	Task Name	Duration	Start	Finish
1	Type rules 1st cut	4 days	Mon 05.12.05	Thu 08.12.05
2	**1. + 2. + 3. + 4.**	**14 days**	**Mon 23.01.06**	**Thu 09.02.06**
3	1. Concurrency challenges + 2 Summary and main re	5 days	Mon 23.01.06	Fri 27.01.06
4	3 Previous work + 4. The original SCOOP model	5 days	Fri 03.02.06	Thu 09.02.06
5	5. Using SCOOP in practice	74 days?	Wed 15.02.06	Mon 29.05.06
6	**6. Generalised semantics of contracts**	**10 days**	**Thu 08.12.05**	**Wed 21.12.05**
7	6.1 Rationale + 6.2 Contracts	6 days	Thu 08.12.05	Thu 15.12.05
8	6.3 Towards proof rule + 6.4 Deadlocks and contrac	6 days	Wed 14.12.05	Wed 21.12.05
9	**10. Refined access control policy**	**10 days**	**Mon 09.01.06**	**Fri 20.01.06**
10	10.1 Attached types	4 days	Mon 09.01.06	Thu 12.01.06
11	10.2 Shared locking + 10.3 Lock passing	6 days	Fri 13.01.06	Fri 20.01.06
12	**7. Type system for SCOOP + 8.**	**28 days**	**Wed 15.02.06**	**Fri 24.03.06**
13	7.1 Rationale + 7.2 Reasoning about object locality	6 days	Wed 15.02.06	Wed 22.02.06
14	7.3 Tagged type system for SCOOP	12 days	Thu 02.03.06	Fri 17.03.06
15	8. Operational semantics + proofs for chapter 7	10 days	Mon 13.03.06	Fri 24.03.06
16	**9. Advanced O-O mechanisms in SCOOP**	**20 days**	**Mon 27.03.06**	**Fri 21.04.06**
17	9.1 Expanded types	3 days	Mon 27.03.06	Wed 29.03.06
18	9.2 Agents	3 days	Thu 30.03.06	Mon 03.04.06
19	9.3 Inheritance + 9.4 Polymorphism and dynamic bind	8 days	Tue 04.04.06	Thu 13.04.06
20	9.5 Genericity	3 days	Wed 19.04.06	Fri 21.04.06
21	11. Implementation issues and solutions	50 days	Mon 24.04.06	Fri 30.06.06
22	12. Conclusions	8 days	Mon 24.04.06	Wed 03.05.06
23	Polishing the dissertation	20 days	Thu 04.05.06	Wed 31.05.06
24	Integration of feedback	25 days	Mon 17.07.06	Fri 18.08.06
25	Thesis defence	1 day?	Wed 30.08.06	Wed 30.08.06
26	**Teaching**	**147 days**	**Mon 05.12.05**	**Tue 27.06.06**
27	**Introduction to programming WS2005/06**	**42 days**	**Mon 05.12.05**	**Tue 31.01.06**
37	**COOP SS2006**	**66 days**	**Tue 28.03.06**	**Tue 27.06.06**
52	Christmas holiday	12 days	Thu 22.12.05	Fri 06.01.06
53	Holiday	3 days	Fri 10.02.06	Tue 14.02.06
54	Holiday	5 days	Thu 23.02.06	Wed 01.03.06
55	Easter holiday	4 days	Thu 13.04.06	Tue 18.04.06



7. TABLE OF CONTENTS FOR THE THESIS REPORT

9. Advanced object-oriented mechanisms in SCOOP
9.1 Expanded types
    9.1.1 Type rules
9.2 Agents
    9.2.1 Type rules
    9.2.2 Agents with open target
9.3 Inheritance
    9.3.1 Domain inheritance
    9.3.2 Multiple inheritance
9.4 Polymorphism and dynamic binding
    9.4.1 Lock weakening rule
    9.4.2 Static vs. dynamic binding of preconditions
9.5 Genericity

10. Refined access control policy for SCOOP
10.1 Attached types and their application to locking
    10.1.1 Refined semantics for argument passing (I)
    10.1.2 Object test
    10.1.3 Covariant redefinition of formal arguments
    10.1.4 Preserving safety
10.2 Shared locking
    10.2.1 Legal interleavings
    10.2.2 Pure queries
    10.2.4 Preserving safety
10.3 Lock passing
    10.3.1 Rationale
    10.3.2 Refined semantics for argument passing (II)
    10.3.3 Preserving safety

11. Implementation issues and solutions
11.1 Library implementation – SCOOPLI
11.2 scoop2scoopli pre-processor and type checker
11.3 Assessment

12. Conclusions
12.1 Scientific contributions
12.2 Limitations of the proposed approach
12.3 Future research directions

8. BIBLIOGRAPHY

[1] G. Agha. Concurrent Object-Oriented Programming, in Communications of the ACM, 1990, 33(9), 125-141

[2] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, Massachusetts, 1986.

[3] G. Agha and W. Kim. Actors: A unifying model for parallel and distributed computing, Journal of Systems Architecture, 45(15), September 1999.

[4] P. America. Pool-t: A parallel object-oriented language, in A. Yonezawa, M. Tokoro (eds), Object-Oriented Concurrent Programming, pages 199–220. MIT Press, 1987.

[5] P. America, M. Beemster. A portable implementation of the language POOL, in Proceedings of TOOLS EUROPE 1989, ed. Jean Bézivin, SOL, Paris, 1989, 347-353

[6] V. Arslan. Applying SCOOP to real-time systems, research plan (draft), available at http://se.inf.ethz.ch/people/arslan

[7] D. F. Bacon, R. E. Strom, A. Tarafdar. Guava: A dialect of Java without data races, in Proceedings of OOPSLA'00, Minneapolis, October 2000

[8] R. Balter et al. Architecture and Implementation of Guide, an Object-Oriented Distributed System, in Computing Systems, 1991, vol. 4

[9] C. Boyapati, R. Lee, M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks, in Proceedings of OOPSLA'02, Seattle, November 2002

[10] C. Boyapati, R. Lee, M. Rinard. Safe runtime downcasts with ownership types, Technical Report TR-853, MIT Laboratory for Computer Science, July 2002

[11] C. Boyapati, B. Liskov, L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases, Technical Report TR-858, MIT Laboratory for Computer Science, July 2002

[12] C. Boyapati, M. Rinard. A parametrized type system for race-free Java programs. In Procediings of OOPSLA'01, Tampa Bay, October 2001

[13] P. Brinch-Hansen. Java's insecure parallelism, ACM SIGPLAN Notices, 34(4):38–45, 1999.

[14] P. Brinch-Hansen. Monitors and Concurrent Pascal: a personal history, in The Second ACM SIGPLAN Conference on History of Programming Languages, pages 1–35, 1993.

[15] P. Brinch-Hansen. Structured multiprogramming, Communications of the ACM, 15(7):574–578, 1972.

[16] P. Brinch-Hansen. The programming language Concurrent Pascal, in IEEE Transactions on Software Engineering SE-1 (2), June 1975

[17] D. R. Butenhof. Programming with POSIX Threads, Addison-Wesley, 1997.

[18] L. Cardelli. Structural subtyping and the notion of power type. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 70–79. ACM Press, 1988.

[19] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism, ACM Computing Surveys (CSUR), 17(4):471– 523, 1985.

[20] D. Caromel. Service, Asynchrony, and Wait-by-Necessity. Journal of Object-Oriented Programming, 2(4):12–18, 1989.

[21] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming, in Communications of the ACM, Volume 36, Number 9, September 1993, 90-102

[22] S. Chaki, S. K. Ramajani, J. Rehof. Types as models: Model checking message-passing programs, in Proceedings of POPL'02, January 2002

[23] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan. Efficient and precise data race detection for multithreaded object-oriented programs, in Proceedings of PLDI'02, June 2002

[24] D. G. Clarke, S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect, in Proceedings of OOPSLA'02, Seattle, November 2002

[25] D. G. Clarke, J. Noble, J. M. Potter. Simple ownership types for object containment, in Proceedings of ECOOP'01, Budapest, June 2001

[26] D. G. Clarke, J. M. Potter, J. Noble. Ownership types for flexible alias protection, in Proceedings of OOPSLA'98, Vancouver, October 1998

[27] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks, ACM Computing Surveys (CSUR), 3(2):67–78, 1971.

[28] W. R. Cook. A proposal for making Ei_el type-safe. The Computer Journal, 32(4):305–311, 1989. Originally in Proc. European Conf. on Object-Oriented Programming (ECOOP).

[29] W. R. Cook, W. Hill, P. S. Canning. Inheritance is not subtyping, in Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 125–135. ACM Press, 1990.

[30] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers", Communications of the ACM, 14(10):667–668, 1971.

[31] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. Extended static checking, Research Report 159, Compaq Systems Research Center, December 1998

[32] E. W. Dijkstra. Cooperating Sequential Processes, Programming Languages, Academic Press, New York, 1968.

[33] E. W. Dijkstra. Hierarchical ordering of sequential processes, in Operating Systems Techniques, pages 72–93, Academic Press, 1972. (EWD310)

[34] E. W. Dijkstra. Two starvation-free solutions of a general exclusion problem, circulated privately known as EWD625, 1977.

[35] A. Dinning, E. Schonberg. Detecting access anomalies in programs with critical sections, in Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD), May 1991

[36] ECMA. Eiffel Analysis, Design, and Programming Language, ECMA Standard 367, June 2005

[37] C. Flanagan, S. N. Freund. Type-based race detection for Java, in Proceedings of PLDI'00, Vancouver, June 2000

[38] R. W. Floyd. Assigning meanings to programs, in J. T. Schwartz (ed.), Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19, pages 19–32, Providence, 1967.

[39] H. Garcia-Molina, J. D. Ullman, J. D. Widom. Database Systems: The Complete Book, Prentice Hall, 2002

[40] A. N. Habermann. Prevention of system deadlocks, Communications of the ACM, 12(7):373–377, 1969.

[41] M. P. Herlihy, J. M. Wing. Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463-492, July 1990.

[42] C. A. R. Hoare. An axiomatic basis for computer programming, Communications of the ACM, 12(10):576-580, 1969.

[43] C. A. R. Hoare. Hints on programming language design, Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Stanford University, 1973.

[44] C. A. R. Hoare. Monitors: an operating system structuring concept, Communications of the ACM, 17(10):549–557, 1974.

[45] C. A. R. Hoare. Communicating Sequential Processes, Prentice Hall International (UK) Ltd, 1985.

[46] A. Igarashi, N. Kobayashi. A generic type system for the pi-calculus, in Proceedings of POPL'01, London, January 2001.

[47] G. Jalloul. Concurrent object-oriented systems: a disciplined approach, PhD thesis, University of Technology, Sydney, Australia, June 1994

[48] J.-M. Jézéquel. Object-Oriented Software Engineering with Eiffel, chapter 9, Addison-Wesley, Reading (Mass.), 1996.

[49] C.B. Jones. Development Methods for Computer Programs including a Notion of Interference, PhD Thesis, Oxford University, June 1981.

[50] M. Joseph (ed.), Real-time Systems: Specification, Verification and Analysis, Prentice Hall International, 1996.

[51] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages, in Proceedings of the Third European Conference on Object-Oriented Programming, July 1989.

[52] L. Lamport. Specifying concurrent program modules, ACM Transactions on Programming Languages and Systems (TOPLAS), 5(2):190–222, 1983.

[53] L. Lamport. Time, clocks, and the ordering of events in a distributed system, Communications of the ACM, 21(7):558–565, 1978.

[54] H. C. Lauer and R. M. Needham. On the duality of operating system structures, in Proceedings of the Second International Symposium on Operating Systems, October 1978. reprinted in Operating Systems Review, 13(2):3-19, April 1979.

[55] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, G. J. Popek. Report on the programming language Euclid, in Sigplan Notices, 12(2), February 1977

[56] B. Liskov, S. Zilles. Programming with abstract data types, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pages 50–59, 1974.

[57] A. Lister. The problem of nested monitor calls, in Operating Systems Review 11(3), July 1977.

[58] K.-P. Löhr. Concurrency annotations, ACM SIGPLAN Notices, 27(10):327-340, 1992.

[59] K.-P. Löhr. Concurrency annotations for reusable software, Communications of the ACM, 36(9):81-89, 1993.

[60] S. Matsuoka, A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Agha G., Wenger P., Yonezawa A. (eds.),

Research Directions in Concurrent Object-Oriented Programming, MIT Press, Cambridge (Mass.), 1993, 107-150

[61] B. Meyer. Eiffel: The Language, Prentice Hall, Englewood Cliffs, N.J., March 1992.

[62] B. Meyer. Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997

[63] B. Meyer (ed.), Special issue on Concurrent Object-Oriented Programming, in Communications of the ACM, 1993, 36(9)

[64] B. Meyer. Systematic Concurrent Object-Oriented Programming, in Communications of the ACM, Special issue on Concurrent Object-Oriented Programming, 1993, 36(9), 56-80

[65] J. Misra. A Discipline of Multiprogramming: Programming Theory for Distributed Applications, Springer-Verlag, 2001

[66] J. Misra, K. M. Chandy. Proofs of Networks of Processes, IEEE Transactions on Software Engineering, 7(4): 417-426, 1981

[67] P. Nienaltowski. Extending the access control policy for SCOOP, draft, available online at http://se.inf.ethz.ch/people/nienaltowski/papers/extended_access_draft.pdf

[68] P. Nienaltowski, V. Arslan. SCOOPLI: a library for concurrent object-oriented programming on .NET, in Proceedings of the 1st International Workshop on C# and .NET Technologies, Pilsen, February 2003

[69] P. Nienaltowski, V. Arslan, B. Meyer. Concurrent object-oriented programming on .NET, IEE Proceedings-Software, 150(5): 308-314, 2003

[70] O. Nierstrasz. A Tour of Hybrid: A Language for programming with Active Objects, in Advances on Object-Oriented Software Engineering, Meyer B., Mandrioli D. (eds.), Prentice-Hall, 1992, 167-182

[71] M. Nuttal. A brief survey of systems providing process or object migration facilities, Operating Systems Review, 1994, 28(4), 64-80

[72] S. Owicki, D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach, Communications of the ACM, 19(5):279-285, 1976

[73] M. Papathomas. Language design rationale and semantic framework for concurrent object-oriented programming, PhD Thesis, University of Geneva, Switzerland, 1992.

[74] D. L. Parnas. A technique for software module specification with examples, Communications of the ACM, 15(5):330–336, 1972.

[75] D. L. Parnas. On the criteria to be used in decomposing systems into modules, Communications of the ACM, 15(12):1053–1058, 1972.

[76] M. Philippsen. A survey of concurrent object-oriented languages, Concurrency: Practice and Experience, 2000, 12, 917-980

[77] C. von Praun, T. Gross. Object-race detection, in Proceedings of OOPSLA'01, Tampa Bay, October 2001.

[78] S. Ren, G. Agha. Rtsynchronizer: language support for real-time specifications in distributed systems, in Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers & Tools for Real-time Systems, 1995.

[79] M. Ruschitzka, R. S. Fabry. A unifying approach to scheduling, Communications of the ACM, 20(7):469–477, 1977.

[80] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs, in Proceedings of the Symposium on Operating Systems Principles (SOSP'97), October 1997

[81] S. Schneider. Concurrent and Real-time Systems, John Wiley & Sons, 2000

[82] M. Shapiro, P. Gautron, L. Mosseri. Persistence and Migration for C++ Objects, in ECOOP 1989, ed. Cook, S., Cambridge University Press, Cambridge (England), 191-204

[83] N. Sterling. Warlock: A static data race analysis tool, in USENIX Winter Technical Conference, January 1993

[84] N. Storey. Safety-critical computer systems, Addison Wesley Longman, 1996

[85] L. H. Turcotte. A survey of software environments for exploiting network computing resources, Technical Report, Mississippi State University, 1993

[86] S. Vaucouleur, P. Eugster. Atomic features, in Proceedings of SCOOL workshop, OOPSLA'05, San Diego, USA, 2005

[87] P. Wegner, A. Yonezawa (eds). Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[88] B. Wyatt, K. Kavi, S. Hufnagel. Parallelism in object-oriented languages: a survey, IEEE Computer, 1992, 11(6), 56-66

[89] Y. Yokote, F. Teraoka, M. Yamada, H. Tezuka, M. Tokoro. The Design and Implementation of the MUSE Object-Oriented Distributed Operating System, in TOOLS 1, Bézivin J. (ed.), SOL, Paris, 1989, 363-370

[90] A. Yonezawa, M. Tokoro (eds.), Object-Oriented Concurrent Programming, MIT Press, Cambridge (Massachussetts), 1987

[91] SCOOPLI 3.2, available at http://se.inf.ethz.ch/research/scoop

[92] scoop2scoopli 2.2, available at http://se.inf.ethz.ch/research/scoop