Efficient Data Race and Deadlock Prevention in Concurrent Object-Oriented Programs

Piotr Nienaltowski
ETH Zurich
8092 Zurich, Switzerland
+41 16 32 44 68
Piotr.Nienaltowski@inf.ethz.ch

ABSTRACT
The main goal of this PhD thesis is to propose and implement a methodology for the construction of programs based on the SCOOP model, and for modular reasoning about their correctness and liveness properties. In particular, the set of correctness rules that guarantee the absence of data races will be refined and formalized; an augmented type system will be proposed to enforce these rules at compile time. Furthermore, an efficient methodology for deadlock prevention, avoidance, detection, and resolution will be developed. A working implementation of SCOOP will be provided. It will take into consideration the proposed mechanisms and serve as a basis for further refinements of the model.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming.* D.1.5 [**Programming Techniques**]: Object-Oriented Programming. D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs*, *programming by contract.*

General Terms
Languages, Verification.

Keywords
Object-oriented concurrency, SCOOP model, deadlocks, data races, ownership types, Eiffel.

1. STATEMENT OF THE PROBLEM
The main goal of this PhD thesis is to propose and implement a methodology for the construction and verification of programs based on the SCOOP model [1]. SCOOP (Simple Concurrent Object-Oriented Programming) was introduced by Bertrand Meyer as an extension of the Eiffel language. The lack of a formal semantics for SCOOP makes it difficult to assess the model with respect to other existing approaches. This thesis should fill in the gap by carrying out an in-depth analysis of the model and proposing adequate solutions to the problems encountered. A methodology for modular proofs of safety and liveness properties of concurrent programs will be proposed. In particular, we will focus on data race and deadlock prevention.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2. SCOOP
SCOOP uses the basic scheme of object-oriented computation: feature call, e.g. *x.f* (*a*). In a sequential setting, such calls are synchronous. 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. If different processors are used for handling the client and the supplier objects, the feature call becomes asynchronous. A declaration of an entity or function may be of the form *x*: **separate** *SOME_CLASS.* The keyword ***separate*** indicates that entity *x* is handled by a different processor, so that calls on *x* should be asynchronous. To provide exclusive locking of objects, SCOOP relies on argument passing. For instance, to obtain exclusive access to a separate object *buf*, it suffices to pass it as an argument of the corresponding call, as in *store* (*buf*, *10*). To provide conditional synchronization, SCOOP introduces a new semantics for preconditions. Preconditions involving calls on separate objects change their semantics: they become *wait conditions*. Such preconditions cause the client to wait until they are satisfied.

3. KEY QUESTIONS AND RESULTS
3.1 Data races
Concurrent programs should be free from data races. Four *separateness consistency rules* and the *separate call rule* [1] of SCOOP seem to ensure this property. Unfortunately, these rules are not strong enough to ensure the absence of data races when Eiffel's *agent*[1] mechanism, introduced after the original SCOOP design, is used. Also, expanded types are not well integrated in SCOOP. We claim, for example, that the separateness consistency rule for expanded types is too restrictive – it rules out useful programs. It is necessary to refine the SCOOP rules to integrate both agents and expanded types. It is impossible to check the rules statically using the standard Eiffel type system because separateness is a property of objects, not classes; the conformance of separate and non-separate entities cannot be expressed statically in terms of subclassing. Therefore, we formalize the refined rules by introducing an augmented type system for Eiffel. A type checker can check the type conformance (thus data race freedom) of SCOOP programs at compile time. The proposed type system (inspired by the ownership type system for JavaCard [2]) augments Eiffel's types with *context tags*.

[1] *Agents* are used in Eiffel to encapsulate routine calls. One can think of them as a more sophisticated form of .NET delegates.

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

$$TaggedType = \{loc, sep\} \times TypeId$$

where *loc* and *sep* are context tags denoting *local* (non-separate) and *separate* types, respectively. 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 subtype relation on *TypeId* :

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

This results in a very simple but sufficiently expressive type system for SCOOP. We illustrate it with the *feature call rule* that ensures the mutual exclusion policy:

$$\frac{\Gamma \vdash e1 :: (\alpha, T),\quad \Gamma \vdash e2 :: (\beta, S),\quad (\alpha, T) * (\beta, S) \prec (\alpha_p, T_p) \quad \alpha = sep \Rightarrow e1 \in FormArg}{\Gamma \vdash e1.f\,(e2) :: (\alpha, T) * (\alpha_r, T_r)}$$

where Γ is the declaration environment, *FormArg* is the set of formal arguments of the routine where the expression is evaluated, (α_p,T_p) is the type of the formal argument of feature *f* and (α_r, T_r) is the type of its result (for simplicity, we assume here that *f* has only one argument). We define the type combinator $* : TaggedType \times TaggedType \to TaggedType$ as:

$$(\alpha,T) * (\beta,S) = \begin{cases} (\beta,S) & \text{if} \quad \alpha = loc \\ (sep,S) & \text{otherwise} \end{cases}$$

In comparison to the state-of-the-art approaches to data race prevention (e.g. [3]), this solution is much simpler and less restrictive — it allows the programmer to use the full potential of the underlying programming language (Eiffel). It does not impose complicated code annotations — one keyword ***separate*** is sufficient. Also, data race freedom is proved compositionally, i.e. if feature *f* has been proved to be data-race-free, the proofs of other features that use *f* can rely of *f*'s interface without using its implementation details. The approach fully supports inheritance and other object-oriented techniques.

3.2 Deadlocks

SCOOP is deadlock-prone. To eliminate deadlocks, we take the following approach. In a first step, we assume that there is a perfect run-time mechanism for deadlock detection and resolution. Secondly, we classify deadlocks according to their nature and devise a strategy for preventing each kind of deadlock. The strategy should be based on statically checkable rules for features; proofs should be modular and automated. By stepwise refinement of these strategies, we weaken the initial assumption on the perfect runtime mechanism for deadlock detection and resolution: since certain classes of deadlocks are excluded, the runtime mechanism can be simplified. We implement the mechanism. Finally, we devise an extended axiomatic system à la Owicki and Gries [4] that will allow for manual proofs of deadlock freedom in case fully automated proofs are impossible.

We can think of at least two interesting properties of features: *global deadlock freedom* and *local deadlock freedom.* Feature *f* is *globally deadlock free* iff it never introduces any deadlock, independently of any other features that might be executed in parallel with *f*. Feature *f* is *locally deadlock free* iff it does not introduce any deadlock, assuming that no other feature is executed in parallel with *f*.

The proofs should be modular. Therefore, in order to prove a property of feature *f*, we can only rely on the interface and the body of *f*, the interfaces of all the features used in *f*, and the invariant of the class where *f* is declared. In Eiffel, the interface of a feature is represented by its signature and its contract (*pre-* and *postconditions*). For reasoning about deadlocks, we need to extend the interface with some additional information: the set of processors that the feature uses. We introduce the concept of *resource* that abstracts a processor (due to space restriction of the present document, we only give informal definitions). The set of all resources on which feature *f* depends is denoted as Dep_f. It includes all the resources associated with the entities that the feature accesses or modifies. We can express some interesting properties ($\overline{a}$ denotes the resource associated with the entity *a*):

$$\forall a \in FormArg_f .(\overline{a} \in Dep_f \wedge \overline{a} \neq \overline{Current} \Rightarrow f \text{ locks } a)$$

$$Dep_f \subseteq \overline{FormArg_f} \cup \{\overline{Current}\} \Rightarrow f \text{ is globally deadlock-free}$$

The first rule refines and formalizes the policy for locking separate formal arguments. The second result shows how to prove deadlock freedom; it could be used in our proof system.

Global deadlock freedom is certainly more interesting than local deadlock freedom – ideally, we would like to prove that our programs are globally deadlock free. Unfortunately, in many cases it is impossible to prove this property compositionally without restricting the amount of potential concurrency. Local deadlock freedom is a weaker property but it is strong enough to rule out a large number of potential deadlocks. Also, compositional proofs of local deadlock freedom are much simpler.

An important contribution of our approach is to integrate static techniques for deadlock prevention with a run-time mechanism for deadlock detection and resolution. Programmers are given the liberty to choose to what extent they want to rely on static checking. It allows them to find the right balance between the guarantee of complete deadlock-freedom and the potential amount of concurrency. This is an important step forward in comparison to the current techniques (see [3]).

4. REFERENCES

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

[2] Müller, P. and Poetzsch-Heffter, A. A Type System for Checking Applet Isolation in Java Card, *Formal Techniques for Java Programs*, 2001.

[3] Boyapati, C. et al.. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks, *OOPSLA'02*, Seattle, November 2002.

[4] Owicki, S. and Gries, D. Verifying Properties of Parallel Programs: An Axiomatic Approach, *Communications of the ACM, 19(5)*, 1972, 279-285.