Transactional Contexts

Harnessing the Power of Context-Oriented Reflection

Sebastián González
Computer Engineering Department
Université catholique de Louvain
s.gonzalez@uclouvain.be

Marcus Denker
Computer Science Department
University of Chile
denker@acm.org

Kim Mens
Computer Engineering Department
Université catholique de Louvain
kim.mens@uclouvain.be

ABSTRACT

The emerging field of context-oriented programming gives a predominant role to the execution context of applications, and advocates the use of dedicated mechanisms to allow the elegant expression of behavioural adaptations to such context. With suitable reflective facilities, language semantics can be adapted to context by reusing the same context-oriented mechanisms that allow base-level adaptability. This kind of meta-level adaptability, in which the computation model itself becomes adaptable to context, gives rise to context-oriented computational reflection. To explore this idea, we set out to implement a simple software transactional memory system that exploits meta-level adaptability by regarding transactions as contexts, and adapting fundamental system behaviour to such transactional contexts. The implementation is succinct and non-intrusive, giving us an indication of the power lying at the crossroads of context-oriented programming and computational reflection.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures, Control structures*

General Terms

Design, Languages, Reliability

Keywords

Context-oriented programming, Computational reflection, Software transactional memory

1. INTRODUCTION

The need for adequate programming techniques that enable application context-awareness has given rise to the emerging field of Context-Oriented Programming (COP) [3]. COP has been approached chiefly from a programming language engineering perspective, by introducing new language abstractions and run-time facilities that help expressing context-dependent adaptations [8, 12, 18]. The combination of COP with computational reflection opens further possibilities for run-time software adaptability [4]. To the extent of our knowledge, there are no documented attempts to adapt the underlying computation model *itself* to context —that is, the meaning of message passing, state manipulation, object creation, inheritance, and so forth.

In this paper we show a first indication of the possibilities offered by context-oriented computational reflection by implementing support for lightweight software-based transactions —actions that are *atomic*, *isolated* and *abortable* [19]— using COP language abstractions and reflection. The language abstractions are part of the Ambient Object System (AmOS) [7], which is implemented on top of Common Lisp. In essence, AmOS is a prototype-based computation model featuring multimethods and subjective dispatch [15].[1] Even though the transaction mechanism we describe can be the basis for a fully-fledged Software Transactional Memory system [13], our goal is rather to make a case for the elegance and usefulness of context-oriented reflection to adapt the underlying computation model.

We use a running example for illustration. To motivate this example, we start from a brief scenario describing typical requirements to be addressed by a transactional system:

> ACME's Ambient Shopping System (AmbiShop) supports automatic payment of goods thanks to Radio Frequency Identification (RFID) tags put on all stock items. AmbiShop uses FIFO accounting for managing the inventory —it regards the first unit that arrived in inventory as the first one sold. Hence, inventories are basically queues from which the dealer draws items and sells them to customers.
>
> Items are drawn from the inventory in order. Each item is accounted for and put in the client's order. If a problem arises in the middle of the checkout operation (for instance, if the client runs out of credit), the whole order is cancelled and the client is notified.

[1] AmOS does not rely on the Common Lisp Object System —the standard object-oriented extension of Common Lisp— in particular because AmOS is not based on a notion of *class*.

COP'09, July 7, 2009, Genoa, Italy.

```
(defproto @inventory (extend @queue))
(defproto @item (clone @object))
(add-slot @item 'description "An inventory item")
(add-slot @item 'entrance-date nil)
(add-slot @item 'exit-date nil)
```

Snippet 1: Definition of inventories and inventory items.

To support proper cancellation of the checkout operation, we use transactions. The transactions we define are atomic "all or nothing" units of work in which arbitrary objects are manipulated. The modifications made to objects are either effective as a whole, or ineffective altogether. If it is decided in the middle of execution of a transaction that the changes should be cancelled, the programmer needs not care about undoing the modifications that have been done to every single object since the transaction began —this might even be impossible, as the programmer does not necessarily hold a reference to all affected objects. Transactions can be aborted intentionally or due to an error.

Besides atomicity, a second important property of lightweight transactions is isolation: the effects of operations made within a transaction should be invisible to operations outside the transaction. Isolation is relevant in concurrent systems. In the scenario, isolation means that a thread should never see the intermediate inventory states of another thread that is currently performing a checkout operation —that is, the states in which only part of the items corresponding to an order have been drawn from the inventory.

In the remainder of the paper we show how the scenario can be implemented straightforwardly thanks to lightweight transactions (Section 2), and how the underlying transactional support can be realised thanks to COP and reflection (Section 3). We discuss the qualities and downsides of our approach and its implementation (Section 4), and finish the paper with the conclusions we have drawn from this experience (Section 6).

2. EXPECTED FUNCTIONALITY

This section shows user code and expected observable behaviour for a system implementing the scenario presented in Section 1. We introduce in passing some of the main concepts behind AmOS to ease comprehension of Section 3.

To implement the inventory, the user creates the `@inventory` and `@item` prototypes shown in Snippet 1. By convention, prototype names are prefixed with the `@` symbol. In this example, an inventory is simply a specialised queue. Inventory items have a description and two timestamps: the date when the item entered the inventory, and the date when it was taken out. If the entrance date is `nil`, the item has not entered the inventory yet; if the exit date is `nil`, it has not left the inventory yet.

In Snippet 2, an inventory and order are created. The inventory contains a few initial items. The state of inventories can be inspected, showing the time at which each item entered the inventory:

```
(defparameter *inventory* (clone @inventory))
(defparameter *order* (clone @inventory))
(enqueue (make-item "potatoes") *inventory*)
(enqueue (make-item "tomatoes") *inventory*)
(enqueue (make-item "oysters") *inventory*)
```

Snippet 2: Sample inventory and order.

```
(atomic
  (loop until (emptyp *inventory*)
    for item = (dequeue *inventory*)
    do (enqueue (process item) *order*)))
```

Snippet 3: Atomic treatment of inventory and order queues.

```
(describe *inventory*)  →
  potatoes (in: 2008/06/02 21:53)
  tomatoes (in: 2008/06/02 21:53)
  oysters  (in: 2008/06/02 21:53)
(describe *order*)  →
  empty inventory
```

During the sell operation, items are dequeued from the inventory, processed in some way, and enqueued in the order. This logic is succinctly encoded as shown in Snippet 3. For the sake of simplicity, the loop is repeated until the inventory is exhausted. There are three main steps being performed in the body of the loop; two of them —`enqueue` and `dequeue`— are standard queue operations. The third is the `process` method which abstracts away the bookkeeping performed for each item that is being transferred from the inventory to the order. The `process` multimethod, defined in Snippet 4, has one argument named `item`, which is specialised on the `@item` prototype.[2] In this example, `process` simply timestamps the item with an exit time and returns the item, without further accounting steps.

The most important part in Snippet 3 is the `atomic` form that surrounds the loop; `atomic` ensures that the enclosed body is performed atomically. Once the `atomic` block finishes, either all changes made are visible at once, or, in case of error, no single change is visible, as if the `atomic` block never executed. If an error arises while processing an item, the programmer needs not determine the point at which the loop stopped working and have a reverse loop undo the changes. Further, the programmer needs no `unprocess` operation to undo the processing performed by `process` for already transferred items. In complex situations, implementing reverse application logic can be difficult and error prone.

If the transaction encoded in the loop of Snippet 3 executes successfully, all items will have been transferred:

```
(describe *inventory*)  →
  empty inventory
(describe *order*)  →
  potatoes (in: 2008/06/02 21:53 out: ... 21:56)
  tomatoes (in: 2008/06/02 21:53 out: ... 21:56)
  oysters  (in: 2008/06/02 21:53 out: ... 21:56)
```

[2]Note that, although similar, the `defmethod` construct we use is unrelated to that of CLOS. AmOS introduces its own method-definition construct.

```
(defmethod process ((item @item))
  (setf (exit-date item) (get-universal-time))
  item)
```

Snippet 4: Simple item processing

```
(defcontext @failing :uses (@standard))

(with-context @failing
  (defmethod process ((item @item))
    (when (> (random 2) 0)
      (error "spurious error"))
    (resend)))
```

Snippet 5: Faulty item processing logic.

The `process` method has timestamped all items with an exit time (we omitted the date in the output to save space).

To show the behaviour of the system in case of failures, we introduce a `@failing` context. In general, contexts are objects representing physical and logical properties of the environment in which the system is running. The `@failing` context in particular represents a situation in which failure testing is being performed on the system. A faulty version of the `process` method is introduced in such context, shown in Snippet 5.[3] The call to `random` yields either 0 or 1 with the same probability. Hence, the new version of `process` will fail roughly half of the time. If it does not fail, it simply invokes the non-faulty (original) behaviour of `process` by means of a `resend` call. The `resend` method is analogous to `call-next-method` in CLOS and `super` in Java.

Starting from the inventory of Snippet 2, but having this time faulty `process` behaviour due to a manual activation of the `@failing` context, we get different output from the code in Snippet 3 when an error occurs:

```
(describe *inventory*)  →
  potatoes (entrance: 2008/06/02 22:44)
  tomatoes (entrance: 2008/06/02 22:44)
  oysters  (entrance: 2008/06/02 22:44)
(describe *order*)  →
  empty inventory
```

Both queues are left untouched, as well as the items they contain —a sign of this in the output is that none of the items has an exit date.

3. TRANSACTIONAL CONTEXTS

Language support for lightweight transactions in modern object-oriented languages is relatively recent [9]. Harris *et al.* [10] show an `atomic` construct that is similar to the one presented in Snippet 3. The mechanisms used to support such atomic transactions are however quite different. We use three advanced features of AmOS: *contexts*, *reflection* and a *become* operation inspired on the Actors model [1].[4] The combination of these features makes the implementation of transactions rather succinct and easy to understand.

[3]Note in passing this further application of COP for software testing, more specifically for fault injection.

[4]Thanks to this operation, an object can be supplanted by another one. Conceptually it remains the same —its identity is preserved— but it assumes new behaviour.

```
1  (defmacro atomic (&body body)
2    `(let ((transaction (clone @transaction)))
3       (handler-case
4           (with-context transaction
5              ,@body)
6         (error (e)
7           (abort transaction)
8           (error e))
9         (:no-error (result)
10          (commit transaction)
11          result))))
```

Snippet 6: Implementation of `atomic`.

```
1  (defcontext @transaction)
2  ($add-slot @transaction 'log (make-hash-table))
3
4  (with-context @transaction
5    (defmethod alter-object (object)
6      (let ((log (log (host-context))))
7        (or (gethash object log)
8            (let ((altobj (clone-object object)))
9              (setf (gethash object log) altobj)
10             (setf (gethash altobj log) altobj))))))
```

Snippet 7: Transactional context definition.

3.1 Transactions as Contexts

The transactions we have defined are simply contexts. We are thereby able to harness the existing context machinery provided by AmOS to execute code in *transaction context*. Thanks to this machinery, the `atomic` form is implemented easily as a thin wrapper around `with-context` —the usual construct for execution of code in a particular context— as shown in Snippet 6. The `atomic` form is simply syntactic sugar (a macro) to make managed invocations of `with-context`. The code wrapped by `atomic` is received in quoted form[5] as the `body` parameter of the macro (line 1). A clone of the prototypical `@transaction` is created (line 2) and the body is executed in that fresh transaction context (lines 4 and 5). If an error is signalled, the transaction is aborted and the error is forwarded to outer error handlers (by signalling it again in line 8). Hence, the `atomic` form is transparent with respect to error handling. If the body completes without error, the transaction is committed and the result of the body's execution is returned.

As shown in Snippet 7, transactions are specialised forms of contexts that have a *log* (line 2). The log is a hash table mapping objects to their *alter-objects* (by analogy to *alter ego*, "the other I" in Latin). Alter-objects are the transactional versions of objects that have been modified in transaction context. Given an object, the `alter-object` method defined on line 5 returns the transactional counter part of the object. The `host-context` call (line 6) yields the context where the currently executing method was found. In the case of `alter-object`, such context will be the trasaction that is currently active (the "host transaction"). The alter-object is thus fetched from the log of the current transaction (line 7). If no alter-object has been defined yet, a new one is created (line 8) and added to the log. Not only the original object is associated with its alter-object (line 9), but also the alter-object is associated with itself (line 10). Without the second

[5]Also called an s-expression in Lisp parlance.

```
(with-context @transaction                                    1
  (defmethod add-slot (object name value)                     2
    (setf object (alter-object object))                       3
    (resend))                                                 4
  (defmethod remove-slot (object index)                       5
    (setf object (alter-object object))                       6
    (resend))                                                 7
  (defmethod (setf slot-value) (value object idx)             8
    (setf object (alter-object object))                       9
    (resend)))                                                10
```

Snippet 8: Destructive operations specialised on transactional context.

association, the system would create transactional versions of alter-objects (alter-objects of alter-objects), and the chain would continue endlessly this way, producing transactional objects of transactional objects. There is no reason to keep protected versions of objects that have already been modified in the current transaction.

The use of the `$add-slot` primitive instead of the regular `add-slot` method on line 2 of Snippet 7 has to do with stopping infinitely recursive calls when the transaction log is read by `alter-object`. The `$add-slot` call creates a *direct* `log` accessor method that bypasses regular message passing, so that its invocation is not intercepted by transaction management logic. Hence, a recursive invocation of `alter-object` on line 6 is avoided. The use of `$add-slot` implies that the case study is not implemented *purely* on top of the object model; this is discussed later on.

3.2 Isolation of Destructive Operations

In transaction context, destructive operations such as `add-slot`, `remove-slot` and `(setf slot-value)` are intercepted and recorded in the current transaction log, instead of letting them modify their target object. Destructive operations are intercepted by defining transaction-specific versions, as shown in Snippet 8. These operations are part of the Meta Object Protocol (MOP) of AmOS. When a transaction context is active, the specialised versions will be applicable, and they will be more specific than the default versions. The three operations use the `alter-object` method defined in Snippet 7 to obtain a transactional version of their `object` argument. For each destructive method, the `object` argument is set to the corresponding alter-object and the message is resent. When invoked, the original versions of the methods will modify the state of alter-objects, instead of modifying original objects. Non-destructive operations need not be specialised.

Besides destructive operations, one more reflective method needs to be specialised on transaction context, although not for logging purposes. As application code executes, the system must make sure that any method invoked in transaction context sees the transactional versions of modified objects (i.e. the alter-objects), rather than the original versions. The `send` method is specialised on `@transaction` context for this purpose, as shown in Snippet 9. The `do-slots` form iterates over the message arguments. It replaces each argument of the original message by the version in the current transaction's log. If the argument has not been modified in transaction context and thus has no associated alter-object, it remains unchanged (`gethash` returns its third parameter

```
(with-context @transaction                                    1
  (defmethod send (selector arguments)                        2
    (let ((log (log (host-context))))                         3
      (do-slots (arguments argument index)                    4
        (setf ($slot-value arguments index)                   5
              (gethash argument log argument))))              6
    (resend)))                                                7
```

Snippet 9: Message sending behaviour for transactional contexts.

```
(defmethod reset ((transaction @transaction))                 1
  (setf (log transaction) (make-hash-table))                  2
  transaction)                                                3
                                                              4
(defmethod clone ((transaction @transaction))                 5
  (reset (resend)))                                           6
                                                              7
(defmethod abort ((transaction @transaction))                 8
  (forget transaction))                                       9
                                                              10
(defmethod forget ((transaction @transaction))                11
  (reset transaction)                                         12
  (resend))                                                   13
                                                              14
(defmethod commit ((transaction @transaction))                15
  (maphash 'become (log transaction))                         16
  ;; rough edge starts here                                   17
  (dolist (dependent (find-combinations-including             18
                       transaction))                          19
    (unless (object-equal dependent transaction)              20
      (let ((original (combination-excluding                  21
                        dependent                             22
                        (list transaction))))                 23
        (merge-contexts dependent original))                  24
      (forget dependent))))                                   25
```

Snippet 10: Transactional context definition.

if the given key is not found in the hash table; in this case the third parameter is the original `argument` of the message). Note that we use the `(setf $slot-value)` primitive on line 5 to set the value of each argument, instead of invoking the regular `(setf slot-value)` method. Sending a message within the `send` method leads to infinite recursion. For this reason, *all* invocations shown in Snippet 9 are primitives, thus avoiding recursive calls of `send`.

This completes the explanation of how MOP methods are specialised on transactional context to manage destructive operations and thus achieve transaction isolation. Once more, we observe that our support of transactions is not implemented *purely* on top of the object-oriented model. As illustrated throughout this section, we must use primitives at certain places to avoid infinite recursion chains. This is a natural consequence of interceding the most fundamental mechanisms of the model such as `send` and `(setf slot-value)`. Inifinite recursion is a well-known artefact of reflection [2].

3.3 Transaction Management

Snippet 10 shows the implementation of main transactional operations. The `clone` method is overloaded for transactions so that each newly created transaction has a fresh log (created by `reset`). Aborting a transaction simply means forgetting the context, which is a standard operation in AmOS:

`forget` removes the given context from the context registry maintained by the run-time system. The `forget` method is specialised on transactions to reset the log, thereby freeing held resources (i.e. letting the garbage collector reclaim the alter-objects referenced by the log).

When a transaction is committed, the modified versions of objects recorded in the log replace unmodified versions. To this end, the transactional log is traversed by means of a `maphash` call on line 16. The Actor-inspired `become` operation [1] is used to substitute the modified version stored in the log for the unmodified one.[6] Hence, the alter-objects become the original objects.

Note that if another thread gains control in the middle of a `commit` operation, it might see an inconsistent state in which some alter-objects have become the original ones, but others have not. To remedy this situation, `commit` should prevent other threads from accessing any of the objects in the transaction log until it finishes. We did not address this issue in our case study.

Rough Edge

Normally, having every alter-object become the original object as explained previously should be sufficient to commit the transaction. However, in our implementation, context objects are not protected by transactions. This means that the context objects that were created during the transaction need to be copied back to non-transactional context by hand (lines 18–25 of Snippet 10). For the sake of brevity, we do not explain this process here.

4. DISCUSSION

The transactional system we have presented is mainly a lateral thinking experiment on novel ways to use COP. Instead of exploring a typical COP application, we set out to explore the feasibility of using contexts as transactions. This study of COP for transactions is different from typical COP examples in that it "goes meta" —the application that is being adapted is AmOS itself. Fundamental system behaviour is thus specialised on transactional context and can vary at run time depending on context.

Conceptually, we find it intuitive to think that a transaction constitutes a special context in which code is run. Objects are seen from a transactional perspective, and from this special point of view, their semantics is different than the semantics observed from non-transactional perspectives, in that changes are isolated and can be rolled back atomically. This matches neatly the notion of subjective objects [16]. Technically, we observe that considering transactions as plain contexts leads to a rather straightforward implementation that exploits existing machinery. We intercept all message sends and all calls to destructive operations by specialising reflective operations, so that every single object is protected in transaction context.

AmOS did not need any modification to accommodate the transactional extension shown in the case study. This was made possible thanks to its MOP. At the meta level, the MOP of AmOS can be seen as a framework with hooks that allow adaptation to different contexts (in this case, transactional context). This is analogous to base-level frameworks in which behaviour is defined through a well-designed set of objects and hook methods. Clean framework protocols increase the possibilities for context adaptation, whether these protocols are at the base level or at the meta level.

In our implementation, adaptation code needs to bypass the computation model. We believe however that this should not be regarded as a paradigmatic shortcoming of COP. Rather, it is a paradigmatic trap of reflection: such paradoxes are observed regularly in systems that are about themselves. Examples are the metaclass regression problem encountered in Smalltalk [6] or the problem of unwanted meta-level call recursion as described for CLOS [2]. Such regression problems must be shortcut in some way. In this paper, we have realised a simple solution. We avoid recursive calls by using primitive functionality at specific parts of adapted reflective behaviour. Since we did not place a shortcut *within* the computation model, the transaction mechanism we present cannot be regarded as an extension of AmOS, but rather as an integral part of the model. A solution that we plan to explore in the future is to use COP itself to solve the recursion problem more elegantly. The problem of unwanted recursion is, in essence, a problem of context. When executing meta-level behaviour, we do not want to execute the meta-level behaviour again. This problem of calling the right behaviour depending on meta vs. base-level execution is easily described as a context [5].

On a final note, we did not explore nested transactions yet, although we foresee no fundamental impediment in supporting them with our approach: each transaction has its own log, and at most one transaction can be active in the current execution thread.

5. RELATED WORK

Context-Oriented Programming. ContextL [3] realizes a context-oriented programming language. *Layers* provide behavior variations and can be enabled explicitly for the execution of a function and thus provide a form of execution context. Reflection in ContextL has recently been explored [4], but the focus has been reflective layer activation, rather than making reflection itself context-aware.

Subjective Programming. Us [16] extends the prototype-based language Self [17] to support subject-oriented programming [11]. In Us, message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContexL. The paper discusses the usefulness of subjectivity for controlling access to reflection APIs, but it does not go as far as making reflection itself subjective.

The MetaHelix. The challenge of unwanted meta-level call recursion has been explored by Chiba *et al.* [2]. The first problem discussed is related to changes of structure. For example, fields added as part of the implementation of a reflective change are visible globally and thus destroy the

[6] Common Lisp's standard `maphash` function calls the function passed as first argument for each key/value pair stored in the hash table passed as second argument. This means that `become` is called for each object/alter-object pair.

reflective model. The second problem noted is recursion: any introduction of new meta-behavior can lead to an endless loop of calls to the same meta-behavior. The solution proposed is the *MetaHelix*. All meta-objects have a field `implemented-by` that points to a version of the code that is not reflectively changed. The MetaHelix thus is very similar to the solution presented in this paper. The system provides the programmer with the original version of the code and the programmer can choose to call this version to break recursion.

Meta Context. One of the authors has explored the notion of contextual reflection for modeling meta-level execution [5]. Here reflection is made context-aware to solve the recursion problem. As we discuss in Section 4, the same problem can be seen in AmOS. We think that combining both ideas is interesting and we plan to work on combining both models in the future.

Transactional Memory for Smalltalk. Renggli *et al.* [14] have realised transactional memory for Smalltalk without changes to the virtual machine. The implementation therefore heavily relies on reflection to implement transactional memory. Reflection is used for realising context-dependent code execution, yet the system does not provide an object model where reflection in general is contextual.

6. CONCLUSIONS

Context-oriented reflection is a particular case of context-oriented programming in which the object model itself becomes adaptable according to context. Despite a few glitches in our implementation, the case study we present on lightweight memory transactions constitutes a first indication that COP makes a powerful combination with computational reflection. Fundamental system behaviour is adapted to situations in which a memory transaction is taking place. Such adaptation is succinct and easy to understand. This experiment leads us to conclude that context-oriented reflection is worth further exploration, in particular the idea of defining lightweight memory transactions as plain contexts.

7. ACKNOWLEDGEMENTS

This work has been supported by the *ICT Impulse Programme* of the Institute for the encouragement of Scientific Research and Innovation of Brussels, by the *Interuniversity Attraction Poles Programme* of the Belgian State, Belgian Science Policy, and by the *Biologically Inspired Languages for Eternal Systems* project of the Swiss National Science Foundation.

8. REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for Actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of International Symposium on Object Technologies for Advanced Software*, volume 1049 of *LNCS*, pages 157–172. Springer, 1996.

[3] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Dynamic Languages Symposium*, pages 1–10. ACM Press, Oct. 2005.

[4] P. Costanza and R. Hirschfeld. Reflective layer activation in ContextL. In *Proceedings of the ACM symposium on Applied computing*, pages 1280–1285. ACM Press, 2007.

[5] M. Denker, M. Suen, and S. Ducasse. The meta in meta-object architectures. In *Proceedings of the International Conference on Objects, Components, Models and Patterns*, volume 11 of *LNBIP*, pages 218–237, 2008.

[6] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[7] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[8] S. González, K. Mens, and P. Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Dynamic Languages Symposium*, pages 77–88. ACM Press, Oct. 2007.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[10] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005.

[11] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. *ACM SIGPLAN Notices*, 28(10):411–428, 1993.

[12] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March–April 2008.

[13] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., June 2004.

[14] L. Renggli and O. Nierstrasz. Transactional memory in a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):21–30, Apr. 2009.

[15] L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 312–336. Springer-Verlag, 2005.

[16] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.

[17] D. Ungar and R. B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22:227–242, 1987.

[18] J. Vallejos, P. Ebraert, B. Desmet, T. Van Cutsem, S. Mostinckx, and P. Costanza. The context-dependent role model. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, pages 277–299. Springer-Verlag, 2007.

[19] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.