Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages

THÈSE

présentée et soutenue publiquement le 18 Février 2013

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Arnaud Jean-Baptiste

Composition du jury

Président :	M. Roel Wuyts	
Rapporteur :	M. Alain Plantec, M. Ciarán Bryce	
Examinateur :	M. Laurence Tratt, M. Roel Wuyts, M. Hervé Verjus	
Directeur de thèse :	Stéphane Ducasse	(Directeur de recherche – INRIA Lille Nord-Europe)
Co-Encadreur de thèse :	Marcus Denker	(Chargé de Recherche(CR1) – INRIA Lille Nord-Europe)

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 41076

Acknowledgments

First of all, I would like to express my gratitude to my supervisors Stṕhane Ducasse and Marcus Denker for giving me the opportunity to be part of this adventure. Thank you for all your support, encouragement and advice throughout the years. Thanks for giving me the desire to continue in research.

My thesis has been done in an especially pleasant context, thanks to the RMOD team members. In particular, thanks to Igor Stasenko for all the discussions we had and for his so particular vision. Thanks to Nicolas Anquetil, Damien Cassou and Damien Pollet for all the time spent providing me advice about writing and presenting. Thanks all for the discussions around all the coffees.

I would like to thank Roel Wuyts, Alain Plantec, Ciarań Bryce, Laurence Tratt and Hervé Verjus for accepting to be part of my PhD committee. I am honored to count with your presence and I really appreciate your comments and advice.

I am deeply grateful to my family and in-laws who have supported me during this experience. Especially thanks to my parents for everything they provide to me, and to all my friends who made these years pass so fast.

Above all, I want to thank my beloved Natacha for being part of my life and for sharing many unforgettable moments with me, supporting and encouraging me through these years. Even if that was not easy.

To Natacha,

Contents

CHAPTER 1
Introduction

Contents

Dynamically-typed programming languages do not provide type information until runtime. Two of their main advantages are that they allow fast prototyping and integrating changes at runtime. These advantages are increasingly required by computing environments, which need to be ubiquitous, pervasive and dynamic [Ancona 2007]. These advantages make dynamically-typed languages popular, as is the case for JavaScript, which is embedded in most web-browsers and mobile phones, and Lua, which is embedded in World of Warcraft and Adobe Photoshop [Tratt 2009]. Some execution environments that are inherently static start to support dynamic typing following their user's requests, for example the Java Virtual Machine.[1]

The ability of dynamically-typed languages to support program changes at runtime and the lack of type information until runtime are also their biggest weaknesses. These weaknesses doom static analysis to failure. Analyzing an application statically means examining the code to extract an understanding without executing it. Static analysis is principally used to ensure that the code structure respects a specific standard of quality and security [Hogg 1991, Almeida 1997, Noble 1998, Müller 1999].

In a context where an application can change at runtime, the system developers must ensure that the application cannot trigger undesirable behavior such that harming the system and collecting private information. The most commonly used way to prevent undesirable behavior is to severely limit the expressivity of the programming language. However, what is really needed is only to limit the impacts the application can have on the rest of the system.

One way to confine the impact an application can have on a system is to control how the application uses references and by enforcing security and safety properties there. Several approaches have been proposed in this direction such as *dynamic ownership* [Gordon 2007], *capability for sharing* [Boyland 2001], *functional purity* [Finifter 2008], and

[1] Sun Microsystem http://jcp.org/en/jsr/detail?id=292

encapsulation [Snyder 1986,Schärli 2004a]. Nevertheless, these approaches are not widely adopted in practice; we think this is because they are so specialized that many of them must be combined in any real world system and each of them requires expertise.

We thus decided to focus our research on providing a general-purpose way to control references in dynamically-typed languages.

1.1 Context

In this section, we study dynamic typing and the constraints it implies on possible solutions to control references.

1.1.1 Dynamically-typed languages

In a dynamically-typed language, type checking is performed at runtime instead of at compilation-time. Dynamic languages such as Smalltalk, Ruby and JavaScript do not store static type information in their compiled code. Variables are not typed and no type information appears until runtime.

A dynamically-typed language is typically more flexible than a statically-typed one: a dynamically-typed language accepts and executes programs that would be ruled as invalid by a static type checker [Gordon 2007]. The counterpart is the lack of type information at compile time: security and refactoring tools benefit from a strong type system to detect potential issues.

Moreover, dynamically-typed languages often come with reflection that undermines static analysis. Reflection allows a program to examine, change and execute behavior of objects at runtime. Since dynamic typing makes it easy to keep programs open to changes at runtime, dynamically-typed languages usually have a powerful reflective API that we cannot ignore while designing a solution to control references.

> Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both incomplete because some parts of the program may not be included in the application call graph, and unsound because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection.
> **Livshits and co.** In *reflection security* [Livshits 2005].

The fact is that Javascript, Lua, Smalltalk, Ruby, Python, and most widespread dynamically-typed languages have a strong reflection API that provides a benefit of expressivity. To be useful in the real world, a solution to control references should thus manage reflection; in this thesis we argue that it is possible to control references in the context of dynamic languages *with* reflection.

1.1.2 Controlling references

In object-oriented systems, a typical program creates a large amount of objects and even more references to those objects, forming a highly interconnected and evolving graph. In that graph, when multiple references point to the same object we speak of *aliasing* [Hogg 1991, Almeida 1997].

Aliasing becomes a problem when one has to limit the extent to which references should be shared, either between objects or across time. In most object-oriented programming languages such as Java, Smalltalk, Python and Ruby, references are the only way to pass and manipulate an object. However, these languages do not control *how* the references are passed among objects and thus references can easily be shared to unexpected objects. Once obtained, a reference allows full access to its object's public interface. Consequently, because object-oriented languages do not enforce any control on aliasing or on object access at the reference level, the consequences of modifying a shared object are difficult to predict.

Despite these difficulties, aliasing remains a core concept in imperative languages, and appears a lot even in well structured object-oriented programs. Since an aliased object can simultaneously be accessed from different perspectives (*i.e.*, different references), we need a mechanism to control these perspectives. One way to do this is to control the references.

1.2 Controlling references in dynamically-typed languages

To clarify, by *controlling references*, we mean determining the behavior and supervising the use of references. In this section, we sketch the three requirements for controlling references within the constraints imposed by dynamic languages with reflection, and thus without taking advantage of static analysis.

Behavior control. We want that an object has different behavior depending on which reference is used to access it. Some references can have more behavior to what the object actually provides (*e.g.*, logging mechanism) or less (*e.g.*, read only). To illustrate this part, a read-only version of an object can be created by dynamically changing its class to a read-only version. However, this ignores aliasing: since the behavior changes at the object level, it is enforced for every alias of that object. In many situations what we really want is read-only behavior from some perspectives, while retaining normal behavior from other perspectives.

We need infrastructure to enforce specific behavior at the reference level.

State isolation. We want that an object has different state depending on which reference is used to access it. This is particularly useful in the context of software transactional memory where a sequence of state changes must only be visible to the references used in the transaction until the end of transaction. From the point of view of software transactional memory, we need to hide then apply state changes at the reference level.

We need infrastructure to isolate state at the reference level.

Propagation control. Controlling a single reference to a single object is not enough in most cases, because the object's graph can contain a back pointer to the object. For example, a read-only reference must guarantee that no modification will be made to the object, even indirectly from the object's graph.

We need infrastructure to propagate this specific behavior and state enforced at the reference level.

1.3 Thesis

As stated above, we want to confine the impact an application can have on a system through the control of references in dynamically-typed languages. To achieve that goal, we identified a list of requirements for controlled references.

We can now state our thesis:

In the context of dynamically-typed languages, reifying references, controlling behavior, and isolating state via such references, is a practical way to control references.

This dissertation focuses on the problem of controlling references and provides solutions based on the three previously identified requirements. First, we explore read-only execution at the reference level. We model read-only behavior at the reference level using transparent proxies, then provide a concrete implementation and benchmark. Second, we raise the abstraction level of this model to support a more generic way to control references. We then implement the infrastructure for controlled references in two steps: we first ensure behavioral control, we then ensure state isolation. To conclude, we present the possible research directions opened by our work.

Contributions. This thesis focuses on the confinement of the impact an application can have on a system through the control of references in dynamically-typed languages. In this context, we made the following contributions:

- *Dynamic read-only objects for dynamically-typed languages* [Arnaud 2010]. We propose dynamic read-only objects (DRO) as an approach to provide read-only behavior at the level of references (Chapter 4). Our approach offers a way to create read-only references to any object in the system. In addition, our approach propagates the read-only property to the object graphs accessed through these references.

- *Handles: behavior-propagating first-class references for dynamically-typed languages* [Arnaud 2013]. We generalize the DRO model and enable behavioral changes. We extend the Pharo environment and programming language[2] with *Handles*, *i.e.*, first-class references that have the ability to change the behavior of referenced objects (Chapter 5).
- *Metahandles: controlling Handles at runtime*. We define Metahandle to offer flexibility and adaptability to controlled references (Chapters 5 and 6).
- *SHandles: controlling the visibility of side effects*. We propose SHandle, an extension of the Handle model to isolate side effects at the level of references (Chapter 6).
- *A working implementation*. As proof of concept, we extend the Pharo virtual machine to support Handles, Metahandles and SHandles (Chapter 7).
- *A formal model*. We formalize the Handles and SHandle models to represent and explain their semantics (Chapter 5 and Chapter 6).

1.4 Structure of the dissertation

Chapter 2 presents the problem of controlling references in the context of reflective dynamically-typed languages. In this chapter, we analyze three approaches representative of the security and safety domains. We then extract from these approaches the requirements to provide controlled references: (i) behavior control, (ii) state isolation and (iii) propagation control.

Chapter 3 discusses the literature about controlled references, especially in the context of dynamically-typed languages.

Chapter 4 presents the implementation of a simple and well-known (read-only) property at the level of references. We use the implementation of this simple property as a first step to understand how to reify references.

Chapter 5 introduces the Handle model that associates behavior modification properties to references. In this chapter, we validate our model by implementing two of the three approaches studied in Chapter 2.

Chapter 6 introduces the SHandle model as an extension of Handles for isolating state at the reference level. In this chapter, we validate our extended model through the implementation of the third approach studied in Chapter 2. We additionally implement the World model [Warth 2008] to ensure that our extended model is generic enough.

[2] http://www.pharo-project.org

Chapter 7 presents the implementation we realized of above two models.

Chapter 8 concludes the dissertation and presents potential research perspectives opened by our work.

CHAPTER 2

Requirements for Controlled References

Contents

In the previous chapter we introduced the problem of enforcing security in the context of dynamically-typed languages. In that context, we focus our research on controlling references.

The goal of this chapter is to extract a set of requirements for implementing controlled references. Towards this goal, we analyze three approaches representative for security and safety domains in the context of reflective dynamically-typed languages. First of all, we explain the constraints inherited from these reflective dynamically-typed languages. Then we present the three approaches and extract the set of requirements to implement controlled references.

2.1 Constraints of reflective dynamically-typed languages

In programming languages a *type* is a set of values, and operations applicable on these values. In many object-oriented languages the class of an object represents the *type* of an object. To ensure that objects only perform valid operations (known and/or primitive messages) the *type* of this object is checked. When the language delegates to runtime the type checks, this is dynamic typing by opposition to static typing where the type checks are done at compile time [Cardelli 1997,Tratt 2009]. A reflective dynamically-typed language is defined by a causal link between its model and itself [Maes 1987]. Languages with this causal link are called *reflective*. Reflection can be separated in two main aspects [Bobrow 1993]:

Introspection is the ability to query the representation of the system (e.g., fetching the value of an instance variable of an object).

Intercession is the ability to change the representation of the system (e.g., adding a new method to a class).

Intercession makes it difficult to anticipate how the behavior of a program will evolve during its execution. One can statically check all methods potentially executed by the program, but if new code is loaded at runtime then this analysis is not valid anymore. In an such an open world system, it is impossible to have complete knowledge of the execution of a program. Consequently, it is not possible to offer a practical solution for controlling references based only on static analysis in the context of reflective dynamically-typed languages.

2.2 Selected approaches

Now we present three approaches representing security or safety features commonly used. The goal is to extract a set of requirements to implement controlled references.

Read-Only Execution ensures that a specific execution does not change any accessed object.

Software Transactional Memory is a concurrency control mechanism controlling access to shared resources in concurrent computing. Software Transactional Memory has been introduced by Nir Shavit [Shavit 1995] and has been the focus of research [Lie 2004, Moore 2006, Shavit 1995] since its introduction, including in context of dynamic languages [Renggli 2007].

Revocable References can be revoked by their owner. Revocable References have been introduced by Redell [Redell 1974] and applied to capability systems by Miller *et al.* [Miller 2003b].

2.2.1 Case one: read-only execution

Mechanisms prohibiting variable mutation are now common in programming languages. Java can declare a field as *final*, and C# has the *readonly* keyword. The two keywords prevent the fields from being modified by something else than the constructor. C and C++ have the *const* keyword: a value declared as *const* can not be changed by any part of the program in any way.

In dynamically-typed languages, few attempts have been made to prohibit variable mutation. One such attempt is *immutability*, a property embedded in the object itself that define once created the object cannot be modified via any alias.

Our solution to prohibit variable mutation is read-only execution per reference. This form of read-only execution ensures the side-effect free evaluation of code. Read-only execution is often needed when objects taking part in the read-only execution are at the same time (e.g., from another thread of execution) referenced normally with the ability to do side-effects. A clear illustration of this need is assertion execution: within the scope of an assert constraint, objects (reachable from the assertion expressions) should not be modified, else this might lead to subtle bugs depending on whether or not assertions are evaluated.

To illustrate, look at the following code:

```
(self checkPrecondition)
   ifTrue: [ self doSomething ].
```

In the previous code, we check a precondition using the method *checkPrecondition*, then if the precondition is *True* the message *doSomething* is sent. Since the precondition is a normal method, *checkPrecondition* may, or may not, have side effects. But we want to ensure that *checkPrecondition* does not do any side effect on any of the objects during its execution. One solution is to enforce read-only behavior when the message *checkPrecondition* is sent to ensure that *checkPrecondition* does not have side effect.

In addition usually an object is referenced from several objects at same time. These other references should conserved the normal behavior. We need to have read-only behavior per reference.

What we want to stress with this example is first the need for read-onlyness. Second, the same object can be referenced at the same time from different perspective, and the read-only behavior should be *enforced for selected perspectives or references*.

Readonly behavior is based on the behavior of the object addressed. A readonly behavior can be created from the class definition itself. But we still need to enforce this behavior at reference-level. From this example we deduce the following requirement:

Per reference behavior shadowing. We need to control behavior at the reference-level. The same objects can be referenced from several perspectives. A read-only behavior should be readonly only for a specific perspective.

2.2.2 Case two: software transactional memory

Concurrent programming is control simultaneous execution of multiple interacting computations [Yonezawa 1987]. The objective is to improve the use of available hardware resources more efficiently. In some domains such as distributed systems, concurrency control is mandatory. Software Transactional Memory (STM) is a modern paradigm, based on transaction synchronization, which takes a different approach from conventional approaches such as mutual exclusion and message passing; software transactional memory, is seen by some as a more user-friendly approach to synchronization [Bieniusa 2009].

Software transactional memory provides high-level mechanisms to control mutual exclusion, based on the concept of *atomic block* used to encapsulate execution. The transaction begins when entering in an *atomic block* and finishes when leaving it.

Software transactional memory ensures that the change introduced by the computation will be fully performed or not at all. there exist several kinds of STM systems, defining different semantics for the *atomic block*, and also different interaction between normal computation and transactional computation [Shavit 1995]. Let us take, the following code:

```
1  LLNode>>atomicallyInsertAfter: aNode
2   | transaction |
3   "Insert a node into a double linked list atomically"
4   transaction := Transaction new: [:nodetoinsert :node |
5      nodetoinsert previous: node. "Step 1"
6      nodetoinsert next: node next. "Step 1"
7      node next previous: nodetoinsert. "Step 2"
8      node next: nodetoinsert. "Step 3"]
9   with: { aNode. self. }.
10  transaction run.
```

The code represents a method inserting atomically a node in a double linked list. In the previous code there are three important steps, (i) initialize the nodetoinsert (lines 5 and 6), then (ii) make the previous pointer to the next node pointing on the nodetoinsert (line 7) and finally (iii) make the next pointer of the current node pointing on the nodetoinsert (line 8). If this behavior was not performed in a transaction and two concurrent processes executed steps 2 or 3 at same time, we will probably reach an inconsistent state.

At the implementation-level, *data versioning* is how the changes are stored which occur during the transaction. There are two cases [Bieniusa 2009]: (i) *eager versioning* performs in-place memory update during a transaction. It saves overwritten values in an undo-log structure for reconstruction on a potential rollback, (ii) *lazy versioning* each atomic block maintains its own local write buffer whose values are later on committed to the shared memory. The two approaches have issues, in *eager versioning* we need to completely freeze the system because a read access of the objects will reach an inconsistent value and in *lazy versioning*, the local copy is expensive to maintain and because it is using a copy, it breaks the identity.

We propose to *isolate state changes at reference-level*. In that context, we do not freeze the concerned objects and we preserve their identity. To do that, we need a mechanism able to isolate state change at reference-level.

From this example we deduce the following requirement:

Per reference state shadowing. The state needs to be isolated at reference-level.

2.2.3 Case three: revocable references

The last approach presented is revocable references. This approach defines specific references that can be revoked when the owner considers they should not be shared anymore. The revocation pattern was introduced by Redell [Redell 1974] and is adapted for dynamic languages by Miller *et al.* [Miller 2003b].

Figure 2.1 shows an example with three objects: Alice can give Bob a reference to Doc. But Alice should be able to revoke it later *i.e.*, Bob cannot access it anymore even if he holds a reference to it. The conceptual solution proposed by Miller *et al.* is to create a revoking facet (R) and pass it to Bob. Such a facet can be seen as an object with a restricted interface or a first-class reference. Note that in this original proposal revocable references are not propagated automatically [Miller 2003b]. The facet needs to be carefully thought to not leak references and only return facets instead.

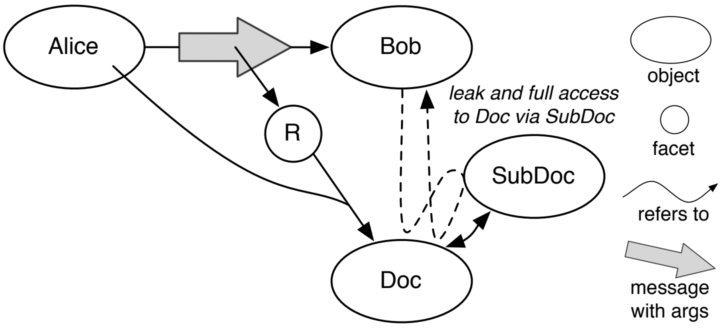


Figure 2.1: Revocable references: Alice can decide Bob should not be able to access Doc. References to SubDoc should also be revokable else Bob can access Doc even after revocation.

In the example, Alice has to make sure to *wrap all objects* discoverable from the reference handed to Bob. Idioms and special safety patterns should be followed by the programmer to make sure that there is no reference leaked by accident. Indeed, imagine that Doc holds a reference to a SubDoc which also has a back pointer to Doc. While Bob cannot access Doc once its reference to Doc is revoked, if Bob gets a reference to SubDoc and this reference is not a revocable one then Bob broke the system and can access Doc even if it should not be able to do so.

The revocable references do not provide a secure mechanism if all the graph access via the reference is not automatically revocable. We need a mechanism to ensure that the different properties defined at the reference-level will be propagated automatically. This example shows that we need to control and automatically apply to a subgraph. From this example we deduce the following requirement:

Propagating control. The change provide at reference-level should be propagated to its subgraph. Moreover we need to provide a way to represent how to propagate these changes.

2.3 Required properties

The previous approaches show the following requirements for controlling references:

Per reference behavior shadowing. Change the behavior of an object from one reference controls how the execution will be performed from this specific reference. We need to control behavior at reference-level.

Per reference state shadowing. Isolate the state of an object from one reference is a way to control the scope of the side effect which occurs from this perspective. We need to isolate state at reference-level.

Propagating control. Control the state and the behavior of a single object at reference-level is not enough to ensure control of this reference, we need to propagate these controls to subgraph elements.

In this chapter we defined the specifications required for providing controlled references. In Chapter 3, we will provide an overview on the current research state about controlling references.

CHAPTER 3
Related Work

Contents

In the previous chapter we defined a set of requirements to implement controlled references in the context of reflective dynamically-typed languages. The goal of this chapter is to present a selected part of the existing research about *controlled references* within dynamically-typed languages.

We first present the four approaches representative of the security or safety domains which can be used in context of dynamically-typed languages and study the usability of these approaches. Second, we present an overview of other solutions existing outside our context.

3.1 Capabilities

Capabilities can be considered as the parent one of our work. It is a way to reify references and control behavior. As a definition, Levy proposed: "A capability is a key to access to a given resource (hardware or software)" [Levy 1984].

In the rest of the chapter, we consider capability-based computer systems, then we we will look at more recent work in the context of object-oriented programming languages.

3.1.1 Capability-based computer systems

In Capability-Based Computer Systems [Levy 1984], M. Levy provides an overview of capabilities. Capabilities provide an unifying mechanism to access software and hardware resources (including memory access). He defines a capability as a tuple composed by an *oid*(object identifier) and an *access right*, see Figure 3.1.

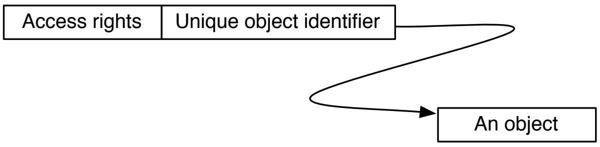


Figure 3.1: A capability: a tuple composed by an *oid*(object identifier) and an *access right*.

In Capability-Based Computer Systems, each user or process has a *list of capabilities* that represents all the objects they can access, and which operations can be performed on them. Capability-Based Computer Systems maintain integrity by prohibiting direct modifications to the *list of capabilities*. Only a few operations are allowed on capabilities:

- Move capabilities in a capability list.
- Remove or restrict a capability (restrict produces a new capability).
- Transfer a capability (via parameter for procedure or as resource to an user).

For example, Figure 3.2 defines an access matrix representing a set of right accesses for each user (The corresponding graph is given in Figure 3.3). As we see in Figure 3.3, Carol does not have any pointer on *Process A* or *File A*, so Coral cannot performed operation on *File A* or *Process A*. Each capability allows a set of operations. So Bob can perform Read, Write operations on *File A*, and Alice can only read *File A*.

			System's ressources	
		File A	**Process A**	**Mailbox A**
Users	*Bob*	*Read, Write*	*Delete, Suspend, Wakeup*	
	Alice	*Read*	*Suspend, Wakeup*	*Send, Receive*
	Carol			*Send*

Figure 3.2: Example: access matrix [Levy 1984]

Object-oriented systems and capability-based systems. In object-oriented systems, objects are identified by an unique identified (their address in memory). Levy adapts the capabilities approach to object-oriented systems. *Resources* (defined in the capabilities model) are the objects and the *operations* are the methods of object.

The Capability-Based Computer Systems approach applied to object-oriented systems succeeds in the sense that it allows one to change behavior at the reference-level (capabilities can be summarized as a reference reification). But this behavior modification is restricted to allow or deny access to methods where we need behavior change. Moreover, Capability-Based Computer Systems requires to prohibit all low-level structures (bits,

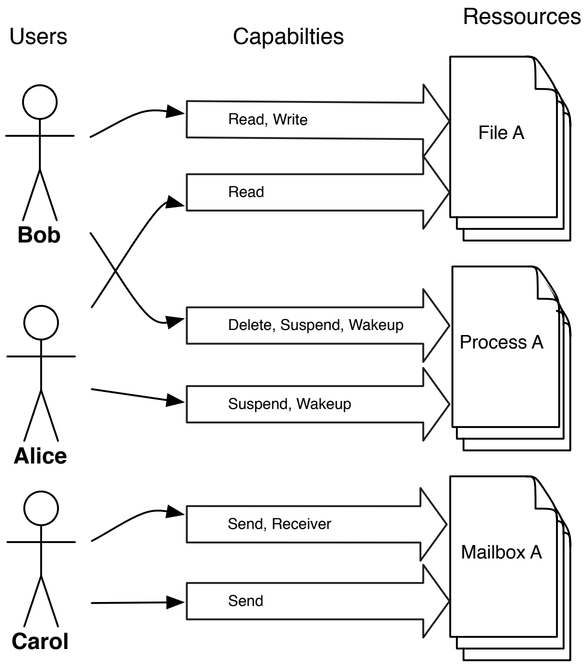


Figure 3.3: Example: access matrix graph produced

pointer, *etc.*) modification of object. Note that, these low-level structures modifications are equivalent to reflective behavior.

3.1.2 Erights

In Erights [Miller 2003a, Miller 2006], capabilities are used to enforce complex security properties. Miller refined capabilities that apply then to object-oriented model. Miller describes capabilities:

> Our object-capability model is essentially the untyped lambda calculus with applicative order local side effects and a restricted form of eval the model Actors and Scheme are based on [Miller 2006].

In [Miller 2006], Miller used the object-capability model to demonstrate that object-capability can enforce revocable, confinement, and *-properties.

Revocable. Revocable properties is the same as what we presented in Chapter 2. As we say before revocation pattern was introduced by Redell [Redell 1974].

To implement the revocable reference, Miller uses an implementation of the Redell's Caretaker pattern [Redell 1974]. The Redell's CareTaker defines a proxy object, that forwards all unknown message sends to a *target* object and has a method revoke.

The idea is to restrict the view on this object using capability, see Figure 3.4

- a Controller reference is a reference restricted by capability to only have the revoke method.
- a Revocable reference is a reference removing all the method access rights.

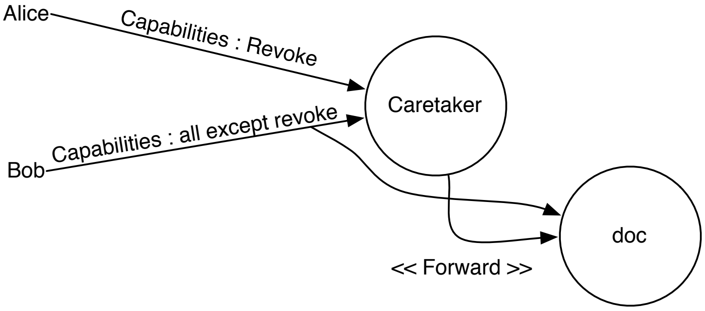


Figure 3.4: Caretaker pattern using capabilities

To illustrate the process, we reuse the same example we described in Chapter 2.

Alice creates a caretaker proxy on the doc. Alice gives Bob the *Revocable reference*, this reference will forward all the messages received to doc, giving full access to doc via the caretaker proxy. Alice keeps the Controller of the proxy, a reference that allows to send the revoke message to the caretaker proxy. Bob will have access to doc, because the caretaker will forward all the messages received to doc. But once Alice decides to revoke the access, Alive sends the message revoke to the Caretaker proxy, then the target of the Caretaker becomes *nil* and all messages sent from the reference hold by Bob will be forwarded to nil.

Confinement. The dictionary definition of confinement is literally: "Confinement is the act to enclose within bounds".

Let us illustrate that with the example of the calculator.

- A Customer wants to use a calculator provided by a Provider in its own application. The Customer wants to ensure the code embedded in the calculator will not access secret information of its application.
- A Provider wants to sell its products to a Customer. But the Provider does not want to leak the calculation algorithm of its calculator to the Customer.

The Provider needs to confine its calculation engine to enforce the two requirements. First the code the calculation engine will not access private data of the Customer. And Customer will not access to private calculation algorithm of the Provider.

To solve this confinement problem, the Customer and the Provider have a mutual access to a FactoryMaker and a Factory. The Factory and the FactoryMaker are stamped and considered as trustable by the two actors. The idea is to package the code of the Provider in FactoryMaker, and pass the Factory created to the Customer via an accept method.

When The Customer instantiates the Factory, he passed the capabilities allowing the behavior that is considered as safe. The instance of the factories created by the customer warrant that its behavior do not bypass the capabilities defined by the customer.

There are two consequences of this design. First the code embedded in FactoryMaker is not visible to the Customer. And second using capabilities the Customer can warrant that no leak will be performed by the code embedded in the Factory, because it only allows specific and known operation using capabilities.

***-property.** For going deeper in confinement problem, Miller shows how to enforce the *-property. The *-property was introduced in [Bell 1974]. The *-property is satisfied if:

> In any state, if a subject has simultaneous "observe" access to A and "alter" access to B, then level(A) is dominated by level (B).

The *is dominated* term comes from graph theory. An Objects A is *dominated* by an object B if all ways to access to object A passed by objects B. To simplify, the *-property is respected if two object A and B where A have a reference to B. Then an object C has a reference on A, it cannot manipulate B except if it passed by A. The *-property allow high-level objects to access low-level objects but disallow low-level object to access high-level objects.

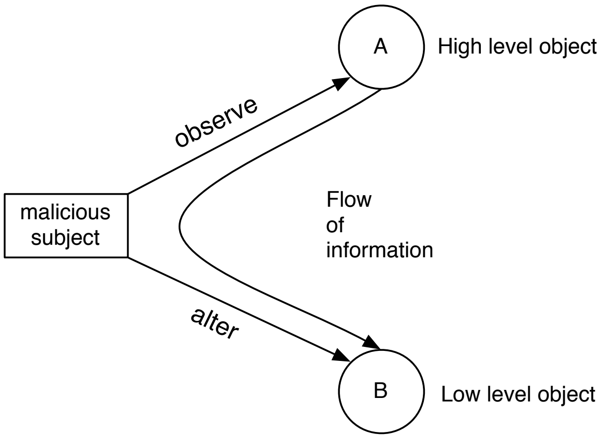


Figure 3.5: Example: *-Property: an object that observes A should not alter B. Here the *-property explicitly disallow this kind of structure. [Bell 1974]

Contrary to [Gong 1989, Kain 1987, Wallach 1997, Saraswat 2003, Boebert 1984], Miller proved that capabilities can be used to create one way channel that supports *-property. He implements its *-property using the same pattern used for confinement. Let us illustrate that by an example. A Customer wants to provide a one way channel between two subjects, a Reader and a Writer. Customer does not trust Reader or Writer. To solve that, Customer creates two factories one with write capabilities on a pool variable and one with read capabilities on this same pool variable. Customers gives read capabilities references to Reader and write capabilities reference to Writer. So Reader can only read the pool variable and Writer can only write into it.

To conclude, Miller rehabilitates capabilities and implements revocable, confinement and *-property using object-capability model [Miller 2006]. The limits of the approach are that these solutions can only be used in restricted a situation. In addition, the revocable pattern can leak a reference to a target object because it does not wrap the subgraph object, if a sub-object references the root object, then it is possible to fetch a full-access reference to the root object.

The model of Miller does not provide automatic propagation of these properties. Each pattern should be manually propagated. Moreover the approaches of Miller are based on patterns, consequently using reflection we can introspect the pattern and break the properties ensured by the pattern previously described (confinement, *-property and revocable).

To conclude, even if Miller successfully proved that the capabilities model regain confinement, revocable and *-property, the capabilities model of Miller still does not fulfill our requirements. The goal is different: We are interesting generally in the question of how to control references, capabilities are just one example.

3.2 Encapsulation

Encapsulation is one of the cornerstone of the object-oriented paradigm. In [Snyder 1986], Alan Snyder define *encapsulation* as:

> Encapsulation is a technique for minimizing inter-dependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. [Snyder 1986]

Most of the solutions to support encapsulation are not compatible with Dynamically-typed languages. MUST [Wolczko 1992] is one of the compatible solutions. MUST introduces a categorization of methods in Smalltalk (*private*, *protected*, *superclass-visible* and *public*) and checks message sends at runtime. This model is static (all is defined at compilation time).

Object-oriented encapsulation for dynamically-typed languages [Schärli 2004a] is another existing solution that can be applied to Dynamically-typed languages, we explain it in detail in the following section.

3.2.1 Object-oriented encapsulation for dynamically-typed languages

In "Object-oriented Encapsulation for Dynamically-Typed Languages" [Schärli 2004a] Schärli *et al.* provide a solution to implement encapsulation for dynamically-typed languages. The solution is based on the syntactic distinction between self-sends (potentially private), object-sends and super-sends and the introduction of policy objects which regulate accesses to objects and their methods (overriding included).

The goals of Schärli are threefold: first, to provide an encapsulation implementation to minimize interdependencies between classes. Second, all the mechanisms should be simple and do not raise the complexity of the language itself (from a semantic point of view). Finally the key difficulty in context of dynamically-typed languages is enforcing encapsulation without static types. Schärli argues that the mechanism used should support this dynamic style of programming.

The solution of Schärli is based on encapsulation policies [Schärli 2004b] to define the encapsulation itself, and change how the message sends are interpreted to enforce the encapsulation (interpretation change are performed at Virtual Machine level).

Encapsulation policies. Schärli provided a working implementation of this per reference encapsulation model.

Encapsulation policies provide encapsulation policy objects that define the *access right* of each method.

- access right **c** means callable. This allows the method with a **c** to be called from an external point of view (external reference or super via inheritance).
- access right **o** means overridable: the method can be overridden. This means that all existing calls to this method are dynamically bound to the overriding method.
- access right **r** means re-implementable: if a subclass does not override but only reimplements a method, existing calls in the superclass continue to be statically bound to the old version of the method. That means a sends of the private message in the superclass do not invoke the re-implemented method in the subclass.

These three different access rights generate eight possibilities. In [Schärli 2004a] Schärli identifies only four specific sets of rights which make sense for Dynamically-Typed languages.

r "hidden", the method can be re-implemented, but not overridden and not called.

or "overridable", the method can be re-implemented or overridden but not called.

cr "callable", the method can be called but not overridden.

cor "Full access", no restriction.

The design is ensured by a specific interpretation that access rights are respected at execution time. Several encapsulation policy objects can be defined on the same object. The encapsulation policy objects can be shared among objects and classes. The encapsulation policy provides restriction on objects at the reference-level. Schärli identifies three of message sends:

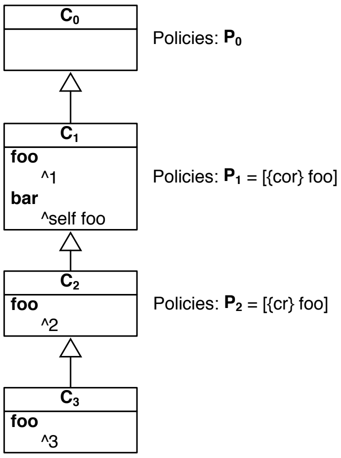


Figure 3.6: Example: Encapsulation Policies illustrated examples [Schärli 2004a]

object-send. Object-sends are messages which are sent to external references (not self). The lookup begins in the class of the object (as usual). But the message send is *valid* from the encapsulation policies point of view if and only if the reference have a **c** (callable access right) on this method in the current encapsulation policies.

super-send. Super-sends are messages which are sent via the super keyword, meaning beginning the lookup in superclass. Here the lookup begins in the static superclass where the method is defined (as usual). But the message is *valid* from the encapsulation policies point of view if and only the method have right **c** on this method in the current encapsulation policies.

self-send. Self-send messages are a special case. The lookup begins in the class of self except for one special case, the case where there is an encapsulation policies in superclass hierarchy that disallow **o**(overriding) of this method, in this specific context the lookup begin in the superclass of this policy.

For example in Figure 3.6, if a bar message is send to an instance of the class C_3, the bar method will execute self foo, the C_3 disallows the **o** right for the foo method. So the lookup begin in the class C_2 (the superclass of the class which disallows the **o** right). The messages found are valid if they perform only local call (all the method calls are yield by the class of self) or if the method are implemented in superclass if the current policy allow **c** right on it.

So this solution is usable in the context of dynamically-typed languages and offer a way to restrict behavior at reference-level. The limit is that we cannot control references in other way than just restricted existing behavior. We want to be able to change behavior and not only restrict it at reference-level. Here a read-only execution implemented using encapsulation policy for removing all accesses to methods that perform a state change will raise a Does Not Understand exception. Moreover the encapsulation policies approach do not manage state.

3.3 Alias control in dynamically-typed languages

Aliasing happens when one object is referenced by multiple other objects. In most object oriented languages this is a very common case. An every day example of a situation where this can lead to problems is the case of the copy of a collection. When a collection is copied using shallow copy instead deep copy, it will alias all the elements of the collection. If an object is changed via one collection, it will affect the other one, too.

3.3.1 Dynamic object ownership

Dynamic Object Ownership [Noble 1999,Gordon 2007] is a proposition to solve the aliasing problem in the context of dynamically-typed languages. Dynamic Object Ownership specifies two invariants: *representation encapsulation* and *external independence*.

Representation encapsulation. It requires to not directly access instance variables, instance variables should be accessed only by message passing through its public interface.

External independence. It means that an object should not be dependent on an external object. An object is external to an aggregate if it is not owned or encapsulated by this aggregate. In Figure 3.7, *ClassRoom*, *School* and *Street* are external to *House*. But *Room 1* and *Room 2* are not external to *House*.

In addition, of these two invariants Dynamic Object Ownership identifies different kinds of message sends, *pure messages* and *oneway messages*.

Pure messages. These are messages which do not access non-final fields, or send messages that would cause information about non-final fields to be returned to the pure send. Uses for pure messages include retrieving information about immutable objects, such as a character from an immutable string object or a co-ordinate from an immutable point object.

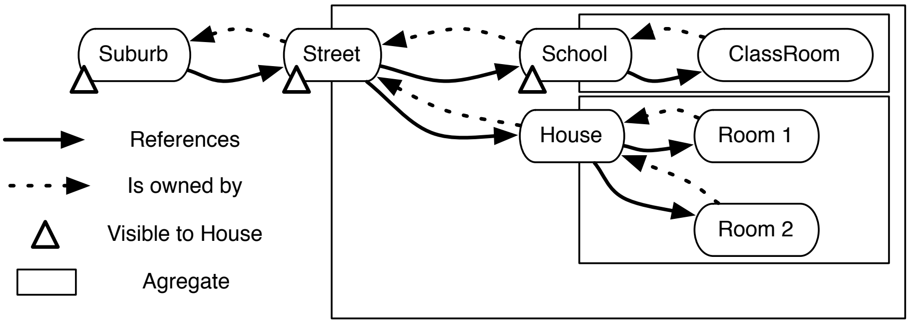


Figure 3.7: Example: Encapsulation for Dynamic Ownership [Gordon 2007]

Oneway messages. These are messages that cannot return a value or throw an exception. But outside this two constraints, *oneway messages* are normal messages.

Relationship of Sender and Receiver	**Message Type**		
	Normal (Unrestricted)	Externally Pure	Independent Oneway
Internal (from normal or oneway)	yes	yes	yes
Internal (from pure method)	no	yes	yes
External (visible but not owned)	no	yes	yes
Not visible	no	no	no

Figure 3.8: Message dispatch Dynamic Ownership [Gordon 2007]

Figure 3.8 shows the dispatch table depending of the relationship between sender and receiver and the kind of message send. Dynamic Ownership completely disallows message sends to non-visible objects. Only *pure messages* and *oneway messages* methods can be send from pure method execution and from external reference. Otherwise it allows every message send.

As an example, in Figure 3.7 *House* can send unrestricted messages to *Rooms 1* and *Room 2*. *House* can send Pure and Oneway messages to *School* and *Street*. But House cannot send any message to *Classroom*.

This solution doesn't need any programming pattern to be used from the developer point of view and is fully applicable to dynamically-typed languages. The restrictions offered by the ownership are global, so automatically enforced by design. The solution, is based only on an unique way to control references by restricting the visibility of objects and distinguishing only two cases to restrict behavior. In addition this solution does not manage state. In our case we want to control reference in more generic way, this solution is interesting but restrictive.

Gordon *et al.* had implemented as an extension of BeanShell[1] call ConstrainedJava. ConstrainedJava is a proof of concept, it is un-optimized. Gordon *et al.* classifies the messages sent by the benchmark in threes categories presented before, 77% of messages are sent to an object directly owned by the message sender (internal send unrestricted messages). In addition, 22% are external sends, restricted to be externally independent. And finally 1.2% internal send within an object. The benchmark itself presents two cases to measure the overhead of ConstrainedJava (i) using the encapsulation enforcement where we have 51,31% of slowdown comparing to normal interpreter (BeanShell 1.3.0) and (ii) without using the encapsulation enforcement (but with virtual machine modification) where we have 37,45% of slowdown comparing to normal interpreter. The performance is poor but the implementation was done as a proof of concept.

3.4 Context-oriented programming

Context-Oriented programming is the ability of a program to change depending of the execution context. A precise definition of context is:

> Any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. **Abowd and co**
> In *Towards a Better Understanding of Context and Context-Awareness* [Abowd 2000]

In Context-Oriented Programming the behavior of an object is modified depending on a context [Hirschfeld 2008]. ContextL [Costanza 2005] provides a notion of *layers*, which define context-dependent behavioral variations. Layers are dynamically enabled or disabled based on the current execution context. Scoping side-effects has been the focus of two recents works. Tanter proposed a more flexible scheme: contextual values [Tanter 2008] are scoped by a very general context function. Context-oriented programming is used in many domains such as autonomic systems [Kephart 2003], self-adaptive [McKinley 2004] and ubiquitous computing [Weiser 1993]. But in our case we are interested in how context-oriented programming can control references.

3.4.1 Contextual values

Contextual values is one attempt to control object's state depending on execution context. Tanter [Tanter 2008] provides two ways to control object's via references *explicit contextual values* and *implicit contextual value*.

Explicit contextual values. *Explicit contextual* values are a generalization of *thread-local* values (values that are specific to a thread), the idea is not to provide a field specific

[1] Pat Niemeyer. *BeanShell.* http://www.beanshell.org

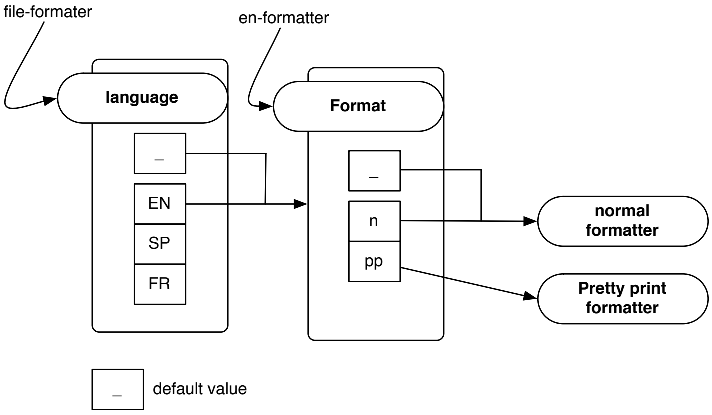


Figure 3.9: Example: Implicit Contextual values [Tanter 2008]

to a thread but to provide any *accessible information* as *thread-local* value. This solution works as soon as the isolation of each thread is perfect, but completely fails if the code did not use specific accessors (contexts are basically a container), or simply if state is shared so the mutation of the state will not be performed automatically. To solve this two issues Tanter provides *implicit contextual values*.

Implicit contextual values. *Implicit contextual* values are in fact an extension of the previous model. First of all, the *implicit contextual* model uses the same contextual values but provides an uniform access and assignment of variables. Register a variable as "Contextual" is still explicit, but accessing a variable is done in the same way that a normal variable. To illustrate, let us take the example in Figure 3.9 *file-formater* provides different kinds of formatter depending of the language used by the current thread. It is possible to chain the contextual values (Tanter call that nested contextual values), in Figure 3.9 *file-formating* is a *nested contextual value* be cause it depends of the **language** and the **format** selected in the thread. It is important to underline the resolution of the value is done when the values are called (otherwise it stays a symbolic link).

The implicit contextual values can be implemented based on thread extension and core object model (for uniform accessing). It is perfectly compatible to dynamically-typed languages and an implementation has been done in Scheme [Tanter 2008]. Contextual values is one way to controlling state via reference for dynamically-typed languages. The solution propagate itself in the same thread. If we want to isolate an execution or a specific process this solution is relevant, but we may want to isolate an object or graph regardless to the thread executing it. Moreover in more complex situation where a thread should have access to the same object from two different points of view because via two different shared objects.

3.5 Conclusion

We take a look to the three major families which can be applied in context of dynamically-typed languages. As result we can see on Figure 3.10 no one fully answers each constraint advanced in the introduction.

	needs answered		
Family's solutions	Behavior	State	Propagation
Capabilities [Miller 2006]	restrict	no	no
Encapsulation [Schärli 2004a]	restrict	no	yes
Ownership [Gordon 2007]	restrict	no	yes
Contextual value [Tanter 2008]	no	yes (thread)	yes (thread)

Figure 3.10: Summarize for each solution

3.6 Other solutions

3.6.1 Aspect-oriented programming

There are a lot of solutions on aspect-oriented programming, we focus only on the idea to dynamically on the fly change code based on control flow has been used for realizing *continuous weaving* [Hanenberg 2004]. Here join-point shadows are introduced in the code on the fly (and lazily) at runtime based on actual control flow. delMDSOC [Haupt 2007,Schippers 2008] is a delegation based machine model for AOP. Hidden proxies are introduced to model join points as *loci of late binding*. The granularity of these solutions are the thread. Depending of the current process, the code injected is different. Here, this solution is interesting, because it can be applied in dynamically-typed languages (that support aspect). But we argue the granularity is still to high, we want to maintain control of reference regardless the thread using it.

3.6.2 Joe-E

Joe-E is subset of Java focused on providing secures programs [Mettler 2010]. Joe-E is based on a verifier to check the code to detect some prohibited patterns and a *tamed* version of the standard libraries.

Code verification. Exceptions in Java lead to number of security exploits. In Joe-E, exceptions are controlled. Joe-E restricts the exception mechanism by prohibiting, *finally* usage, mutable throwable exceptions, prevent catching of Error exception and prohibiting the *finalize* use. All these checks are performed to warrant secure encapsulation. In addition Joe-E restricts some usages such as the use of native methods.

Taming the standard libraries. Joe-E considered that the standards libraries can have side effects or unsafe behavior. The idea is to provide a safe version of standard libraries (call *tamed*) and avoiding the used of unsafe (or uncheck) libraries. The *tamed* libraries usage is enforced statically by the *code verifier* presented before.

The language Joe-E offer a solution to have a secure programming language. The model is limited to be used in context of dynamically-typed languages, the code verification is enforced statically using static analysis and code rewriting (transparent for the developer), we cannot use static analysis in context of dynamically-typed languages.

3.6.3 Alias prevention

Alias prevention provides a way to control aliasing by construction, it disallows some design pattern depending of the context, using program analysis.

JavaSeal. JavaSeal is a Java-based agent kernel to support security constraints on mobile agent systems [Bryce 1999]. The JavaSeal kernel defined *Seals* that are used to encapsulate an agent. A *Seal* has its own object loader and isolates the objects from the system. In addition, threads (called *Strands*) are confined to a *Seal*. To communicate between *Seals*, objects can be serialized into a *Capsule* to be exported on another *Seal*. The medium for transmitting is the *Channel* that is a thrusted *Channel* is allowed to transmit *Capsule* between *Seal*. JavaSeal prohibits cross-referenced objects between agents. This solution is interesting because it provides a confinement of part of the application. But this solution takes place in mobile agent systems and require some adaptation to be applied to object-oriented languages.

Islands. Islands define a groups of objects as an island. The object in an island can access all the objects in the same island [Hogg 1991]. In addition each *island* can define a *bridge*, which is an object used to control and manage communication between islands. Island provides strong isolation for a group of objects, using syntactic mechanisms. Island ensures that no static references can exist across the boundary of an island. This solution can not be applied with reflection or in dynamically-typed languages because static analysis can not deal with the lack of type information at compile time.

Balloon. Balloon objects are objects that can be referenced only once [Almeida 1997]. In addition all reference accessed from the balloon cannot be referenced outside the balloon itself. Balloon Types enforces these restrictions by a compile-time full program analysis. This solution can not be applied with reflection or in dynamically-typed languages because the static analysis can not deal with the lack of type information at compile time.

Flexible Alias Protection. Flexible Alias Protection is a system for enforcing encapsulation [Noble 1998]. Flexible Alias Protection defines three modes (**rep, arg** and **free mode**) that can be used to annotate statement, which are then propagated through the program. **rep**(Representation mode) defines that the object annotated should not be returned to the rest of the system. **arg**(Argument mode): the annotated object cannot be used to access a mutable state. **free** is the default case and normal mode. This set of modes are enforced statically, using static analysis at compile time. Here again, the static analysis fails to manage reflection and cannot be applied in context of dynamically-typed languages.

Type universe. Type universe is a type system where all is defined in a component call universe [Müller 1999]. Universes are used to represent a hierarchical partitioning of the system. Object in an universe can not access object in another universe except if they are defined in its parent universe. There is a root universe and every objects is defined in a specific universe. Type checking prevents leaks of references, and prevents *rep exposure* (leak of a mutable reference out of it scope). This solution uses type checking to enforce the belonging of the object to the universe. Because of type checking at compile time this solution can not be applied in dynamically-typed languages.

3.6.4 Roles and views on objects

Applying different views on objects depending on a given context has been the concern of many papers [Carré 1990, Civello 1993, Smith 1996, Bardou 1996, Herrmann 2007, Warth 2008]. ObjectTeam [Herrmann 2007] supports role in a programming language. A Team is an object that defines the scope of roles (multiple roles collaborating together). *Roles* are fields and methods which can dynamically be bound to objects. A team defines how roles forward or delegate their roles to the team participants. ObjectTeam focuses on *role* introduction in a programming language. ObjectTeam proposes three semantics for views on object: sharing, forwarding as in prototype languages and dispatch without lookup. *Roles* by definition do not automatically propagate through an object graph. Smith and Ungar's Us [Smith 1996], a predecessor of today's Context-Oriented Programming [Costanza 2005], explored the idea that the state and behavior of an object should be a function of the perspective from which it is being accessed. Warth et al. [Alessandro Warth 2011] introduces worlds, a language construct that reifies the notion of program state and enables programmers to control the scope of side effects. An object (with the same identity) can have different states in different worlds. Worlds focus on providing control for state and do not provide a per reference semantic. Us layers are similar to Worlds without a commit operation. Split objects [Bardou 1996] define a model using delegation to create points of views on objects. Split objects consist of *pieces*, where a piece represents a property on the object. The pieces are organized in a delegation hierarchy. The self pseudo-variable is used to address the *"target"* in the split object. In addition, another pseudo-variable called *thisViewpoint* allows one to refer to current point of view.

This kind of solution depending of the role can managed state or behavior but it is not at reference-level. Moreover roles and views do not manage automatic propagation of the defined properties.

3.6.5 Proxies

Proxies are a way to control access on an object by creating an interface object. A proxy is a well known technique in any object-oriented language. Java provides support for proxies as a part of the java reflection library. The standard dynamic proxies can be defined only for interfaces. Uniform proxies for Java [Eugster 2006] is an approach to provide dynamic proxies for classes. All existing Java proxies are limited: proxies do not forward all method calls. As they inherit from Object, all methods implemented in Object cannot be controlled. In addition, there is no solution for forwarding Java operations that are not message sends. Stratified Proxies [Van Cutsem 2010] is a realization of proxies for JavaScript. They identify a subset of specific calls that are trapped by the virtual machine and automatically reify those when using proxies. They provide a meta-level API to use and manage proxy behavior. Stratified Proxies are actually close in spirit of our work. Stratified Proxies do not manage the propagation natively and have therefore to use different means when using proxies in the context of, for example, revocable references. The concept explored to deal with complex object graphs is that of a Membrane that wraps all objects passed through it [Miller 2006]. Ghost Proxies [Martinez Peck 2011] is a realization of proxies for Smalltalk. Ghost offer adaptive proxies, that allow selected interception of messages, uniformity (each object is managed in the same way), transparency and stratified. Ghost is powerful and adaptive but cannot be applied for security because it is uniform (it does not make the difference between the proxies from developer and user point of view).

3.6.6 Stratified reflection

Mirrors are a design principle for structuring the meta-level features of object-oriented languages [Bracha 2004]. The idea is to not have both reflective and non-reflective functionality in all objects, but instead separate reflective from base-level functionality. A Mirror is a simple meta-object: a second object that provides reflection on a normally non-reflective base-level object. This solution changes the behavior from an external object point of view, that offers a clear separation of concern, but does not solve any of our problems.

In this chapter we studied a set of related solutions, which do not answer the requirement expressed in Chapter 2. As providing a solution to control references is not an easy task, in the next chapter we will conduct an experiment. In this experiment, we will ensure a simple and well know behavior modification with propagation, the read-only behavior.

CHAPTER 4

First Experiment: Ensuring Read-Only at Reference-Level

Contents

Ensuring behavior control, state control and propagation at reference-level is difficult so in order we will conduct a really first step, *implementing a simple and well-know property (read-only) at reference-level*. We choose *read-only* behavior because we want to separate behavior and state control to understand deeply the concerns in place. We want to identify clearly the implementation needs to integrate generic properties at reference-level in already existing dynamically-typed languages.

4.1 The challenges of dynamic read-only execution

Read-only execution is appealing for a number of reasons, ranging from easier synchronization to lowering the number of potential bugs due to side effects, aliasing and encapsulation violation [Almeida 1997,Gordon 2007,Finifter 2008]. Mechanisms prohibiting variable mutation are now common in programming languages (*e.g.*, Java has the final keyword, C++ and C# have const). This prevents the reference and not the referee to be modified.

In dynamically-typed languages, few attempts have been made to provide immutable objects. VisualWorks Smalltalk[1] and Ruby support immutable objects using a per object flag that tells whether the fields of the object may be modified. Since immutability is a property embedded in the object itself, once created, immutable objects cannot be modified via any alias. In this section, we discuss the challenges of realizing read-only execution in dynamically-typed languages.

[1] http://www.cincomsmalltalk.com

Different views for multiple usages. Read-only execution is often needed in cases where objects taking part in the read-only execution are at the same time (*e.g.*, from another thread of execution) referenced normally with the ability to do side-effects. A clear illustration of this need is assertion execution: within the scope of an assert method invocation, objects should not be modified, else this could lead to subtle bugs depending on the evaluation of assertions.

What we want to stress here is not only the need for immutability but also the fact that such a property can be dynamic and that the same object can be simultaneously referenced read-only from one object, and write-enabled from another object.

The case of object creation. One interesting question appears when thinking about objects that are created locally and used in a method but never stored in an object. As far as the rest of the system is concerned, these objects are disconnected: changing them does not change the overall state of the system. It is natural to think that newly created objects are not read-only even when referenced from a read-only execution context. Storing a newly created object within a read-only view should raise an error.

Flexibility to support experiments. Having read-only object references is one solution of a larger problem space. Several possible reference semantics and variants have been proposed in the literature such as Lent, Shared, Unique or Read-only. In addition, read-only is just one element in the larger spectrum of capability-based security model [Miller 2006]. Therefore the current model should be flexible to be able to explore multiple different semantics. Just providing one fixed model for how the read-only property works is not enough (see Section 4.5). The model and its implementation should be flexible enough to be able to grow beyond pure read-only behavior.

Challenges for read-only execution in dynamically-typed languages. The key challenge is then:

> How do we provide flexible read-only execution on a per reference-basis in the context of complex object graph with shared state without static type analysis support?

Some work such as encapsulation policies [Schärli 2004b,Schärli 2004a] offer the possibility to restrict an object's interface on a per reference basis, but they do not address the problem of propagating the read-only property to aggregate references. Encapsulation policies are static from this point of view. ConstrainedJava offers dynamic ownership checking [Gordon 2007] but the focus is different in the sense that ConstrainedJava makes sure that object references do not leak while we are concerned about offering a read-only view.

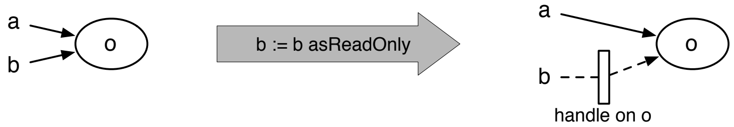


Figure 4.1: A reference to an object can be turned into a read-only reference.

4.2 Dynamic read-only references

Our approach to introduce read-only in a dynamically-typed language is based on dynamic read-only references. Dynamic read-only references can offer different behavior for immutability to *different* clients. Such references are based on the introduction of a special object, called handle, by which clients interact with the target object. A handle is a *per reference* transparent proxy, *i.e.*, identity is the same as the object they represent. The handle therefore forms a reification [Friedman 1984] of the concept of an object reference: a reference is now modeled by an object. Similar reifications have been realized in other contexts, one example is *Object Flow Analysis*, which models aliases as objects [Lienhard 2008]. Another example is *delMDSOC*, an execution model for Aspect Oriented languages [Haupt 2007].

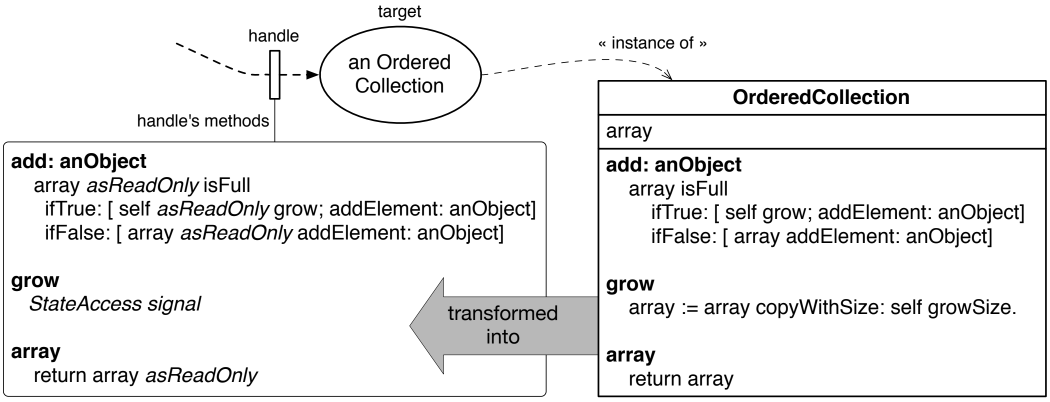
4.2.1 Handle: a transparent per reference proxy

Conceptually, a handle is an object that represents its target; it has the same identity as its target, but redefines its behavior to be read-only.

At the implementation level, handles are special objects. When a message is sent to a handle, the message is actually applied to the target object, but with the handle's behavior —*i.e.*, the receiver is the target object, but the method is *resolved in the handle*. In essence, a handle never executes messages on itself, rather it replaces the behavior of messages that are received by its target.

Handle creation example. Figure 4.1 illustrates the creation of a read-only reference. First, we get an object and we assign it into the variables a and b. Second, via the variable b, the object is asked to become read-only. This has as effect to create a handle.

Identity of handle and object. Contrary to traditional proxies [Gamma 1995] and similarly to Encapsulators [Pascoe 1986], a handle and its target have the same identity — the handle is transparent. Not having transparency could lead to subtle bugs because most of the code is not (and does not have to be) aware of the existence of handles: for example, without transparency, adding a target and its handle to a set would break the illusion that the handle is the same object as the target since both would be added to the set.



add: anObject
    array *asReadOnly* isFull
        ifTrue: [self *asReadOnly* grow; addElement: anObject]
        ifFalse: [array *asReadOnly* addElement: anObject]

grow
    StateAccess signal

array
    return array *asReadOnly*

add: anObject
    array isFull
        ifTrue: [self grow; addElement: anObject]
        ifFalse: [array addElement: anObject]

grow
    array := array copyWithSize: self growSize.

array
    return array

Figure 4.2: The handle holds the read-only version of the target's code, with instance variable accesses and affectations rewritten.

4.2.2 Enabling read-only behavior

As explained above, a handle can have a different behavior than its target for the same set of messages. Specifically, we redefine all the methods to offer the expected read-only behavior. To install a handle on a target, we rewrite the target's methods and install them into a *Shadow class* that the handle references:

1. Store accesses in globals and in instance variables signal an exception;

2. Read accesses to globals, instance variables and to the self pseudo-variable are dynamically wrapped in read-only references.

Figure 4.2 illustrates this behavioral transformation. The class OrderedCollection defines an instance variable array, and three methods: the method grow which modifies the state of the collection by modifying the contents of the instance variable array, the method add: which conditionally invokes other methods and the method array which returns a reference to this object. The Shadow class of the handle for this object contains three methods. The method grow which raises an exception. The method array which returns a read-only reference to the internal array. The method add: is transformed according to the transformation we described above: self/this, instance variables and globals are asked to be read-only and modification raise an error.

Objects referenced by temporary variables are not transformed into read-only references because they do not change the state of the object. Objects referenced by arguments may be read-only references, but only because of an earlier transformation.

Note that this transformation is recursive and dynamic. It happens at runtime and is driven by the control-flow. This recursive propagation is explained in the next section.

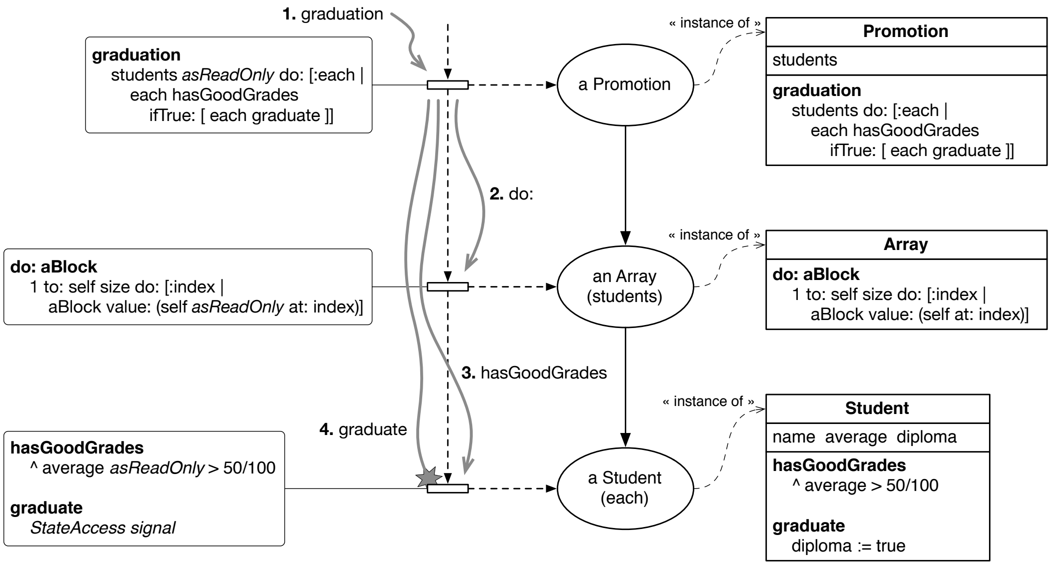


Figure 4.3: When we send the graduation message via a read-only reference to the Promotion instance, execution proceeds via read-only handles, until the graduate method attempts a side-effect.

4.2.3 Step by step propagation

The read-only property dynamically propagates to an object's state when it is accessed. Figure 4.3 illustrates how immutability is propagated to the state of a list of students. When the graduation: message is sent to a read-only Promotion instance, first the message do: is sent to the read-only students instance variable. This invokes the read-only version of the Array»do: method, and in turn the read-only version of Student»hasGoodGrades. Since hasGoodGrades does not modify its receiver (here the Student instance), the execution continues this way until it reaches a student with grades good enough to pass. At that point, we are still in the read-only version of the Promotion»graduation method, so we execute the read-only version of graduate, which throws an exception, since in its original version it modifies the student's diploma instance variable.

This example shows that immutability is propagated to the references of instance variables and global variables recursively and on demand, based on the execution flow of the application. Note that execution fails only when it reaches an assignment to an instance variable or a global variable.

Global variables and classes. Access to global variables is controlled the same way as access to instance variables: write access is forbidden (it raises an exception), any read is wrapped in a creation of a read-only handle. As classes are objects in Smalltalk, accesses to classes leads to the creation of a read-only handle for that class. Any change of the structure of the class, *e.g.*, adding or removing methods or changing the inheritance hierarchy is therefore forbidden.

Newly created objects. Even though referencing a class will result in a read-only handle, using the class to create a new Object will not result on this object being wrapped in a read-only handle. A new object has only one reference, therefore changing this object will not lead to a side effect: all other objects of the program remain unchanged. We can therefore just have new objects be created normally, allowing modification. Any try to store the newly created object will result in an error, making sure that the object never has any influence on the rest of the system.

Temporary variables. Temporary variables only live for the extend of the execution of one method. They are not stored in an object if not done explicitly. Therefore, we can safely keep temporary variables with read-write semantics: reading does not wrap the reference in a handle, writing is allowed.

Arguments. We do not wrap arguments or return values. If a value is used as an argument, it has to come from somewhere: it is either read from an instance variable, and therefore already read-only. Or it was created locally in a method and is not connected to any other object in the system. We can therefore hand over arguments without any special wrapping: either the reference passed is already read-only or it is read-write, it is handed over unchanged.

Block closures. For closures, there are two cases to distinguish: executing a block read-only and passing a block to a method of a read-only data-structure as an argument. For read-only execution, the objects representing closures need to support a read-only execution mode. For this, all instance and global variable reads are wrapped in a handler. In addition, we need to wrap all variable reads defined outside of the closure and the arguments.

4.3 Examples

We now present how DRO is applied to solve two critical problems: side effect-free assertions and read-only collections.

4.3.1 Example revisited: side effect free assertions

An assertion is a boolean expression that should hold at some point in a program. Assertions are usually used to define pre- and post-conditions: in a service contract, if the precondition is fulfilled, then the postcondition is guaranteed. Assertions are supported by a number of languages, including Eiffel, Java, Smalltalk, and C#. It is reasonable to expect a program to behave to the same bit when assertions are removed to gain performance.

However, none of the languages supporting assertion can enforce behavior preservation when removing assertions, because the assertion code could perform any method call or side effect.

Writing an assertion for *e.g.*, a ServerSocket object requires that the object is not changed during the execution of the assertion (which can be repeated multiple times), while it can be modified outside the contract. To illustrate our point, consider the following method definition extracted from the trait implementation of Pharo:

```
Behavior>>flattenDown: aTrait
    | selectors |
   self assert: [self hasTraitComposition and:
        [self traitComposition allTraits includes: aTrait]].
    selectors := (self traitComposition transformationOfTrait: aTrait)
    selectors.
    self basicLocalSelectors: self basicLocalSelectors , selectors.
    self removeFromComposition: aTrait.
```

The assertion contains five message sends (hasTraitComposition, and:, traitComposition, allTraits, and includes:). If one of these message leads to a side effect, then the assertion will be not easily removable and may lead to subtle bugs.

Using dynamic read-only object references, we define a safe alternative for the assertion mechanism as follows:

```
Object>>safeAssert: aBlock
   self assert: aBlock asReadOnly
```

Like assert:, the method safeAssert: is defined in the class Object, the common superclass of all classes in Pharo. All classes will therefore be able to call this method. The parameter aBlock is a closure that will be evaluated by the original assert: with aBlock value. However, before delegating to the original implementation, we make the block read-only by rewriting all accesses the block does to variables defined outside itself. This includes all instance variable accesses, accesses to temporary variables from an outer block or method, as well as the special variables self and thisContext. Temporary variables are not wrapped in read-only handlers. The new assertion in our example thus becomes:

```
    self safeAssert: [self hasTraitComposition and:
        [self traitComposition allTraits includes: aTrait]].
```

The preservation of object identity is important in our case: an object has the same identity as its read-only counterpart. When defining assertions, the self reference within the block provided to safeAssert: is the same identity than outside the block.

With DRO, the contract code can be executed guaranteeing that no side-effect occurs. Therefore, no bugs concerning unwanted state changes can happen.

4.3.2 Read-only data structures

In general, read-only references provide the programmer with the possibility to hand out safe references to internal data-structures. As soon as the reference that is handed to a client as a read-only handle, it is guaranteed that no modification may happen through this reference. For example, we can use read-only structures to make sure that the programmer gets notified if (s)he attempts to modify a collection while iterating on it. This is indeed a subtle way to shot oneself in the foot; bugs introduced this way are difficult to track because they may rely on the order and identity of the elements. To be safe, the idiomatic way is to iterate on a copy of the collection and modify the original.

Java provides support for requesting an *unmodifiable collection* for any collection. This is a wrapper object that protects the collection from modification, while allowing the programmer to access or enumerate the content like a standard collection object. In Smalltalk, it is common to hand out a copy of a collection if the collection itself is used internally.

With dynamic read-only references, one solution is to offer a safeDo: method that propagates read-only status to the collection. Attempts to change the collection will then lead to an error.

```
Collection>>fullSafeDo: aBlock
   ^ self asReadOnly do: aBlock
```

Note here that the read-only status gets propagated to the block arguments, as we can easily see in the implementation of do:. The parameter passed to the block is read via the instance variable array, which is automatically wrapped in a read-only handler. This read-only reference provides read-only versions of all methods of class Array, including at:, resulting in a read-only reference passed to the block.

```
OrderedCollection>>do: aBlock
   "Override the superclass for performance reasons."
   | index |
   index := firstIndex.
   [ index <= lastIndex]
      whileTrue:
         [ aBlock value: (array at: index).
          index := index + 1 ]
```

All other variables referenced in the block, *i.e.*, globals, temporary variables of the enclosing methods or instance variables of the class are not read-only. The execution of the block itself is not done in a read-only context, just accesses to the collection (and all objects contained) are read-only.

Our solution with DRO provides a way to protect accesses to the entire object graph, starting at the read-only reference to the collection. In Section 4.5, we discuss that a better way of controlling the propagation is needed. We now formalize the model described in Section 4.2 by providing an operational semantics for Dynamic Read-Only execution using the formalism of ClassicJava [Flatt 1999]. The goal of this formalization is to provide the

$$\begin{array}{rcl}
e & = & \ldots \mid \textbf{readonly}\ e \\
E & = & \ldots \mid \textbf{readonly}\ E \mid \lfloor E \rfloor \\
v, o & = & \ldots \mid \lfloor oid \rfloor
\end{array}$$

$$\begin{array}{rcll}
P & \vdash & \langle E[\textbf{readonly}\ o], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor], \mathcal{S}\rangle & [\textit{read-only}] \\
P & \vdash & \langle E[\lfloor o \rfloor .f], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor v \rfloor], \mathcal{S}\rangle & [\textit{get}] \\
 & & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = v & \\
P & \vdash & \langle E[\lfloor o \rfloor .f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[\textbf{error}], \mathcal{S}]\rangle & [\textit{set}] \\
P & \vdash & \langle E[\lfloor o \rfloor .m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor\ [\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle & [\textit{send}] \\
 & & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \langle c, m, x^*, e\rangle \in^*_P c' & \\
P & \vdash & \langle E[\textbf{super}\langle \lfloor o \rfloor, c\rangle .m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[\lfloor o \rfloor\ [\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle & [\textit{super}] \\
 & & \text{where } c \prec_P c' \text{ and } \langle c', m, x^*, e\rangle \in^*_P c'' \text{ and } c' \leq_P c'' &
\end{array}$$

Figure 4.4: Extensions made on SmalltalkLite resulting in Smalltalk/R

necessary technical description when implementing DRO in one's favorite language. It should be noted that this formalism models the read-only behavior. To keep it simple, it does not follow the implementation. This means that we do not model references. The time when the read-only behavior is propagated therefore is different: it happens at the time of message send, not handle creation. We decided to provide this simplified formalism as a first step, we plan to extend it as future work when we go beyond pure read-only behavior.

For this purpose we extend SmalltalkLite whose description is given in appendix. SmalltalkLite has been presented in our previous work [Bergel 2008], it should therefore not be considered as a contribution. SmalltalkLite is a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and **super** sends. It is similar to ClassicJava, but removes interfaces and static types, and fields are private, so only local or inherited fields may be accessed. SmalltalkLite is generic enough to be considered as a formal foundation targeting languages other than Smalltalk (e.g., Ruby).

We provide the necessary extension of SmalltalkLite to capture the semantics of Dynamic Read-Only execution. We call Smalltalk/R the resulting extended language (Figure 4.4).

First, we augment the set of expressions and evaluating contexts with a new keyword, readonly. This keyword takes an expression as parameter and, at execution time, evaluates the expression and makes the result as read-only. Read-onlyness is expressed with $\lfloor \ldots \rfloor$. A value enclosed into these half square brackets cannot be mutated. A read-only value is written $\lfloor v \rfloor$ and can only result from evaluating a readonly expression. The readonly keyword belongs to the surface syntax, whereas $\lfloor \ldots \rfloor$ designates a read-only reference. The expression readonly e evaluates to a $\lfloor o \rfloor$ reference.

The read-only property is propagated through instance variable and global access such as class references (Section 4.2.2). In SMALLTALKLITE, the only global accesses permitted are class references in new expressions.

A new rule for the readonly keyword has to be added. The expression readonly o is simply reduced into $\lfloor o \rfloor$.

Subsequently, reduction rules must be adjusted to take the immutability property into account: field assignment leads to an error, while field access propagates the read-only behavior to the accessed value; calling a method propagates the read-only object reference into the method body; the read-only object reference is also propagated with super calls.

Message lookup is achieved through the notation $\langle c, m, x^*, e\rangle \in_P^* c'$: we look for a method called m, with the arguments x^*, and a method body e. The lookup begins in the class c and c' is the first-class among superclasses of c that defines this method. $\langle m, x^*, e\rangle$ is a method definition, $\langle c, m, x^*, e\rangle \in c'$ is a different notation that says $\langle m, x^*, e\rangle$ is looked up starting in c.

No special treatment is needed for super calls. If the receiver is readonly (as in **super**$\langle \lfloor o \rfloor , c\rangle$), then the method lookup starts in the super class of c, that we call c'. Following the notation of CLASSICJAVA, direct subclass is designed using $c \prec_P c'$. The method to lookup is $\langle m, x^*, e\rangle$ and it is searched starting in c'. It is found in c'', using the operator $\in_P^*$. Naturally, we have the relation $c' \leq_P c''$. The formulation illustrates the simplicity of readonly feature of the DRO model.

4.4 Implementation

We implemented DRO in Pharo by (1) extending the virtual machine and (2) using a bytecode rewriting engine [Denker 2006].

VM changes. The Squeak/Pharo Virtual machine was modified to support transparent proxies. Our implementation is more general than required for strict read-only execution. It offers a per reference handle as presented in Section 3, while for DRO we only need a per class method dictionary containing the rewritten methods. We did that because we see dynamic read-only references as just one case of a more general scheme for security alternatives in the realm of object capability model [Miller 2006].

Virtual machine modifications were required because in Smalltalk, reflection would allow one to easily change handles. Thus the user of a handle could corrupt the handle integrity and send forbidden messages. By implementing handles at the VM level we ensure that handles are tamperproof. In addition, handles have a different method lookup than normal objects: when a handle receives a message, the VM looks the method up in the handle's shadow class, but applies it on its target object. Also, the current implementation, never makes read-only integers, booleans, or nil objects, since those are naturally immutable. We changed the virtual machine to treat a handle and the object pointed to as identical. Technically this is realized by a special check in the byte-code implementing identity.

Runtime infrastructure. In addition to the VM changes, Figure 4.5 presents the principles of our implementation at the runtime level. When an object should offer a read-only behavior, a handle is created. The creation of the handle leads to the definition of a shadow class and its inheritance chain - in the style of Encapsulators [Pascoe 1986, Ducasse 1999] A shadow class is an anonymous class which in the context of DRO will be used by the modified handle method lookup. In such shadow classes, methods are rewritten to implement the semantics described before.

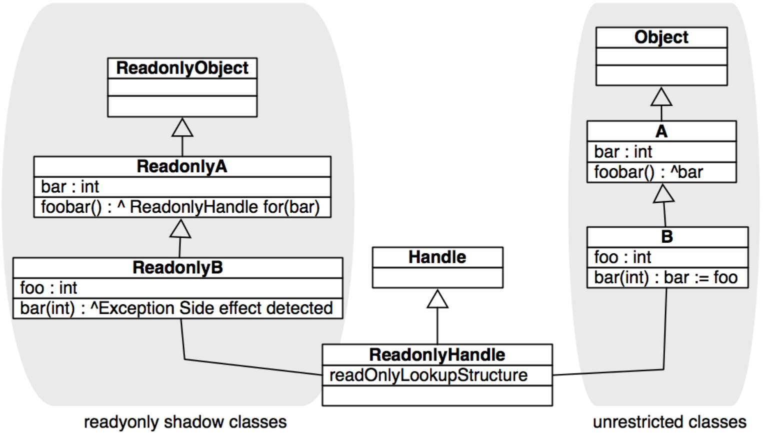


Figure 4.5: Shadow classes as an implementation of DRO.

We use the byte-code transformation framework ByteSurgeon [Denker 2006] to dynamically rewrite methods and install them in the shadow classes.

Shortcomings of the current implementation. In the current implementation we do not support primitive virtual machine calls and weak references (*i.e.*, referenced ignored by the garbage collector). In addition the solution is tailored towards read-only and does not offer a generic framework to implement different security properties.

Benchmarks. We present preliminary benchmarks to asses the cost of implementing DRO on a real system [2]. We compare the performance of a DRO VM, a virtual machine where the DRO is enabled, and a standard VM, for both the case of using standard references and using read-only references. We take a method returning a simple literal and execute it one million times using the MessageTally profiler to measure the slowdown of executing a message send. In addition, we measure the slowdown of object creation by executing one hundred thousand iterations. As we can see in the table, message sends are slowed down by 8%. In the case of read-only references, the overhead is maximal 16%.

[2] The benchmarks were done on a MacBook Pro Core2Duo 2.4Ghz.

	Standard VM	DRO VM	Slowdown	DRO VM	Slowdown
				(to handle)	
send	1109.3 ms	1200.5 ms	8.22 %	1290.1 ms	16.30 %
creating instance	595.9 ms	780.4 ms	30.96 %	NA	NA

Very time-consuming is the creation of a handle in the case when the shadow class needs to be created. In this case, all methods of the class of the object and all superclasses are copied and modified. This naturally takes time but needs to be done only once for each class.

4.5 Discussion

Scoping propagation. In our approach, we cannot control the scope of immutability propagation: for the whole subgraph of objects reached during execution we construct the read-only overlay. Therefore a collection accessed as read-only cannot have elements with a different access behavior: in the following example, accessing the elements of the collection will propagate the immutability to the elements:

```
    (aCol asReadOnly at: 1) changeStateOfElement
```

Our approach supports argument exposure [Boyland 2001] in the sense that when the collection is accessed as read-only, its elements are read-only too. A related problem is the question of how to allow harmless side-effects. For example, often programs need to cache calculated values. With a strict read-only execution model, this is not possible.

The problem is that it is not clear how a scoping mechanism for limiting read-only propagation should look like. Care needs to be taken to not break the read-only model completely. This is part of our future work: we imagine both a static scoping mechanism based on program annotations and one based on dynamically scoped variables.

Implementation alternatives. The implementation approach chosen uses rewriting bytecodes to change behavior of methods combined with hidden (per reference) object proxies that are realized by changing the virtual machine. Alternatively, we could have generalized the idea of a per-object immutability bit and encoded immutability in the object pointer. Dynamic languages already encode small integers directly in the pointer using a tag bit. Extending tagging to multiple bits has been done and is especially feasible in systems with 64bit pointers.

We decided against this implementation strategy as it would constrain the model to just be about immutability, with no way of controlling the propagation. As we discussed before, scoping the propagation of read-only behavior is essential. Besides propagation, with having handles be special objects (hidden by the virtual machine), but in the end fairly normal nevertheless, we can start to experiment with other mechanism besides pure read-only behavior. All these experiments would not be possible in a system that encodes behavioral modification in a single bit.

An interesting question is how much of the implementation of the transparent handles can be realized without virtual machine changes. The reason for the VM change is the need to hide the proxy completely: the proxy is indistinguishable from the object and there is no way (not even using reflection) to reference the object directly. Purely reflective approaches (*i.e.*, rewriting bytecode for identity) are always visible to introspection and therefore not easy to realize in a completely transparent way.

4.6 Conclusion

In this chapter we conducted a first step based on read-only where we presented a solution for read-only and then a model with formalism and its own implementation.

In the following chapter, we will use what we learned from this first step to ensure generic purpose properties based on behavior control. In addition we will provide a detailed implementation with benchmark and we will rewrite the formal model to propose a *clean* and usable formal model not based on rewriting rules.

CHAPTER 5
Ensuring Behavioral Properties at Reference-Level

Contents

In the previous chapter, we identified the implementation requirements to integrate a well-know property (*read-only*) at the reference level. This implementation is based on a bytecode rewriting engine in combination with a virtual machine change.

In this chapter we will generalize the approach and focus to ensuring generic behavior control and propagation. We will use the Revocable property presented in Chapter 2 and we will revisite the Read-only property implemented in previous chapter.

Our approach is structured as a framework: first, the programmer has to specify how a class holding a property (for read-only, raising an error on field write, for revocable, blocking execution when revoked) is derived from a class to which the property has to be applied. Second the handle construct mechanism ensures the propagation and the absence of leakage of that property at execution time.

The contributions of this chapter are:

1. Handles: a generic model for first-class references that propagate their behavior and their formal description,

2. the application of this framework to implement read-only execution and revocable references, and

3. a precise description of the implementation.

5.1 The case for handles

This section presents one example that stresses the specific requirements we need to support to control object graphs in dynamically-typed languages.

5.1.1 One motivating example

Revocable References. Miller *et al.* show that capabilities can be used to support confinement and revocable references [Miller 2003b]. Figure 5.1 shows an example with three objects: Alice can give Bob a reference to Doc. But Alice should be able to revoke it later *i.e.*, Bob cannot access it anymore even if he holds a reference to it. The conceptual solution proposed by Miller *et al.* is to create a revoking facet (R) and only pass such facet to Bob. Such a facet can be seen as an object with a restricted interface or a first-class reference. Note that in this original proposal revocable references are not propagated automatically. The facet needs to be carefully thought to not leak references and only return facets instead.

As we have seen the example presented Chapter 2, Alice has to make sure to *wrap all objects* discoverable from the reference handed to Bob. Idioms and special safety patterns should be followed by the programmer to make sure that there is no reference leaked by accident. Indeed, imagine that Doc holds a reference to a SubDoc which also has a back pointer to Doc. While Bob cannot access Doc once its reference to Doc is revoked, if Bob gets a reference to SubDoc and this reference is not a revocable one then Bob broke the system and can access Doc even if it should not be able to do so.

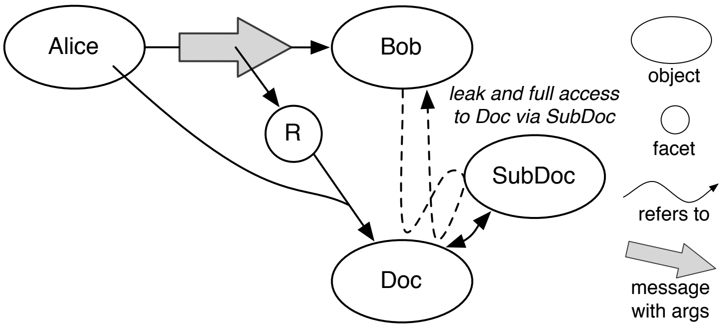


Figure 5.1: Revocable references: Alice can decide Bob should not be able to access Doc. References to SubDoc should also be revokable else Bob can access Doc even after revocation.

5.1.2 Analyzed required properties

In Chapter 2, we presented the requirements for providing features at references level.

Per reference behavior shadowing. The previous examples show that we need to control behaviors at reference-level. The same objects can be referenced from multiple perspectives. A read-only behavior should be readonly only from a specific point of view and not on object itself.

Propagating. An important aspect is how specific properties propagate during program execution to the object graph. In revocable reference examples, the properties of make a reference *revocable* do not making sense if the revocable behaviors do not apply to a subgraph. The same remark can be done for the read-only property and the Software Transactional Memory state protection.

- When we want to ensure that in a precondition, all objects are read-only (*i.e.*, the state of the objects do not change), this property should be propagated to all the objects involved in the precondition execution.
- Similarly, when a *user* is granted a revocable reference, the propagation of such behavior to the object graph participating in a control flow is important: when the reference is revoked all the references to this object graph made during the execution should be revoked as well.

Such a propagation should not be limited to a thread but should nevertheless follow execution.

Transparent. There should be no special case for the object manipulated. When Bob accesses Doc via a revocable reference, it should be able to perform any actions on it and should not be aware that it is using a revocable reference.

5.2 The generic model of behavioral handles

Handles are first-class references that propagate behavioral changes dynamically to the object subgraph during program execution. We present it formally in Section 5.3.

Vocabulary. We call *target* the object on which handles are created. When adequate, we distinguish between the *creator* of a handle and one of its *users* (*i.e.*, programmers that access an object via a handle obliviously). A creator is able to create handles and control them if necessary. A *user* simply uses a handle. When the user has only access to a handle, he cannot access the original object.

A two step approach. Our approach is structured in two parts: First the language designer has to define in his own way how the property that he wants to support is implemented. He does this by specifying how a *class* is transformed into a *shadow class*. The result of such a transformation is a *class* that holds the property applied to a target class. For example, to implement read-only, all the write accesses in a given class should raise exception (Figure 5.2). Second, once handles are created, they ensure at runtime that the property is propagated dynamically reference by reference through the object subgraph. They ensure that the target object cannot leak and that the property is preserved. In this article, we focus on the second step.

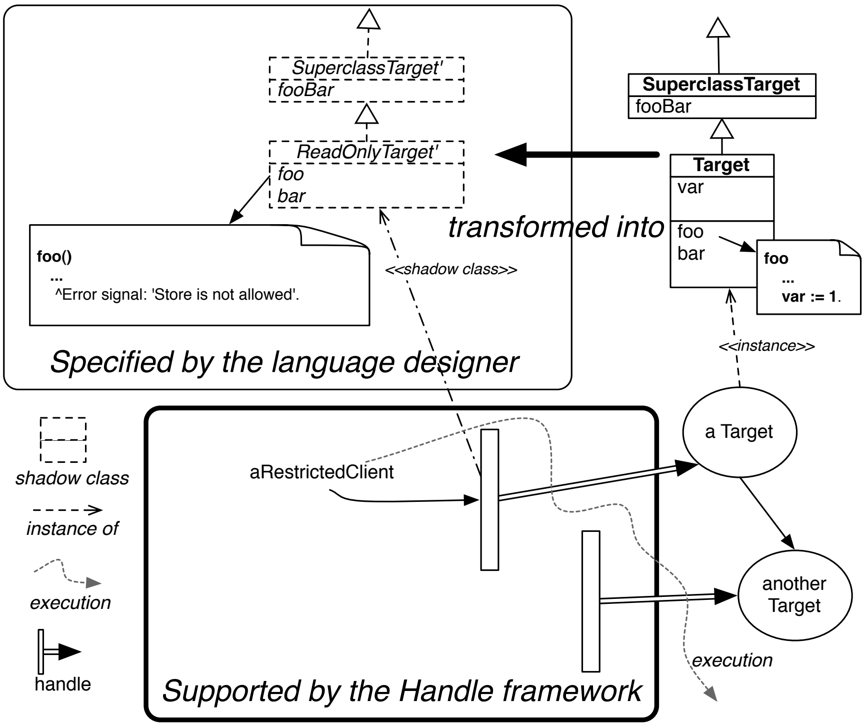


Figure 5.2: Handle supports the management of reference properties and their propagation at runtime. The language designer has to provide the transformation that defines the semantics of his new construct: here read-only.

5.2.1 Handle model

Figure 5.3 describes the underlying principle of Handles: (1) a handle is a transparent reference to a target object, (2) a handle can define different behavior than the target object. When the message foo is sent via the reference restrictedClient1 the handle executes its *foo'* method *using the identity and the state of the target object*. In addition, restrictedClient1 has only access to the transformed target behavior which is stored in a *shadow class*. There is no lookup mechanism between the shadow class and the target class. If a method is not defined in the handle, but in the target, it will *not* be accessible to the handle client. Multiple handles can have the same target object. A target can be accessible via a normal reference fullAccessClient1 and controlled ones such as restrictedClient1. It is the responsibility of the infrastructure built on top of handles to ensure the adequate use of properties and references. In its current version, the Handle framework does not keep the target class and its shadow synchronized automatically.

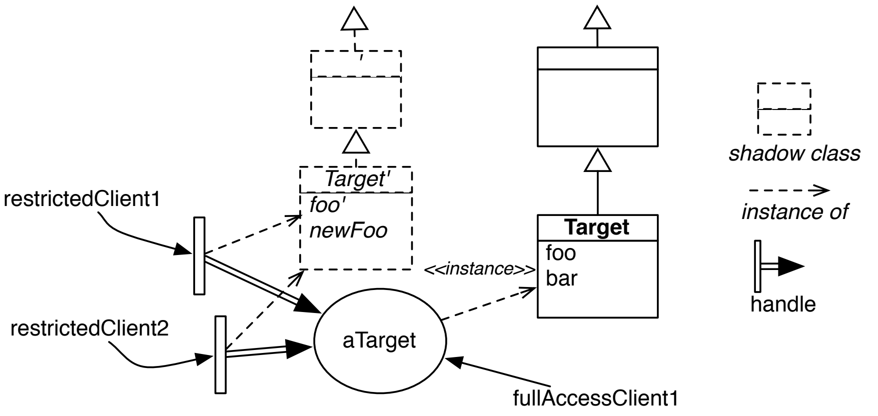


Figure 5.3: Handle Principle: a handle uses target state and identity and defines a specific behavior (potentially adapted from target class). Several handles can co-exist on the same target.

State access through handles. When a handle method accesses state, it accesses the state of the target object. Thus changing state from a handle reference is not local to the handle (the handle does not shadow the state of the target). Instead, if the specific handle behavior changes state it is the state of the target object that will change. Handles as any other objects can be stored in instance variables.

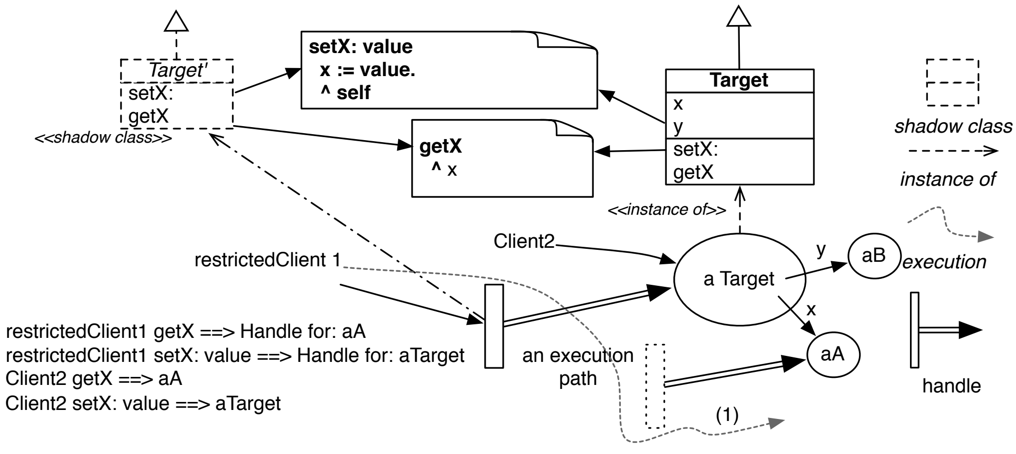


Figure 5.4: Handle Propagation Principle. All accesses via a handle to target object state are wrapped with a handle and propagated at runtime.

Handle Propagation through Execution. Figure 5.4 presents handle propagation. From a handle client point of view, *any instance variable read creates a handle on the accessed object*. In particular, sending a message that returns an instance variable of the target object, returns a handle on this object. In Figure 5.4 restrictedClient1 getX[1] returns a handle on the object aA. This propagation is recursive and follows the application execution (1 in Figure 5.4). From *a handle*, the receiver of a message is the handle via which the object was accessed. restrictedClient1 setX: (Object new) stores a new object (not a handle on this new object) in the target object and returns the handle used by restrictedClient1.

A handle performs a per reference modification of its target behavior. restrictedClient1 is interacting via a handle with aTarget. The handle ensures its semantics and propagates it from object to object in the subgraph of aTarget. Client2 is using a normal reference and the execution gradually interacts with objects in the object graph. The semantics is formally described in Section 5.3.

The case of self. Since sending a message to a handle leads to the application of a handle method to the target object, this raises the question of self/this. In particular, a handle method returning its receiver could leak the target object and this is clearly not what we want. In our approach, self/this represents the receiver of the message.

- When accessed from a non-handle, self/this in a (target) method represents the target object as in traditional object-oriented languages.
- When accessed from a handle, self/this refers to the handle.

5.2.2 Handle creation and the metahandle

Handle creation. To ensure that once a handle is created, there is no possibility for the programmer to access the target directly, we divide handle lifetime in two distinct periods:

- *Initialization.* A handle is initialized with the configuration options managing its behavior. Immediately after its initialization the system activates it.
- *Handle activation.* Once a handle is activated, it represents a *view* on the target object. It is impossible to directly send messages to the handle. Such messages are automatically managed as messages sent to the target and follow the behavior described earlier. This behavior is implemented at virtual machine level and cannot be reverted.

Metahandle: controlling a handle. Handles face an important tension: on the one hand, handles should forward messages they receive to their target. They should transparently

[1] While the implementation is done in Smalltalk, to ease reading the examples are writing using a Java syntax.

represent other objects. On the other hand we want to be able to control their behavior. For example, to implement revocable references we need to be able to mark a graph of handles. To solve this tension, the model offers metahandles.

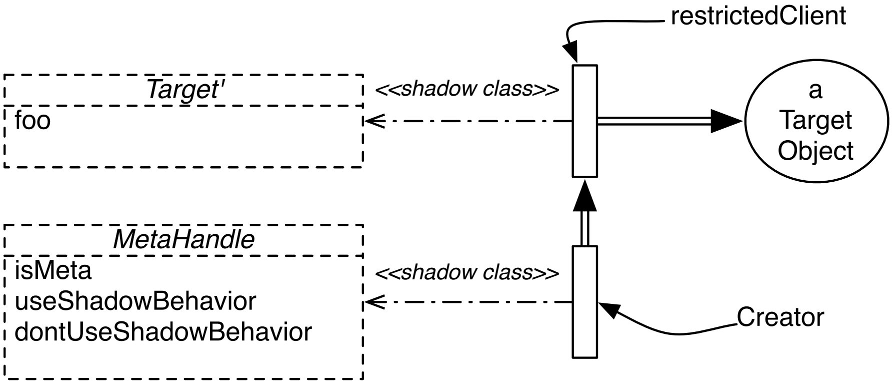


Figure 5.5: Metahandle Principle: a metahandle is a handle whose target is a handle.

A metahandle is a handle whose target is another handle (see Figure 5.5). Since an activated handle applies its method to its target, sending a message to a metahandle can modify a handle. That way handles can be configured, for example, to use either the shadow behavior (useShadowBehavior) or the target behavior (dontUseShadowBehavior). An important point is that a metahandle can only be created on inactive handles, as soon as activated a handle applies the modified target methods to its target.

A handle creator can keep a reference to a metahandle to later configure or change handle behavior. When the handle creator does not keep a reference to the metahandle, there is no way to change the behavior of a handle. In addition, handles can specify if they return their metahandle or not, by default they don't. Therefore as soon as the handle creator does not give away a reference to a metahandle, there is no way to interact with the handle (the behavior is the one described earlier: a message sent to a handle looks for method in the handle shadow class and applies it to the handle's target object, not the handle itself).

Primitives. One of the implementation issue is the primitives how manage primitives procedure call via an handle. We choose as implementation choose to create a separate table of primitives (copy of the default primitive table), and create for each primitive (more than three hundreds primitives procedure) a handle version of the primitive, it is a painstaking job but from implementation point of view it is required to have a viable Virtual Machine. And add a really cool properties to automatically manage reflection a usual process.

5.3 HANDLELITE: handle operational semantics

In this section we present the operational semantic in the current model of handles. Then we will explore a simple code example to see how it is translated to the formal model.

Property representation. Handles are motivated by approaches coming from the safety and security domain such as read-only execution and revocable references. Our model needs to be able to express such properties at the reference-level.

A property is a mechanism created by the language developer which defines how the control should be performed at the reference-level. This includes the change of behavior, the configuration of a handle and how the handle should propagated. The developer creates a mechanism for providing a class c' holding the property p. If we take the example of the read-only property, it may recompile some methods to raise errors on field assignment. We add a new syntax for defining property application: p(c). It takes as argument the class c which we need to control by property p, see Figure 5.6.

$$p(c) \quad = \quad c'$$

Where *c*' respects the properties *p*.
And *c* is a class.

Figure 5.6: handle: property translation

We only model properties and their propagation and not their actual implementation: Remember that the language designer has to provide a way to map a class and a property to a class. The property will be enforced by handles at reference-level.

Handle representation. We add a new construct to SMALLTALKLITE for defining handle creation: handle(o, p). It takes as arguments the target object o and a property p. We use h_p^o as a compact form for handle(o, p) and we add it to the redex of SMALLTALKLITE (see Figure 5.7).

$$\varepsilon \quad = \quad [...] \mid h_p^o$$

Figure 5.7: Handle: new redex for HANDLELITE

We have now the infrastructure for modeling manipulating message passing and state access at reference-level. We present how to enforce behavior shadowing, transparency and propagation.

5.3.1 Per reference behavior shadowing

To enforce behavior shadowing, a handle needs to keep a behavior. Adding handle to redex allows one to send message to a handle $h_p^o.m(\varepsilon^*)$, and to pass a handle as a parameter of a method send. In addition we add the possibility to send a message to a superclass to a handle (ie. **super**$\langle h_p^o, c\rangle.m(\varepsilon^*)$). Moreover, we add two rules to the reductions rules of SmalltalkLite to change how message and super send are managed (Figure 5.8) when performed on a handle. We create a [handled send] reduction and [handled super send] reduction for handle that we explain now:

[handled send] represents how message sends are managed (lookup and evaluation) on a handle. The rule [handled send] defines the expression $\langle E[h_p^o.m(v^*)], \mathcal{S}\rangle$ (see Figure 5.8), which evaluates the method body $[\![e[v^*/x^*]]\!]_{c''}$ found by searching in class c'' beginning at class c', where c' is class holding the property p (provided by $p(c) = c'$ where c is the class of the object o). This means, when h_p^o receives a message($h_p^o.m(\varepsilon^*)$), the lookup begins in class c', where c' is provided by the handle property p. Note that self is bound to h_p^o.

[handled super send] represents how super message sends are managed when performed on a handle. The rule [handled super send] defines the expression $\langle E[$**super**$\langle h_p^o, c\rangle.m(v^*)], \mathcal{S}\rangle$ (see Figure 5.8), which evaluates the method body $[\![e[v^*/x^*]]\!]_{c''}$ found by looking up in class c'' beginning at class c', where c' is the superclass of the class obtain using the property p of the handle h_p^o. This means, when h_p^o receive a message(**super**$\langle h_p^o, c\rangle.m(\varepsilon^*)$), the lookup begins in class c'', where c'' is the superclass of the c' is the class containing the method using *super*. To resume the *super send* mechanism is applied to handle within binding the receiver value to the handle. By construction if we are in a case of [handled super send] that mean we are already in a shadow behavior execution and the static binding is correct.

$$
\begin{array}{lll}
P \vdash & \langle E[h_p^o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[h_p^o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle & [\textit{handled send}] \\
& \text{Where } \langle c', m, x^*, e\rangle \in_P^* c'' & \\
& \text{And } c' \; is\; a\; class\; supporting\; the\; property\; p \;\; (via\; p(c) = c') & \\
& \text{And } c \; is\; the\; class\; of\; the\; object\; o & \\
& & \\
P \vdash & \langle E[\textbf{super}\langle h_p^o, c'\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[h_p^o[\![e[v^*/x^*]]\!]_{c'''}], \mathcal{S}\rangle & [\textit{handled super send}] \\
& \text{Where } c' \prec_P c'' & \\
& \text{And } \langle c'', m, x^*, e\rangle \in_P^* c''' & \\
& \text{And } c'' \leq_P c''' & \\
& \text{And } c' \; is\; the\; class\; containing\; the\; method\; using\; \textbf{super} & \\
& \text{And } c' \; is\; a\; class\; supporting\; the\; property\; p \;\; (via\; p(c) = c') &
\end{array}
$$

Figure 5.8: Behavior related reductions for HandleLite

Using the rules [handled send] and [handled super send] we enforce shadow behavior at reference-level because the behavior is changed when the messages are received by the handle (using the property p). In addition, when a message is received by a handle, *self* is bound to the handle.

5.3.2 Transparent proxies

Handles are transparent proxies, so when the identity of a Handle is requested, it should answer the identity of the target object. In SMALLTALKLITE, identity is a value embedded in the object. This can raise an issue for Handles, as reading the identity value is fixed:

$$o \quad = \quad [...] \mid \mathit{oid}$$

In our model, we require to see the identity as a function, so when we request the identity of object, we return the identity of the object:

$$\mathit{oid}(o) \quad = \quad \mathit{oid}$$

But when identity is requested from a handle, the identity of the target object is returned:

$$\mathit{oid}(h_p^o) \quad = \quad \mathit{oid}(o) \quad = \quad \mathit{oid}$$

5.3.3 Property propagation

Handles propagate the properties that its ensures. Adding Handle to redex allows one to evaluate the expression $h_p^o.f$ and $h_p^o.f{=}\varepsilon$ on a handle and write a handle into a field $\varepsilon.f{=}\,h_p^o$. Handles are transparent proxies and require to manage propagation. This implies to manage the state accesses in a different way when they are performed from a handle. So as we see in Figure 5.9, we add the following two reductions:

[handled get] represents how fields are read from a handle. The [handled get] reduction has two steps. First fetch the state of the target object. And as second step, [handled get] propagates the properties p. Thus instead of reducing $\langle E[h_p^o.f], \mathcal{S}\rangle$ to $\langle E[v], \mathcal{S}\rangle$, we wrap the return value $\langle E[h_p^{\prime v}], \mathcal{S}\rangle$ into a new handle $h_p^{\prime v}$ respecting the same properties p.

[handled set] represents how fields are written from a handle. The [handled set] reduction shows how the state is written in the target object. This rule shows that handles do not keep their own state (the state is stored in target object).

Using [handled get] and [handled set] we enforce the propagation of the properties held by handles. Moreover we use [handled get] and [handled set] to update the state of the target object.

$$\begin{array}{lll} P & \vdash & \langle E[h^o_p.f], \mathcal{S}\rangle \hookrightarrow \langle E[h'^v_p], \mathcal{S}\rangle \qquad [\textit{handled get}] \\ & & \text{Where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \\ & & \text{And } h' \; is \; a \; new \; handle \\ & & \text{And } \mathcal{F}(f) = v \end{array}$$

$$\begin{array}{lll} P & \vdash & \langle E[h^o_p.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[h^o_p \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle \quad [\textit{handled set}] \\ & & \text{Where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \end{array}$$

Figure 5.9: State related reductions for HANDLELITE

5.3.4 Example

We illustrate the current formalism with an example of code to show that an execution of the handle propagates correctly the property and that it does not leak references to the target object.

In the following example, we expect to have a class (generated from the class of the target object) that implements the property that a handle applies and propagates on the target subgraph (it raises an exception on write access).

To illustrate the handle mechanism, we have a BankAccount object with an instance variable *user*.

```
BankAccount>>user
 ^ user
```

```
BankAccount>>myself
 ^ self
```

The class defines two methods: *user* that returns user and *myself* that returns *self*.

```
1 | hba ba |
2 ba := BankAccount new.
3 hba := ReadOnlyHandle for: ba.
4 hba myself.
5 hba user.
```

We create a BankAccount (line 2), then we create a read-only handle on it (line 3). Finally we execute *myself* and *user* methods.

The example shows that given a handle we cannot leak references to the target object: first,in *myself* we see that *self* returns the handle and not the target. Second, accessing instance variables when executing the user method, we get a handle on the value (potentially other objects not shown in the example). Now we see how this code is formally translate.

```
1 | hba ba |
2 ba := BankAccount new.
3 hba := ReadOnlyHandle for: ba.
```

$$\begin{array}{ll}
a & \mathbf{handle}(ba, ReadOnlyProperty) \\
b & \Rightarrow h^{ba}_{ReadOnlyProperty}
\end{array}$$

In previous code in lines *1 2 3*, we create a BankAccount *ba* and a read-only Handle *hba* on it. The handle is created and its corresponding formalism is: $\mathbf{handle}(ba, ReadOnlyProperty)$ (line *a*) and the reduced form $h^{ba}_{ReadOnlyProperty}$ (line *b*). Second, we send the message *myself* to the handle:

```
4 hba myself.
```

$$\begin{array}{lll}
a & \left[\left[h^{ba}_{ReadOnlyProperty}.myself()\right]\right] & [handled\ send] \\
b & \Rightarrow h^{ba}_{ReadOnlyProperty}[\![\mathrm{self}]\!]_{ReadOnlyBankAccount} & \\
c & \Rightarrow h^{ba}_{ReadOnlyProperty} &
\end{array}$$

In the previous code, we send a message to *hba* and fetch the self value, the message is transformed as a read-only send (since it is looked up in the class ReadOnlyBankAccount see *line a*, which is created by the property ReadOnlyProperty). The message send is unchanged, because it is a return self message (*line b*). It is impossible to leak a reference to the target object, as the self value is bound to $h^{ba}_{ReadOnlyProperty}$ (*line c*).

```
5 hba user.
```

$$\begin{array}{lll}
a & \left[\left[\mathrm{h}^{\mathrm{ba}}_{\mathrm{ReadOnlyProperty}}.\mathrm{user}()\right]\right] & [handled\ send] \\
b & \Rightarrow h^{ba}_{ReadOnlyProperty}[\![\mathrm{self.user}]\!]_{ReadOnlyBankAccount} & \\
c & \Rightarrow h^{ba}_{ReadOnlyProperty}\left[\left[h^{ba}_{ReadOnlyProperty}.\mathrm{user}\right]\right]_{ReadOnlyBankAccount} & [handled\ get] \\
d & \Rightarrow h^{ba}_{ReadOnlyProperty}\left[\left[h^{user}_{ReadOnlyProperty}\right]\right]_{ReadOnlyBankAccount} & \\
e & \Rightarrow h^{user}_{ReadOnlyProperty} &
\end{array}$$

In the code, we send a message to *hba* and get the instance variable *user* (*lines a b c*, [handle send]). The message is transformed as a read-only send (since it is looked up in the class ReadOnlyBankAccount). In this specific case, the message send is equivalent to the original send (because it is a read access). Second, the user value is fetched via $h^{ba}_{ReadOnlyProperty}.user$ (*line d*). The value of the field *user* is obtained in the target object and we create a new handle for this reference, $h^{user}_{ReadOnlyProperty}$ (*line e*). This example shows how instance variables are wrapped on access.

This example shows that:

- Handles ensure the read-only property: the read-only property is propagated to the object ba and all its subgraph from the handle point of view.

- *ba* does not leak any references to the original object, even if methods in the target object return references to themselves (*myself* in the example).

5.4 Read-only References with Handles

Now we rewrite how this new handle model can be used to implement the read-only behavior and its propagation. The Chapter 4 presented a version dedicated and specific to read-only. In this first version, there was no metahandle, the handle propagation was done by on-the-fly bytecode rewriting and was only supporting read-only behavior. Figure 5.10 shows that the read-only handle shadow class contains rewritten methods of the target class such that they raise error. The error raising behavior is based on rewriting store bytecodes as in the previous model Chapter 4. We now present the how the generic handle model can be used to implement the read-only behavior. The framework provides two entry points to the language designer:

- He should define the behavior of the handle creation method named for: aTarget. This class method takes as argument a target object.

- He should specify how an handle is created during the object graph propagation by defining the method propagateTo: aTarget. This method expects again a target object.

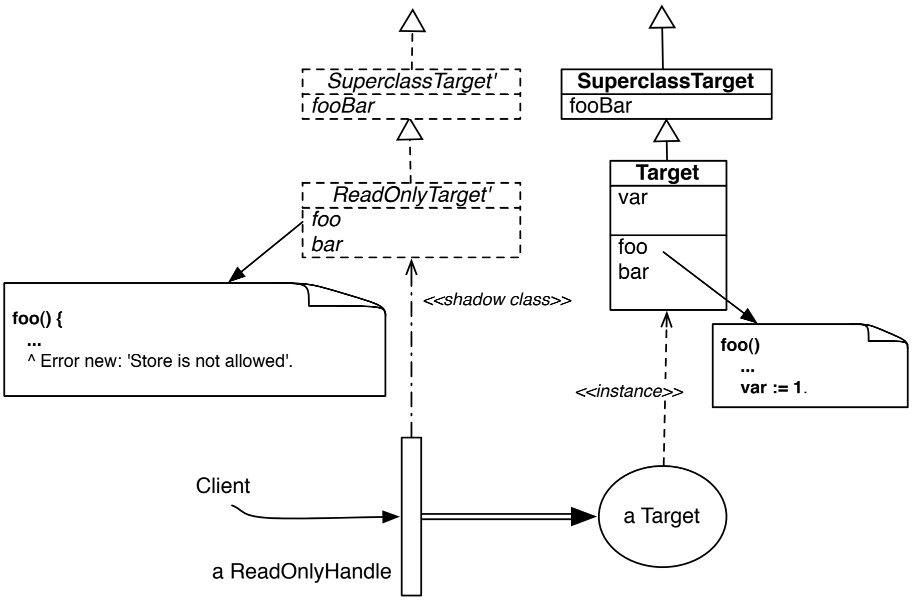


Figure 5.10: Read-only handles: handle shadow classes contain rewritten methods so that they raise an error.

Handle creation. Practically, we create a specific handle class ReadOnlyHandle subclass of Handle. We specify the handle creation as follows:

```
1 ReadonlyHandle class>>for: aTarget
2  | handle aROShadowClass |
3  aROShadowClass := self createROShadowFor: aTarget.
4  handle := self initializeFor: aTarget to: aROShadowClass.
5  handle useShadowBehavior.
6  handle activateHandle.
7  ^ handle
```

Line 3: A class is obtained as a transformed (readonly) version of the target object class – All store accesses to instance variable will raise an exception. Such behavior is not the concern of the handle framework but of the language designer that should provide it. Line 4 we create a deactivated handle associated with the read-only class. Line 5 specifies that messages sent to the handle are applied to the target. Line 6 activates the handle. From then on we cannot access the handle itself anymore. Line 7 returns it.

Propagation. In addition, the framework asks us to define the creation of handles during the propagation by defining the class method propagateTo: which is invoked by the virtual machine. Here we simply create a read-only handle on the argument.

```
ReadonlyHandle class>>propagateTo:anObject
 ^ ReadonlyHandle for: anObject.
```

This message is sent when an instance variable is accessed. Its value is returned in place of the instance variable. This code transformation dynamically propagates the read-only behavior to the object graph.

5.5 Revocable references with handles

The idea behind a revocable reference is to create a reference to an object that can be controlled and revoked [Miller 2003b]. Our revocable reference implementation uses handles and metahandlesf (as shown in Figure 5.11). A revocable reference named doc' with a handle on Doc is created. A controller reference named c-doc', a metahandle on the handle doc', is created. Alice gives to Bob doc' (the revocable reference). When Alice wants to revoke this reference it uses the controller reference. Our implementation is based on the possibility to toggle the shadow behavior using a metahandle: When on (*i.e.*, reference is revoked) the shadow class will raise errors, when off the messages are normally handled (*i.e.*, messages sent to a handle are not looked up in the shadow class but in the target class).

5.5.1 Revocable references implementation

We implement the Revocable References using Handles in three steps.

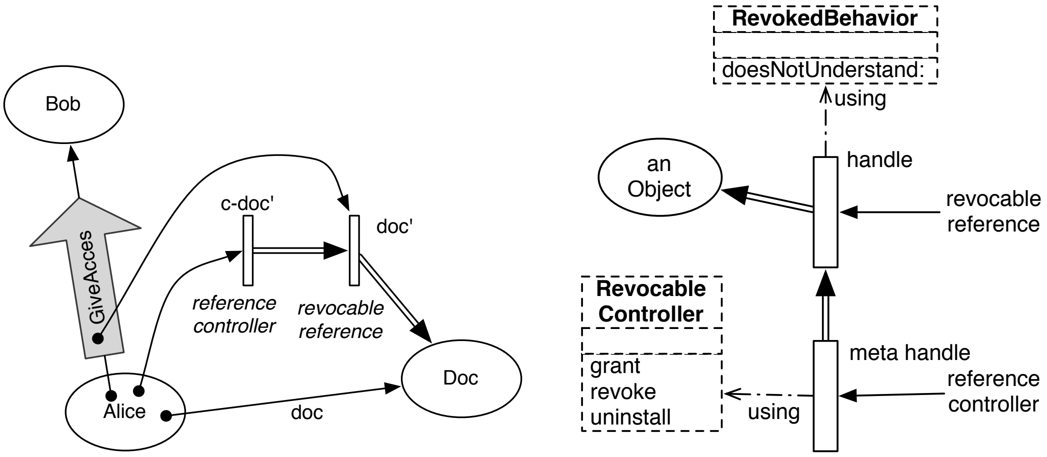


Figure 5.11: Right: Revocable References using Handles. Left: Revoked references have as a shadow classes that does nothing but raising errors.

Step one: error raising behavior. Sending messages to an object via a revoked reference should raise errors. To implement such revoked behavior, we create a class named RevokedBehavior which inherits from nil. This class does not define any method besides the doesNotUnderstand: method which raises an error [Pascoe 1986, Ducasse 1999]. Any message send will then raise the exception AccessRevoked. RevokedBehavior will play the role of a shadow class for all the revoked references. Again such behavior is part of the language designer task to define the semantics he wants for his language constructs.

```
RevokedBehavior>>doesNotUnderstand: aMessage
 ^ AccessRevoked signal.
```

Step two: RevocableReference. Second we define a new subclass of Handle named RevocableReference.

```
1 RevocableReference class>>for: aTarget
2     | revocableHandle controller |
3     revocableHandle := self initializeFor: aTarget to: RevokedBehavior.
4     controller := RevocableReferenceController for: revocableHandle.
5     controller dontUseShadowBehavior.
6     revocableHandle activate.
7     ^ {revocableHandle . controller}
```

Line 3 a new handle is created and associated with the revoking behavior created in Step 1. Here we do not need to get a shadow class per target class since we want to always raise errors and RevokedBehavior is playing this role for all the target object classes. Line 4, a metahandle (created in Step 3) is created on the handle. Line 5 configures the handle not to use the revoking behavior. Line 6 activates the handle. Line 7 returns an array with the handle and its controller (a metahandle).

Step three: RevocableControllerReference. To control the handle (the revocable reference), we define a new metahandle class named RevocableReferenceController. This class implements two methods revoke and grant.

```
RevocableReferenceController>>revoke
 self useShadowBehavior.
```

```
RevocableReferenceController>>grant
 self dontUseShadowBehavior.
```

When Alice sends the message revoke to the controller, this message applies the method revoke on the revokedRef (doc') handle. The revocable reference uses then the shadow class behavior which leads to error for any messages.

The rest of this section shows how we can use the natural propagation of properties inside the object subgraph to enhance revocable references.

5.5.2 Propagation of revocable references

Revocability of references should propagate to a graph of used objects. Since SubDoc is reachable from Doc it may leak a reference of Doc to Bob. Such a reference should not break the fact that Doc reference to Bob is revocable. Therefore SubDoc should pass to Bob only revocable references when accessed via Doc. In addition, all the references reachable from Doc subgraph should also be revocable (Figure 5.12).

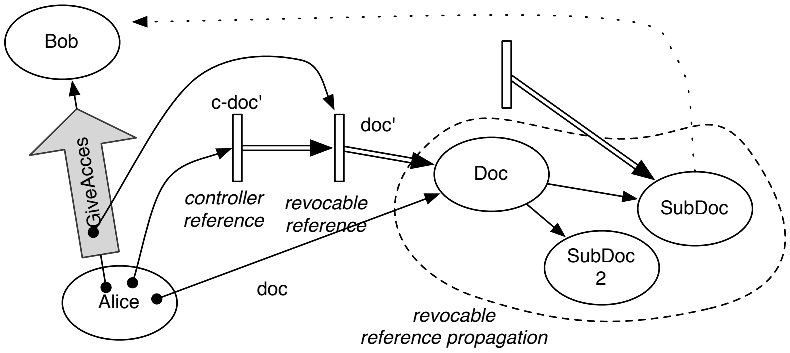


Figure 5.12: Propagation of revocable references: When Bob accesses SubDoc via Doc, SubDoc should be revocable too.

Revocable reference *propagation* is a bit more complex than the one of the read-only property, because one should only *revoke* the references coming from a specific graph. The basic idea behind the solution is that:

- All the handles in a revocable reference graph *coming from the same original handle* have the same identifier.

- We introduce a factory that creates metahandles, and keep per graph a list of reference controllers (metahandles) to be able to revoke references. It offers the API to revoke references.

5.5.3 Using revocable references

Now everything is in place. The following code illustrates the behavior of the system. Alice asks the Factory to provide a revocable reference on Doc (Line 1). She obtains a pair consisting in an instance of factory and a revocable reference on Doc (named hDoc). This instance of factory contains only the reference controllers for the specific references created by this invocation. Later it may contain different reference controllers gathered during the propagation which occurs during hDoc doSomething execution (Line 4) but all the references will have the same identifier. Alice can pass hDoc to anybody. Later, Alice asks the factory to revoke the references (Line 6). Now references hDoc raise exceptions when used.

```
1  pair := ControllerFactory for: Doc.
2  "pair is an array with aFactory and hDoc"
3  hDoc := pair second.
4  hDoc doSomething.
5
6  pair first revoke.
7  "we ask the factory to revoke the references to Doc from this specific
     reference"
8  hDoc open  "raises an exception"
```

5.6 Virtual machine level implementation

In this section, we present the virtual machine changes needed to support Handles.

Bytecodes. To implement that a handle is transparent, we modify the identity primitive and associated bytecode of the virtual machine. The primitiveIdentical tests if two objects are identical (if the pointer in memory is the same). We modified it so that when invoked on handles, the VM compares their target objects.

The difficult case of primitives. At the language level, when a message is sent to a handle, the found method (if any) is applied to the target object. In addition, it is not possible to distinguish the handle and the target object. However, from the virtual machine point of view a handle is an object, therefore we had to modify the virtual machine to take into account handles at the level of primitives. (A primitive is a functionality that is implemented at the VM level and invoked from the language level. Primitives exist for low-level operations such as integer or float manipulation, memory allocation, object offset access (basicAt:, basicAt:put:), method execution (perform:, executeMethod, pointer swapping (become:),....) All together, there are around 150 primitives. In the Squeak/Pharo VM, primitives acts as message sends but shortcut the normal bytecode dispatch loop and invoke directly their associated VM C function.

The challenges we faced is that primitive invocations should not be freely executed as this may lead to a leak of the target. The key points are:

- A handle is in charge of deciding which methods can be executed when a message is sent to it. If the shadow class hierarchy does not include a selector, even for primitive methods, it is not be accessible and executed. The handle designer is in charge of the semantics and elements he wants to provide access to. Our design decision is that by default nothing is possible.
- Primitive invocation on handle objects related to state access are executed as if they were sent to the target object.
- Reflection cannot bypass handles. Our implementation takes care that reflective features cannot bypass the handle semantics and propagation. All the primitives were rewritten to take care of handles.
- Certain meta operations such as invoking directly methods or performing method lookup (perform:) use the shadow class of the handle.

We adapted the virtual machine primitives to behave as described. Primitives have to be analyzed case by case.

5.6.1 Controlling behavior

By design, a handle controls object execution and dynamically changes target object behavior. To implement this, we modified the method lookup location in the VM. If during a message send the receiver is an *activated* handle, we modify where the lookup starts: the shadowClass or the class of the target object (when the option is to not use the shadow behavior).

Here is the normalSend method of the Squeak/Pharo VM implemented in SLang (A Smalltalk subset which is transformed to C) [Guzdial 2001]. A normal send is invoked for each method invocation (except primitive ones). It is inlined.

```
Interpreter>>normalSend
   "Send a message, starting lookup with the receiver's class."
   "Assume: messageSelector and argumentCount have been
   set, and that the receiver and arguments have been pushed
   on the stack,"
   "Note: This method is inlined into the interpreter dispatch loop."
```

```
| rcvr |
...
(self activatedHandle: rcvr)
   ifTrue: [ (self handleUseShadowBehavior: rcvr)
      ifTrue: [ lkupCls := self handleClassLookupOf: rcvr]
      ifFalse: [ lkupCls := self fetchClassOf: (self handleTargetOf:
 rcvr)]].
self commonSend.
```

- The cost of adding a test in each message send is not marginal. We discuss this in Section 5.7. We experimented with alternative designs such as changing the class of the Handle at activation time but it leads to a more static solution and was not satisfactory.

- At this step we do not change the receiver of the message, it is still the handle.

5.6.2 Propagation

There are two aspects of propagation: (1) what is propagated and (2) at which moment the propagation occurs. The first aspect is delegated to the Handle class itself by calling the class's propagateTo: method. We presented this point in previous sections. At the VM level, it requires to lookup this method and execute it.

For the second aspect, we send the propagateTo: message to each target instance variable read access. This ensures that all the objects of an object graph get a chance to be wrapped with a handle during one execution flow. At the VM level, we change the pushInstVarAt bytecode so that when the propagation is enabled, we substitute on the stack the pushed instance variable by the corresponding handle (given by the previous step - *i.e.*, calling the propagateTo: method). This is enough to implement the semantics described previously.

5.7 Evaluation: performance analysis

To validate our approach, we present a short analysis of the performance and overhead. For the Handle implementation, we need to analyze two different aspects: first, we modify the virtual machine to support handle execution. This implies modifications to perform check for handles that slow down normal execution (*i.e.*, code not using handles). Second, we analyze the performance when using handles for different scenarios.

Base performance. We measure the performance of our modified VM compared to the normal VM. For this we execute two examples: a binary tree and a simple n-body simulation[2]. We execute it without actually using handles on both the normal virtual machine and on our Handle virtual machine. The two virtual machines used are compiled from the

[2] http://shootout.alioth.debian.org/

Squeak/Pharo VM version *4.2.2b1*. They are generated manually with the exact same build environment.

- The *binary-trees* benchmark. We build binary trees and then iteratively remove all nodes, until a deepness of 16. We execute this benchmark 50 times. In this benchmark we see an overhead of 5.45% of execution for the Handle VM.

	Means	Standard deviation
normal VM	21167.00*ms*	106.26*ms*
handle VM	22321.57*ms*	66.35*ms*

- The *n-bodies* is a model of the orbits of planets. This benchmark is interesting because it stresses state access. In this benchmark we see an overhead of 7.36% of execution time when using the Handle VM. We execute this benchmark with argument N=100000, 50 times to make this measurement.

	Means	Standard deviation
normal VM	4444.50*ms*	38.36*ms*
handle VM	4772.80*ms*	20.68*ms*

So we see a slowdown of less than 8% in both cases for executing code on our special handle VM prototype. The reason for the slowdown is coming from the checks for handles vs. objects when accessing state, message sends, and identity. Schaerli *et al.* [Schärli 2004a] reports an overhead of 15% for their implementation of encapsulation policies and they also modify a virtual machine to introduce references. We used the same virtual machine but a more recent version. The difference is probably due to the fact that we spent more time optimizing our implementation. Note that using a more recent version is not really an advantage since introducing changes in a more optimized system usually results in more overhead because the standard case to compare against is better optimized.

Discussion. Handles require to modify and control message sends. In Smalltalk, message sends are the most frequently used primitive instruction. Therefore overhead in message sends will induce a cost for all computation.

Cost of handle execution. In addition to the general slowdown of the VM, we are especially interested in the overhead of actually using handles in a program. It is clear that the slowdown depends on the behavior that the handles introduce as well as how the handles are used. The slow-down will therefore be different for the kind of handle used (*e.g.*, revocable references, read-only) and in addition will depend on the scenario of actual use.

For revocable references, we perform the two previous benchmarks *n-bodies* and *binarytrees*, they are an especially stressful benchmarks to show the cost of some specific operations on objects via a handle.

- In the *n-bodies* benchmark, we create a revocable reference and we have 24000000 access to integers (in additions of the algorithm execution operation).

	Means	Standard deviation
revocable nbody	8172.12ms	31.01ms
nbody	4772.80ms	20.68ms

  We see an increase in runtime of 71%. This slowdown is substantial, but explained by an implementation detail of the virtual machine: integers are not objects, they are instead encoded in the pointer and operations are optimized by special bytecodes. As soon as we use handles, the execution uses normal objects and message sends for the handle object. Even for this worse-case, the slow-down does not prohibit real world use.

- The *binarytrees* benchmark is performed to focus on the slow-down introduced by instance variable propagation and RevocableReference initialization.

	Means	Standard deviation
revocable binarytrees	68094.23ms	70.06ms
binarytrees	22321.57ms	66.35ms

  We see a slowdown of 205%, the reason for the slowdown is the number of graphs managed and their size (1747535 different object graphs with size between 4 and 65536 nodes). The example is very extreme in the number of revocable data structures managed: even with a very large number of revocable graphs managed, the mechanism stays usable in practice.

  The two previous benchmarks show a significant overhead. But these benchmarks focus on showing that even in extreme cases, the system is practically usable. In practice, revocable references are not used to manage such a large number of different objects graphs. To measure the usual cost of using a Revocable Reference, we take another benchmark *regex-dna*[3]. Here we read as input a DNA sequence and match and translate into nucleotide code. We protect the input value by a revocable references. We see a slowdown of 8.8%.

	Means	Standard deviation
revocable regex-dna	1095.12ms	13.46ms
regex-dna	1006.32ms	11.44ms

In the current state the Handle prototype is implemented in a relatively naive way to explore and validate the model. In the future we want to evaluate whether a VM dealing directly with first-class references provides better performance. Another possible improvement is to have a fast bytecode rewriting engine at the VM level.

[3] http://shootout.alioth.debian.org/

Memory usage. The exact cost of using a handle can be calculated easily. A handle is allocated as a normal but compact object in the system. In Squeak/Pharo a compact class is represented differently than normal classes. A list of maximum 32 classes can be turned into compact class to save space. The object header of their instances consists of only a single 32-bit word and contains the index of their class in a compact classes array. This makes handles small, and more importantly, it allows the virtual machine to check whether an object is a handle or a real object by looking at the object header alone. In addition a handle object has three instance variables. This means an instance of the handle has a size of 16 bytes (one word for the header, three words for the instance variables). For each object that a handle is generated for, we pay 16 bytes. In addition, one need to count the generated classes. The cost for those depends on the exact handle. *e.g.*, for read-only, we have to copy the class hierarchy, while revocable just needs one revoking behavior class.

5.8 Conclusion

In this chapter we have presented Handles, an approach to support behavior-propagating first-class reference as a language construct. We explored how handles are used to apply security semantics dynamically to object graph at runtime. Handles allow several security related language extensions to be implemented. We have presented an object-capability system, and read-only references and validated our implementation with benchmarks. In the following chapter (Chapter 6), we will extend Handles to support state shadowing in addition to behavior modifications.

Chapter 6
Stateful Handles

Contents

The current Handle model provides a mechanism for supporting the definition of properties and for automatically propagating these properties to a complete subgraph. What our previous approach lacked is any ability to handle state (*i.e.*, how the state of an object can be modified and visible only from one reference). With read-only execution (see Chapter 4), state modification was forbidden by changing behavior on a per-reference basis. But in many cases, we need to be able to control state change, not just disable it completely.

This leads to the question:

> *How handle (at reference-level) can support object state isolation?*

This chapter proposes SHandle, an extension of the previous Handle model (presented in Chapter 5) as one answer to this question. SHandle realizes handles that provide the possibility to shadow state of the target object on a per reference-level.

The contributions of this chapter are:

1. The SHandle framework, which provides a way to isolate object state per references using it.
2. The application of this extended framework to implement two examples: a simple transactional memory model and Worlds [Alessandro Warth 2011].

6.1 SHandles: handles with state

SHandle extends the model of first-class references presented in the previous chapter. In addition to behavior modifications and propagation, we want to be able to keep object state changes local to the reference themselves.

The most important changes are:

- State shadowing: the state of the object can be shadowed by a SHandle. If a 'handled' reference changes the state of the target object, the change will only be visible from the reference itself.

- Since the underlying language (Smalltalk in our case) supports identity comparison using the message == and equality using the message =, we had to revisit them in presence of handles and to introduce a new operator that can distinguish between handles and objects (to support reflecting about handles).

- Metahandle: We extend handle control especially related to state: enable/disable state shadowing and merging the state of the handle into the target.

Figure 6.1 describes the underlying principle of SHandle:

(1) A shandle is a transparent reference to a target object.

(2) A shandle can define different behavior than the target object.

(3) A shandle can define its own state different from the state of the target object. In Figure 6.1, restrictedClient1 can only have access to the state *a target'*. Once it is created, there is no longer any relationship between the state embedded in the Handle (*a target"* in Figure 6.1) and the state of *the target object*.

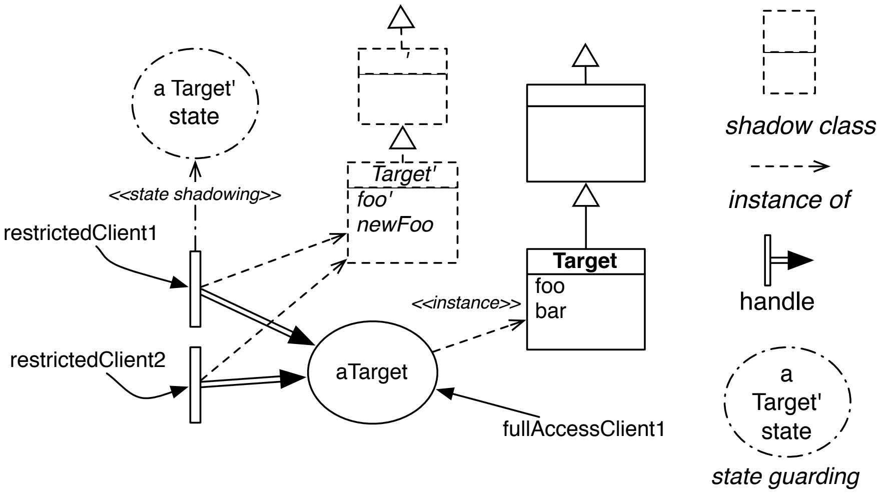


Figure 6.1: Shandle principle: a shandle keeps its own version of the target state.

Shandle creation. To ensure that once a shandle is created, there is no possibility for the programmer to access the target directly. Similarly to the previous model, we divide the lifetime of the shandle in two distinct periods:

- *Initialization.* A shandle is initialized with the configuration options managing its behavior. Immediately after the initialization the system activates the shandle.
- *SHandle activation.* Once a shandle is activated, it represents a *view* on the target object. It is impossible to directly send messages to the shandle. Such messages are automatically managed as messages sent to the target and follow the behavior described earlier. This behavior is implemented at virtual machine level and cannot be reverted.

6.1.1 State access through shandle

The goal of SHandle is to keep side effects isolated from a reference perspective. As we say before, each shandle can keep its own version of the target object's state. In Figure 6.2, when restrictedClient1 invokes the getX or setX: accessor, they access the state of the instance variable x' of a different version than the target object. All changes performed by the setX: method are happening to the shandle. Note that a shandle as any other objects can still be stored in instance variables.

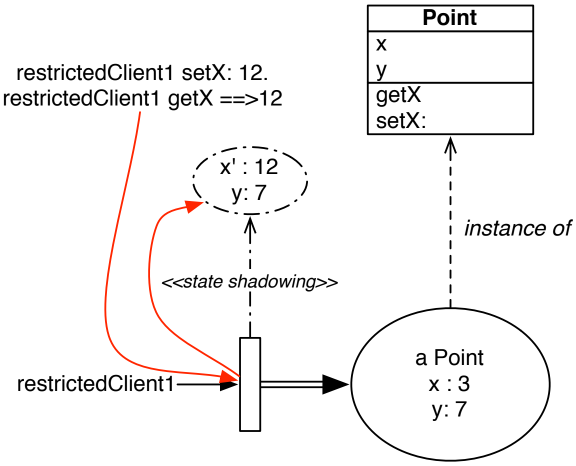


Figure 6.2: State shadowing: a shandle keeps the target state on a per reference-level.

6.1.2 Propagation

The propagation mechanism is a cornerstone of the Handle model and it is kept in the SHandle model. When an object state is accessed via a shandle, the property encoded

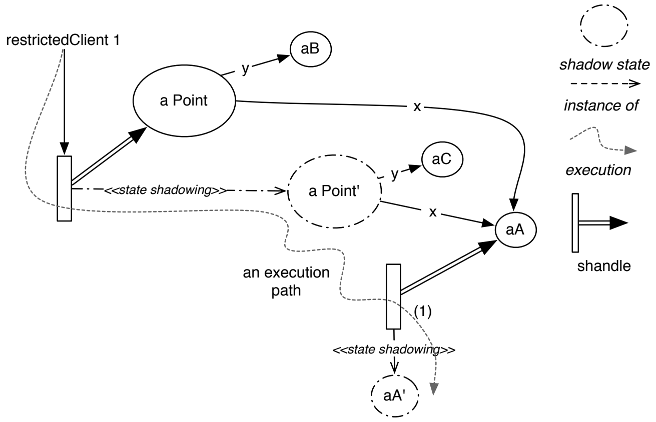


Figure 6.3: SHandle Propagation Principle. All accesses via a shandle to target object state are wrapped with a shandle and propagated at runtime.

by the shandle is propagated. For example, for a simple state handling, a new shandle is automatically created on that state.

The propagation is performed on a value extracted from a handle state object (See Figure 6.3). From a shandle client point of view, *any instance variable read access propagate the handle property*. In particular, reading an instance variable of the target object now returns a shandle on this object (*(1)* in Fig. 6.3). In Figure 6.3 restrictedClient1 getX returns a shandle on the object aA. This propagation is recursive and follows the application execution.

From *a shandle*, the receiver of a message is the shandle via which the object was accessed. Note that restrictedClient1 setX: (Object new) stores a new object (not a shandle on this new object) in the target object and returns the shandle used by restrictedClient1. This point will be clarified with the formal definition given in Section 6.2.

Propagation and state shadowing relationship. The propagation property of shandle is orthogonal to the state shadowing property. The propagation property represents *how* the value should be accessed and propagated through the object subgraph at runtime, and the state shadowing represents *which* value should be accessed.

6.1.3 Equality, similarity and identity

One side effect of state shadowing addition to the Handles model is that it forces to take care about *equality*. Indeed, with SHandle, we get a kind of perspective on an object and this perspective caches locally the state of the observed object, therefore there is a need to redefine identity and equality in presence of a shandle.

To keep the system coherent we revisit the equality and identity primitive notions and

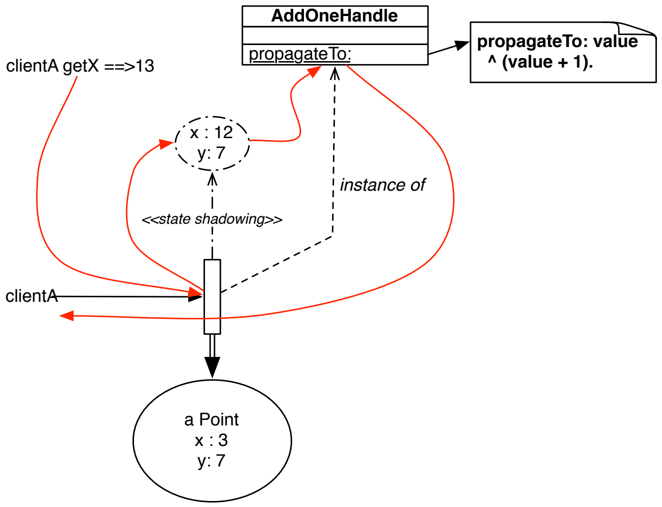


Figure 6.4: State and Propagation: first the shandle get the value in shadow state, then shandle propagates the property.

we add another kind of identity checking that takes into account shandle differentiation for low level operations (usually required for meta-level infrastructure).

This way we obtain:

Equality = The message = expresses the equality between objects. It returns true when the two objects have the same values.

Similarity == The message == expresses the similarity between objects. Two references pointing on the same *target object* are similar. Two references can have different behavior or different states. Object similarity does not imply equality or identity. In Smalltalk, the message == compares the identity between objects. In our system it compares the identity of the target objects.

Identity === The message === expresses that two references are identical when their references are the same. Using the message === allows one to distinguish between a handle and its target, even if at a conceptual level, we have the same identity. Identity still implies equality and similarity. The identity is a low level operation that should not be used by end-users.

6.1.4 Metahandle: controlling a shandle

We want to be able to control the behavior of a shandle. As we see before in Chapter 5, a metahandle is a handle whose target is a handle. As we extend handles, we need to adapt metahandle to manage the new abilities of shandles.

We added two mechanisms:

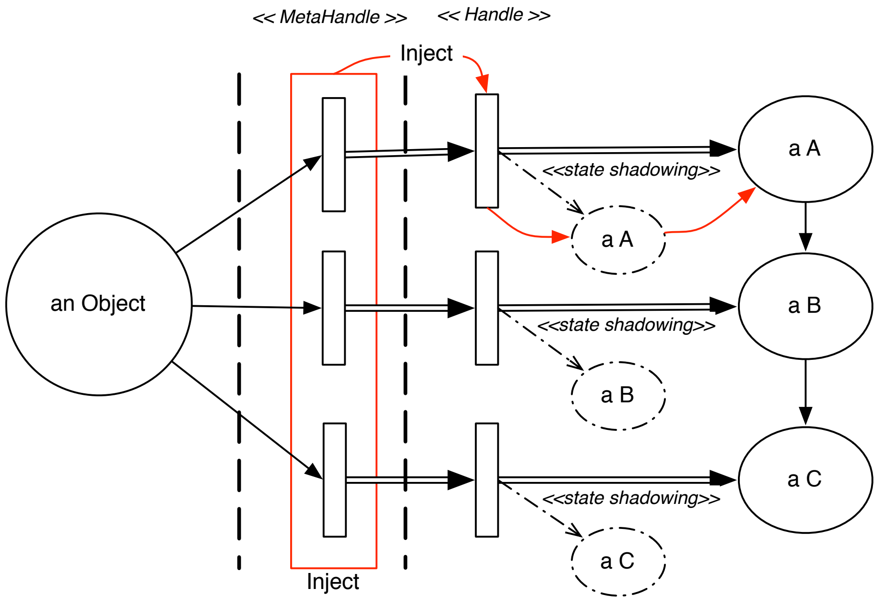


Figure 6.5: Injecting process of Metahandle

(De)/Activate State Shadowing. We introduce the possibility to manage the state shadowing behavior of a shandle: only from a metahandle we have the possibility to activate/deactivate the shadowing behavior of the target handle.

Handle vaporisation. In addition, it is also possible to *vapor* a handle. Vaporisation remove the handle and inject the state in the target object (if required). First we *inject* the state held by a shandle into its target (only if the state shadowing is active) and we migrate all pointers pointing to the handle to the real target object.

6.2 SHandleLite: shandle operational semantics

In the previous example, we used a SHandle to provide an isolated graph of objects. We reuse the example of the BankAccount presented in Chapter 5. The goal is to underline how SHandle provides an isolated graph and to describe how the state protection works.

6.2.1 Similarity to previous operational semantics

In Chapter 5, we have presented HandleLite, an operational semantics for our previous handle model. This operational semantics required some small changes to be adapted to the new Shandle model. As the proposed model is an extension of our previous model, most of the operational semantics presented in Chapter 5 is the same, only the part about identity and instance variable access has been rewritten to support Shandle.

6.2.2 Per reference state shadowing and propagation

SHandles need to keep their own state. As we see in Chapter 5 adding shandle to redex allows reading a field on shandle $h^o_p.f$ and writing a field on shandle $h^o_p.f{=}\varepsilon$, write shandle into field $\varepsilon.f{=}\ h^o_p$. State guarding and propagation imply to manage the state accesses in a different way when there are performed via shandles. So as we see in Figure 6.6, we rewrite the two reductions:

handled get represents how the instance variable is read from a shandle. The [handled get] reduction has two goals, first to manage state guarding read access, shandles keep their own version of the state. So $\langle E[h^o_p.f], \mathcal{S}\rangle$ implies that the field f is stored in the shandle itself. In second time, [handled get] should propagate the properties p, so instead of reducing $\langle E[h^o_p.f], \mathcal{S}\rangle$ to $\langle E[v], \mathcal{S}\rangle$, we wrap the returned value $\langle E[h'^v_p], \mathcal{S}\rangle$ into a new shandle h'^v_p respecting the same properties p.

handled set represents how the instance variable is written to a shandle. The [handled set] reduction manages state shadowing write access. SHandles keep their own version of the state. So $\langle E[h^o_p.f{=}v], \mathcal{S}\rangle$ implies that the field f is stored in the shandle itself. By consequence all changes which occur from this handled reference, remain local to this shandle and do not impact the target object itself.

$$\begin{array}{lll}
P \vdash & \langle E[h^o_p.f], \mathcal{S}\rangle \hookrightarrow \langle E[h'^v_p], \mathcal{S}\rangle & [\textit{handled get}] \\
& \text{Where } \mathcal{S}(h^o_p) = \langle c, \mathcal{F}\rangle & \\
& \text{And } \mathcal{F}(f) = v & \\
& \text{And } h' \; is\ a\ new\ shandle & \\
& \text{And } c' \; is\ a\ class\ supporting\ the\ property\ p\ \ (via\ p(c) = c') & \\
& \text{And } c\ is\ the\ class\ of\ the\ object\ o & \\
& & \\
P \vdash & \langle E[h^o_p.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[h^o_p \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle & [\textit{handled set}] \\
& \text{Where } \mathcal{S}(h^o_p) = \langle c, \mathcal{F}\rangle & \\
& \text{And } c' \; is\ a\ class\ supporting\ the\ property\ p\ \ (via\ p(c) = c') & \\
& \text{And } c\ is\ the\ class\ of\ the\ object\ o &
\end{array}$$

Figure 6.6: State related: reductions for SHandleLite

6.2.3 Addendum: transparent proxies

As we see in Chapter 5, the identity management in SmalltalkLite should be changed to a function. In this chapter, state handle added a new form of identity. We need now to deal with *Equality*, *Similarity* and *Identity*. To solve that in the operational semantics, we add to HandleLite the possibility to identify shandle as itself. We add a primitive function that will bypass the transparent properties of the shandle. We call it *rid* (for reference identity).

$$\begin{aligned} \neg\,(rid(h_p^o) &= rid(o)) \\ rid(o) &= oid(o) \end{aligned}$$

This function return the identity of the receiver, for normal object it is equivalent to *oid*, but in case of a shandle it will return and the real identity of the shandle. This primitive function is designed to be used in reflective code to check if the two references are same reference. Consequently, a target object can be distinguished from a shandle. To summarize *Similarity*, it is managed by the function *oid* presented in Chapter 5 and the *Identity* is managed by the function *rid*.

6.2.4 Example

We illustrate the current formalism with an example of code to show that an execution of the shandle ensures the state shadowing properties. To illustrate the handle mechanism, we have a BankAccount object with an instance variable *user*.

```
BankAccount>>user
 ^ user
```

```
BankAccount>>setUser: value
 user := value.
 ^ self
```

The class defines two methods: *user* that returns user and *setUser:* that sets user to an argument. Note that *setUser:* returns the receiver.

```
1 | hba ba |
2 ba := BankAccount new.
3 hba :=  IsolationHandle for: ba.
4 hba setUser: 42.
5 hba user.
```

We create a BankAccount (line 2), then we create an IsolationHandle (shandle designed to isolate the state) on it (line 3). Finally we execute *setUser:* and *user* methods. The example shows how a shandle keeps its own state: first, *setUser:* will set the *user* field to *42* value. Second, accessing instance variables when executing the user method, we get a shandle on the value that we just set before.

```
| hba ba |
ba := BankAccount new.
hba := IsolationHandle for: ba.
```

$$\begin{array}{ll} a & \mathbf{handle}(ba, IsolationProperty) \\ b & \Rightarrow h^{ba}_{IsolationProperty} \end{array}$$

In the previous code in lines *1 2 3*, we create a BankAccount *ba* and an IsolationHandle *hba* on it. The shandle is created and its corresponding formalism is: $\mathbf{handle}(ba, IsolationProperty)$ (line a) and the reduced form $h^{ba}_{IsolationProperty}$ (line b).

```
4 hba setUser: 42.
```

$$
\begin{array}{lll}
a & \left\llbracket h^{ba}_{IsolationProperty}.setUser(42) \right\rrbracket & [handled\ send] \\
b & \Rightarrow h^{ba}_{IsolationProperty} \llbracket \text{self}.\text{user} = 42 \rrbracket_{BankAccount} & [handled\ set] \\
c & \Rightarrow h^{ba}_{IsolationProperty} \llbracket \text{self} \rrbracket_{BankAccount} & \\
d & \Rightarrow h^{ba}_{IsolationProperty} &
\end{array}
$$

In the previous code, we send a message to *hba* and set the value of the field *user* to 42, the message is not transformed (since it is looked up in the class BankAccount but applied to the shandle, see *line a*). Then the message itself is executed and the field *user* of the shandle is set to 42. Finally the message returns the *self* value (still bound to the shandle).

```
5 hba user.
```

$$
\begin{array}{lll}
a & \left\llbracket \mathrm{h}^{\mathrm{ba}}_{\mathrm{IsolationProperty}}.\mathrm{user}() \right\rrbracket & [handled\ send] \\
b & \Rightarrow h^{ba}_{IsolationProperty} \llbracket \text{self}.\text{user} \rrbracket_{BankAccount} & \\
c & \Rightarrow h^{ba}_{IsolationProperty} \left\llbracket h^{ba}_{IsolationProperty}.\text{user} \right\rrbracket_{BankAccount} & [handled\ get] \\
d & \Rightarrow h^{ba}_{IsolationProperty} \left\llbracket h^{42}_{IsolationProperty} \right\rrbracket_{BankAccount} & \\
e & \Rightarrow h^{42}_{IsolationProperty} &
\end{array}
$$

In the previous code, we send a message to *hba* and get the instance variable *user* (*lines a b c*, [handle send]). The message is not transformed by the shandle. Second, the user value is fetched via $h^{ba}_{IsolationProperty}.user$ (*line c*). The value of the field *user* is obtained in the shandle (*line d*) and we create a new shandle for this reference, $h^{42}_{IsolationProperty}$ (*line e*). This example shows how instance variables are wrapped on access and the isolation is maintained.

This example shows that:

- SHandles ensure that the isolation property is propagated to the accessed objects from the shandle point of view.
- *ba* reference does not leak any references to the original object or state, even if methods in the target object returns references to themselves by return the receiver (*setUser:* in the example).

6.3 Validation: software transactional memory

As a first validation, we present how SHandle can be used to implement *Software Transactional Memory* (STM).

The objective of STM is to improve the use of available hardware resources. In some domains such as distributed systems, concurrency control is mandatory. Software Transactional Memory (STM) is a modern paradigm, based on transaction synchronization, which takes a different approach from conventional approaches such as mutual exclusion and message passing; software transactional memory is seen by some as a more user-friendly approach to synchronization [Bieniusa 2009].

STM is based on the concept of *atomic block*, used to encapsulate execution that should be performed safely. The transaction begins when entering in the *atomic block* and finishes when leaving it. STM ensures then that the side effect of the computation will be fully performed or not at all. There are several kinds of STM systems implementing different semantics [Shavit 1995].

It should be made clear that our goal is not to provide a fully engineered real world implementation of STM. Instead we want to present a prototype of STM realised with SHandles as a proof of concept.

6.3.1 Implementation choices

Annette Bieniusa did a *classification STM systems with respect to the most important design choices* [Bieniusa 2009]. We select some of these design choices and propose to implement a STM system supporting all these properties.

- **Atomicity** is level of isolation. We chose *Strong Atomicity* that ensures that an atomic block executes in isolation with respect to all other computations.
- **Conflict detection.** We chose to use *optimistic conflict detection* that postpones data validation until the end of an atomic block.
- **Granularity of conflicts.** We chose *object level granularity*. This means that changes are detected at the object level.
- **Data versioning** defines how the changes are managed by the transaction to allow one to undo all state change in case of aborting the transaction. There are different possibilities for the implementation. One example is to keep a log of all changes applied, another to keep a copied version of the object state. With *SHandle* we chose to use the *SHandle* object itself. SHandles shadow state changes and therefore store all changed state without modifying the original object.
- **Nesting** defines how a transaction should be managed when it is defined in another transaction. We chose to implement *closed nesting* that performs a commit check and possible rollback of a nested transaction, but ensures that the nested transaction effects only become globally visible when the outermost transaction commits.

We will present how Software Transactional Memory can be implemented following these properties using SHandle.

6.3.2 STM and Shandles

Implementation in a nutshell. SHandles have the ability to keep their own version of state. Together with propagation, they can be used to isolate the state changes of a complete object graph. To illustrate how SHandle can be used to isolate the change that occurs during a transaction, we look at the following example:

The following code protects the changes

```
1 protectedReference := Array with: #(1 2 3).
2 transaction := Transaction new: [:reference || value |
3     value := reference at: 1. "get the variable"
4     reference at: 1 put: (value * 2).]
5 with: {protectedReference} "wrap the reference passing in parameter"
6 transaction run. "run the transaction"
>>> #(2 2 3)
```

At *line 1* we create an array with respectively at first, second and third index the value one, two and three. At *lines 2 to 5* we create a transaction take as arguments an *atomic block* and *argument array*. *Atomic block* such at *lines 2 to 4* where we change the first index of the array by twice of the stored value. And an *argument array* that are root references of the transaction. Finally we do not run the transaction until we send the execute transaction.

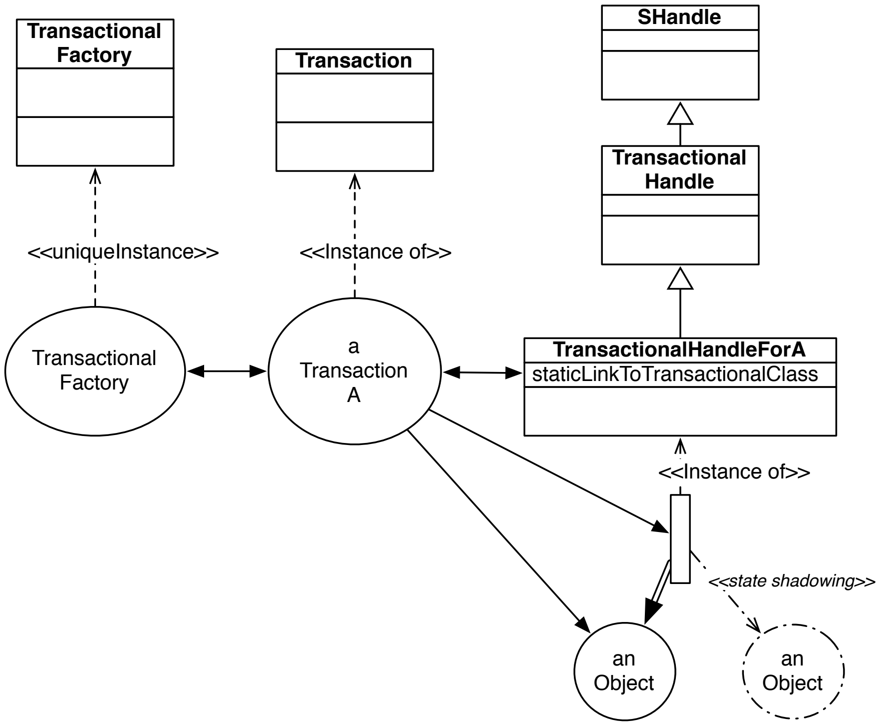


Figure 6.7: State and Propagation

Implementation. Our implementation of STM is based on three classes. The three classes represent the three abstraction levels needed to be managed, see Figure 6.7. It is composed of three elements: transaction, handle and transactional factory.

- **Transactional Factory**: it manages the relationship between transactions. There there is one instance Transactional Factory in the system. The Transactional Factory object manages conflict detection and conflict resolution. It detects modifications at the level of the transaction, initiates the abort of the transaction and establishes a schedule for re-executing aborted transactions.

- **Transaction**: it manages relationships between the objects accessed by a transaction. A transaction object manages the creation and commit process for all objects taking part in the transaction. A transaction keeps a Metahandle on each SHandle to be able to commit changes in the commit phase (using vaporize).

- **SHandle**: it manages changes at object level. The SHandle shadows all state changes at reference level. The SHandle framework provides the automatic propagation of the state shadowing properties during the execution of the *atomic block*.

It should be stressed that shandle itself is the base for the implementation. A shandle keeps its own version of the state so that the Handle shadows state changes and makes sure that they are not visible until the commit. Second, a shandle provides propagation: in a case of STM, each instance variabled access is automatically wrapped and object read are registered with the transaction (to detect conflict). In addition object reads are registered to the factory to manage conflict detection and resolution. Basically, a transaction keeps a shandle for each object. In addition, the factory keeps a copy of the original object for each object to detect conflicts.

At the end of a transaction (*Optimistic Conflict Detection*), we check if a conflict is present. To do this, we should check two things: first, if the original object has been changed by other computation during our transaction, then we immediately reset and re-execute the transaction (*strong atomicity*). As second step we check if another transaction has used one of the objects changed by the transaction. If this is the case, we abort all problematic transactions. It is important to notice that here the two tests are required because during transaction state modification is not visible until the commit.

Now we show how support nested transaction with our implementation.

Nested. The idea behind a *closed* Nested STM that all the side effects of the nested transaction should be injected in the upper transaction (that can be nested also). All side effects will be installed in the real objects only at the end of the most outer transaction. Let us illustrate that by an example in the following code :

```
point := 1@1.
TransactionA := Transaction
   newWith:[:thePoint |
   thePoint x: (thePoint x + 1). "Step 1"
   TransactionB := Transaction
      newWith:[:thePointAgain |
      thePointAgain x: (thePointAgain x + 1) "Step 2"]
      references: {thePoint}.
      TransactionB run.
      thePoint x: (thePoint x * 2) "Step 3"]
references:{point}.
TransactionA run.
```

In the previous code we define two transactions A and B, the transaction B is created inside the transaction A, the transaction B is a nested transaction of transaction A. This means that all the side effects that occur in transaction B should be committed in the transaction A like any other state change of transaction A.

The solution for realizing nested transactions follows naturally from the properties of the Shandle framework itself. As seen in Figure 6.8, we create a shandle on the shadowed state (through the metahandle). When transaction B will end and commit its changes, all the changes will be automatically forwarded to the upper transaction.

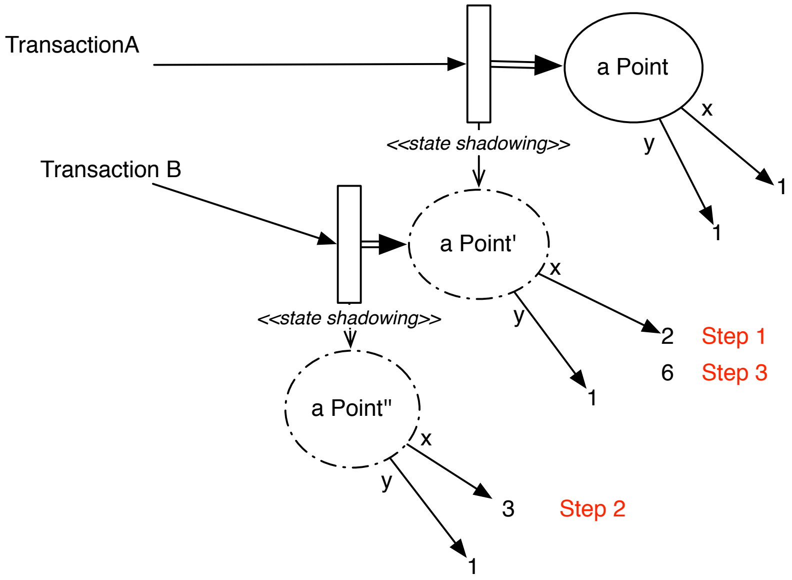


Figure 6.8: Shandle support for nested transaction

6.3.3 STM: conclusion

Weakness of the implementation. Again, it should be made clear that our goal was not to provide a fully engineered real world implementation of STM. Especially, the transactional factory should have been managed by the virtual machine. In our implementation we focus on the state aspect of transactional memory instead to the concurrency aspects.

As we have presented, we can support Software Transactional Memory using shandles. The implementation supports *strong atomicity*, *optimistic conflict detection* and *object level granularity*. For data versioning we use *shandle* and finally we support *closed nesting*. In the following section, we present the second validation, how to support Worlds [Alessandro Warth 2011] using shandle.

6.4 Validation: worlds

As a second validation, we show how to implement *Worlds* to control side effects using SHandle.

Worlds is a language construct that reifies the notion of program state [Warth 2008, Alessandro Warth 2011]. Worlds can be used to control the scope of side effects. The principle is that a computation can be performed in a world enclosing it which isolates all changes from the rest of the program execution. Worlds can be nested.

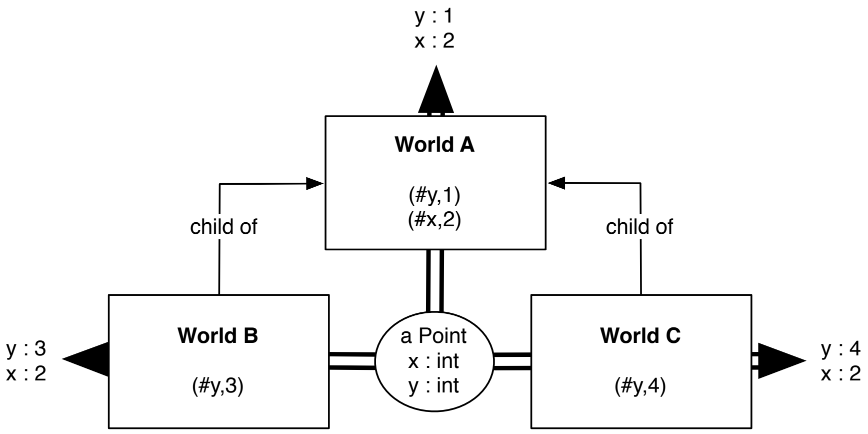


Figure 6.9: World principle [Warth 2008]

A world encloses all the side effects occurring during its computation, as we illustrated in Figure 6.9. Imagine that we have a Point, which has two integers x and y. In world A the value for this point is (1,2), while in another one it can have a different value. The same *Point* which is defined in world A, can be changed in world B. For example, we can change x to 3. The change will be limited to B. If a value is not changed, the access will be delegated to world A. Moreover, a world can inherit from another world. To summarize the value of different instance variables of the same point can be different depending of the world as we can see in Table 6.1:

Worlds can have multiple uses in practice, as shown by Warth [Warth 2008]. Examples are exceptions that restore state when raised, extension methods for JavaScript or reversible

World	A	B	C
X value	2	2 (found in A)	2 (found in A)
Y value	1	3	4

Table 6.1: Different values existing for the same Point depending of the world A, B or C

computation in general. One obvious application of Worlds is sandboxing.

What we propose is to implement the Worlds model using SHandle. The World infrastructure at the reference-level will provide a clear separation between all the World infrastructure and base object behavior.

6.4.1 Worlds and Shandles

For implementing Worlds, we follow the original implementation [Alessandro Warth 2011]. First, the idea is to capture all the instance variables or global read access to provide hooks for the lookup in the World hierarchy.

Our implementation of Worlds is based on two classes, World and a special SHandle.

World. World is the object representing a world in the system, it is used as a factory to create and manage the shandle. World objects have two responsibilities, first *finding the value of the field accessed* as we explained before. In the original model the changes are kept local to the current world and delegated the un-changed instance variable to the parent World. Second, a world needs to *maintain isolation* of changes. A world is used as a factories to maintain an isolation of state at level of the world (we used the same pattern for STM present in previous section and revocable reference in Chapter 5).

Shandle. We used shandle to keep the *state isolated from the world* and the propagation mechanism to *search the value of instance variable in the current world*. The state kept by the SHandle is used to keep all modifications performed on a specific object. To keep the *copy-on-write* property of the world model [Warth 2008], we have an initial state, each new shandle has all instance variables (and indexed variable in our implementation) replaced by an association to a mock object (unique per world) and the index of the current instance variable (with a prefix for indexed field).

The mock is used do detect when the value should be found in the parent world, and the index that represents which instance variable should be accessed.

Detailed approach. In detail, when a object is accessed in a world we create a shandle on it and initialize all instances variables with a tuple composed with a mock object (unique per world, ensure that pattern is unique) and the index of this instance variable (see Figure 6.10). This tuple represents the access itself and not the value.

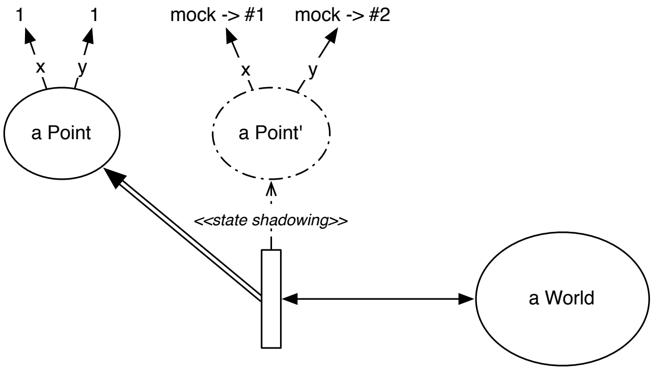


Figure 6.10: World initial state of the shandle

The propagation method will request to the world the value accessed. The world receives a message with, the shandle (to identify who is the receiver) and the accessed value. There are two cases: first we have a normal value then return it. Second we have a mock object then begin the lookup process by using the shandle to identity of the accessed object and the index that should be accessed to the parent world. We repeat the process until a value is found. Once we found a value we wrap the value into a shandle or if it was already wrapped before we return the value stored into the world.

Commit into the upper world. As in the original model, the commit phase should be confined to the direct parent world, we choose to use the same structure we presented before in Section 6.3.2 (see Figure 6.11). Just before the commit phase the world will request all the values in the direct parent world store it in this own object state and using metahandle fix the target of the shandle to point to the shadow state of the direct parent world. Finally we use the commit method to install the state in the parent world. In addition the original implementation required to check if the committed objects have been changed in the parent since the last read. To do that, we maintain a copy of each object read and as in original model if this object as been changed in parent world we failed the commit and raised a exception (as in original model). In Figure 6.11 after the commit phase of the world B, in the world A, *a Point* will have *2* and *2* respectively in *x* and *y*.

Global variable and thisWorld. To wrap global variable and to implement the message *thisWorld* proposed in the original implementation, we used the shadow behavior abilities of shandle to redirect the behavior to class created from the original one.

Worlds: conclusion. We show that we can implement Worlds using shandles. Our implementation has the possibility to be used on every objects (regardless to its inheritance tree). This represents an enhancement on the original model (based on inheritance).

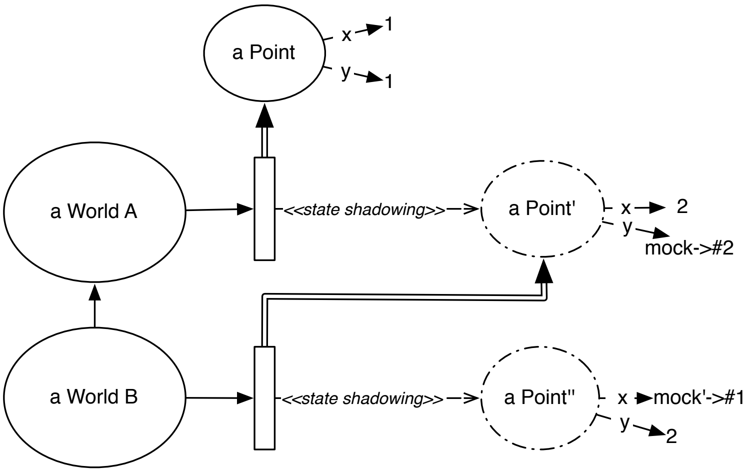


Figure 6.11: Inheritance in our Worlds implementation

6.5 Conclusion

In this chapter, we extended the Handle model previously presented in Chapter 5 to include the ability to isolate state at reference-level. We proposed a formal model to clearly describe the proposed model. We validate our extension by implementing two existing approaches software transactional memory [Shavit 1995] and Worlds [Warth 2008]. In the next chapter, we will present the implementation choices and advances we realized.

CHAPTER 7
Implementation

Contents

In the previous chapter we have presented the Stateful Handle model that supports state shadowing per reference and automatically propagating of properties (related to behavior and state shadowing) to a complete subgraph.

Besides the conceptual model, a large part of our work was to extend the Pharo Virtual machine to be able to implement the Handle and SHandle models. The changes require by Handle and SHandle models to be implemented in a language are:

An infrastructure for handles. We need to be able to manipulate the handle from the virtual machine.

Manage the message passing. We need to change *how* the message passing related bytecode are interpreted by the virtual machine to deal with the special case of handles.

Manage the state access. We need to change *how* states are accessed when accessed via the handle.

Provide the propagation infrastructure. We need to create the infrastructure and a solution to propagate properties in the virtual machine.

Manage the primitives. There are some existing primitive functions, which by definition can be called on any object. We need to manage them to provide the expected behavior when perform on a handle or with a handle as an argument.

Offer a languages side interface. We need to provide different language constructs that allow the create and manipulate the handle from the language side (not only inside the virtual machine).

The goal of this chapter is to discuss some implementation aspects. First, we present the context of the implementation, the Pharo Virtual Machine, then we discuss what we had to change at the virtual machine level to support Handle. Finally, we will explain how to use the Handle prototype from the Pharo environment itself.

7.1 Virtual machine

Pharo is a modern open-source development environment, it is a fork of Squeak Smalltalk, a language derived originally from Smalltalk-80 [Goldberg 1983].

The Pharo language and the environment evolved already considerably compared to Smalltalk-80. Over the last years especially the build environment for the VM improved considerably: use of continuous integration, up-to-date documentation and a user friendly build process that allows anyone to compile its own virtual machine in less than half a hour. Compared to this, when the work on handles was started in 2009, the lack of documentation and a complicated build infrastructure meant that a considerable effort was needed to extend the virtual machine for Handles.

VMMaker technology. The Pharo virtual machine build environment is based on a tool (VMMaker) that is used for generating the C source files of the virtual machine. The idea behind VMMaker, is to implement the Smalltalk virtual machine in a subset of Smalltalk. Basically VMMaker generates C files for the interpreter and for all additional plugins. The code is writen in Slang, which is a subset of Smalltalk that can be easily compiled to C [Ingalls 1997, Guzdial 2001].

7.2 Handle: virtual machine part

In our implementation, the live cycle of the Handle has two steps: first a *configuration* part where the handle is still a normal object. In the configuration step we can fill all the instances variables and prepare the handle to be used. Second, as part of the *activation* step, the handle is turned into a real Handle: from that point it is a reference from the virtual machine point of view.

In addition, to simplify benchmarking and handle manipulation, we can configure the handle in real time. We can activate or deactivate shadow behavior, state shadowing and finally propagation.

Our representation of a handle is an object that have four instances variables:

targetObject is the target object that the handle points to.

shadowBehavior is the class where the method lookup will be redirected if the shadow behavior is activated.

state is where the state is kept if the state shadowing is activated.

configuration is a simple bit word, the value represents the configuration of the handle.

Once all these fields are correctly filled, we can activate the handle so that it is turned into a real reference for the virtual machine (represent the target object and has the handle expected behavior).

Activation primitive. The following code is the actual code of the activation primitive, which is the primitive used turn on the *activation* phase:

```
1  primitiveAsHandle
2     | rcvr |
3     <inline: true>
4     <export: true>
5     rcvr := self stackTop.
6     (self isInstanceOfHandle: rcvr) ifTrue: [
7        (self isActiveMetaHandle: rcvr)
8           ifTrue: [^self primitiveFail.]
9           ifFalse:[objectMemory setHandleBit: rcvr]].
```

The pragmas <inline: true> and <export: true> are keywords for the C generator. export declares that the method is a named primitive that can be called from the image. The pragma <inline: true> tells the C generator that the code should be inlined in the caller.

In the previous code we first take the top of the the stack that is the receiver of the primitive, check if this receiver is an inactive (line 7) handle (line 6). If these two conditions are respected, activate the handle by changing a bit in the object header using setHandleBit:. Once activated a handle cannot be deactivated and its behavior is the one presented in Chapter 6.

7.2.1 Behavior

In this section, we explain what do we changed in virtual machine to support the handle shadow behavior properties.

Handle send. As we presented before when a message is received by a handle, the method lookup should begin in the shadow behavior class. To implement shadow behavior properties, we override the normalSend method that implements the message send bytecode. The exact change can be seen in the following code:

```
normalSend
   | rcvr |
   <sharedCodeNamed: 'normalSend' inCase: 131>
   rcvr := self internalStackValue: argumentCount.
   self assert: lkupClass ~~ objectMemory nilObject.
   (self internalIsHandle: rcvr) ifTrue:[
      (self handleInlineUseBehavior: rcvr) ifTrue:[
         lkupClass := (self handleClassLookupOf: rcvr).]
      ifFalse: [
         lkupClass := objectMemory fetchClassOf: (self
handleReceiverOf: rcvr)]]
   ifFalse:[
      lkupClass := objectMemory fetchClassOf: rcvr].
   self commonSend.
```

In the previous code the lines 1 to 5 are identical to the original code. Then Line 6, we test if the message send has been performed on an activated handle. If it is the case, we check if the current handle actually uses the shadow behavior properties (line 7). Here again if, it uses the shadow behavior then we initialize the method lookup from the class returned by the method handleClassLookupOf:. Otherwise we initialize it, with the class of the target object (bypassing the handle) as we can see in line 10. Else if the first test returns false (line 6), as the receiver is not a handle, we just reconnect with the normal object send process (lines 11 and 12). It is important to notice we do not change the receiver, we only change where the method lookup begins.

It should be noted that there are in the virtual machine several methods managing message passing (most of them for optimization). We do not present all the code in this section as all these methods are changed the same way following the example shown above.

7.2.2 State and propagation

To manage State shadowing and propagation, in the virtual machine we intercept all the virtual machine behavior (primitives, bytecode, etc.) related to, (i) storing a value in a instance variable or indexed variable, (ii) all behavior related to push a instance or indexed variable on stack.

As an example we will show the most interesting case, the pushReceiverVariable: method (called when the related bytecode is executed and when instance variables are read using the reflective API).

```
pushReceiverVariable: fieldIndex
| rcvr value |
rcvr := self receiver.
(self internalIsHandle: rcvr) ifTrue:[
   self hPushReceiverVariable: fieldIndex on: rcvr]
ifFalse: [
   value := (objectMemory fetchPointer: fieldIndex ofObject: rcvr ).
   self internalPush: value]
```

First of all we need to determine if the pushReceiverVariable: request is performed on a handle or a normal object. Obviously in the case of an handle we need a special treatment: we call hPushReceiverVariable:on:. In case of a normal object, we just execute the original code.

In the virtual machine the state shadowing implementation and the propagation is highly connected as they happen at the same time. Figure 7.1 shows the process when a instance variable or an indexed field (for Arrays like objects) are pushed on the stack, that is when instance or indexed variables are read.

As we can see in Figure 7.1, there are two steps. First, we need to determine where to fetch the value, second the value needs to be propagated (or not). The first step is simple: we fetch a value in the state instance variable of the handle (if state shadowing is activated),

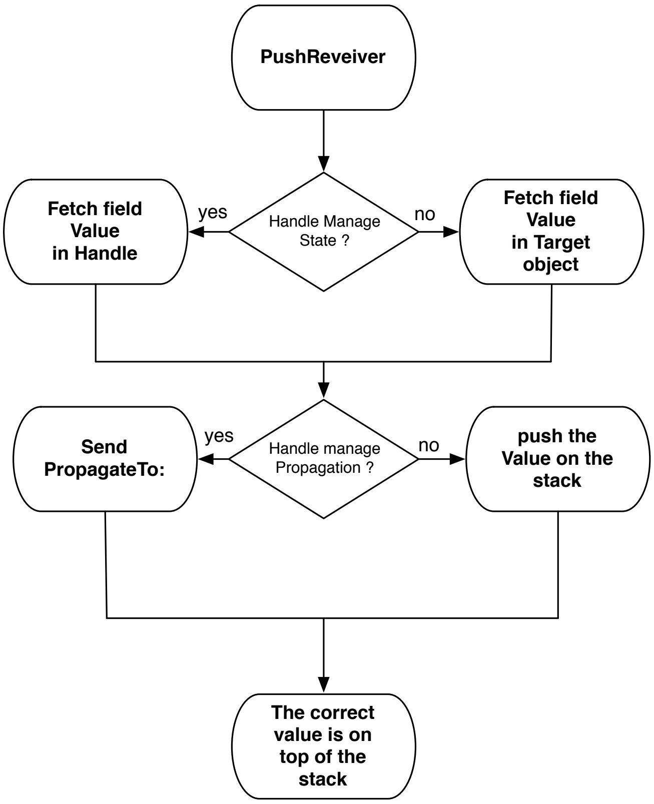

PushReveiver
Handle Manage State ?
yes
no
Fetch field Value in Handle
Fetch field Value in Target object
Handle manage Propagation ?
yes
no
Send PropagateTo:
push the Value on the stack
The correct value is on top of the stack

Figure 7.1: State and Propagation Process

or we fetch the standard target object state. In a second step, we check if the value should use the propagation mechanism or not.

Propagation mechanism. We wanted to have a really simple and effective propagation mechanism, the solution chosen is the following. Each handle on their class side declares a propgatedTo: method that takes one argument, anObject which is the value that should be propagated by the handle behavior. To illustrate the mechanism, we show the code implementing propagation for readOnly references:

```
propagateTo: anObject
      ^ReadOnlyHandle for: anObject.
```

As we can see in the previous code, we just initialize a new handle for anObject and return it. In the virtual machine, the code is the following:

```
pushHandle: aHandle propagateValue: aObject
|   handleClass handleMetaClass |
 <inline: false>
 <export: true>
```

```
5     "initialize Part"
6     handleClass := objectMemory fetchClassOf: aHandle.
7     handleMetaClass := objectMemory fetchClassOf: handleClass.
8     self push:  handleClass.
9     self push: aObject.
10    messageSelector := self handlePropagateSelector.
11    argumentCount := 1.
12    "Sending message Part"
13    "From here  "
14    self lookupMethodInClass: handleMetaClass.
15    lkupClass := handleMetaClass.
16    self activateNewMethod.
```

Basically, in lines 6 and 7 we fetch all the required values (handle class and handle metaclass, where the method lookup begins). Then in line 8 we push the receiver, in line 9 we push the argument (the value of the instance variable or the indexed variable fetched previously). Then we initialize the virtual machine variables (argument counter and sector that should be used that is a constant in this case). And finally we do the method lookup and activate the method found by its lookup.

When the message finishes to be executed, we will have the return value on top of the stack.

7.2.3 Primitives

Primitives are functions in the virtual machine that are called directly from Smalltalk. The VM has three kinds of primitives: numbered (old primitives), named primitives, and named primitives stored in plugins. All counted, there are more or less two hundred primitives. As a handle is a special object, we need to manage the primitives specially. For each of them, we should read it to understand what they do and determine the correct behavior expected when performed on the handle.

For the existing primitives, we identify two cases: either we simply redirect the behavior on the target object (most of the identified cases), or we create a complete new primitive with the expected behavior for the handle. The last case is needed for primitives that are part of the reflective API, all together around thirty primitives.

But in addition of the existing primitives, additional primitives are needed for a fully functional implementation. Examples are primitives such as the new identity primitive, installing primitive for stateful handle and of course the activation primitive shown earlier. The complete list of new primitives:

primitiveAsHandle activates the receiver if is it a handle.

primitiveInstall installs (see Chapter 6) shandle in one step.

primitiveInstallAll installs (see Chapter 6) a full array of handles in one step.

primitiveSameObject (==, see Chapter 6) checks if the argument and the receiver represent the same object, that is that the two objects point on the same *target object*. The two references can have different behavior or different state.

primitiveIdentical (===, see Chapter 6) checks if the argument and the receiver are the same reference. They have the same state and same behavior.

We have described briefly all the underlying mechanisms to support the Handle implementation from a virtual machine point of view. Now we will explain how this infrastructure is used from within the Pharo environment.

7.3 Handle: image part

Pharo Smalltalk stores the entire program state (including both class and non-class objects) in an image file. The image is a snapshot of the complete system state that can be reloaded by the virtual machine.

We already discussed primitives which are a set of procedures defined as part of the virtual machine that can be called from the image. In addition, there has to be a way to share objects easily between the virtual machine world and the image. For example, we need to be able to access the class that represents handles from the VM side.

7.3.1 Special objects array

The *Special objects array* is the structure that enables a pre-defined list of objects to be statically referenced in the virtual machine. It is used to identify and manipulate these objects. For more information, please refer to section *Objects Used by the Interpreter* in the bluebook [Goldberg 1983].

In our implementation we use this mechanism. We register the Handle class and the selector of the method used to propagate behavior. In addition we have a meta API on the image side to provide a flexible way to manage this information. For example we can dynamically change the Handle class and the method used for propagation. This flexibility is important for debugging and experimentation: the virtual machine does not need to be recompiled even when the Handle class is changed.

7.3.2 Handle implementation

In this section, we take a look to the Handle implementation in the image side. We do not go in detail, instead we give an overview of the interface provided by the Handle class and instance side.

Handle instance side. The handle default behavior is divided in four categories, (i) *Activating* that contains only the primitive used to activate the handle. (ii) *Options*, containing all the methods used to activate the different options. (iii) *Initialize*, containing all method used to initialize the method, we declare all the steps of the process to be able to overwrite it in a subclass without redefining all the process. (iv) *Testing*, containing a set a method used to identify the handle. All the behavior contained in the four categories can only be send to a not-activated handle or using a Metahandle.

Handle class side. The handle class default behavior is divided in two categories, (i) *Instance creation*, containing several methods to create an instance of the handle with different configurations. (ii) *Propagation*, containing only the mandatory propagateTo: method. The handle class side defines syntactic sugar to regroup the different configurations of handles and the propagation method.

The implementation of handle and metahandle is simple, on the instance side there are eighteen methods (including the accessors) and on the class side only six methods (five syntactic sugar instance creation methods and the propagateTo: default behavior).

Metahandle. The metahandle in our implementation is just a handle on a handle. We create a subclass to able to distinguish handles and metahandles and give a structure for subclassing. As we explained before we can only create metahandle on a not-activated handle.

7.4 Conclusion

In this chapter we presented the logic behind all the changes we introduced in the virtual machine. We discussed all the language side infrastructure used to manipulate the Handle concept. In the next chapter, we present possible future work, both concerning engineering improvements and new directions for research.

CHAPTER 8
Conclusion

In this chapter we briefly summarize the contributions of the thesis and discuss possible future work.

8.1 Contributions of the thesis

The contributions of this thesis are:

- *Dynamic read-only objects for dynamically-typed languages* [Arnaud 2010]. We propose dynamic read-only objects (DRO) as an approach to provide read-only behavior at the level of references (Chapter 4). Our approach offers a way to create read-only references to any object in the system. In addition, our approach propagates the read-only property to the object graph accessed through these references.

- *Handles: behavior-propagating first-class references for dynamically-typed languages*. We generalize the DRO model and enable behavioral changes. We extend the Pharo environment and programming language[1] with *Handles*, *i.e.*, first-class references that have the ability to change the behavior of referenced objects (Chapter 5).

- *Metahandles: controlling Handles at runtime*. We define Metahandle to add flexibility and adaptability to controlled references (Chapters 5 and 6).

- *SHandles: controlling the visibility of side effects*. We propose SHandle, an extension of the Handle model to isolate side effects at the level of references (Chapter 6).

- *A working implementation*. As proof of concept, we extend the Pharo virtual machine to support Handles, Metahandles and SHandles (Chapter 7).

- *A formal model*. We formalize the Handles and SHandle models to represent and explain their semantics (Chapter 5 and Chapter 6).

[1] http://www.pharo-project.org

8.2 Discussion

In the following, we discuss some interesting points that concern the design choices of the handle model.

Handle composition. Handles and SHandle do not allow the possibility to create a handle to an already *activated* handle. This means it is not possible to change the behavior of a handle by composing handles. We limited the models explicitly at this step to enforce that it is not possible to change the handle behavior if not planned (by using a metahandle before the handle activation). In addition as we explained, our approach is to offer a reference mechanism that holds and propagate properties, not to define the properties themselves. Therefore we cannot control if a given property defined by the language designer can be composed with another one. The possible change is just too large. One idea is to restrict what a handle (or shandle) changes and to declaratively specify these changes. With this, we plan to explore the idea of chaining handles.

Deactivating a handle. Once *activated* a handle (or shandle) cannot be deactivated. It is a design choice to ensure that handle behavior cannot be changed. We provide a way to control the handle via a metahandle but a metahandle can only be activated on a non active handle. This means that handle control needs to be planned in advance.

Storing handles. In the current models, we can store handles in instance variables. We do not de-wrap handles on store. The reason is that the target where the handle is stored could be accessed by a non restricted client and this would lead to a leak to the target object protected by the handle. Imagine that we have a revocable reference (see Chapter 5) to a document, storing such a document in an instance variable should preserve the revocable property since this revocable reference could be stored on an object accessed without a revocable property. We should be able to revoke the reference and the store instance variable should hold a revoked reference and raise the expected errors.

Concurrent use. Since handles may introduce behavior changes and different threads can create and access through different handles, it may happen that some problems arise leading to incoherent behavior. We minimize the problem of incompatibility by, with the current model, not allowing handles to be chained. In addition, we believe that this point is linked to the semantics of the constructs implemented using handles and how different clients can get access to different semantics. A handle ensures that the target object cannot escape the handle boundary, it does not prevent mixing incompatible semantics.

General safety and reflection. The claim of Handles is not that because we use them, a language implementing them will be safer. Handles only ensure that their target object cannot leak and that the handle behavior is used in place of target one. We believe that Handle can be part of the solution by supporting the control of the reflective features offered.

8.3 Future work

The SHandle model can be enhanced in several ways. We can distinguish two mains categories, for one engineering level improvements related to the implementation, and on the other hand direction for future research.

8.3.1 Engineering

Performance The current implementation leads to a 15% slowdown even when not using handles, more when handles are used. There are some interesting ways to speed up execution, for example by providing different code depending on if handles are active or not. Especially the JIT generated code is interesting in this regard. We have not yet used at all the fact that there is a runtime translation happening in the system before execution.

Interpreter Even the interpreter implementation can be improved, we plan to create a separate interpreter to only be active on demand.

Refactoring After the last iteration, the code provides some simplification opportunities of subsystems coming from the older versions.

8.3.2 Research

Shadow classes: Shadow classes should be made more general. One idea is to model the shadow class using handles themselves. This means, a Shadow class is just a Handle on a class, modeling all differences to the class as part of the state shadowing of the target object (the class). Figure 8.1 shows a possible solution.

State representation State representation should improved. Currently, we copy the target object. Instead, it should be possible to have a better structure. This would for one save memory, but in addition would allow for example to associate meta data much easier.

Composition model: We need a way to compose handles. How can we associate the behavioral change of two different kinds of handles at the same time to one object without conflicts?

Safe Reflection Handles are a low level mechanisms that the underlying language should have to get a chance to offer safer dynamic and reflective behavior. But how provide secure reflection?

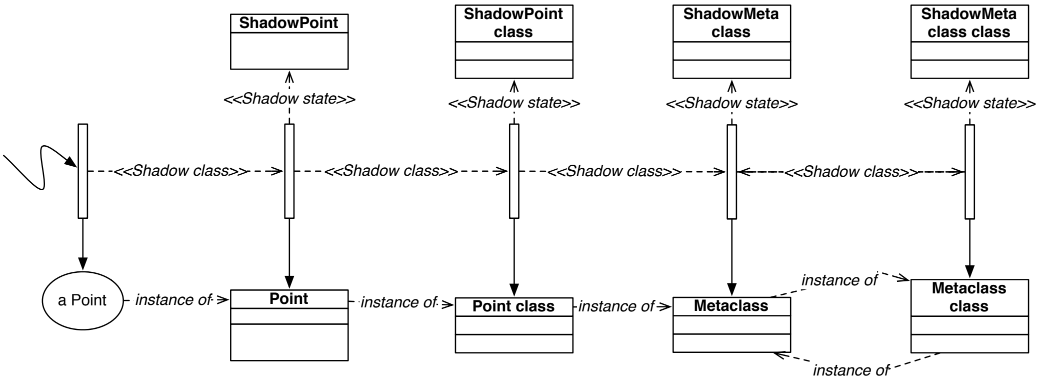


Figure 8.1: Shadow class as Handle on a class

Besides improves the handle model (research and engineering), another important future work is to use the handle model for large systems. For example the possibility to control side effects is very interesting for testing or developing system while it is running. In both cases we want to be able to run code while making sure that the main system is not impacted.

Appendix

APPENDIX A

SMALLTALKLITE

Contents

The present appendix provides a description of SMALLTALKLITE [Bergel 2008]. This is not a contribution of this thesis and we added it to help the reader. SMALLTALKLITE is based on the model used by Flatt *et al.* [Flatt 1998]. SMALLTALKLITE is a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and **super** sends. SMALLTALKLITE is similar to CLASSICJAVA, but removes interfaces and static types. Fields are private in SMALLTALKLITE, so only local or inherited fields may be accessed.

A.1 SMALLTALKLITE reduction semantics

The syntax of SMALLTALKLITE is shown in Figure A.1. SMALLTALKLITE is similar to CLASSICJAVA, while omitting the features related to static typing.

To simplify the reduction semantics, SMALLTALKLITE adopts an approach similar to that used by Flatt *et al.* [Flatt 1998]. It annotates field accesses and **super** sends with additional static information that is needed at "run-time". This extended redex syntax is shown in Figure A.1. The figure also specifies the evaluation contexts for the extended redex syntax in Felleisen and Hieb's notation [Felleisen 1992].

Predicates and relations used by the semantic reductions are listed in Figure A.3. (The predicates CLASSESONCE(P) *etc.* are assumed to be preconditions for valid programs, and are not otherwise explicitly mentioned in the reduction rules.)

$P \vdash \langle \varepsilon, \mathcal{S} \rangle \hookrightarrow \langle \varepsilon', \mathcal{S}' \rangle$ means that we reduce an expression (redex) ε in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new expression ε' and (possibly) updated store $\mathcal{S}'$. A redex ε is essentially an expression e in which field names are decorated with their object contexts, *i.e.*, f is translated to $o.f$, and **super** sends are decorated with their object and class contexts. Redexes and their subexpressions reduce to a value, which is either an object identifier or nil. Subexpressions may be evaluated within an expression context E.

The store consists of a set of mappings from object identifiers *oid* $\in$ dom$(\mathcal{S})$ to tuples

$$\begin{array}{rcl}
P & = & \mathit{defn}^* e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^* \mathit{meth}^*\ \} \\
e & = & \textbf{new}\ c \ \mid\ x \ \mid\ \textbf{self} \ \mid\ \mathsf{nil} \\
 & \mid & f \ \mid\ f{=}e \ \mid\ e.m(e^*) \\
 & \mid & \textbf{super}.m(e^*) \ \mid\ \textbf{let}\ x{=}e\ \textbf{in}\ e \\
\mathit{meth} & = & m(x^*)\ \{\ e\ \} \\
c & = & \text{a class name} \ \mid\ \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}
\qquad
\begin{array}{rcl}
\varepsilon & = & v \mid \textbf{new}\ c \ \mid\ x \mid \textbf{self} \mid \varepsilon.f \mid \varepsilon.f{=}\varepsilon \\
 & \mid & \varepsilon.m(\varepsilon^*) \mid \textbf{super}\langle o, c\rangle.m(\varepsilon^*) \mid \textbf{let}\ x{=}\varepsilon\ \textbf{in}\ \varepsilon \\
E & = & [\,] \mid o.f{=}E \mid E.m(\varepsilon^*) \mid o.m(v^*\ E\ \varepsilon^*) \\
 & \mid & \textbf{super}\langle o, c\rangle.m(v^*\ E\ \varepsilon^*) \mid \textbf{let}\ x{=}E\ \textbf{in}\ \varepsilon \\
v, o & = & \mathsf{nil} \mid \mathit{oid}
\end{array}$$

Figure A.1: SMALLTALKLITE syntax and Redex syntax

$$\begin{array}{rcl}
o[\![\textbf{new}\ c']\!]_c & = & \textbf{new}\ c' \\
o[\![x]\!]_c & = & x \\
o[\![\textbf{self}]\!]_c & = & o \\
o[\![\mathsf{nil}]\!]_c & = & \mathsf{nil} \\
o[\![f]\!]_c & = & o.f \\
o[\![f{=}e]\!]_c & = & o.f{=}o[\![e]\!]_c \\
o[\![e.m(e_i^*)]\!]_c & = & o[\![e]\!]_c.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{super}.m(e_i^*)]\!]_c & = & \textbf{super}\langle o, c\rangle.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{let}\ x{=}e\ \textbf{in}\ e']\!]_c & = & \textbf{let}\ x{=}o[\![e]\!]_c\ \textbf{in}\ o[\![e']\!]_c
\end{array}
\quad
\begin{array}{rcl}
\textbf{new}\ c\ [v/x] & = & \textbf{new}\ c \\
x\ [v/x] & = & v \\
x'\ [v/x] & = & x' \\
\textbf{self}\ [v/x] & = & \textbf{self} \\
\mathsf{nil}\ [v/x] & = & \mathsf{nil} \\
f\ [v/x] & = & f \\
f{=}e\ [v/x] & = & f{=}e[v/x] \\
e.m(e_i^*)\ [v/x] & = & e[v/x].m(e_i^*[v/x]) \\
\textbf{super}.m(e_i^*)\ [v/x] & = & \textbf{super}.m(e_i^*[v/x]) \\
\textbf{let}\ x{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x{=}e[v/x]\ \textbf{in}\ e' \\
\textbf{let}\ x'{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x'{=}e[v/x]\ \textbf{in}\ e'[v/x]
\end{array}$$

Figure A.2: Translating expressions to redexes (left) and variable substitution (right)

$\langle c, \{f \mapsto v\}\rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

Translation from the main expression to an initial redex is specified by the $o[\![e]\!]_c$ function (see Figure A.2). This binds fields to their enclosing object context and binds **self** to the *oid* of the receiver. The initial object context for a program is nil. (*i.e.*, there are no global fields accessible to the main expression). So if e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial redex.

The reductions are summarized in Figure A.4.

new c [*new*] reduces to a fresh *oid*, bound in the store to an object whose class is c and whose fields are all nil. A (local) field access [*get*] reduces to the value of the field. Note that it is syntactically impossible to access a field of another object. The redex notation $o.f$ is only generated in the context of the object o. Field update [*set*] simply updates the corresponding binding of the field in the store.

When we send a message [*send*], we must look up the corresponding method body e, starting from the class c of the receiver o. The method body is then evaluated in the context

$\prec_P$	Direct subclass $c \prec_P c' \iff$ **class** c **extends** $c' \cdots \{\cdots\} \in P$
$\leq_P$	Indirect subclass $c \leq_P c' \equiv$ transitive, reflexive closure of $\prec_P$
$\in_P$	Field defined in class $f \in_P c \iff$ **class** $\cdots \{\cdots f \cdots\} \in P$
$\in_P$	Method defined in class $\langle m, x^*, e\rangle \in_P c \iff$ **class** $\cdots \{\cdots m(x^*)\{e\} \cdots\} \in P$
$\in^*_P$	Field defined in c $f \in^*_P c \iff \exists c', c \leq_P c', f \in_P c'$
$\in^*_P$	Method lookup starting from c $\langle c, m, x^*, e\rangle \in^*_P c' \iff c' = min\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$
ClassesOnce(P)	Each class name is declared only once $\forall c, c'$, **class** $c \cdots$ **class** $c' \cdots$ is in $P \Rightarrow c \neq c'$
FieldOncePerClass(P)	Field names are unique within a class declaration $\forall f, f'$, **class** $c \cdots \{\cdots f \cdots f' \cdots\}$ is in $P \Rightarrow f \neq f'$
FieldsUniquelyDefined(P)	Fields cannot be overridden $f \in_P c, c \leq_P c' \implies f \notin_P c'$
MethodOncePerClass(P)	Method names are unique within a class declaration $\forall m, m'$, **class** $c \cdots \{\cdots m(\cdots)\{\cdots\} \cdots m'(\cdots)\{\cdots\} \cdots\}$ is in $P \Rightarrow m \neq m'$
CompleteClasses(P)	Classes that are extended are defined range($\prec_P$) $\subseteq$ dom($\prec_P$) $\cup$ {Object}
WellFoundedClasses(P)	Class hierarchy is an order $\leq_P$ is antisymmetric
ClassMethodsOK(P)	Method overriding preserves arity $\forall m, m', \langle m, x_1 \cdots x_j, e\rangle \in_P c, \langle m, x'_1 \cdots x'_k, e'\rangle \in_P c', c \leq_P c' \implies j = k$

Figure A.3: Relations and predicates for SmalltalkLite

of the receiver o, binding **self** to the receiver's *oid*. Formal parameters to the method are substituted by the actual arguments (see Figure A.2). We also pass in the actual class in which the method is found, so that **super** sends have the right context to start their method lookup.

super sends [*super*] are similar to regular message sends, except that the method lookup must start in the superclass of class of the method in which the **super** send was declared. When we reduce the **super** send, we must take care to pass on the class c'' of the method in which the **super** method was found, since that method may make further **super** sends. **let in** expressions [*let*] simply represent local variable bindings.

Errors occur if an expression gets "stuck" and does not reduce to an *oid* or to nil. This may occur if a non-existent variable, field or method is referenced (for example, when sending any message to nil). In this appendix we are not concerned with errors, so we do not introduce any special rules to generate an error value in these cases.

$$
\begin{array}{lll}
P \vdash & \langle E[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle E[\mathit{oid}], \mathcal{S}[\mathit{oid} \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in^{*}_{P} c\}\rangle]\rangle & [\mathit{new}] \\
& \text{where } \mathit{oid} \notin \mathrm{dom}(\mathcal{S}) & \\
P \vdash & \langle E[o.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle & [\mathit{get}] \\
& \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = v & \\
P \vdash & \langle E[o.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle & [\mathit{set}] \\
& \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle & \\
P \vdash & \langle E[o.m(v^{*})], \mathcal{S}\rangle \hookrightarrow \langle E[o\llbracket e[v^{*}/x^{*}]\rrbracket_{c'}], \mathcal{S}\rangle & [\mathit{send}] \\
& \text{where } \mathcal{S}[o] = \langle c, \mathcal{F}\rangle \text{ and } \langle c, m, x^{*}, e\rangle \in^{*}_{P} c' & \\
P \vdash & \langle E[\textbf{super}\langle o, c\rangle.m(v^{*})], \mathcal{S}\rangle \hookrightarrow \langle E[o\llbracket e[v^{*}/x^{*}]\rrbracket_{c''}], \mathcal{S}\rangle & [\mathit{super}] \\
& \text{where } c \prec_{P} c' \text{ and } \langle c', m, x^{*}, e\rangle \in^{*}_{P} c'' \text{ and } c' \leq_{P} c'' & \\
P \vdash & \langle E[\textbf{let}\ x{=}v\ \textbf{in}\ \varepsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\varepsilon[v/x]], \mathcal{S}\rangle & [\mathit{let}]
\end{array}
$$

Figure A.4: Reductions for SMALLTALKLITE

Bibliography

[Abowd 2000] Gregory D. Abowd and Anind K. Dey. *Towards a Better Understanding of Context and Context-Awareness*. In Proceedings of the CHI 2000 Workshop on the What, Who, Where, When and How of Context-Awareness. ACM Press, New York., 2000. 23

[Alessandro Warth 2011] Yoshiki Ohshima Alessandro Warth, Ted Kaehler and Alan Kay. *Worlds: Controlling the Scope of Side Effects*. In Proceedings of the 25th European Conference on Object-Oriented Programming(ECOOP'11). LNCS, 2011. 27, 65, 78, 79

[Almeida 1997] Paulo Sérgio Almeida. *Balloon types: Controlling sharing of state in data types*. In Proceedings of ECOOP '97, LNCS, pages 32–59. Springer Verlag, June 1997. 1, 3, 26, 29

[Ancona 2007] Davide Ancona, Massimo Ancona, Antonio Cuni and Nicholas D. Matsakis. *RPython: a step towards reconciling dynamically and statically typed OO languages*. In DLS '07: Proceedings of the 2007 symposium on Dynamic languages, pages 53–64, New York, NY, USA, 2007. ACM. 1

[Arnaud 2010] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel and Mathieu Suen. *Read-Only Execution for Dynamic Languages*. In Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS'10), Malaga, Spain, June 2010. 4, 91

[Arnaud 2013] Jean-Baptiste Arnaud, Marcus Denker and Stéphane Ducasse. *Behavior-Propagating First Class References For Dynamically-Typed Languages*. Science of Computer Programming, 2013. Under-revision. 5

[Bardou 1996] Daniel Bardou and Christophe Dony. *Split Objects: a Disciplined Use of Delegation within Objects*. In Proceedings of the 11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96), pages 122–137, October 1996. 27

[Bell 1974] D.E. Bell and L. LaPadula. *Secure Computer Systems*. Rapport technique, ESD-TR-83-278, April 1974. 17

[Bergel 2008] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz and Roel Wuyts. *Stateful Traits and their Formalization*. Journal of Computer Languages, Systems and Structures, vol. 34, no. 2-3, pages 83–108, 2008. 37, 97

[Bieniusa 2009] Annette Bieniusa. *Consistency, Isolation, and Irrevocability in Software Transactional Memory*. PhD thesis, Uni freiburg, 2009. 9, 10, 74

[Bobrow 1993] Daniel G. Bobrow, Richard P. Gabriel and J.L. White. *CLOS in Context — The Shape of the Design*. In A. Paepcke, editeur, Object-Oriented Programming: the CLOS perspective, pages 29–61. MIT Press, 1993. 7

[Boebert 1984] W. E. Boebert. *On the inability of an unmodified capability machine to enforce the *-property*. In In Proceedings of the 7th DOD/NBS Computer Security, 1984. 17

[Boyland 2001] John Boyland, James Noble and William Retert. *Capabilities for Sharing, A Generalisation of Uniqueness and Read-Only*. In Proceedings of the 15th European Conference on Object-Oriented Programming(ECOOP'01), numéro 2072 de LNCS, pages 2–27. Springer, June 2001. 1, 40

[Bracha 2004] Gilad Bracha and David Ungar. *Mirrors: design principles for meta-level facilities of object-oriented programming languages*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices, pages 331–344, New York, NY, USA, 2004. ACM Press. 28

[Bryce 1999] Ciarán Bryce and Centre Universitaire. *The JavaSeal mobile agent kernel*. In Autonomous Agents and Multi-Agent Systems, pages 103–116, 1999. 26

[Cardelli 1997] Luca Cardelli. *Type Systems*. In Allen B. Tucker, editeur, The Computer Science and Engineering Handbook, chapitre 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997. 7

[Carré 1990] Bernard Carré and Jean-Marc Geib. *The Point of View Notion for Multiple Inheritance*. In Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications(OOPSLA/ECOOP '90), volume 25, pages 312–321, October 1990. 27

[Civello 1993] Franco Civello. *Roles for composite objects in object-oriented analysis and design*. In Proceedings of the 16th International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'93), ACM SIGPLAN Notices, volume 28, pages 376–393, October 1993. 27

[Costanza 2005] Pascal Costanza and Robert Hirschfeld. *Language Constructs for Context-oriented Programming: An Overview of ContextL*. In Proceedings of the first Dynamic Languages Symposium (DLS'05), pages 1–10, New York, NY, USA, October 2005. ACM. 23, 27

[Denker 2006] Marcus Denker, Stéphane Ducasse and Éric Tanter. *Runtime Bytecode Transformation for Smalltalk*. Journal of Computer Languages, Systems and Structures, vol. 32, no. 2-3, pages 125–139, July 2006. 38, 39

[Ducasse 1999] Stéphane Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999. 39, 57

[Eugster 2006] Patrick Eugster. *Uniform proxies for Java*. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA'06), pages 139–152, New York, NY, USA, 2006. ACM. 28

[Felleisen 1992] Matthias Felleisen and Robert Hieb. *The revised report on the syntactic theories of sequential control and state*. Theor. Comput. Sci., vol. 103, no. 2, pages 235–271, 1992. 97

[Finifter 2008] Matthew Finifter, Adrian Mettler, Naveen Sastry and David Wagner. *Verifiable Functional Purity in Java*. In Proceedings of the 15th ACM International Conference on Computer and Communications Security (CCS'08), pages 27–31, 2008. 1, 29

[Flatt 1998] Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen. *Classes and Mixins*. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 171–183. ACM Press, 1998. 97

[Flatt 1999] Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins*. Rapport technique TR 97-293, Rice University, 1999. 36

[Friedman 1984] Daniel P. Friedman and Mitchell Wand. *Reification: Reflection without metaphysics*. In LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming, pages 348–355, New York, NY, USA, 1984. ACM. 31

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional, 1995. 31

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk 80: the language and its implementation. Addison Wesley, Reading, Mass., May 1983. 84, 89

[Gong 1989] Li Gong. *A Secure Identity-Based Capability System*. In In Proceedings of the 1989 IEEE Symposium on Security and Privacy, pages 56–63, 1989. 17

[Gordon 2007] Donald Gordon and James Noble. *Dynamic ownership in a dynamic language*. In Pascal Costanza and Robert Hirschfeld, editeurs, Proceedings of the 2007 symposium on Dynamic languages (DLS'07), pages 41–52, New York, NY, USA, 2007. ACM. 1, 2, 21, 22, 25, 29, 30

[Guzdial 2001] Mark Guzdial and Kim Rose. Squeak — open personal computing and multimedia. Prentice-Hall, 2001. 60, 84

[Hanenberg 2004] Stefan Hanenberg, Robert Hirschfeld and Rainer Unland. *Morphing aspects: incompletely woven aspects and continuous weaving*. In AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, pages 46–55, New York, NY, USA, 2004. ACM. 25

[Haupt 2007] Michael Haupt and Hans Schippers. *A Machine Model for Aspect-Oriented Programming*. In Proceedings of European Conference on Object-Oriented Programming (ECOOP'07), volume 4609 of *LNCS*, pages 501–524. Springer Verlag, 2007. 25, 31

[Herrmann 2007] Stephan Herrmann. *A precise model for contextual roles: The programming language ObjectTeams/Java*. Appl. Ontol., vol. 2, pages 181–207, apr 2007. 27

[Hirschfeld 2008] Robert Hirschfeld, Pascal Costanza and Oscar Nierstrasz. *Context-Oriented Programming*. Journal of Object Technology, vol. 7, no. 3, March 2008. 23

[Hogg 1991] John Hogg. *Islands: aliasing Protection In Object-Oriented Languages*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91), ACM SIGPLAN Notices, volume 26, pages 271–285, 1991. 1, 3, 26

[Ingalls 1997] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), pages 318–326. ACM Press, November 1997. 84

[Kain 1987] Richard Y. Kain and Carl E. Landwehr. *On Access Checking in Capability-Based Systems*. IEEE Transactions on Software Engineering, vol. 13, pages 202–207, 1987. 17

[Kephart 2003] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing*. Computer, vol. 36, no. 1, pages 41–50, January 2003. 23

[Levy 1984] Henry Levy. Capability-based computer systems. Butterworth-Heinemann, Newton, MA, USA, 1984. 13, 14

[Lie 2004] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, May 2004. 8

[Lienhard 2008] Adrian Lienhard. *Dynamic Object Flow Analysis*. Phd thesis, University of Bern, December 2008. 31

[Livshits 2005] Benjamin Livshits, John Whaley and Monica S. Lam. *Reflection Analysis for Java*. In Proceedings of Asian Symposium on Programming Languages and Systems, 2005. 2

[Maes 1987] Pattie Maes. *Concepts and Experiments in Computational Reflection*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 147–155, December 1987. 7

[Martinez Peck 2011] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Efficient Proxies in Smalltalk*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011), Edinburgh, Scotland, 2011. 28

[McKinley 2004] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten and Betty H. C. Cheng. *Composing Adaptive Software*. Computer, vol. 37, no. 7, pages 56–64, July 2004. 23

[Mettler 2010] Adrian Mettler, David Wagner and Tyler Close. *Joe-E: A Security-Oriented Subset of Java*. In Proceedings of Annual Network and Distributed System Security Symposium (ISOC NSSS), pages 375–388, 2010. 25

[Miller 2003a] Mark Samuel Miller and Jonathan S. Shapiro. *Paradigm Regained: Abstraction Mechanisms for Access Control*. In Proceedings of the Eigth Asian Computing Science Conference(IAFOR'03), pages 224–242, 2003. 15

[Miller 2003b] Mark Samuel Miller, Ka-Ping Yee and Jonathan Shapiro. *Capability Myths Demolished*. Rapport technique, Combex Inc, 2003. 8, 10, 11, 44, 56

[Miller 2006] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 15, 18, 25, 28, 30, 38

[Moore 2006] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill and David A. Wood. *LogTM: Log-based Transactional Memory*. In Proceedings of the 12th International Symposium on High-Performance Computer Architecture, pages 254–265. IEEE Computer Society, February 2006. 8

[Müller 1999] P. Müller and A. Poetzsch-Heffter. *Universes: A Type System for Controlling Representation Exposure*. In A. Poetzsch-Heffter and J. Meyer, editeurs, Programming Languages and Fundamentals of Programming. Fernuniversität Hagen, 1999. 1, 27

[Noble 1998] James Noble, Jan Vitek and John Potter. *Flexible Alias Protection*. In Eric Jul, editeur, Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, pages 158–185, Brussels, Belgium, July 1998. Springer-Verlag. 1, 27

[Noble 1999] James Noble, David Clarke and John Potter. *Object Ownership for Dynamic Alias Protection*. In Proceedings of the 37th International Conference on Objects, Models, Components, Patterns (TOOLS'99), November 1999. 21

[Pascoe 1986] Geoffrey Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80*. In Proceedings of the ninth International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86), ACM SIGPLAN Notices, volume 21, pages 341–346, November 1986. 31, 39, 57

[Redell 1974] David D. Redell. *NAMING AND PROTECTION IN EXTENDABLE OPERATING SYSTEMS*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974. 8, 10, 15

[Renggli 2007] Lukas Renggli and Oscar Nierstrasz. *Transactional Memory for Smalltalk*. In Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pages 207–221. ACM Digital Library, 2007. 8

[Saraswat 2003] Vijay Saraswat and Radha Jagadeesan. *Static support for capability-based programming in Java*, 2003. 17

[Schärli 2004a] Nathanael Schärli, Andrew P. Black and Stéphane Ducasse. *Object-oriented Encapsulation for Dynamically Typed Languages*. In Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04), pages 130–149, October 2004. 2, 18, 19, 20, 25, 30, 62

[Schärli 2004b] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Roel Wuyts. *Composable Encapsulation Policies*. In Proceedings of European Conference on Object-Oriented Programming (ECOOP'04), volume 3086 of *LNCS*, pages 26–50. Springer Verlag, June 2004. 19, 30

[Schippers 2008] Hans Schippers, Dirk Janssens, Michael Haupt and Robert Hirschfeld. *Delegation-based semantics for modularizing crosscutting concerns*. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications, pages 525–542, New York, NY, USA, 2008. ACM. 25

[Shavit 1995] Nir Shavit and Dan Touitou. *Software Transactional Memory*. In Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 204–213, 1995. 8, 10, 74, 81

[Smith 1996] Randall Smith and Dave Ungar. *A Simple and Unifying Approach to Subjective Objects*. TAPOS special issue on Subjectivity in Object-Oriented Systems, vol. 2, no. 3, pages 161–178, 1996. 27

[Snyder 1986] Alan Snyder. *Encapsulation and Inheritance in Object-Oriented Programming Languages*. In Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, pages 38–45, November 1986. 2, 18

[Tanter 2008] Éric Tanter. *Contextual values*. In Proceedings of the 2008 symposium on Dynamic languages(DLS '08), pages 1–10, New York, NY, USA, 2008. ACM. 23, 24, 25

[Tratt 2009] Laurence Tratt. *Dynamically Typed Languages*. Advances in Computers, vol. 77, pages 149–184, 2009. 1, 7

[Van Cutsem 2010] Tom Van Cutsem and Mark Samuel Miller. *Proxies: design principles for robust object-oriented intercession APIs*. In Proceedings of the 2010 symposium on Dynamic languages(DLS '10), volume 45, pages 59–72. ACM, oct 2010. 28

[Wallach 1997] Dan S. Wallach, Dirk Balfanz, Drew Dean and Edward W. Felten. *Extensible security architecture for Java*. In In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 116–128, 1997. 17

[Warth 2008] Alessandro Warth and Alan Kay. *Worlds: Controlling the Scope of Side Effects*. Rapport technique RN-2008-001, Viewpoints Research, 2008. 5, 27, 78, 79, 81

[Weiser 1993] Mark Weiser. *Some computer science issues in ubiquitous computing*. Commun. ACM, vol. 36, no. 7, pages 75–84, July 1993. 23

[Wolczko 1992] Mario Wolczko. *Encapsulation, delegation and inheritance in object-oriented languages*. IEEE Software Engineering Journal, vol. 7, no. 2, pages 95–102, March 1992. 18

[Yonezawa 1987] Akinori Yonezawa, Etsuya Shibayama, T. Takada and Yasuaki Honda. *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*. In A. Yonezawa and M. Tokoro, editeurs, Object-Oriented Concurrent Programming, pages 55–89. MIT Press, Cambridge, Mass., 1987. 9