

Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages

Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker, Camille Teruel

▶ **To cite this version:**

Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker, Camille Teruel. Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages. Science of Computer Programming, 2015, Science of Computer Programming, 98 (3), pp. 318-338. ⟨10.1016/j.scico.2014.07.011⟩. ⟨hal-01060537v2⟩

HAL Id: hal-01060537

https://inria.hal.science/hal-01060537v2

Submitted on 31 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages

accepted to Science of Computer Programming

Jean-Baptiste Arnaud[a], Stéphane Ducasse[a], Marcus Denker[a], Camille Teruel[a]

[a]*RMoD INRIA Lille Nord Europe*

Abstract

Controlling object graphs and giving specific semantics to references (such as read-only, ownership, scoped sharing) has been the focus of a large body of research in the context of static type systems. Controlling references to single objects and to graphs of objects is essential to build more secure systems, but is notoriously hard to achieve in absence of static type systems. In this article we embrace this challenge by proposing a solution to the following question: What is an underlying mechanism that can support the definition of properties (such as revocable, read-only, lent) at the reference level in the absence of a static type system? We present handles: first-class references that propagate behavioral change dynamically to the object subgraph during program execution. In this article we describe handles and show how handles support the implementation of read-only references and revocable references. Handles have been fully implemented by modifying an existing virtual machine and we report their costs.

Keywords: Security, Dynamic language, First class references, Language design

1. Introduction

Controlling references is essential to build secure systems. Monitoring references and giving them specific properties have been the focus of a large body of research in the context of statically typed languages [Hog91, CPN98, CD09]. For example, references are qualified as read-only, lent, shared, immutable [BNR01]. The problem addressed by such approaches is central to building more secure systems [Bis03]. Some works proposed to control the interface of an object [HLR+99, FZ04]. However such approaches are not adequate in presence of open-world and dynamic type systems [GN07].

Before going further we define two terms used in this article with precise meaning: *property* and *capability*.

Property. We use the term property to denote the general behavior that an object or a computation should exhibit. For example, when we write that an object should ensure a read-only property, it means that during a computation the state of this object should not be changed.

URL: `http://jeanbaptiste-arnaud.eu` (Jean-Baptiste Arnaud), `http://stephane.ducasse.free.fr` (Stéphane Ducasse), `http://marcusdenker.de` (Marcus Denker), `http://rmod.lille.inria.fr` (Camille Teruel)

Our use of the term property should not be confused with properties in the sense of fields, attributes or instance variables of objects.

Capability. A capability is a key to access a resource. In the *object capability model* proposed by Miller [Lev84, MS03, Mil06], a capability is a reference to an object and the available operations are the methods of this object. An object has to offer a limited interface to limit access to its operations.

We continue our introduction by presenting approaches that tried to control references. Few approaches have been proposed in the context of dynamically-typed languages: encapsulation policies propose different per-reference encapsulation interfaces [SBD04]. Dynamic ownership proposes to control *access* to object parts by changing message passing with an execution cost up to 51% [GN07]. In addition, most approaches to control references are concerned with controlling a single reference. In practice, one is often interested in controlling the complete graph of objects that is accessible at run time from the reference.

Several works use the object capability model as a way to control references [Lev84, MS03, Mil06, FMSW08]. The idea behind capabilities-as-objects is that the reference itself is a capability that is a key to access a resource or behavior ; without this key the resources are not accessible. This means that if a client has a reference to an object, it has a capability that is equal to what the object can do. In such approaches, there is no way to restrict what a reference can do, other than just giving another reference to a different object that has a constrained interface. The programmer thus must follow idioms and patterns to make sure that there is no reference leaked with the full interface, or the safety would be compromised. Capability-based implementations such as Joe-E are again based on a static type system [FMSW08]. Such approaches do not fully address our needs since some properties should propagate through all the objects reached during a particular execution and this only from the perspective of a given reference.

These works raise the following question: *What is an* underlying *mechanism that can support the definition of properties (such as revocable or read-only) at the reference level in the absence of a static type system?* This article proposes Handles as one answer to such question.

A handle is a first-class reference[1] which acts as a viewpoint on the object it refers to. In addition, a handle enforces the semantics that it embeds (such as read-only) on the referenced object but only when such object is accessed via this handle. Finally, handles act as an overlay on the dynamic object graph in which they are automatically propagated at run time.

Our approach is structured as a framework: firstly, the language designer 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. For example, when the class Point should be accessed via a handle, the read-only property is applied to the class Point to create a read-only version of this class. Secondly, the handle mechanism ensures the systematic propagation of that property at execution time.

The contributions of this article are:

1. the presentation of challenges to control references and object graphs in the context of dynamically-typed languages,

[1] *i.e.*, reification of reference

2. Handles: first class references that propagate their behavior and their formal description,
3. the application of this framework to implement read-only execution and revocable references, and
4. a precise description of the implementation in Pharo, a Smalltalk derivative [BDN+07].

The differences with our previous work [ADD+10] are: (1) a generalization and new definition of the handle concept to be able to convey different semantics such as revocable references, (2) the introduction of metahandles to allow run-time control over handles, and (3) a new and minimal implementation of handle propagation not based on bytecode rewriting.

In the following section we present two problems to show that dynamic languages need better control over references. Section 3 presents Handles: behavior-propagating first-class references. Its formal model is presented in Section 4. We then proceed to use handles to realize read-only execution (Section 5.1) and revocable references (Section 5.2). Low-level virtual machine implementation details are shown in Section 6, followed by an evaluation in Section 7. In Section 8 we present an overview of the related work. In Section 9 we conclude by summarizing the presented work and outlining future work.

2. A Case for Handles

This section presents two examples, the goal is to stress the specific requirements needed to control references.

2.1. Constraints brought by Dynamically-Typed Languages

Controlling references is a hard problem for any programming language. But in the case of reflective dynamically-typed languages, we face some additional problems:

No static types. For example, using a static type system, Birka *et al.* add a read-only type qualifier, which ensures that all state that is transitively reachable from a read-only reference is also read-only [BE04]. Dynamically-typed languages do not provide static type information, seriously compromising any static analysis at compile time.

Open world. System openness (with dynamic code loading and dynamically-typed systems) has two consequences. First, there is a need for more control (for example, when loading untrusted extensions at run time). Second, it makes analysis harder since there is no fixed system to be analyzed before execution.

2.2. Supporting Read-Only Execution

We analyze preconditions as an example of the need for read-only references. From this example we extract requirements to support read-only references in a dynamically-typed language. The challenge is that the execution of a method precondition should not change the state of the participating objects (receiver and arguments). Most existing languages supporting pre and post conditions are either based on coding conventions (programmers should not invoke methods changing the state of the objects and arguments) or on copying the objects (which is unrealistic in most of the cases).

An ideal solution is to define the scope of execution of a precondition and ensure that any modification will raise a run time error. Remember that we do not have static types so the error

can only occur at run time. Imagine the definition of the method submorphsDo: of the class Morph written in Smalltalk [2]:

```
Morph>>submorphsDo: aBlock
    <precondition: self submorphs isOrdered >
    ...
    self submorphs isOrdered
    ...
```

Within the precondition scope (denoted here using <>) the programmer should get the warranty that the state of the receiver *and its object subgraph* are unchanged and that an error is raised if an attempt is performed. Not only the receiver but any modification of the objects reached during such execution should raise such an error [ADD+10]. In contrast, using exactly the same expression (here self submorphs isOrdered) in the method body can modify the receiver and its object subgraphs.

2.3. Supporting Revocable References

Miller *et al.* show that capabilities can be used to support confinement and revocable references [MYS03]. Figure 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. 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.

In the example, the facet needs to be carefully thought to not leak references and only return facets instead. This example shows that there is a need for more advanced way to control references than traditional ones. Membranes [Mil06, VCM13] is such an advanced way that is discussed in Section 8.

2.4. Requirements

The previous examples show the following requirements for controlling object graphs:

Reference-based. The previous examples show that we need to control object graphs per reference. The same graph can be referenced from multiple objects that do not use the same property or do not use properties at all.

[2] In Smalltalk, as a first approximation, messages follow the pattern receiver methodName1: arg1 name2: arg2 which is equivalent to the Java syntax receiver.methodName1name2(arg1, arg2). Hence, self submorphs isOrdered is equivalent to this.submorphs.isOrdered(). In addition, a method definition starts with a method definition without type definition. Hence, the method signature submorphsDo: aBlock is equivalent to void submorphDo(b Block).

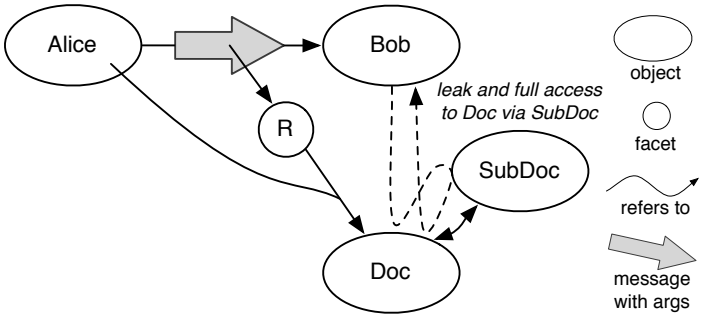


Figure 1: Revocable references: Alice can decide Bob should not be able to access Doc. References to SubDoc should also be revokable, else Bob can access Doc even after revocation.

Propagating. An important aspect is how specific properties propagate to the object graph during program execution. Controlling a single object is not enough, as all non-trivial programs use multiple objects to model data structures which form a graph of objects. We want to be able to control such a graph of objects, even though we might only reference only one object of the structure.

- When we want to ensure that the execution of a precondition is read-only, the read-only property should be propagated to all the objects involved in the precondition execution [ADD+10].
- Similarly, when a *user* is granted a revocable reference, the propagation of such behavior to the objects 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. Sending messages via controlled reference should not be different than normal sends. For example, when Bob accesses Doc a controlled reference, it should be able to perform any actions on it and should not be aware that it is using a special reference. We discuss transparency and the relationship with identity in Section 7.2.

3. Handles

Handles are first class references that propagate behavioral changes dynamically to the object subgraph during program execution. We present them formally in Section 4.

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 handle's target object.

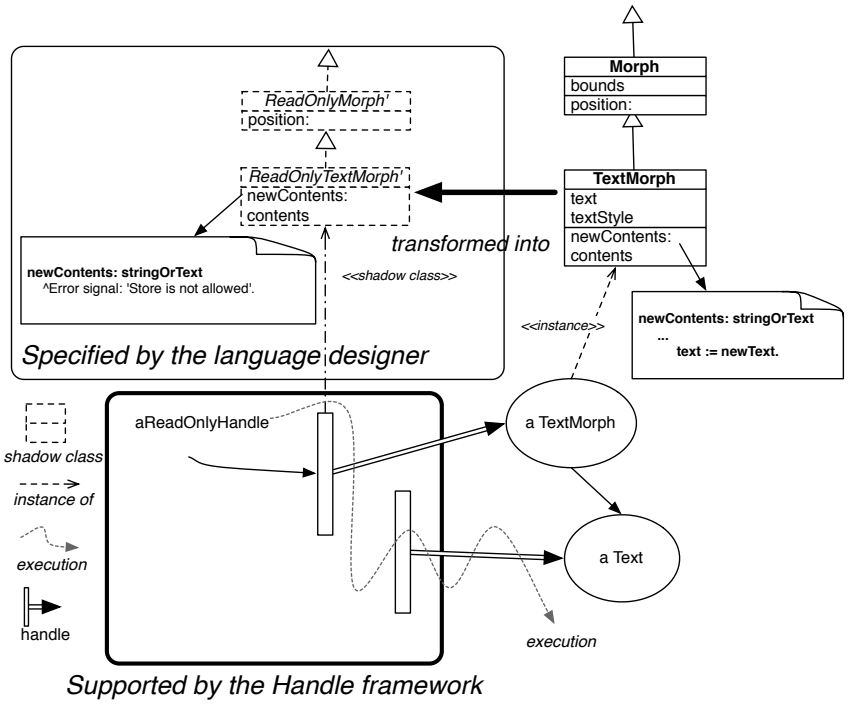


Figure 2: Handle supports the management of reference properties and their propagation at run time. The language designer has to provide the transformation that defines the semantics of his new construct: here read-only.

A Two-Step Approach. Our approach is structured in two parts as illustrated by Figure 2. 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 has the property applied to the class of the target object. For example, to implement the read-only property, all the write accesses in a given class should raise an exception (Figure 2). Second, once handles are created, the framework ensures at run time 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.

As a complement to Figure 2, the following code snippet shows how a programmer will get a read-only reference using the message asReadOnly provided by the language designer using the Handle framework. aReadOnlyClient is a handle on the object anTextMorph.

```
"Used by the programmer"
aReadOnlyHandle := anTextMorph asReadOnly.

"Defined by the language designer"
Object>>asReadonly
   ^ ReadOnlyHandle for: self
```

3.1. Handle Model

Figure 3 describes the underlying principle of Handles: (1) a handle is a transparent reference to a target object. By transparent we mean that a programmer cannot detect by using pointer equality that he has a pointer on an object or on a handle on the same object. (2) a handle can define different behavior than the target object. When the message contents is sent via the reference authorizedClient1 the handle executes its *contents'* method *using the identity and the state of the target object*. In addition, authorizedClient1 has only access to the transformed target behavior which is stored in a *shadow class*. Obviously, the shadow class can have additional methods as we can see in Figure 3: the *primitiveRedraw* method can only be accessible using authorizedClient1 or authorizedClient2. There is no lookup mechanism between the shadow class and the target class. If a method is not defined in the shadow class but is defined in the class of 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 normalAccessClient and controlled ones such as authorizedClient1. 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 class synchronized automatically.

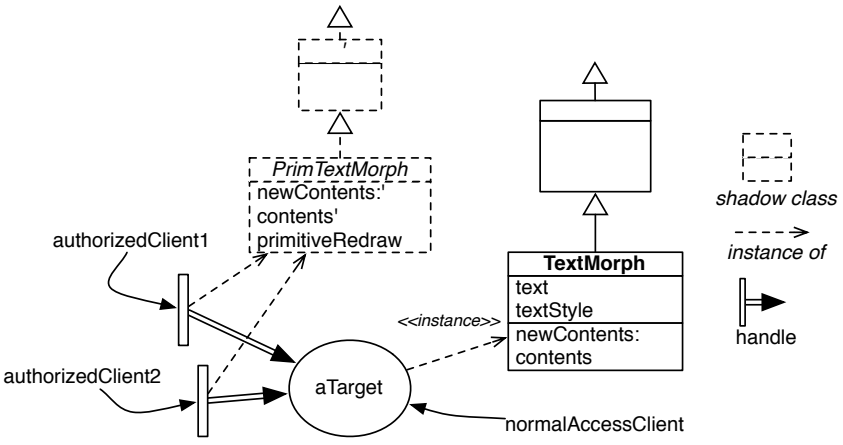


Figure 3: Handle Principle: a handle uses target state and identity and defines a specific behavior (potentially adapted from target class). Several handles can co-exist on the same target.

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, the state of the target object that will change. Like objects, handles can be stored in instance variables.

Handle Propagation through Execution. Figure 4 presents handle propagation. When accessed via a handle, *any instance variable read wraps the instance variable,* i.e.*, creates a handle on the object held in the variable*.

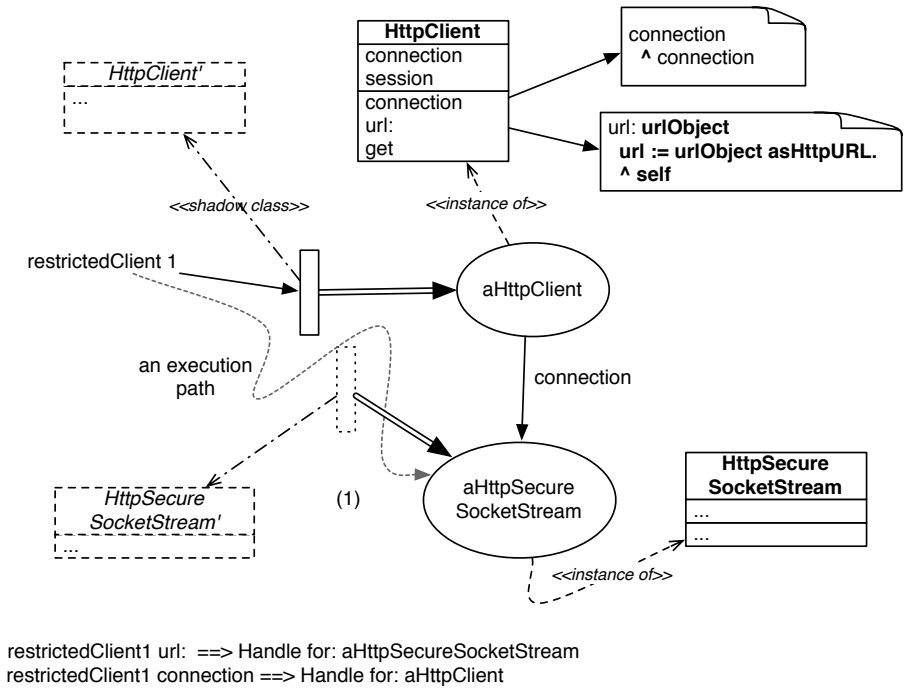


Figure 4: Handle Propagation Principle. All accesses via a handle to target object state are wrapped with a handle and propagated at run time.

- Sending a message that returns an instance variable of the target object, returns a handle on this object. In Figure 4 the expression restrictedClient1 connection returns a handle on the object held by the instance variable: aHttpSecureSocketStream.
- This propagation is recursive and follows the application execution (1 in Fig. 4).
- Finally, a handle do not wrap on store and self references stay wrapped. The expression restrictedClient1 url: 'https://www.google.com' stores a new httpURL (and not a handle on this new object) in the target object and returns the handle (accessed via restrictedClient1).

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 what self/this refers to. In particular, a handle method returning its receiver could leak the target object and this is clearly not what we want [Lie86].

- When a method is executed via a reference that is not a handle, self/this in a (target) method represents the target object as in traditional object-oriented languages.
- When a method is executed via a reference that is a handle, self/this refers to the handle.

Handles conserve the invariant that self/this represents the receiver of the message.

3.2. Handle creation and 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 information relative to its target object. Immediately after its initialization the system activates the handle.
- *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 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.

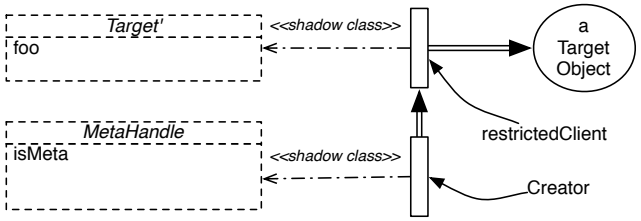


Figure 5: Metahandle Principle: a metahandle is a handle whose target is a handle.

A metahandle is a handle whose target is another handle (see Figure 5). Since an activated handle is a point of view on its target and as such may change the target behavior, sending a message to a metahandle can modify a handle. Using metahandles, handles can be reconfigured dynamically. Nevertheless the behavior of the metahandle can be partial, depending of the shadow class used. An important point is that a metahandle can only be created on inactive handles. The reason is that a handle user should not be able to alter the handle.

A handle creator can keep a reference to a metahandle to later configure or change the associated 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, depending on their class (and the language designer needs), during their initialization, handles will return their metahandle or not (by default they do not return it). Therefore, as long 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 a method in the handle shadow class and applies it to the handle's target object, not the handle itself).

Of course, it is possible to create a metahandle on a metahandle. This meta-metahandle could be used to restrict the use of a metahandle that has been handed over to a client. For example, one could hand over a metahandle that is a revocable reference. The meta-metahandle then controls revocation of the metahandle.

4. HANDLELITE: handle operational semantics

We present the operational semantics of Handles by extending SMALLTALKLITE [BDNW08]. SMALLTALKLITE is based on CLASSICJAVA [FKF98] but adapted to dynamically-typed languages: it does not support any notion of static type, interface, and considers fields as private. SMALLTALKLITE formalisation is given in appendix. Our goal is to provide a precise description of handle execution. Then we explore an example to show how properties are ensured and propagated during program execution.

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 designer which defines how to control an object graph at the reference-level. This includes the change of behavior, the configuration of a handle and how the handle should be propagated. The designer creates a mechanism for transforming a class c into 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: $p(c)$. The property p takes as argument the class c and return a new class c'. The intuition behind c' is that it enforces property p. See Figure 6.

$p(c) \quad = \quad c'$
Where c' respects the property p.
And c and c' are classes.

Figure 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 reducible expressions of SMALLTALKLITE (see Figure 7).

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

Figure 7: Handle: new reducible expressions for HANDLELITE

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

4.1. Per reference behavior shadowing

To enforce behavior shadowing, a handle needs to keep a behavior. We add handles to reducible expressions to allow one to send message to a handle $h_p^o.m(\epsilon^*)$, and to pass a handle as a parameter to a method send. In addition, we add the possibility to perform a super-send on a

handle (ie. **super**$\langle h_p^o, c\rangle.m(\epsilon^*)$). Moreover, we add two rules to the reduction rules of SMALLTALKLITE to change how message and super-send are managed (Figure 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 8), which evaluates the method body $[\![e[v^*/x^*]]\!]_{c''}$. This method body is found in class c'' using a traditional method lookup 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 ($h_p^o.m(\epsilon^*)$), the lookup begins in class c', where c' is provided by the handle property p (see [BDNW08] Fig 8 page 17). Note that self is bound to h_p^o. To summarize, the usual self-send mechanism is applied, except that the receiver is bound to the handle.

[handled super-send] represents how super-sends are managed when performed on a handle. The rule [handled super-send] defines the expression $\langle E[\textbf{super}\langle h_p^o, c'\rangle.m(v^*)], \mathcal{S}\rangle$ (see Figure 8), which evaluates the method body $[\![e[v^*/x^*]]\!]_{c'''}$. This method body is found using a traditional lookup starting from the superclass of the class containing the super-send expression. This means, when h_p^o receives a message(**super**$\langle h_p^o, c'\rangle.m(\epsilon^*)$), the lookup begins in class c'', where c'' is the superclass of the c' and c' the class containing the method using *super*. c''' is the class containing the matching method. To summarize, the usual *super-send* mechanism is applied except that the receiver is bound 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.

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

$$
\textit{where } \in_P^* \quad \text{Method lookup starting from } c \qquad \langle c, m, x^*, e\rangle \in_P^* c' \iff c' = \textit{min}\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}
$$

Figure 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.

4.2. 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 treat 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 for a handle, the identity of the target object is returned:

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

4.3. Property propagation

Handles propagate the properties that they provide. We add the Handle concept to reducible expression that allows one to evaluate the expression $h_p^o.f$ and $h_p^o.f{=}\epsilon$ on a handle and write a handle into a field $\epsilon.f{=}\, h_p^o$. 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 9, we add the following two reductions:

[handled get] represents how fields are read from a handle. The [handled get] reduction first fetches the state of the target object and then 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 $\langle E[h_p^v], \mathcal{S}\rangle$ into a new handle h_p^v respecting the same properties p.

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

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

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

Figure 9: State related reductions for HANDLELITE

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.

4.4. Example

We illustrate the 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 have a shadow class that implements the tracing property that a handle applies and propagates on the target subgraph. The tracing property registers all the message sends on handles and is denoted TP.

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

```
BankAccount>>user
  ^ user

BankAccount>>applyMonthlyCharges
  ... "Some messages that apply monthly charges"
  ^ self
```

The class defines two methods: *user* that returns user and *applyMonthlyCharges* that computes all the charges performed during the month and then return *self*. In our context, the important point is that such method leaks a reference to *self*.

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

We create a BankAccount (line 2), then we create a tracing handle on it (line 3). Finally we execute *applyMonthlyCharges* and *user* methods.

We apply now each rule and the example shows that given a handle we cannot leak references to the target object. First, in *applyMonthlyCharges* we see that *self* returns the handle and not the target (see below the line 4 and the rule application). 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 show the formal execution of this code.

Line 2 create a bank account ba and line 3 creates a handle $hba = h^{ba}_{TP}$ for ba. Line 4 is evaluated according to the given semantics as follow:

$$\begin{array}{lll} a & \left[\!\left[h^{ba}_{TP}.applyMonthlyCharges()\right]\!\right] & [handled\ send] \\ b & \Rightarrow [...] & \\ c & \Rightarrow h^{ba}_{TP}[\![\text{self}]\!]_{TPBankAccount} & \\ d & \Rightarrow h^{ba}_{TP} & \end{array}$$

In the previous listing, we send a message applyMonthlyCharges() to *hba* and fetch the self value. The method is looked-up in the class $TPBankAccount = TP(BankAccount)$ (see *line c*). The message returns self (*line c*) that is bound to the handle h^{ba}_{TP} (*line d*). Line 5 is evaluated according to the given semantics as follow:

$$\begin{array}{lll} a & \left[\!\left[h^{ba}_{TP}.\text{user}()\right]\!\right] & [handle\ send] \\ b & \Rightarrow h^{ba}_{TP}[\![\text{self.user}]\!]_{TPBankAccount} & \\ c & \Rightarrow h^{ba}_{TP}\left[\!\left[h^{ba}_{TP}.\text{user}\right]\!\right]_{TPBankAccount} & [handle\ get] \\ d & \Rightarrow h^{ba}_{TP}[\![h^{user}_{TP}]\!]_{TPBankAccount} & \\ e & \Rightarrow h^{user}_{TP} & \end{array}$$

In the previous listing, we send a message user to *hba* and get the instance variable *user* (lines *a b c*). The message is looked-up in the class $TPBankAccount$. 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 $h^{ba}_{TP}.user$ (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}_{TP} (line *e*). This example shows how instance variables are wrapped on access.

This example shows that:

- Handles ensure that the tracing 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 self (*applyMonthlyCharges* in the example).

5. Uses Cases for Handles

In this section we describe some use-cases for Handles. First we present several approaches implemented using handles. Second we detail two well-known approaches, Read-only and Revocable references.

ProfilingReferences define references that are able to instrument all the message sends and instance variable accesses performed through a specific reference and notify another object. This property propagates through the graph.

MirroredReferences is an implementation of Mirrors [BU04] using handles. Mirrors provide access to reflective features by inversion of control: this is not an object that accesses its meta-object, but the meta-object (called a mirror) that provides access to its subject. Mirror-based architecture follow the principle of stratification that stipulates that base-level facilities should be separated from meta-level facilities. Classical mirror-based architecture follow this principle because a mirror and its target are two different objects. Handles offer another way to follow this principle by providing two different kinds of reference on a same object. A mirror reference embeds meta-level facilities while normal references do not. This solution ensures that a mirror reference and its target object share the same identity. Ambiguous methods that could belong to both base-level and meta-level (such as printString) become less verbose because no mirror instantiation is needed.

RestrictedMethodScope proposes an approach to restrict the methods accessible from a specific reference. The user can specify a filter to exclude methods of the API of the target object class.

AuthentifiedReferences proposes an approach to change the API of group of objects based on an authentification process.

All these approaches have been implemented and are functional. In the following subsection we will present the implementations of two security properties using handles.

5.1. Read-only References with Handles

Now we present how handles allows one to implement the read-only behavior and its propagation[3]. Figure 10 shows that the read-only handle shadow class contains rewritten methods of the target class so that they raise error.

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 a handle is created during the object graph propagation by defining the method propagateTo: aTarget. This method expects again a target object.

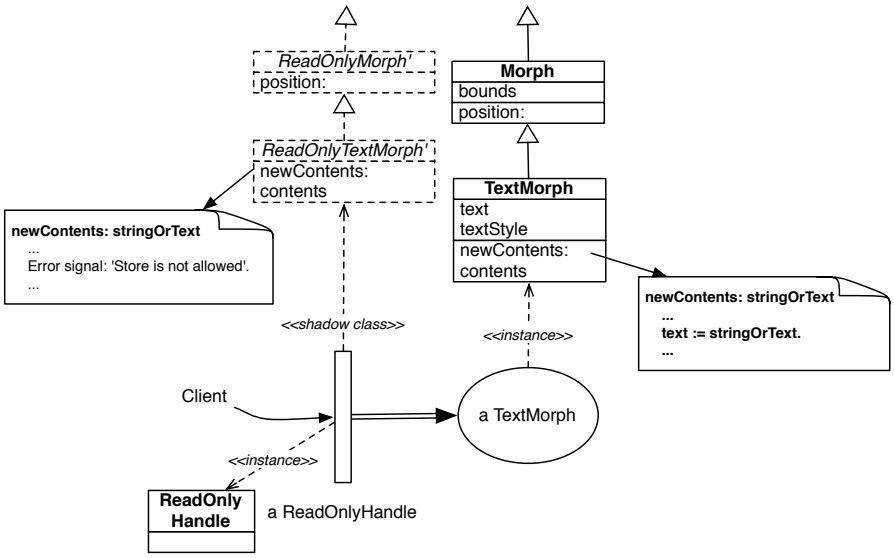


Figure 10: Read-only handles: handle shadow classes contain rewritten methods so that they throw exception on state write.

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 activateHandle.
6       ^ handle
```

[3] A previous article shows a first specific version of Handles [ADD+10]. 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.

At line 3, a class is obtained as a transformed (read-only) version of the target object class – all store accesses to instance variables 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 create a deactivated handle associated with the read-only class. Line 5 activates the handle. From then on we cannot send messages to the handle itself anymore, all messages are forwarded to the target object. Line 6 returns the handle.

Propagation. In addition, the framework asks the language designer 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.

```
1 ReadOnlyHandle class>>propagateTo: anObject
2       ^ ReadOnlyHandle for: anObject.
```

This method is invoked when an instance variable is accessed. The value returned by this method is returned in place of the instance variable value. This mechanism dynamically propagates the read-only behavior to the object graph.

5.2. Revocable References with Handles

The idea behind a revocable reference is to create a reference to an object that can be controlled and revoked [MYS03]. Our revocable reference implementation uses handles and metahandles (as shown in Figure 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 handled normally.

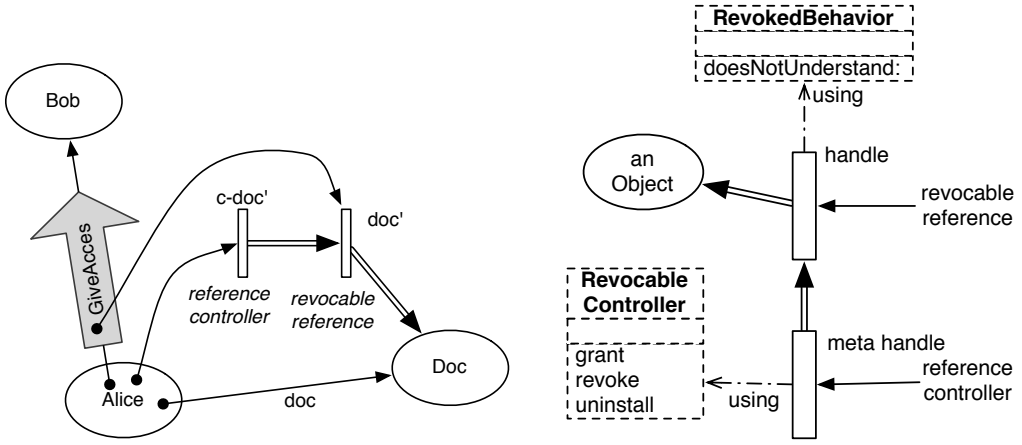


Figure 11: Right: Revocable References using Handles. Left: Revoked references have as a shadow classes that does nothing but raising errors.

5.3. Revocable References Implementation

We implement the Revocable References using Handles in three steps.

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 [Pas86, Duc99]. 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. He has to design the semantics that 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 grant.
6    revocableHandle activate.
7    ^ {revocableHandle . controller}
```

At line 3, a new handle is created and associated with the revoking behavior created in Step 1. 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 creates a metahandle on the handle. Line 5 configures the handle to not apply the revoking behavior. Line 6 activates the handle. Line 7 returns an array with the handle and its controller (the 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
  shadowBehavior := nil.

RevocableReferenceController>>grant
  shadowBehavior := target class
```

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

5.4. 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 SubDoc to Bob. Such a reference should not break the fact that Doc reference to Bob is revocable. Therefore, all references accessed by Bob from its revocable reference should be revocable too. In our example, the reference to SubDoc obtained by Bob via its revocable reference should be revocable. All the references reachable from Doc subgraph should also be revocable when accessed from the handle on Doc (Figure 12).

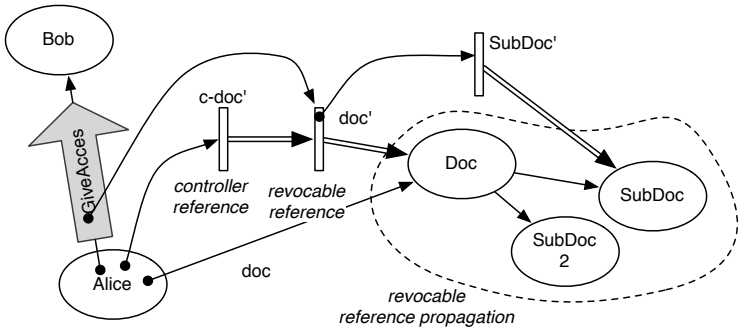


Figure 12: Propagation of revocable references: When Bob accesses SubDoc via Doc, SubDoc should be revocable too.

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 a list of reference controllers (metahandles) per graph to be able to revoke references. It offers the API to revoke references.

5.5. Using Revocable References

Now that the core model has been explained, we present an example. 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 of an instance provided by the factory and a revocable reference on Doc (named Doc'). The instance 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 Doc' doSomething execution (Line 4) but all the references will have the same identifier. Alice can pass Doc' to anybody. Later, Alice asks the factory to revoke the references (Line 6). Now references Doc' raise exceptions when used.

```
1  pair := ControllerFactory for: Doc.
2  "pair is an array with aFactory and Doc'"
3  Doc' := pair second.
4  Doc' doSomething.
```

```
5    Alice pass: Doc' to: Bob.
6    pair first revoke.
7    "we ask the factory to revoke the references to Doc from this specific reference"
8    Doc' open  "raises an exception"
```

6. Virtual Machine Level Implementation

In this section, we present the virtual machine changes to the Pharo Virtual Machine [BDN+09] needed to support Handles[4].

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

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:),.... There are around 150 primitives. In the Squeak/Pharo VM, primitives act 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 object. 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 method, even for primitive methods, this method cannot be accessed 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.

[4]The source code of the VM and the example given in this article are available at: http://jeanbaptiste-arnaud.eu/phd/

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 shadow class 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) [GR01]. 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 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.

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 defined by the class-side method propagateTo: of the Handle class that takes an object as argument. This method can decide to return this object or to return an handle on this object. The second aspect is handled by calling this method on each state read access. 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 result of propagateTo: method. This is enough to implement the semantics described previously.

7. Evaluation and Discussion

To validate our approach, we present a short analysis of the performance and overhead. We discuss various properties of handles and how they relate to our previous work.

7.1. Performance Analysis

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 a 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[5]. 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 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 depth of 16. We execute this benchmark 50 times. In this benchmark we see an overhead of 5.45% of execution for the Handle VM.

	Mean	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.* [SBD04] 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 an 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*. These benchmarks are stressful to show the cost of some specific operations on objects via a handle.

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

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

	Mean	Standard deviation
revocable nbody	8172.12*ms*	31.01*ms*
nbody	4772.80*ms*	20.68*ms*

  We see an increase 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.

	Mean	Standard deviation
revocable binarytrees	68094.23*ms*	70.06*ms*
binarytrees	22321.57*ms*	66.35*ms*

  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. To measure the usual cost of using a Revocable Reference, we take another benchmark *regex-dna*[6]. 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%.

	Mean	Standard deviation
revocable regex-dna	1095.12*ms*	13.46*ms*
regex-dna	1006.32*ms*	11.44*ms*

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.

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

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

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, we need to take into account the cost of the generated classes. This cost depends on the behavioral change. For example, for read-only reference we have to copy the class hierarchy, while revocable just needs one revoking behavior class.

7.2. Discussion

We discuss now some aspects of the handle design.

Differences with our previous work. The Handle model is a generalization and re-design of our previous research about read-only execution [ADD+10]. The model developed here therefore is different, the most important changes are:

- The new model is more general. The previous model was only designed to support read-only references. Now a handle offers a more general mechanism that can be tailored to different scenarios, but takes into account that handles should not leak references to the target object.
- Metahandle. Having all the message sends to a handle offers the possibility to add the metahandle protocol to control handles.
- Simple propagation implementation. In the previous version the propagation to an object graph had to be generated by rewriting bytecode. Now the propagation is simpler, the language designer just needs to specify only one method in the handle class. The propagation is much more efficient, it does not need to visit all bytecodes to detect instance variable accesses, and rewrite them. It is automatically done by the virtual machine.
- Handles can be stored in instance variables.

Handle Composition. Handles 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 model explicitly at this state 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 propagates 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. One idea is to restrict the expressivity of Handles. With this model, we can specify the changes produced and compose them. We plan to explore the idea of chaining handles. As the designer has no way to figure out if a handle is already installed, it should be possible to use another handle in addition.

Deactivating a Handle. Once *activated* a handle 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. So handle control should be planned in advance.

Storing Handles. In the current model, we can store handles in instance variables. We do not dewrap 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 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 of instance variable should hold a revoked reference and raise the expected errors.

The Problem of Identity. The current model intentionally makes handles indistinguishable from their target objects. In particular, the identity of a handle and its target are the same. The reason for this is that handles on the same object represent the same object (not just a similar one). In addition, handles are supposed to be as transparent as possible: only behavior makes them different from the object represented. One possibility can be to introduce two kinds of identity: being the same object and referencing the same object. We need to distinguish real identity and referential identity as two concepts.

Handles and State. It is clear that shadowing state with handles would be interesting. Nevertheless, we think that presenting the model already in the current form is important. As shown in this paper, handles are already useful without embedding their own state. A generalization of the presented model that takes handle state into account is developed in [Arn13].

8. Related Work

The work presented in this paper takes place in the context of a large spectrum of other works ranging from ownership control to capabilities, via controlling interfaces and context-oriented programming. We present here the most significant work with a stress on dynamically-typed solutions, but the list is not exhaustive.

Roles and Views on Objects. Applying different views on objects depending on a given context has been the concern of many papers [CG90, Civ93, SU96, BD96, Her07, WOKK11]. ObjectTeam [Her07] supports roles 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 be dynamically bound to objects. A team defines how roles forward or delegate their roles to the team participants.

The main difference between our work and ObjectTeam is the focus of our work on first-class references and the propagation of behavior into the object graph. ObjectTeam focuses on *role* introduction in a programming language. ObjectTeam proposes three semantics for views on object: delegating as in prototype languages, traditional forwarding and method call interception. *Roles* by definition do not automatically propagate through an object graph.

Smith and Ungar's Us [SU96], a predecessor of today's Context-Oriented Programming [CH05], 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. [WOKK11] 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.

Subject-oriented programming is another attempt to control objects depending on execution context [HO93]. Subject-oriented programming introduces *Subjects*. *Subjects* are objects that regulate state and behavior accesses for each object. There are two differences with our work: (1) Subject-oriented programming does not limit the *activation* of *subjects*. This point severely limits the usage of subject-oriented programming in the context of security. (2) Subject-oriented programming is not based on references.

Contextual values are one attempt to control object's state depending on thread execution context [Tan08]. Contextual values is a generalization of thread-local values implemented in Scheme. There are two differences with our work: (1) the state of objects is contextualized and not their behavior, (2) the solution is local to one thread execution. Our approach is not limited to thread execution: we can have multiple handles on the same objects executed by the same thread. In addition a handle can crosscut multiple thread executions.

Split objects [BD96] 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. There are several differences between our work and split objects. First the Handle model does not manage state, it is only concerned with behavior. In addition, Handles do not operate by delegation. A handle replaces the default behavior of the object by the shadow behavior. Finally the treatment of self is different. 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. In the Handle model, *self* represents the handle.

Context-Oriented Programming. ContextL [CH05] provides a notion of *layers*, which define context-dependent behavioral variations. Layers are dynamically enabled or disabled based on the current execution context. To some extent, the work presented in this paper is related to context-oriented programming: the behavior of an object is modified depending on a context [HCN08]. But unlike prior work on context oriented programming, the context built by the handle is not purely defined by the thread of execution. With Handles, the propagation of changed behavior is dynamic and lazily follows the control flow of the application.

Proxies. A proxy is a well known technique in many object-oriented language. As a handle is a reification of an object reference, it shares some similarities with a proxy. Java provides support for dynamic proxies as a part of the Java reflection library. Java dynamic proxies are limited because they do not forward all method calls but only those defined in specific interfaces. For example, all methods defined in the root class Object cannot be intercepted as they are not defined in an interface. Uniform proxies for Java [Eug06] is an approach to provide dynamic proxies that work for the whole interface of arbitrary classes. Uniform proxies can thus mimic arbitrary objects. However, there is no solution for intercepting operations that are not method calls.

Stratified Proxies [VCM10] improve that point in the context of JavaScript. They can intercept additional operations like object state accesses. Stratified Proxies are actually close in spirit to our work. The difference of our approach is identity preservation and propagation. 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. For example, the concept explored to deal with complex object graphs is that of a *membrane* [Mil06, VCM13]. A membrane is a way to dynamically isolate an object graph with a layer of proxies. These proxies wrap the arguments the argument and the return value of messages they receive. A membrane starts with a

single proxy whose target is the root of the object graph to isolate. The other proxies are created on-demand as the application runs. In the context of revocable references, a membrane transitively imposes revocability on all references exchanged via the membrane. Compared to Handles, a membrane does not imply a notion of propagation over the dynamic extent of the program execution, it just ensures that objects are consistently wrapped on each side of the membrane. With Handles, because of propagation, revoking an object graph takes effect immediately: any thread that is currently in the dynamic extent of a message sent to a revocable reference will be stopped immediately. With Membranes however, because only objects that pass through its boundary are wrapped, such a thread could keep running forever even if the membrane is revoked: it would be stopped by a thrown exception only if it sends a message to a revocable reference obtained from the outside of the membrane.

Other variants of proxies have been proposed. Chaperones and impersonators [STHFF12] are both kinds of proxies that are used to implement the contract system of Racket. Impersonators can only wrap mutable data types, while chaperones can wrap both mutable and immutable data types. Chaperones can only further constrain the behavior of the value that it wraps. When a chaperone intercepts an operation, it must either raise an exception, return the same result that the wrapped target would return, or return a chaperone for the original result. Impersonators, on the other hand, are free to change the value returned from intercepted operations. There is no notion of propagation of the wrapped behavior.

Delegating Proxies [WNTD14] are proxies that can be used to implement a notion of propagation similar to the one proposed by Handles: Delegation Proxies scope behavioral changes to dynamic extents. As of today, this is the closest work to Handles. It should be noted that the original work on Handles [ADD+10] is prior to delegating proxies. One difference is that with Handles it is not possible to unwrap a handle using a reflective API. Another difference is that the Handles implementation is based on deep modifications of the underlying virtual machine such as modified primitives and message passing while propagation is implemented reflectively with Delegating Proxies.

Encapsulation Policies. Encapsulation policies restrict interfaces [SBD04]. Like Handles, encapsulation policies have per-reference semantics. An object can expose different interfaces based on its different references. However, these approaches have two differences. First, there is no propagation in encapsulation policies. Second, Handles are not limited to change the original object interface. Changes can be done inside method body depending on the execution path, while encapsulation policies just control a method as a whole.

Object Capability Model. In E [Mil06], capabilities are used to enforce security by not giving the possibility for a client to by-pass the object interface. Contrary to Handles, here capabilities are modeled as normal objects. They are not hidden or treated specially by the VM. Propagation is not supported and the programmer needs to use security patterns to control access to objects.

Joe-E is a subset of Java based on an object-capability model supporting purely functional methods and type checking [FMSW08]. In Joe-E, a purely functional method does not have side-effects and its behavior only depends on its arguments. In our example of contracts, ensuring functional purity would be too strong: a pre-condition can perfectly rely on internal non state related side-effects. Functional purity is also difficult to ensure in the presence of both late binding and imperative code, without resorting to an entirely different programming style [FMSW08, MWC10].

Stratified Reflection. Mirrors are a design principle for structuring the meta-level features of object oriented languages [BU04]. The idea is to not have both reflective and non-reflective functionality in all objects, but instead separate reflective functionalities from base-level functionalities. A Mirror is simple meta-object: a second object that provides reflection on a normally non-reflective base-level object. Interesting differences of handles to mirrors are that for one, mirrors are normal objects (Mirrors do not apply their methods to their base-level objects), not specially protected by VM-level changes and in addition, mirrors do not support propagation.

Alias Control and Dynamic Object Ownership. Dynamic object ownership [NCP99, GN07] is one of the rare propositions to control aliasing in the context of dynamically-typed languages. Dynamic object ownership implements Flexible Alias Ownership [NVP98]: every object which is part of the representation of an aggregate object is owned by that object and should not be visible outside the aggregate. The ownership of every object is stored into a dedicated field and it is used to verify the validity of every message send. Dynamic ownership enforces *representation encapsulation*, which states that an encapsulated object can only be accessed via its encapsulating object, and *external independence*, which states that an object should not depend on the mutable state of an object that is external to it. ConstrainedJava distinguishes two kinds of externally independent messages: pure (that do not access state) and oneway message (that do not return results). The problem solved by dynamic object ownership is different but related to the one solved by the Handles. The goal of Handles is not to enforce encapsulation per se, but to change interfaces of the same objects, dynamically and to different clients. We do not distinguish object ownership or containment, nor do we enforce that components should be accessed through their owner. The implementation reports up to 51% method execution slowdown due to the tests do be done at run time.

Read-Only References. Hakonen *et al.* [HLR+99] propose the concept of *deeply immutable references*; they only discuss possible implementation strategies without presenting a working implementation. In Javari, Birka *et al.* [BE04] extend Java with a static type system of transitively read-only references. These works are the most similar to our read-only references; the main difference is that they are proposed for statically typed languages. In particular, Javari methods have to be declared read-only *à priori*; unmodified Java code is conservatively considered to have side-effects. In contrast, our example of read-only references does not require any modification besides the initial creation of a read-only reference.

9. Conclusion and Future Work

To be able to build more secure systems, controlling references to single objects and to graphs of objects is essential. In dynamically-typed languages we cannot rely on static type systems for controlling references. In addition, dynamic code loading and reflective change makes it hard to rely on any kind of static analysis.

In this paper 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 an object graph at run time. 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.

As future work, we will extend Handles to support state shadowing in addition to behavior modifications. An open question is how to leverage modern VM technology (just in time compilation) to speed up the execution in the presence of first class references.

Acknowledgments. We thank L. Fabresse, D. Pollet, O. Nierstrasz, A. Kellens and the anonymous reviewers for their helpful comments that allowed us to substantially improve the quality of this paper. S. Ducasse and M. Denker gratefully acknowledge the financial support of Inria. This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013' and by the French Directorate for Armament

References

[ADD+10] 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.

[Arn13] Jean-Baptiste Arnaud. *Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages*. PhD thesis, Université de Lille, 2013.

[BD96] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *Proceedings of OOPSLA '96*, pages 122–137, October 1996.

[BDN+07] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[BE04] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA'2004*, pages 35–49, 2004.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Pearson Education (Singapore) Pte. Ltd., 2003.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing, a generalisation of uniqueness and read-only. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 2–27. Springer, June 2001.

[BU04] 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.

[CD09] N. R. Cameron and S. Drossopoulou. Existential quantification for variant ownership. In *18th European Symposium on Programming*, number 5502 in Lecture Notes in Computer Science. Springer, 2009.

[CG90] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 312–321, October 1990.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS'05)*, pages 1–10, New York, NY, USA, October 2005. ACM.

[Civ93] Franco Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 376–393, October 1993.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64. ACM Press, 1998.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[Eug06] 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.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[FKF98] 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.

[FMSW08] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *CCS'08*, pages 27–31, 2008.

[FZ04] Philip W. L. Fong and Cheng Zhang. Capabilities as alias control: Secure cooperation in dynamically extensible systems. Technical report, Department of Computer Science, University of Regina, 2004.

[GN07] Donald Gordon and James Noble. Dynamic ownership in a dynamic language. In Pascal Costanza and Robert Hirschfeld, editors, *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 41–52, New York, NY, USA, 2007. ACM.

[GR01] Mark Guzdial and Kim Rose. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

[Her07] Stephan Herrmann. A precise model for contextual roles: The programming language objectteams/java. *Appl. Ontol.*, 2:181–207, apr 2007.

[HLR+99] Harri Hakonen, Ville Leppänen, Timo Raita, Tapio Salakoski, and Jukka Teuhola. Improving object integrity and preventing side effects via deeply immutable references. In *Fenno-Ugric Symposium on Software Technology*, pages 139–150, 1999.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[Hog91] 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.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, November 1986.

[Mil06] 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.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of the Eigth Asian Computing Science Conference*, pages 224–242, 2003.

[MWC10] 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.

[MYS03] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, Combex Inc, 2003.

[NCP99] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *Proceedings TOOLS '99*, November 1999.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *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.

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 341–346, November 1986.

[SBD04] 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.

[STHFF12] T.S. Strickland, S. Tobin-Hochstadt, R.B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA'12*, 2012.

[SU96] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[Tan08] Éric Tanter. Contextual values. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2008. ACM.

[VCM10] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010.

[VCM13] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP'13*, 2013.

[WNTD14] Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stéphane Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, Lugano, Suisse, apr 2014.

[WOKK11] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. Worlds: Controlling the scope of side effects. In *ECOOP 2011*. LNCS, 2011.

Appendix A. SMALLTALKLITE

The present appendix provides a description of SMALLTALKLITE [BDNW08]. This is not a contribution of this article and we added here to help the reader. SMALLTALKLITE is based on the object model used by Flatt *et al.* [FKF98]. 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. Similarly it ignores features that are not relevant such as reflection or class-side methods. Fields are private in SMALLTALKLITE, so only local or inherited fields may be accessed.

To simplify the reduction semantics, SMALLTALKLITE adopts an approach similar to that used by Flatt *et al.* [FKF98]. It annotates field accesses and **super**-sends with additional static information that is needed at "runtime". The syntaxes of expressions (e) and extended reducible expressions (ϵ) of SMALLTALKLITE are shown in Figure A.13. The figure also specifies the evaluation order thanks to the evaluation contexts (E) in the style of Felleisen and Hieb [FH92].

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

Figure A.13: SMALLTALKLITE syntax and reducible expression syntax

Translation from expressions to reducible expressions is specified out by the $o[\![e]\!]_c$ function (see Figure A.14). 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). If e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial reducible expression.

Predicates and relations used by the semantic reductions are listed in Figure A.15. These predicates are assumed to be preconditions for valid programs, and are not otherwise explicitly mentioned in the reduction rules.

The reductions rules are summarized in Figure A.16. The notation $P \vdash \langle \epsilon, \mathcal{S}\rangle \hookrightarrow \langle \epsilon', \mathcal{S}'\rangle$ means that we reduce a reducible expression ϵ in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new reducible expression ϵ' and (possibly) updated store $\mathcal{S}'$. The store consists of a set of mappings from object identifiers $\mathit{oid} \in \mathrm{dom}(\mathcal{S})$ to tuples $\langle c, \{f \mapsto v\}\rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

The rule [*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 reducible expression 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.

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

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

Figure A.14: Translating expressions to reducible expressions (left) and variable substitution (right)

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 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.14). 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 paper we are not concerned with errors, so we do not introduce any special rules to generate an error value in these cases.

$\prec_P$ 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

$\langle c, m, x^*, e\rangle \in_P^* c' \iff c' = \mathit{min}\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$

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$(\prec_P) \cup \{\mathsf{Object}\}$

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.15: Relations and predicates for SMALLTALKLITE

$$P \vdash \langle E[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle E[\mathit{oid}], \mathcal{S}[\mathit{oid} \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P^* c\}\rangle]\rangle \quad [\mathit{new}]$$
where $\mathit{oid} \notin \text{dom}(\mathcal{S})$

$$P \vdash \langle E[o.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle \quad [\mathit{get}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = v$

$$P \vdash \langle E[o.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle \quad [\mathit{set}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle \quad [\mathit{send}]$$
where $\mathcal{S}[o] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in_P^* c'$

$$P \vdash \langle E[\textbf{super}\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle \quad [\mathit{super}]$$
where $c \prec_P c'$ and $\langle c', m, x^*, e\rangle \in_P^* c''$ and $c' \leq_P c''$

$$P \vdash \langle E[\textbf{let}\ x{=}v\ \textbf{in}\ \epsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\epsilon[v/x]], \mathcal{S}\rangle \quad [\mathit{let}]$$

Figure A.16: Reductions for SMALLTALKLITE