Experiments with Worlds

Aran Lunzer and Yoshiki Ohshima

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2013-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201   t: (818) 332-3001 f: (818) 244-9761

Experiments with Worlds

Aran Lunzer and Yoshiki Ohshima

1st March 2013

1 Introduction

This memo documents some experiments carried out in 2011 and 2012 based on Worlds, a mechanism for creating and using isolated code-execution contexts. The starting point was the Worlds/Squeak implementation created primarily by Alex Warth and Yoshiki Ohshima around the middle of 2011, and described (alongside a basic JavaScript implementation, Worlds/JS) in a paper presented at ECOOP that year [6][1]. The various additions and enhancements we have made to the Squeak code are in the latest version of the Worlds package[2].

We report on two directions of investigation. The first was an effort to extend Worlds/Squeak so that instances of OrderedCollection and IdentityDictionary, two widely used Squeak classes for handling growable collections, would provide the same useful inter-world features as were already assured for fixed-size objects and arrays. The second was to continue the investigation, already started in the ECOOP paper, into the variety of ways in which the protective properties of Worlds could be put to good use in applications.

At the end of the memo, in Section 4, we reflect on what we have learned from our various isolated investigations, and speculate on how these findings could be brought together in generalised form within a future generation of Worlds support.

1.1 World-aware versions of growable collection classes

As explained in the ECOOP paper, the Worlds/Squeak implementation takes the form of a library of classes that programmers can use to incorporate worlds into their applications. Apart from the WWorld class itself, support centres on WObject, an abstract superclass for classes whose instances are to be world aware (i.e., capable of holding different instance-variable values in different worlds). In addition, the classes WArray and WBitmap provide world-aware versions of Array and Bitmap, the key difference from WObject being that their slots are accessed by integer indices rather than by names.

All the classes in the original Worlds/Squeak have the property that the number of slots an instance has is set at creation and doesn't change thereafter. However, it is common in Squeak applications to use object structures that include instances of the growable collections OrderedCollection and IdentityDictionary. We saw it as a priority to provide world-aware versions of these classes.

The first issue to consider was whether the collection classes' standard implementations could be made world aware just by providing world-aware values for their instance variables. At first glance this seems plausible: for any nested object structure—or, more generally, an object graph—the entire structure will be world aware if every object in the graph is either (a) world aware or (b) immutable (such as a number or symbol). Given that both OrderedCollection and IdentityDictionary are implemented using an instance of Array to hold their contents, changing them to use WArray instances instead will ensure that changes made to a collection in some world w will have no effect on the collection as seen in w's parent or sibling worlds.

However, being usefully world aware is not just a matter of supporting independent updates in separate worlds. A further crucial feature of worlds is their so-called safety properties, described in

[1]This memo is best read in conjunction with that paper, which can be downloaded from `www.vpri.org/html/writings.php`; look for date 2011-06-03.

[2]Accessible as a Monticello package at `tinlizzie.org/updates/exploratory/packages/Worlds-<version>.mcz`

the ECOOP paper as being intended to prevent programming with worlds from being ‘error-prone or dangerous’. The following properties should hold between a world w_{parent} and another, w_{child}, that has been spawned from it:

no surprises: changes in w_{parent} ... should never make variables appear to change spontaneously in w_{child}

Implementation: Once a variable (or slot, memory location, etc.) has been read or modified in a world w, subsequent changes to that variable in w's parent world are not visible in w.

consistency: a commit from w_{child} should never leave w_{parent} in an inconsistent state

As in database management, a computation executed in w_{child} needs to appear to w_{parent} as if it was executed without any interruption. In other words, the entire computation done in w_{child} needs to be placed as an atomic (inseparable) unit in the serial progression of w_{parent}'s state changes. To ensure this property, at the time of a commit we check the serialisability of the computation executed in the child world.

Implementation: We employ an optimistic concurrency control scheme, in which a commit from w_{child} to w_{parent} is only allowed to happen if, at commit, all of the variables (or slots, memory locations, etc.) that were read in w_{child} have the same values in w_{parent} as they did when first read by w_{child}. This serialisability check ensures that the computation was done atomically. If the check fails an exception is thrown, and the commit is abandoned before any values in w_{parent} have been changed.

The fact that these properties are expressed in terms of individual object slots has an important implication for nested world-aware data structures, such as the example above of numbers held in nested arrays: if any non-leaf object in the structure is replaced as a side effect of implementation logic (as opposed to under intentional control by the application), there is a risk that the serialisability check will flag a clash where none really exists.

We illustrate this problem by considering a naive attempt to create a world-aware version of OrderedCollection. An OrderedCollection works by holding an Array that contains the items currently in the collection, and that has some number of empty slots available for new items. Roughly speaking, when an item is added and there is an empty slot, the item goes straight into that slot, and when the empty slots run out the collection ‘grows’ by replacing the internal array with a larger one and copying all the existing items into it. A naive world-aware implementation would replicate all the OrderedCollection methods in a new subclass of WObject, and would use a WArray for the internal array. This would, in fact, work fine across multiple worlds if the internal array never runs out of empty slots, but the code below shows what can happen when the collection needs to grow:

```
oc := WNaiveOrderedCollection new.  "say it starts with 3 empty slots"
oc add: 'a'.
oc add: 'b'.

w1 := thisWorld sprout.
w2 := thisWorld sprout.

w1 eval: [ oc printString ].  "reads all IVs, including the internal array"
w2 eval: [ oc printString ].  "ditto"

oc add: 'c'.  "still fits in the existing array"
w1 commit.    "no problem"

oc add: 'd'.  "grows oc, by replacing the internal array with a larger one"
w2 commit.    "*** refused ***"
```

Even though there are no evident updates in world w2 that would clash with the updates in its parent world, the serialisability check sees an incompatibility. The code evaluated in the parent world replaced the internal array after the code in w2 had accessed that array, so w2 is now considered uncommittable.

This is a general issue for managing nested data structures in the presence of worlds. In particular, it implied that implementing a useful world-aware version of OrderedCollection (and similarly IdentityDictionary, which also uses an internal array and grows by replacing it) required us to come up with an alternative to the naive approach. Some results of our experiments are presented in Section 2.

1.2 Patterns of use for Worlds

The original conception of Worlds as a way of controlling the scope of side-effects led to the canonical example being safe experimental execution of a fragment of code: a world is spawned as a kind of sandbox, the code fragment is run within this sandbox world, then only if the code succeeds are its changes committed to the parent world that represents the main program context. If the code fails, the child world can be abandoned and the main context will not have been harmed.

There are other ways of benefiting from the context isolation that worlds offer. The ECOOP paper shows some examples, such as the branching undo structure for a simple graphical editor. In Section 3 we provide a categorisation of these and other patterns of use, illustrating some with implementation examples.

2 World-aware growable collections

In this section we document our experiments in developing world-aware versions of Squeak's IdentityDictionary and OrderedCollection classes, including the work of extending the policies for inter-world safety properties (as described in the Introduction) to work on objects that can change their numbers of slots on the fly.

First, it's important to be clear on some limits to the protection offered by the safety properties. Consistent with the principal goal of Worlds being to protect an execution context from changes (side effects) caused by execution that happens in a child world, the *consistency* check does offer a strong guarantee of protection for a parent world. By contrast, the *no surprises* property is relatively weak: all it guarantees is that *for a single slot, in a single object* the child world will see the same value each time it accesses the slot, even if between those accesses the slot is being updated in a parent world (or in a more distant ancestor). The property does not guarantee anything about consistency *between* slots accessed in the child world. Consider the following code:

```
array := WArray new: 2.
array at: 1 put: 10.
array at: 2 put: 20.
w := thisWorld sprout.
w eval: [ a1 := array at: 1 ].  "a1 = 10"
array at: 1 put: 100.
array at: 2 put: 200.
w eval: [ array asString ].     "evaluates to { 10. 200 }"
```

The no-surprises policy ensures that the first element of array, which was found to be 10 when initially accessed from child world w, stays as 10 whenever subsequently used in w. But this example shows that a child world's code is still open to another form of surprise: that of seeing incompatible combinations for values of slots (such the two elements of array) to which the child's accesses were interleaved with updates in the parent.

The example is also an illustration of a further trap: now that world `w` has incompatible values for the slots within `array`, it will be prevented by the serialisability check from committing *any* changes (not just to `array`) into its parent world. This is the strong *consistency* property working as designed, protecting the parent, but the inability to commit makes `w` into what we call a zombie world.

It is important to bear in mind that the protections built into Worlds were never intended to act as concurrency controls that would prevent problems like this from arising. If a multi-world application is at risk of ending up with inconsistent or meaningless values because of interleaved evaluation between child and parent worlds, the programmer needs to put in place additional protection mechanisms, just as for any multi-threaded body of code. One form of protection (or, more accurately, of repair) is the use of Squeak's powerful exception-handling facilities to work around any serialisability failures that do arise. An exception handler can respond to exceptions by making application-specific fix-ups, then resuming execution. Later (in Section 4.5) we show how this could work.

In the mean time, while being aware of the limits to the powers of the safety properties, we decided that our implementations of the world-aware growable collections should at least incorporate *no surprises* and *consistency* protections of similar strength of those that apply to fixed-size objects. We also felt free to experiment with extra forms of protection that could prove helpful, such as the 'snapshot' feature described below as part of our world-aware implementation of OrderedCollection.

2.1 WIdentityDictionary

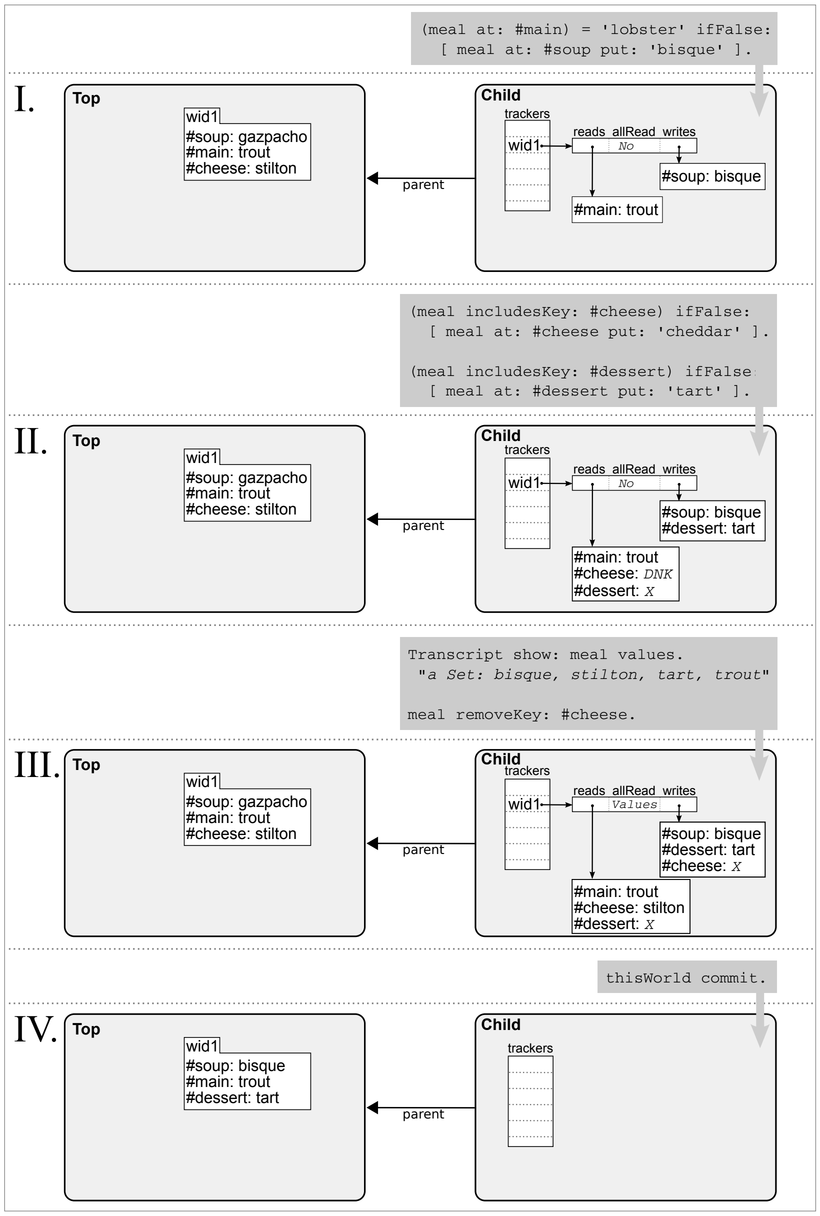
WIdentityDictionary is a world-aware version of the IdentityDictionary class, in which values are stored under keys that can be arbitrary objects[3]. Implementing WIdentityDictionary involved adapting the existing per-slot access mechanisms to handle per-key access, taking account of the fact that dictionary entries can be added and removed, on the fly, in any world that references the dictionary. It is to be expected that a dictionary can have some keys in a child world that it doesn't have in the parent, and vice versa. This called for a new design for the update trackers—the data structures used for tracking the different values held for a given object slot in the different worlds in which the object participates (i.e., the role performed by the *reads* and *writes* collections used by WObject, WArray etc).

We decided that the trackers needed to distinguish between an access to just a key (e.g., using the `includesKey:` method), and access to a full key/value pair (e.g., using `at:`). As an illustration of why this is important, consider a piece of code that is handling dictionaries that represent database-like records. At some point the code may query the keys of such a record, just to decide what kind of record it is—say, if it has a 'balance' key then the code can treat it as a bank account. If this check also created a *reads* record for the current value held under that key (despite the code not requesting it), the serialisability check for a subsequent `commit` would fail if the balance value had changed in the mean time. That would be an undesirable outcome.

Deciding to distinguish the handling of `includesKey:` and `at:` messages in this way felt obvious and reasonable, but—as will be seen below—there were other places where trying to 'do the right thing' in WIdentityDictionary code to help applications run in predictable and safe ways was much less clear cut. We made what we believe are reasonable decisions, but can see that for some applications the resulting protections could cause their own problems. Part of the discussion in Section 4 addresses how to give programmers a say in how protection should work in their own applications.

Figure 1 shows a simple example of WIdentityDictionary usage, including the workings of the trackers.

[3]The 'Identity' aspect signifies that a value stored under a key k can only be accessed later by specifying exactly the same object k. In Squeak this means, for example, that a String instance is typically not a useful key for an IdentityDictionary, because multiple instances of a character sequence such as `'abc'` are all distinct String objects. Instances of Symbol (such as `#abc`), by contrast, are unique for a given sequence of letters and are therefore suitable for use as IdentityDictionary keys.


```
(meal at: #main) = 'lobster' ifFalse:
  [ meal at: #soup put: 'bisque' ].
```

```
(meal includesKey: #cheese) ifFalse:
  [ meal at: #cheese put: 'cheddar' ].

(meal includesKey: #dessert) ifFalse
  [ meal at: #dessert put: 'tart' ].
```

```
Transcript show: meal values.
 "a Set: bisque, stilton, tart, trout"

meal removeKey: #cheese.
```

```
thisWorld commit.
```

Figure 1: Specialised per-world update trackers for WIdentityDictionary. (I) The variable `meal` points to a WID that has been created and given three entries in the Top world. Code accessing and updating `meal` in the Child world (that was directly spawned from Top) has created *reads* and *writes* records. (II) Checking for existence of a key, without accessing the value, creates a *reads* record with value `Don't Know` (DNK). A key that does not exist in the parent is recorded with `Not There` (X). (III) Querying all values creates a union of values from the child and ancestor worlds, as usual; the `Don't Know` value in *reads* is replaced by the read value. Removing a key creates a `Not There` *writes* record. (IV) Committing to the parent succeeded, because all *reads* records were consistent with the dictionary's state in Top. Changes and removals made in Child have been applied.

2.1.1 'No surprises' protection

The original conception of the 'no surprises' principle, as reproduced in the Introduction, is that changes in the parent world should never make variables appear to change spontaneously in the child. The current implementation of WIdentityDictionary extends that principle as needed to cope with the possibility that code in the parent will add or remove keys, rather than just changing the values associated with them. In particular:

- If code in the child world attempts to access a key *k* that is not in the dictionary, we record the *absence* of *k* as part of the state read from the parent. Even if code in the parent later adds a value under *k*, it will not be seen by the child.
- Conversely, if the first lookup of *k* finds that it is present in the dictionary, *k* will remain visible in the child world even if it is removed in the parent.
- When the child world accesses the value under *k*, that value is recorded so that future accesses to *k* will always return the same value.

  One corollary of keeping separate records of key and value access is the possibility that a child world initially tests for the presence of *k* (using `includesKey:`) and finds that it is there, but at some time later attempts to access the value at *k* (using `at:`) and finds that in the parent world it has now been deleted. We regard this as a form of surprise that the Worlds mechanisms are not responsible for preventing; it could happen just as easily in non-world-aware but multi-threaded code, in which one thread tests for the existence of a key then later attempts to access that key, but in the mean time a second thread has removed it. The WIdentityDictionary, in this situation, will behave as if the key is now absent, and will change the *reads* record to reflect this absence. Although changing the record seems to contravene the no-surprises rule, the surprise—*k* was there, and now it's gone—has already happened (and through no fault of the Worlds code). Recording the absence now will at least ensure that no further surprises occur.

  Note that if code in a child world removes a key, this is also recorded as an access to the value under that key. This is in line with standard Smalltalk practice of returning the removed value as the result of any `remove:` operation.
- If code in the child world enumerates the dictionary's entire contents, not only do we record each key/value pair that is found in (or via) the parent world, but also make a note of the fact that *all* the parent's keys have now been seen. This guarantees that subsequent enumerations will be given the same keys.

  The *allRead* annotation in fact has two states other than its initial `false`: `Keys` or `Values`, representing whether the child world has accessed just the keys, or the values too. The use of `Values` is primarily for efficiency, in that it allows future enumerations of the dictionary in the child world to happen without any need to access ancestor worlds.

2.1.2 Consistency protection

Consistency protection hinges, as usual, on a serialisability check that confirms that the state of slots read by the child world is the same as the state currently prevailing in the parent. Again, the ability for slots to come and go requires some special handling:

- If the child world has read a key/value pair, we must confirm that the dictionary has that same pair in the parent world.
- If the child has recorded the presence of a key *k*, but not its value, we check only that *k* is still present in the dictionary in the parent world.

- Similarly, if the child has recorded the *absence* of a key, we confirm that the key is still *not* present in the parent world.

In pseudo-code, the procedure for serialisability-checking a WIdentityDictionary is therefore as follows:

1. If there is no *reads* collection, **OK**.
2. (not currently implemented) If the child world has recorded that it has seen all keys, ask the parent world for all the keys now in the collection. If the set of keys seen in the child does not match this, **FAIL**.
3. For each key in *reads*
    - If the value is `Not There`, and the parent (or other ancestor) has the key, **FAIL**.
    - If the value is `DNK`, and the parent (or other ancestor) doesn't have the key, **FAIL**.
    - For any other value, if the parent doesn't have the same value, **FAIL**.
4. **OK**.

The second check is not currently implemented because it would require more detailed *reads* records than we currently maintain. The intention is that if there is a record of having seen all the collection's keys in the ancestor worlds, in principle the `commit` should not proceed if the keys as seen now in the parent world are different. This is because the code in the child world might have generated some value based on the keys it has seen (perhaps just based on the overall number of keys), and if this value is committed into the parent it could be inconsistent with the current state of the dictionary there.

The reason for wanting to add this kind of condition to the serialisability check is that, when working with objects that have variable numbers of slots, it is common to aggregate slot values and to make code decisions based on these secondary properties. If a child world has calculated values based on a dictionary having 10 keys, say, it would be dangerous to allow these values to be committed into a world where its size is now 11. And, strictly speaking, the code in such a case would indeed not be properly 'serialisable' in the parent.

This was not an issue when Worlds only catered for fixed-slot objects, but the cat is now decisively out of the bag. And there is no general way to catch all cases: as we considered the possible ways to protect a parent world, we found that we could think of some application examples that called for strict checking, and others for which that same checking would be obstructive (typically by refusing to commit changes that would in fact—for that application context—be safe). This started us thinking about how to let programmers take control for themselves over the boundaries of the checks; some possible directions are outlined in Sections 4.4 and 4.5.

2.2 WCollection

Instances of Squeak's OrderedCollection have variable length, and allow addition or removal at any point. WCollection, the world-aware replacement for OrderedCollection, is designed to work similarly.

Like WIdentityDictionary, the implementation of WCollection must allow for a given collection to have more slots in some worlds than in others, and for items—and hence slots—to be added or removed freely. However, when we considered the impact of such changes between worlds we realised that WCollection was going to be especially fragile, in the sense of how readily operations in a parent world would lead to a child world reading inconsistent values (and thus becoming a 'zombie').

A typical path to creation of a zombie world was illustrated on page 3, with a section of code that creates and modifies an array. Expressed abstractly, a child world becomes a zombie if its accesses to object values (slots) are interleaved with updates in the parent world as follows:

1. child reads some object slot s_a
2. parent replaces the value in s_a, and also in some other slot s_b
3. child reads s_b

Note that slots s_a and s_b can even be in different objects. The risk thus resembles roulette: when a number of slots are updated in a given world, will they happen to include some slots that have already been accessed from a child world, and other slots that the child is about to access? In the example on page 3 the parent-world code replaced the entire contents of a WArray, harming a child world that had read just part of the contents and was now going on to read the rest... and one could say that replacing the entire contents of a collection was bound to invite trouble. By contrast, for a simplistic implementation of a world-aware OrderedCollection even the following code would create a zombie world:

```
meal := WSimpleOrderedCollection with: 'starter' with: 'main' with: 'dessert'.
w := thisWorld sprout.
w eval: [ Transcript show: meal first ].    "starter"
meal addFirst: 'soup'.  "a typical operation, in the parent world"
w eval: [ Transcript show: meal second ].   "starter ??"
```

At the end of this code segment, not only is the child world `w` now a zombie (blocking *any* changes from being committed to the parent, not just changes to the `meal` collection), but the value it has just read from index 2 is exactly the one it previously read from index 1. The usefulness of the collection's behaviour in `w` has been compromised.

As an antidote to this fragility, we decided to use WCollection as the place to try out a 'snapshot' feature that would provide some additional protection. The idea is to give a child world a snapshot of the values in the slots of one or more objects, such that any later changes to those slots in (or above) the parent world are hidden from the child. Code in the child can then read the slot values at leisure, without risk of being wrong-footed by interleaved parent-world changes. In the case of WCollection the snapshot is taken automatically on the first access to any aspect of the collection, and covers the collection's contents at that instant. With this feature, the above code runs as follows:

```
meal := WCollection with: 'starter' with: 'main' with: 'dessert'.
w := thisWorld sprout.
"*** first access to meal installs (in w) a snapshot of the whole collection ***"
w eval: [ Transcript show: meal first ].    "starter (read from snapshot)"
meal addFirst: 'soup'.  "in the parent world"
w eval: [ Transcript show: meal second ].   "main (read from snapshot)"
```

Note that a snapshot itself does not create any *reads* records. These are created, as usual, when code in the child world explicitly accesses slots, and the serialisability check in any later `commit` will examine only those slots. What the snapshot does is to guarantee that if and when the child world accesses the collection, it will find contents that are internally consistent (assuming, of course, that they were consistent in the parent world at the time of the snapshot). The snapshot also means that operations on a WCollection will never create a zombie world.

A side benefit of the snapshot feature is efficiency: once a child world has a snapshot of the contents of a WCollection, no further access to the collection will call on the computationally expensive slot-lookup mechanism typically used by world-aware objects. Given how common it is for Smalltalk code to use OrderedCollection instances, we found this saving to be crucial in maintaining acceptable performance of Squeak applications when they are run in a multi-world setup.

Although the current implementation of snapshot is embedded within the code for WCollection, it could be useful for world-aware objects in general. It would also be possible to make snapshot a feature that programmers can choose to invoke explicitly, rather than happening automatically on first access to an object. These possibilities are considered in Section 4.3.

2.2.1 'No surprises' protection

Just as for WIdentityDictionary, the flexible nature of a WCollection required some decisions on how to augment the standard protections offered by worlds. Our decision to deploy the snapshot semantics, described above, automatically provides the following blanket form of protection for a child world:

- As soon as code in the child world accesses any aspect of the collection—reading or overwriting any element, inserting or removing an element, enquiring about the collection's size or about the existence of some index—a snapshot of the collection is taken from the parent world. Thereafter, no changes to the collection in the parent will have any impact on the collection as seen in the child.

2.2.2 Consistency protection

While the snapshot feature makes it trivial to describe the child world's view of a WCollection, the serialisability check for determining whether an updated collection can be committed to a parent world requires more detailed specification. As emphasised above, the normal serialisability rules still apply—namely, only values that have explicitly been read in the child world are to be checked for consistency with the parent world at the time of `commit`. But there are some subtleties to be taken into account in applying the rules:

- If code in the child world has requested the collection's size—either explicitly, through the `size` method, or implicitly by any operation that changes the size or that tests it (e.g., by attempting to access an element that is off the end)—confirm that the collection in the parent now has the same size as when the snapshot was taken.

  Note again that *any* operation on the collection—reading or writing—will cause a snapshot to be taken. But not every operation accesses the size, and if the child world has not accessed the size then it would be obstructive to insist on size agreement at the time of `commit`. For example, consider a child world in which the only operation has been to read and then overwrite the first element of a non-empty WCollection. This change can be committed to the parent world as long as the first element in the parent still has the value that was read in the child, regardless of whether the overall size of the collection has changed. By contrast, if the child world had performed the operation explicitly on the *last* index in the collection (for example, using a method such as `last` to read it) then serialisability is only maintained if the size in the parent world is the same as it was before; in this case, the access to size is recorded and its agreement must now be checked.

- If the child world has read value v from index i, in the parent world the same value must appear at the same index.

  As a corollary, if in the parent world the collection now has fewer than i elements, the check will fail.

  Removing the ith element is treated as reading the value v from index i then overwriting it. The parent must still have v at index i. Removal from anywhere other than the end of the collection will cause the subsequent elements to move one place nearer the start, while removal from the end will cause overwriting of the last element with the `Not there` value. In the latter case, if `commit` goes ahead the `Not there` will cause truncation of the collection in the parent.

- If the child world has updated the element at index *i*, in the parent world the collection must have at least *i-1* elements.

  Say the collection had four elements when it was read from the child world, and the child has overwritten the element at index 3 (reminder: Smalltalk's indexed collections are numbered from 1, and indices must be contiguous). This rule says the change can be committed to the parent world as long as it still has at least two elements in the collection. If there are exactly two, the change will cause an append. If there were only one, making an update at index 3 would raise the question of what should appear at index 2. We avoid this question by rejecting the `commit`.

- If the child world has run a search through the collection (using methods such as `includes:`, `indexOf:`, `lastIndexOf:`, etc), the parent must have values that would cause the same result to be obtained. Some example cases:

  – A successful `includes:` search records a read just at the index where the sought value was found. The parent must still have that value at that index.

  – A successful `indexOf:` or `lastIndexOf:` search records a read of all items from the start or end of the collection (respectively) as far as the first match. The parent must have all the same values, at the same indices.

  – An unsuccessful search records a read of all elements in the collection at the time of the search. It also records a read of the collection size, if it had not already been recorded. Again the parent must have all the values, at the same indices.

Based on these rules, the procedure for serialisability-checking a WCollection is as follows:

1. If the size of the collection was read, and its size in the parent is now different, **FAIL**.
2. If no elements have been read or written, **OK**.
3. If an element read from the parent at index *i* is different from the parent's current value at *i*—including if the parent now has fewer than *i* elements—**FAIL**.
4. If no elements have been written, **OK**.
5. If an element has been written to index *i*, and the parent has fewer than *i-1* elements, and no element has been written to index *i-1*, **FAIL**
6. **OK**.

It is probably clear to the reader, from the discussion of how we treat access to the WCollection's size, and the serialisability impact of successful and unsuccessful searches, that we are again in territory similar to the questions of how strictly a WIdentityDictionary should record failed key lookups. There is no single way to 'do the right thing', so we believe that in the future it would make sense to allow programmers to tailor the protections, along the lines sketched out in Section 4.4.

2.2.3 Performance

As one of our experiments, we made some rough measurements of the performance impact of deploying a world-aware version of OrderedCollection.

Even with the performance benefits that accrue from the snapshot semantics, and delegating internally to the existing highly optimised OrderedCollection class, there is still a considerable overhead in using our WCollection. We ran some tests using the following simple adding/removing benchmark in the two classes (not showing here the loop iteration used to improve the time-measurement accuracy):

```
simpleBenchmark: n
  "In a child world, add n items to the start of a collection
   then remove one by one from the end"
  WWorld thisWorld sprout eval: [
    collection := self new.
    1 to: n do: [ :i | collection addFirst: i ].
    [ collection isEmpty ] whileFalse: [ collection removeLast ]
  ].
```

The average timings for one loop (add n items, then remove them) for OrderedCollection were $n = 10$: 0.00095ms; $n = 100$: 0.0089ms; $n = 1000$: 0.082ms. The equivalents for WCollection, and the factors relative to OrderedCollection, were $n = 10$: 0.015ms (16x); $n = 100$: 0.20ms (22x); $n = 1000$: 2.7ms (33x).

In addition, and driven partly by our interest in the eventual unification of all world-aware object types, we ran some tests on a version of WCollection implemented as a subclass of WIdentityDictionary. It is called WSimpleOrderedCollection, and uses integer keys to represent the collection indices. Relative to its WIdentityDictionary superclass it adds or overrides around 60 methods (as compared to the roughly 170 methods used to implement WCollection). The implementation benefits from the fast VM-plugin primitive methods we had written for WIdentityDictionary, but does not take snapshots—indeed, the main reason we built this class was to explore how the world safety properties would work out on an OrderedCollection implementation with and without the snapshot semantics.

Performance of WSimpleOrderedCollection on the add-at-start benchmark used above, even though it performs no reads from the parent world (where having a snapshot would make a difference), is predictably poor: $n = 10$: 0.38ms (400x); $n = 100$: 103ms (11600x); $n = 1000$: 81000ms (1 million times). These figures can be seen as a reminder that designing for performance, though not something that we want to divert us from seeking designs with elegance and simplicity, is also not something that can be ignored indefinitely.

3 Ways of using Worlds

The ECOOP paper described a number of possible uses for the Worlds mechanisms. Here we gather those examples, and a couple more, into a broad categorisation of how worlds can be useful. We begin by defining a set of categories that, although there is some overlap between them, capture what we feel are useful distinctions among alternative modes of use for worlds:

Isolated local state If a world is spawned as the child of the main execution context, and some aspects of the execution-context state are modified in the child, it can be retained indefinitely (practically speaking, for the lifetime of the context from which it was spawned) as a form of sandbox in which those modifications are in force. Code evaluated in the sandbox will be subject to the modifications, but they will not leak through to the main execution context.

One example of this mode of use is the simple module system for JavaScript described in Section 3.3 of the ECOOP paper. The 'undo for applications' example in Section 3.2 of that paper can also be seen as a use of this concept, insofar as the application initially sets itself up in a world isolated from the main execution context (though in that particular example the application then goes on to spawn further child worlds, making it an instance of the 'Cumulative histories' usage described below).

Safe experimental execution This is perhaps the archetypal use of worlds: a child world is spawned, some code that the programmer knows might fail is executed there, and only if the code's results are satisfactory are they committed into the parent world representing the main execution context. Otherwise the child world, along with its uninteresting partial results, can simply be abandoned; the main execution context suffers no harm.

Section 3.1 of the ECOOP paper describes this mode of use as providing a clean way to unwind from exceptions. Section 7 of the paper shows how it can also form a useful basis for a computation that relies on being able to backtrack, the example given there being ordered choice within a Parsing Expression Grammar.

Progressive branching alternatives An alternative to the experiment-and-backtrack approach suggested above for parsing a grammar is to spawn multiple worlds at a choice point—one for each choice—and feed subsequent tokens on the input stream to all the resulting branches in parallel. If a branch encounters a further choice point, it recursively spawns sub-branches. When a branch fails to match the input against its next expected token, it is deemed to have failed and is dropped (harmlessly) from the list of active branches. A branch that handles the entire input stream represents a successful parse.

Using worlds to hold branching alternatives can be applied not just to grammars, with their clear success/failure signals, but to any computation that involves processing a fixed number of steps (like the tokens in an input stream) using an algorithm that encounters branching points. In Section 3.3 below we show its application to line breaking for text.

Parallel alternatives Instead of spawning multiple independent child worlds on the fly when encountering a branch in an algorithm, this mode of use is based on starting out with a pre-defined set of worlds initialised with usefully distinct states that a user/programmer wants to work with. Like the 'progressive branching alternatives' case, typically all the worlds are then driven in parallel with the same sequence of updates, but respond to those updates in accordance with their differing setups.

Using worlds in this way would be the basis for supporting a subjunctive interface: an interface in which alternative scenarios can be set up, displayed, and manipulated in parallel [2, 3]. In Section 3.4 we show a simple demonstration of such an interface based on a spreadsheet-like table object built within the STEPS project.

Cumulative histories An application that is currently executing in some world w_i can spawn a child world w_j and switch to executing there instead. The results of the computation will not be affected, but w_i will remain as a holder of the application state as it was at the time of the switch. Some time later, w_j can spawn its own child world in turn and switch execution to that. The growing chain of worlds will in effect hold snapshots of the application's state at the switching timepoints.

This is in essence nothing more than a nested case of 'safe experimental execution', but the practice of retaining each spawned world, rather than discarding it after a single experiment, provides a new level of usefulness. Section 3.2 in the ECOOP paper explains how timing the context switches to coincide with the invocation of commands in an API allows the worlds chain to be used to implement 'undo'. Section 6.1 of that paper shows the concept expanded to a branching tree of application states, supporting 'tree undo' in a graphical editor.

In the following sections we provide a little more detail for each of these modes of use, and in some cases present examples of their deployment.

3.1 Isolated local state

The example described in Section 3.3 of the ECOOP paper shows how, in a dynamic language such as JavaScript, a world can be used as an isolated context in which some methods have been given different definitions from those found in the main execution context. Code evaluated in this world will reflect those alternative definitions.

That example showed only the use of pure functions, but for some kinds of module it may also be useful that the world can be used to capture state that persists between the evaluations it is asked to perform; this is in part what the application example in Section 3.2 of the ECOOP paper is doing. Indeed, in a language with support for such flexible modification of methods there is no meaningful distinction between state held in method dictionaries and state held in other object slots.

3.2 Safe experimental execution

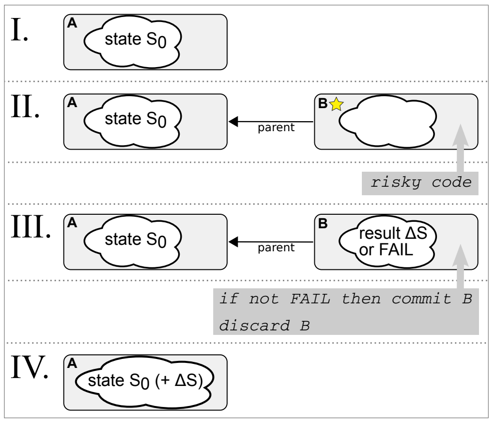


Figure 2: Safe experimental execution. (I) World A holds the system state before the experiment. (II) World B is spawned from A (we use a yellow star to highlight that this is the first time B has been seen), then some experimental code is evaluated in the context of B. (III) If the code succeeded, B now has a resulting delta to the system state that can be committed to A; otherwise, no `commit` will take place. (IV) World B has been discarded; A is either updated with the change of state or, if the code failed, is unchanged from before.

Figure 2 represents the use of worlds for safe experimental execution. It shows a case where the experiment can either succeed or fail; only in the case of success are the experiment's results committed into the world representing the main execution context.

Apart from isolating the execution context from side effects accumulated during an evaluation that eventually failed, there is another way in which a world can be used for conditionally committing the results of an experiment. Even if the code within the experiment world completes successfully, the `commit` into the parent represents an additional checkpoint with strong protections, as discussed at length earlier in this memo. The serialisability-based check used within `commit` is exactly what would be needed to support safe processing of transactions, as follows:

- One world w is treated as holding the application's mainstream data.
- For each transaction t_k a child world w_{tk} is spawned from w, and the transaction's operations are performed in the child.
- If the operations complete successfully, the application attempts to commit the resulting changes from w_{tk} into w.
- A commit will only go ahead if w_{tk} passes the serialisability check. If not, the commit is rejected and t_k as a whole is marked as having failed.

- A transaction that fails—either because of a problem encountered while executing it, or a problem in committing—has no effect on w; no cleanup or 'rollback' is required. The application is free to re-try the transaction, if the reason for failure indicates that it might succeed on a second attempt.

Bank accounts are a standard way to illustrate transaction processing. Consider the following two bank transactions:

- t_p: Move $700 from the savings account of customer A to her checking account.
- t_q: Move $350 from the same savings account to the savings account of customer B.

If t_p and t_q happen to run in parallel, both will find the same balance in the savings account. One or the other will come to commit first, and the (atomic) commit operation will allow it through, changing the account's balance. Then the other transaction will try to commit, and will discover that the account now holds a different balance from the one it used. The commit of this second transaction will be rejected by the serialisability check. However, because the system responsible for processing transactions can see that the only problem was the commit, it can re-run the rejected transaction repeatedly (in a newly spawned world each time) until it goes through.

3.3 Progressive branching alternatives

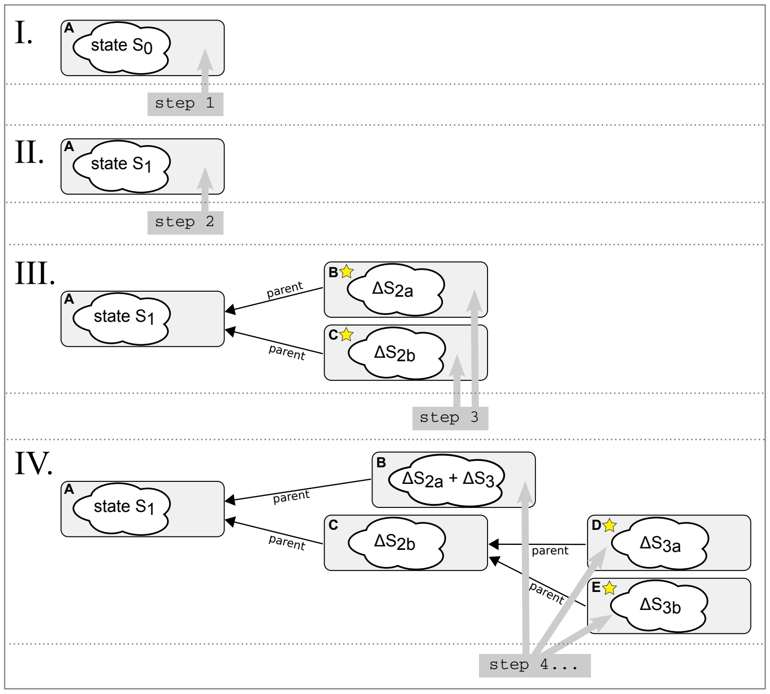


Figure 3: Progressive branching alternatives. (I) Starting state, ready for step 1. (II) Step 1 did not involve multiple options, so there is still just one world. (III) Step 2 had two optional processing paths, so worlds B and C were created and now hold the respective deltas from pursuing those paths. Step 3 will be applied to both these worlds. (IV) In world B step 3 did not have multiple options, but in world C it caused a further split into two. Step 4 will be applied to all three leaf worlds... and so on.

Figure 3 represents the use of worlds for carrying out a sequence of processing steps, where each step may be discovered to have two or more options for its evaluation. When this happens, separate child

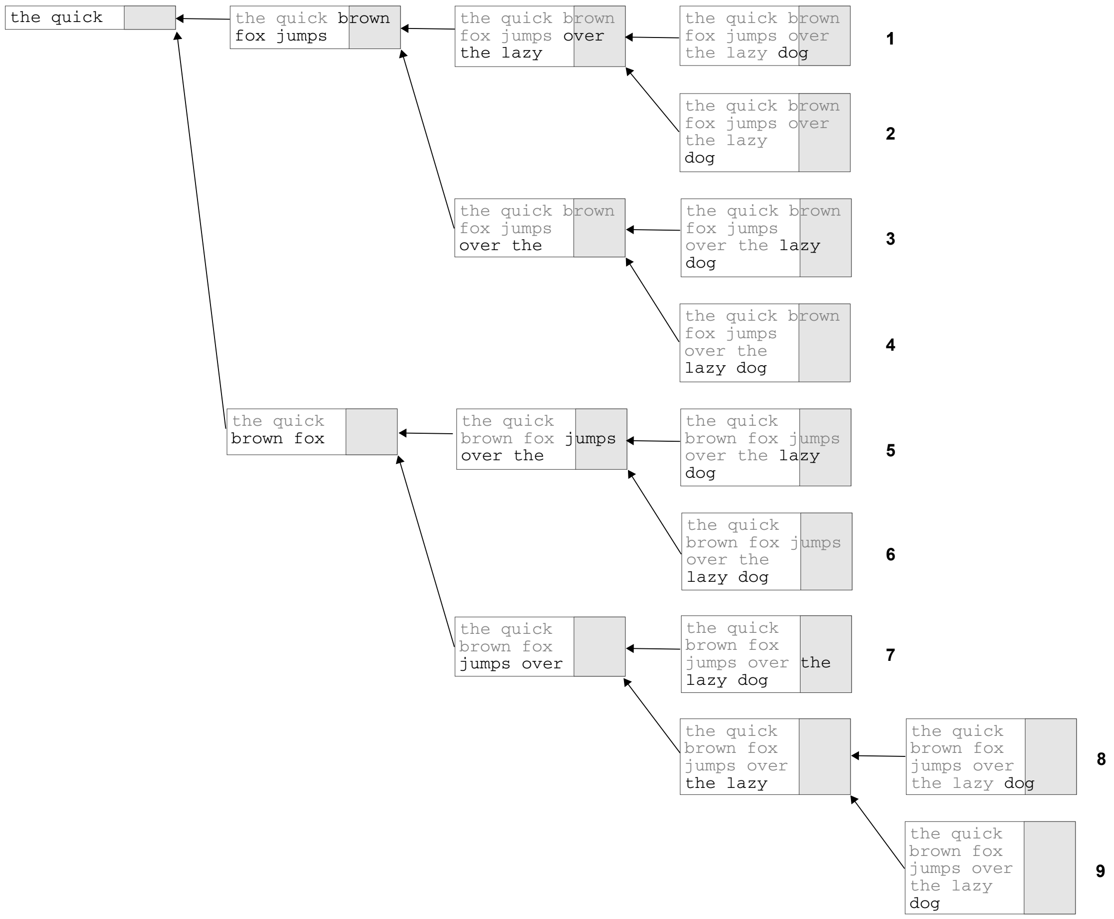


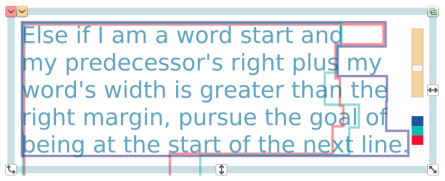
Figure 4: Text layout with progressive branching. Each box corresponds to a world in which layout has been carried out for some sub-sequence of words in the paragraph, with a pointer back to the parent world from which it was spawned due to a branch point in the layout. The shaded zone at the right of the text box represents the user-chosen range for the right margin. The first word that ends within this range is 'brown', so that is the first branching point. In the two worlds after a branch, the word-placement state inherited from the parent is shown in grey, while the world's own 'delta' is in black. After all words have been added, in this case there are nine leaf worlds representing nine possible layouts.

worlds are spawned to be the evaluation contexts for each of the options; this gives rise to a tree of worlds, with the leaves holding the latest state. Subsequent processing steps are notionally evaluated in all the leaf worlds, although for some applications there may be a way to check whether the state in a given world is no longer sufficiently interesting to merit further pursuit, in which case that world can be discarded.

We deployed this mode of use for worlds in an algorithm for laying out a paragraph of text into a box, where instead of specifying a single right-margin position the user can indicate flexibility in acceptable text width by specifying the margin as a range. The margin range is interpreted as meaning that each line of text can be broken ahead of any word that would end somewhere within the range. If this range is large (and especially if words are short), as the line comes close to being full there may be several points where the next word could either be added to the current line or used to start the next line. At each such point, our algorithm spawns two child worlds to hold the result of placing the word on either line. In a little more detail:

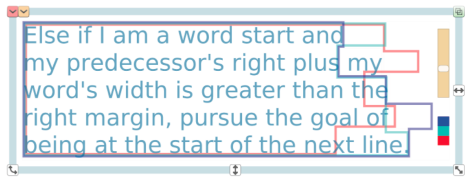
- A world represents an in-progress layout, embodying some set of choices about where to insert line breaks.
- A single step in the layout involves placement of a single word. Each currently existing world is instructed to place that word (i.e., assign it an x, y position), and does so in accordance with the state of the layout in that world.
- Whenever there is a choice as to whether to start a new line, one world splits into two. Each continuing world's inherited state includes all layout details (i.e., word positions) that have been decided up to the point of the split. After the split the worlds are independent.
- Once the last word has been placed in all worlds, a complete layout is read from each world.

A small worked example of this algorithm is shown schematically in Figure 4. As seen in this figure, the nine alternative results obtained by pursuing the simplistic algorithm represent a wide range of shapes for this short paragraph, many rather ungainly. Figure 5 shows one attempt to provide a helpful interface to this form of layout procedure, by ranking the results according to the evenness of their line lengths and revealing a few of the highly ranked results' outlines for the user to compare.



Else if I am a word start and
my predecessor's right plus my
word's width is greater than the
right margin, pursue the goal of
being at the start of the next line.

(a)



Else if I am a word start and
my predecessor's right plus my
word's width is greater than the
right margin, pursue the goal of
being at the start of the next line.

(b)

Figure 5: An interface to multi-world branching text layout, implemented within the STEPS document editor. Candidate layouts for a paragraph have been ranked using a simple score that favours consistent line lengths. The words are shown in their positions for the top-ranked layout, while coloured outlines show the shapes of the next three layouts in the ranking, to help the user decide whether the top layout is indeed the best. The two figures are for alternative maximum widths; although the top-ranked layout is the same in each case, the greater space available in (b) turns out to have enabled all four highly scored layouts to fit within five text lines.

3.4 Parallel alternatives

Figures 6 and 7 represent two patterns of using worlds to set up and drive a number of alternative application states in parallel. The difference between the patterns is in whether updates are applied directly to the leaf worlds or to a single root world that the alternatives all share as a parent. The latter notionally involves less processing, suggesting that it could be more efficient, but it only suits cases in which the differences between the branches do not diverge beyond their initial customisation.

An example where the first pattern is appropriate (updates are applied to each leaf world in turn) is a spreadsheet application in which the alternatives represent different values for a cell in the midst of the calculation. The leaf worlds will differ not just in the value of that cell, but in the values of all cells derived from it.

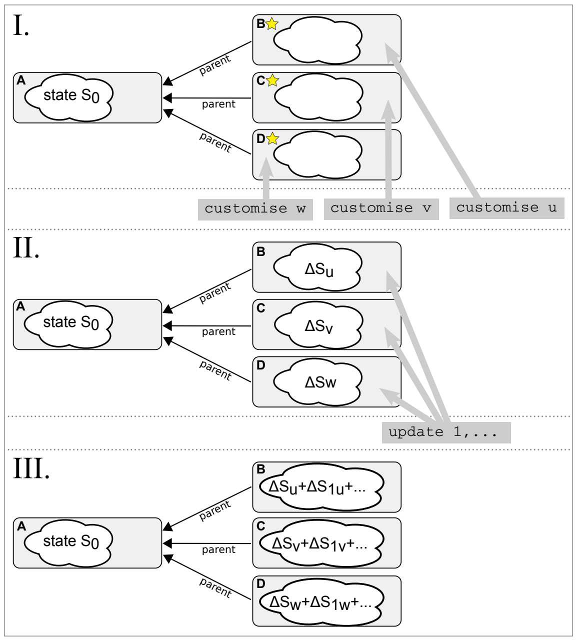


Figure 6: Parallel alternatives. (I) Worlds B, C, D have been spawned from A and are ready to be customised. (II) The customisation has given each child world a delta to the system state. Code representing further updates to the application will be fed to all three leaf worlds. (III) As the updates proceed, each leaf world accumulates state changes that depend on its customisations.

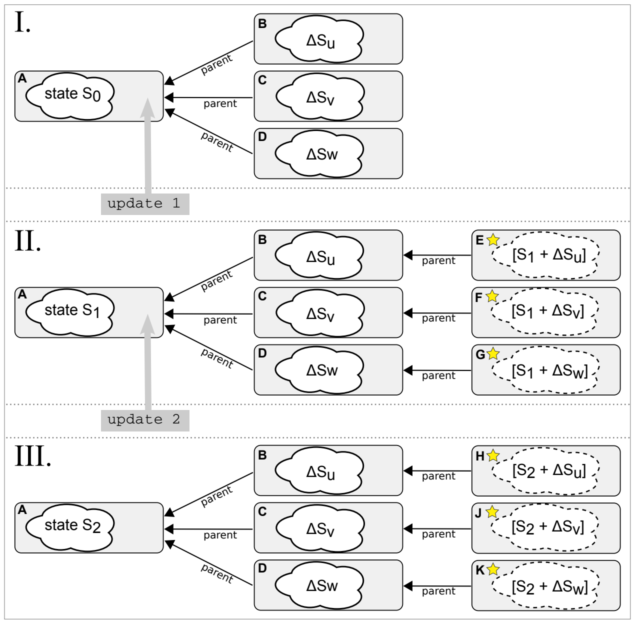
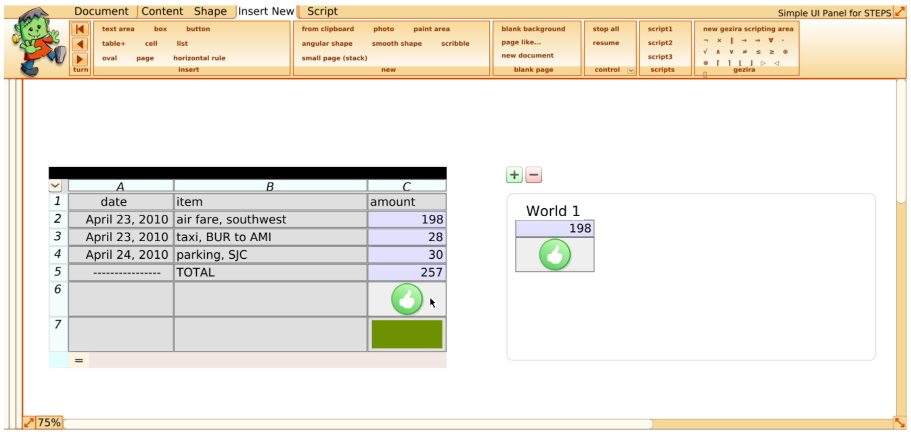


Figure 7: Parallel alternatives with evolving shared state. (I) The branches have been set up, as in panel II of Figure 6—but in this case application updates are going to be applied to world A, the main execution context. (II) After update 1, worlds E, F, G are spawned to act as views onto the branches' respective states without imposing any lasting change. (III) For update 2, worlds E, F, G have been discarded and new ones H, J, K spawned as views onto the updated state.

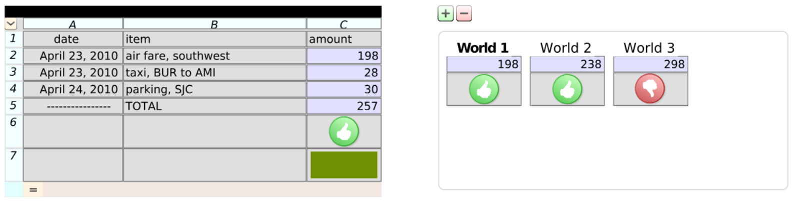


	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	30
5	----------------	TOTAL	257
6			👍
7			🟩

=

World 1
198
👍

(a)

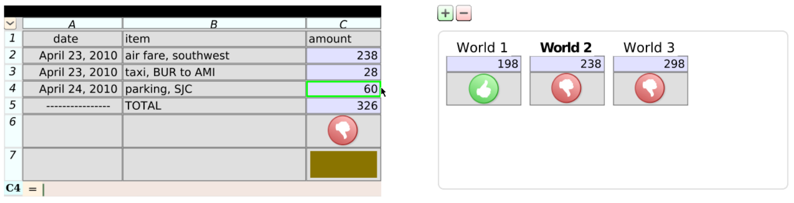


	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	30
5	----------------	TOTAL	257
6			👍
7			🟩

=

World 1	World 2	World 3
198	238	298
👍	👍	👎

(b)

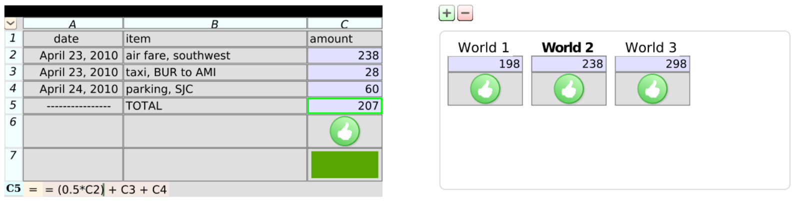


	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	238
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	60
5	----------------	TOTAL	326
6			👎
7			🟫

C4 = |

World 1	**World 2**	World 3
198	238	298
👍	👎	👎

(c)



	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	238
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	60
5	----------------	TOTAL	207
6			👍
7			🟩

C5 = = (0.5*C2) + C3 + C4

World 1	**World 2**	World 3
198	238	298
👍	👍	👍

(d)

Figure 8: A spreadsheet operating with parallel alternatives. In the top figure (which shows the sheet as a document in the STEPS project's Frank editor) the user has chosen the air fare cell and the thumb-up/down cell (indicating whether the total falls below the $300 budget-approval threshold) to designate them as cells whose values in each alternative are of interest. Copies of these two cells appear in the world palette on the right-hand side. Panel (b) shows the table and world palette after the user has created two extra alternatives, and has entered different air fares for each. In panel (c) the user has entered an increased parking charge (for a longer trip); this cell is still shared by and affects all the alternatives, producing a thumb-down even for the middle of the three air fares. Finally, in panel (d) the user has decided to see what happens if he includes in the total just half of the air fare; again, the new formula affects all cases, and now they are all under the threshold for a thumb-up.

By contrast, the second pattern suits an application where the alternatives have no persistent impact, acting akin to lenses or filters on the main application state. An example of this would be a set of alternative modules holding different versions of code for displaying the application state: after each update to the state its values would be displayed using each of the modules in turn, but this display action would generate no persistent state that the modules themselves should hold onto.

Note, however, that as shown in Figure 7 this second pattern in general requires that new leaf worlds be created each time the application's state is to be accessed after an update. This is because the evaluation of any code in a leaf world, such as to read state from the application in order to display it, will set up *reads* records for all the slots that have been accessed. Any attempt to reuse a leaf world to access future updates will fail, because the 'no surprises' protection will deliver the same values as seen before.

Figure 8 shows an example of the updates-to-leaf-worlds pattern (Figures 6), applied to a spreadsheet table in the STEPS project. The main features of its implementation are as follows:

- Each world represents a single consistent state of the sheet: cells connected by formulas, with input values (into non-dependent cells) and corresponding derived values.
- The user chooses which cell properties (values or formulas) are to be split—i.e., given distinct values in separate worlds. All other properties are shared across worlds.
- When a property that has not been split is edited, its new value is applied as an update to each of the leaf worlds, causing recalculations where necessary.
- The user interface displays the sheet as a whole as it would appear in one of the worlds, switchable by the user. The cells that have been designated as being split are shown separately, in a way that helps the user correlate the choices that have been made in each world.

Considering spreadsheets as just one instance of a more general pipeline style of processing, similar world-aware processing and interaction can be provided for applications such as search (with different search terms, result ranking criteria, etc) or data visualisation/exploration.

3.5 Cumulative histories

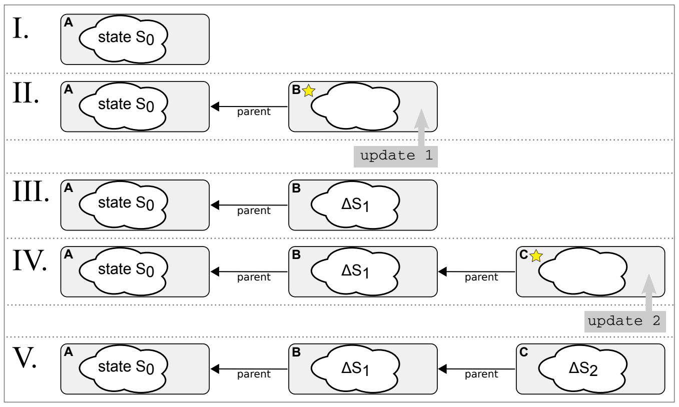
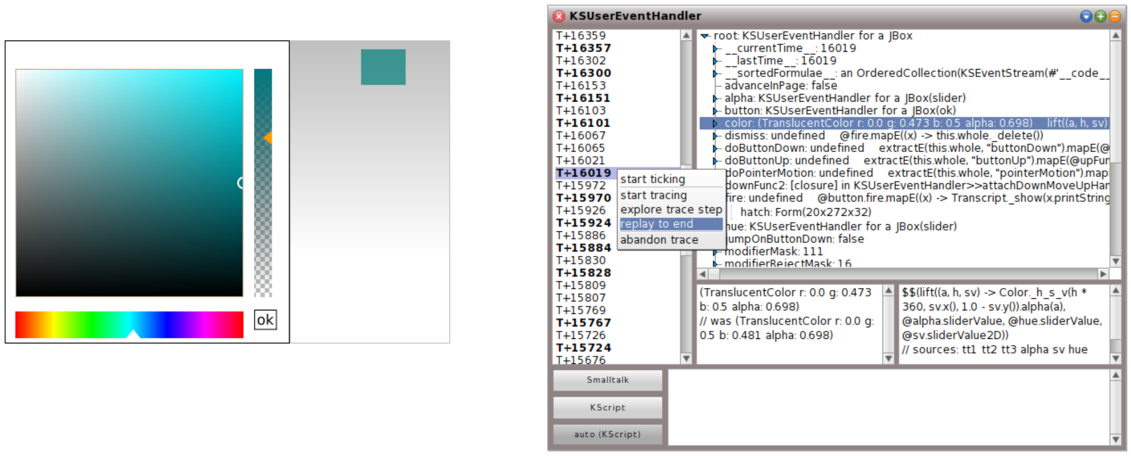


Figure 9: Recording progress increments in a cumulative history. (I) World A holds the initial system state. (II) World B is spawned prior to execution of update 1. (III) B now holds a delta resulting from update 1. (IV, V) C is similarly spawned as a child of B, and holds the delta resulting from update 2.

Figure 9 represents how worlds can be used to build up histories of application state. For each step in code execution a new world is spawned, creating for that step an isolated context that inherits, but does not affect, the entire application state up to that point. No world is told to `commit` into its parent, so each remains as a record of the code step for which it was created. The figure only shows worlds added in a linear chain, whereas the world-aware bitmap editor discussed in the ECOOP paper, which is also an example of this mode of using worlds, supports the construction of a branching tree of world (and workpiece) states. That editor has the following properties:

- The delta held by a single world corresponds to a single addition to the bitmap—namely, the pixel changes making up one stroke.
- After selecting a non-leaf world in the displayed tree of states, the user can invoke a 'branch' command to create a new child world, and hence a new branch of the tree.
- At any given point (world) in the tree, the state of the drawing as seen at that point is an aggregate of the world's delta and the state held by each of its ancestors. However, to avoid having to create bitmap-sized *reads* records this inherited state is never explicitly read from code evaluated within such a world; instead, the worlds' contributions are accumulated in a so-called flattening operation.
- To save a drawing as seen from a particular world, you select a world and flatten its state.
- Though not supported in the existing implementation, each world could be overwritten at any stage. Its contribution to the aggregate state of descendant worlds (i.e., worlds further down the branches on which it sits) would change accordingly.
- Also not supported yet, but potentially valuable, is a facility to allow the copying or moving of a branch (i.e., a sequence of strokes) to a different point in the edit tree.

Figure 10 shows a further example of worlds holding a cumulative history, in an explorer tool that we built for working with KScript-based applications in the STEPS project [4, 5]. The explorer has a tracing facility that spawns a new world to capture the changes within each evaluation time step (i.e.,



```
T+16359
T+16357
T+16302
T+16300
T+16153
T+16151
T+16103
T+16101
T+16067
T+16065
T+16021
T+16019
T+15972
T+15970
T+15926
T+15924
T+15886
T+15884
T+15830
T+15828
T+15809
T+15807
T+15769
T+15767
T+15726
T+15724
T+15676
```

```
root: KSUserEventHandler for a JBox
  __currentTime__: 16019
  __lastTime__: 16019
  __sortedFormulae__: an OrderedCollection(KSEventStream(#'__code_
  advanceInPage: false
  alpha: KSUserEventHandler for a JBox(slider)
  button: KSUserEventHandler for a JBox(ok)
  color: (TranslucentColor r: 0.0 g: 0.473 b: 0.5 alpha: 0.698)    lift((a, h, sv)
  dismiss: undefined    @fire.mapE((x) -> this.whole._delete())
  doButtonDown: undefined    extractE(this.whole, "buttonDown").mapE(@
  doButtonUp: undefined    extractE(this.whole, "buttonUp").mapE(@upFur
  doPointerMotion: undefined    extractE(this.whole, "pointerMotion").map
  downFunc2: [closure] in KSUserEventHandler>>attachDownMoveUpHar
  fire: undefined    @button.fire.mapE((x) -> Transcript._show(x.printString
     hatch: Form(20x272x32)
  hue: KSUserEventHandler for a JBox(slider)
  jumpOnButtonDown: false
  modifierMask: 111
  modifierRejectMask: 16
```

```
(TranslucentColor r: 0.0 g: 0.473
b: 0.5 alpha: 0.698)
// was (TranslucentColor r: 0.0 g:
0.5 b: 0.481 alpha: 0.698)
```

```
$$(lift((a, h, sv) -> Color._h_s_v(h *
360, sv.x(), 1.0 - sv.y()).alpha(a),
@alpha.sliderValue, @hue.sliderValue,
@sv.sliderValue2D))
// sources: tt1 tt2 tt3 alpha sv hue
```

Figure 10: Examining the trace of a sequence of user interactions with a KScript colour-picker widget. The object-structure explorer on the right offers selective tracing of the evaluations happening in the tree of objects being explored, over a time period controlled by the user. In this case a trace has been accumulated for a few seconds of the user dragging the colour picker's sliders. The list in the left-hand pane of the explorer shows time steps, labelled with milliseconds since the start of tracing. Each step corresponds to a world.

tens of times a second). After creating a trace over a time period of interest, the user can select any step in the trace list and temporarily revert all the objects in the application to the state as captured at that instant. This makes it easy to step back and forth through time by moving the selection up and down the step list, watching the changing values in the explorer's main view—an expandable tree of objects and their fields—and the reinstated visual state of the traced objects (in this case, the moving sliders and changing colours of the widget). To assist in exploring the changes to individual objects, selecting an object in the main view causes the time steps at which that object's value changed to be highlighted in bold. Our intention was to reproduce some of the facilities seen in omniscient or back-in-time debuggers [1].

Hooking the tracing mechanisms into the KSWorld environment was straightforward. All KScript objects (instances of KSObject) are essentially dictionaries, so making the KSObject class inherit from WIdentityDictionary was sufficient to allow all KSObjects to become world-aware. Causing each significant pseudo-time step (i.e., each step in which at least some object values changed) to spawn a new world was achieved using a simple check hooked into the KScript evaluation cycle.

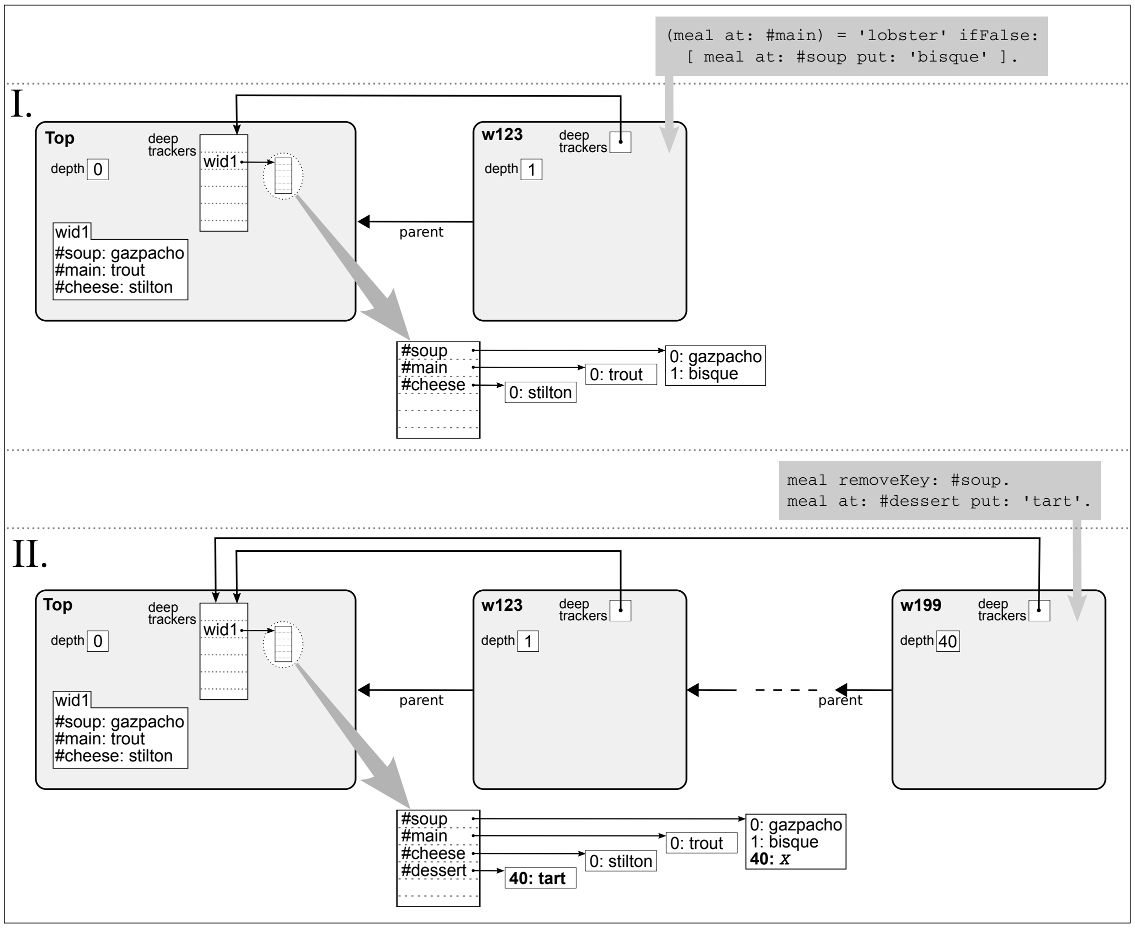


Figure 11: Specialised update trackers (for writes only) for WDeepIdentityDictionary. (I) As in Figure 1, a dictionary has been created and given three entries in the Top world. The update to key `soup` made in world w123 is recorded in a dedicated OrderedCollection for this key in this dictionary: for depth 0 the value is 'gazpacho' while for depth 1 the value is 'bisque'. (II) Thirty-nine time steps later, code evaluated in world w199 deletes the key `soup`, recorded using a `Not There` value, and adds a new `dessert` key that creates a further entry in the keys table (but the existence of this key will remain invisible to any world at a depth less than 40).

However, the high frequency of creating new worlds and evaluating code in them, and the long parent-child chains generated in recording a multi-second trace (regularly reaching hundreds or even thousands of worlds), led to a severe performance bottleneck. Part of the burden was the creation of *reads* records for every accessed object in every world, especially since a fundamental part of each KScript evaluation cycle is to examine all streams in all KScript objects to decide which ones need to be updated. Another part of the burden arose in accessing fields within newly created KScript objects: the first access to each key in such an object would result in a lookup through every world in the ancestor chain.

To work around these issues we created WDeepIdentityDictionary, a new form of WIdentityDictionary specialised for working in deeply nested child worlds where updates to any single slot tend to be distributed sparsely through the world chain. WDeepIdentityDictionary also saves time by not creating *reads* records, which is reasonable given that the history support for which it is designed does not require the normal child/parent world protections. As shown in Figure 11, the fundamental difference from WIdentityDictionary is that instead of each object (dictionary) having update trackers in every world in which it has been accessed, it uses just a single set of update trackers that is shared between all the worlds in a chain. The trackers include lists of updates for each key in the dictionary, each update annotated with the depth of world at which it took place. Accessing a key from a world involves going straight to the trackers for that key, and finding the closest depth of parent world in which the key was updated. To further reduce the lookup time, the annotated updates are sorted by depth and can thus be accessed using a binary search.

4 Reflections and possible future directions

The experiments reported here were not part of any grand plan; we were just tentatively pushing the boundaries of Worlds support and usage. In this section we list some observations from this work, and some further aspects of Worlds that we feel it would be interesting to explore; some of the aspects are addressed in more detail in dedicated subsections. The overall goal is to look at where the experiments have taken us, and to sketch out a possible future in which the various threads of these experiments are drawn together in a new, unified architecture and implementation for Worlds.

The first aspect that our experiments helped to clarify is that there is a spectrum of levels at which the use of worlds can be reflected in an application's code. On the one hand there are cases where the use of worlds is integral to the computations, such as the progressive branching mode demonstrated on text layout in Section 3.3. We call these cases 'world dependent'. At the other extreme is the use of worlds for debugger-like tracing, as demonstrated in Section 3.5, in which the application code being executed has no knowledge that worlds are being used to track it (and, indeed, if the application made any explicit use of worlds this would interfere with the tracing). These cases can be called 'world blind'. These extremes correspond to whether there is a clear separation between the application code and a kind of supervisory layer in which worlds are created and manipulated. Between the two ends are 'world adapted' cases, such as the use of worlds to handle transactions, as suggested in Section 3.2, in which the world facilities are not part of the computation itself but are used to provide concurrency protection; and the spreadsheet example shown in Section 3.4, where there is an underlying application that usually works without worlds, but into which worlds can be woven as a way to provide extra features, such as for running experiments or exploring alternatives.

Code that is world blind, but can be tracked using worlds, requires simply that all the long-lived objects making up the application state (i.e., everything other than ephemeral values such as method temporaries) be world aware. The most straightforward way to arrange this is for all the application objects to be instances of a single world-aware class (or of a subclass), as is the case for KSObject, the JavaScript-like dictionary class that acts as the root for all KScript objects. Our experiences with the earlier STEPS spreadsheet and document editor, written in Smalltalk, was almost as easy because all objects in the application framework and user interface were descended from a single LObject class. In that case, however, we had to take the further step of ensuring that the code also only used world-aware collection classes (WArray, WIdentityDictionary, and WCollection).

A second result from the experiments is that our experiences of using worlds led us to a better understanding of the basic facilities of worlds themselves and the objects they refer to, including how non-world-aware objects can usefully be thrown into the mix, for example as a way to communicate between worlds. In Section 4.1 we introduce some new 'convenience methods' that we feel worlds should support, and in Section 4.2 discuss how non-world-aware and world-aware objects could one day be cut from the same cloth.

A third result is our experience in implementing and using specialised features such as the snapshot facility built into WCollection. We would like to turn this feature into a mechanism that is more general—applying to most or even all types of world-aware object—and is also optional, so that the programmer of a new application can use it as appropriate to provide the desired balance between inter-world protection and sharing. One direction for this generalisation is discussed in Section 4.3.

Along similar lines to the above, we gained experience in designing the policies for enforcing the *no surprises* and *consistency* safety properties for new object classes. We came to the conclusion that there is no single way to 'do the right thing' on inter-world safety properties for all applications that might ever be built with worlds, suggesting that a better approach is to give application programmers more control over these properties. In Section 4.4 we look into making the serialisability checks tailorable, so that applications can select how much checking the worlds mechanisms will do and when exceptions will be thrown, then in Section 4.5 we sketch out how application code can respond appropriately to exceptions when they are thrown.

Standing over all the proposals for future work is a desire to arrive at simple, unified designs for all aspects of Worlds support. This means not only generalisations such as making our experimental snapshot mechanism available for all object types rather than just WCollection, but also finding an elegant way to unify the snapshots with the update trackers, and similarly a way to unify the update trackers for growable collections with those for fixed-size objects. It may require several more rounds of isolated experiments to find the right designs, but a clean final architecture to offer to programmers is a worthy end goal.

4.1 Convenience methods on world instances

The original Worlds/Squeak had a minimal set of operations for creating and manipulating instances of WWorld, the class for world objects themselves. A world could only be created by sending `sprout` to an existing WWorld instance (starting with the Top world, which is always accessible). The receiver is registered as the new world's parent, and the child starts with an empty dictionary of update trackers. Once some changes have accumulated in a world, the `commit` method is used to commit them into the parent—provided the serialisability check goes through, of course. And that's about it.

Here are some additional methods that we have experimented with:

`copy` Being able to create a copy of some child world seems a generally useful operation to have. The `copy` message sent to a world w produces a world that is a sibling of w, and whose state is the same as if the new sibling had been created at the same time as w and had replicated every subsequent code evaluation.

Implementing `copy` requires ensuring the new world has update trackers for each object that has been accessed through w. The trackers are created as a copy of those in w, copied deeply enough that any additional *reads* or *writes* records created either in w or in its copy will not be visible to the other.

`reparent` This operation is trivial to implement—it just replaces the world's `parent` instance variable—but of course has profound implications, and can cause chaos if used incautiously.

One use is in a situation such as the tracing explorer described in Section 3.5. If a user wants to examine the trace in, say, the last few seconds leading up to an event whose occurrence cannot easily be predicted, one approach is to start the trace ahead of time and just keep it running, stopping it (manually or otherwise) once the event happens. But if the system being traced is complex, the update frequency is high, and the delay until the event long, this can lead to prohibitively large trace records. What would be useful is a way to discard trace steps that are older than a certain threshold, but without losing all their state information (which subsequent steps may yet need to access). So suppose the tracer has ascertained that the second world in its list is old enough to be discarded. It can do so with the following code:

```
discardedWorld := traceWorlds removeAt: 2.
discardedWorld commit.
(traceWorlds at: 2) reparent: (traceWorlds at: 1).
```

This commits all the changes in `discardedWorld` into the origin world (the first one in the trace), then makes the former child of `discardedWorld` be a direct child of the origin instead.

Another use for `reparent` is the complementary action of inserting a world between a world and its existing parent. For example:

```
frozenWorld := w copy.    "starts as a sibling"
w reparent: frozenWorld.  "make the frozen world w's parent"
w initializeTrackers.     "and wipe clean w's recorded updates"
```

The inserted world is referred to here as frozen because it captures everything about `w` at the time of the copy, and stays that way. The application state at `w` can now be forked by sending `w`'s new, frozen parent a `sprout` message for each new desired fork. All the resulting sibling worlds start with exactly the same overall state of the application. Of course, this would also be true if they had been created by repeatedly sending `sprout` to `w`, or by repeatedly sending `copy`. The differences are that (a) `w` can continue to record updates without affecting the new worlds (which would not be true if they were its children), (b) the `sprout` can be sent even after `w` has recorded further updates, but the new worlds will still faithfully represent `w`'s state at the time of the freeze, (c) whereas forks created using `copy` will have duplicates of all their *reads* records, which makes it likely that attempting to commit later changes from multiple forks will run into serialisability problems, here the likelihood is much lower because the forks only capture reads that happened after the freeze.

As a final example of the use of `reparent`, we propose a generalisation of the standard `commit` operation, which normally assumes it is committing into the world from which the receiver was originally spawned. With `reparent` we can offer a 'commit into' operation that will commit (or at least attempt to commit) the receiver's changes into any destination world of the programmer's choice. Whether the relationship of the world to its new parent is one for which the serialisability check and the merging of updates are meaningful is at the programmer's discretion, and risk.

`eval:onError:` This truly is just a convenience method, but acknowledges an important gap in the existing protections for a parent world: even though the parent is entirely protected from the value-setting effects of code evaluated in a child, the parent world's execution can still be brought to a halt by an error or uncaught exception in child code. As a matter of simple precaution, it would be a good habit to back up all such execution by specifying how to proceed should an error occur.

4.2 Optional world awareness

While most of the interesting things a Worlds implementation makes possible are based on having objects that are world aware, there can also be times when a non-world-aware object is useful. The primary reason for wanting a non-world-aware object is when some sequence of updates are to be shared between worlds, given that inter-world sharing (between siblings, from children to parents, from parents to children after the first access) is precisely what the standard Worlds mechanisms are there to suppress.

In Worlds/Squeak there is already an easy way to create a non-world-aware object: use any object whose class does not inherit from WObject. And the fact that the main Squeak environment—with all its kernel objects, global variables, and development tools—is built from non-world-aware classes is, for the most part, just what one needs.

However, consider the transition into maturity of a Worlds-supportive environment such as KSWorld, the personal computing platform we built using KScript. This maturation includes two strong trends: first, there is a desire to build more and more of the environment's tools out of its own classes, rather than those of the hosting platform; second, the application-level code already matches well the interface to the environment's core classes (such as collections), so it would be inconvenient to have to create additional classes just so they could be used to communicate between worlds. Both trends suggest that it would be useful to be able to create non-world-aware versions of the objects that are normally world-aware. The question is how to support this.

One possibility would be to equip all world-aware objects with a setting that subverts how their methods access `thisWorld`, the world in which the method is being run. If `thisWorld` always appears to be the top world, all accesses and updates will take place there. Any objects instantiated and installed to act as components for such a non-world-aware object would need to be infected with the same subversive behaviour. Note that we believe it only makes sense for this setting to be decided at the time of instantiation; it would not be useful or manageable for an object to be allowed to flip from one state to another on the fly.

A second possibility would be to create a dual hierarchy of classes, maintained largely automatically, such that developing a world-aware class would simultaneously build a non-world-aware dual. When code accesses a class to instantiate an object, it can either explicitly state which of the hierarchies should be used, or by default access the hierarchy in which its own class appears (thus perpetuating its aware/non-aware nature).

That's about as far as we've considered on this issue. Building an elegant solution could be an interesting little challenge.

4.3 Controlling visibility of parent-world updates

In Section 2.2 we introduced the practice of having a child world take a snapshot of a WCollection at the time of first access to it, thus guaranteeing that any collection elements subsequently read within that world will be internally consistent, regardless of interleaved evaluations and updates happening in ancestor worlds. This is a specialised form of 'no surprises' protection for the child world.

While such protection can be useful, we do not regard the current status of snapshotting—as a silently applied feature of the WCollection class alone—as a viable long-term situation. The protection is obtained at the cost of opennness to change: even in code written in a context where the programmer can be sure that interleaved updates in the parent world will never lead to a problem of the child world seeing inconsistent values, the use of a WCollection means that the child world will be blocked from seeing any updates that happen after the world's first access to the collection. The fact that this behaviour is unique to WCollection (in particular, is different from the closely related WArray) is also an inconvenient idiosyncracy for the programmer to have to remember.

Our proposal, therefore, is that snapshot be made available as a facility that can be applied to any world-aware object, under programmer control. Its purpose remains as protecting a child world from interleaved updates in ancestor worlds, in cases where this could lead the child world to read mutually inconsistent values. But its selective, program-controlled use would allow the protection to be applied at precisely the point in time where it is needed, and to precisely the set of objects whose consistency needs to be guaranteed. For example, in a child world that is being used to perform an experimental adjustment to the values in a spreadsheet, snapshot could be applied to all cells in the sheet before embarking on the experiment. Whichever cells turn out to be implicated in the experimental calculation, the snapshot will ensure that they all deliver consistent values, regardless of any changes happening in parallel in the parent world. Another example is calculating the combined total of all accounts within a bank: taking a snapshot of all the account balances will allow a child world to work its way through the accounts at leisure, without concern for the possible shifting of money between them. Of course, both this case and the last depend on the snapshot itself being atomic with respect to updates.

In addition to the use of snapshots at the start of code sections for particular calculations, applications of the 'world dependent' type (as defined at the start of this section) could be written to apply a snapshot to the critical application objects in each new world at the time when the world is spawned.

It is important to bear in mind the effect that taking a snapshot has on a subsequent `commit` of child-world updates into the parent. In general the use of snapshot appears benign, in that having a snapshot of an object that the child world turns out never to access causes no harm to the child's ability to commit. But it is still the case that for any object that the child world does access, if that object has different slot values at the time of the `commit` then the serialisability check will flag the clash. So if the child world applies a snapshot, early in its existence, to objects that are not critical to its calculations but that it does access at some point, the risk of an obstructive (and needless) clash with the parent at the time of `commit` is increased.

An extreme case of commit clash would happen with the bank-account example described above. Once the child world has worked its way through all the accounts, it has a *reads* record for every account balance. Unless the bank had been completely dormant throughout this period, there is no chance that every balance is still the same at the end of the calculation, and therefore no chance for the child to commit any derived values into the parent world.

However, this is a case where the application programmer could assert that, despite any changes in slot values that have been read, the slot values that have been written (perhaps a set of records constituting a summary report on the state of the accounts) still deserve to be committed.

One way to allow this is if we provide a further operation that is in some ways a complement to the snapshot: a `noCache` setting that tells the objects to which it is applied not to create *reads* records for any slots that are accessed. In the bank-account case, combining a snapshot of all bank accounts with a `noCache` setting for the same objects will ensure that the values seen are consistent, but the values written for the report can still be committed.

Thinking further about `noCache`: whereas using snapshot on a set of objects makes it impossible for a child world to see changes to those objects happening in the parent, `noCache` makes it impossible for the child to *ignore* changes. Every access to a slot will be forwarded to the parent, and will obtain its latest value. Therefore, applying `noCache` at the level of an entire child world would be a way to avoid the contortions illustrated in Figure 7, where it is the *reads* caching that forces the application to spawn new worlds just to read the system state, only to discard them after a single use.

The snapshot and `noCache` operations can be used separately or in combination, depending on an application's needs. Clearly they both have a dramatic impact on the semantics of world usage and, in particular, on the default protections offered by worlds. But if used responsibly, by code that has the oversight necessary to assure that the outcomes are predictable and valid for the application in question, we believe they are powerful tools to have in the drawer.

4.4 Customised consistency checks

While the previous section addresses how to give programmers control over the 'no surprises' safety feature that protects a child world, here we consider similar powers for controlling the consistency checks that protect a parent.

In the current Worlds implementations, the consistency checks are based around the maintenance and use of *reads* records. The features discussed in this section therefore depend on the programmer not having disabled such recording for an object by use of the `noCache` setting introduced above.

As explained in the Introduction, the standard consistency check for fixed-slot objects involves confirming that, for every slot that was read at some point from the parent, a read carried out now would give the same value as it did at the time. In Section 2 we showed how we went about mapping this form of check onto objects that can change the number and names of their slots, and how we came to the conclusion that different applications may call for different policies—in particular, for handling the outcome of searches and tests carried out on the object.

This led to the thought that we could offer programmers a way to specify, perhaps for each growable collection created in their code, how consistency protection should be enforced for that collection. As an illustration of the possible variation, imagine three policies for dealing with failed key lookups in a WIdentityDictionary: under the *Relaxed* policy, failed lookups impose no commit-time constraints at all; *Specific* means that all failed lookups are explicitly recorded, to check whether values for those keys in particular have appeared later (this is the behaviour of our initial implementation of WIdentityDictionary); and under *Strict* any failed lookup constrains `commit` to fail if any new keys have appeared.

The *Strict* level may sound too draconian, but the principle is straightforward: if code in a child world tests for existence of a key, and the test fails, we can consider that the code has examined every key in the dictionary (which it would indeed do, if search were implemented simplistically as a linear iteration). Therefore it would be a violation of serialisability if any new key were later to have appeared in the parent world. This level also provides the strength of protection needed to assure that any value derived in the child world based on all the dictionary's keys would be the same in the parent. For example, 'the number of keys that start with a vowel', or 'the first and last keys in alphabetical order'. Even so, there are still limits to what can be asserted: in a Squeak dictionary, for example, 'first key in an enumeration' could not be assured, because the enumeration sequence depends unpredictably on the hashing used to store keys.

Implementing a set of policies like these based on the existing way in which *reads* and *writes* records are used would involve tailoring the dictionary's behaviour in each of the methods where the policy has an impact. For example, the choice of policy may affect how the dictionary caches the results from key lookups, how it gathers keys from child and parent worlds during enumeration, and how it compares child and parent states in its serialisability check. The most flexible approach, from the point of view of an application programmer hoping to implement some particular policy, would therefore be if the dictionary code provided hooks into all these methods. But our initial investigations suggested that the complexity of this approach would be prohibitive: coordinating the behaviour across the hooked methods would require deep understanding of the update-tracing mechanisms, and would be error-prone.

One way to reduce the complexity for the application programmer would be for the dictionary implementation to incorporate a pre-designed range of policies, and let the programmer choose (by name) which policy each instance should enforce—with some default setting if no choice is made. This simplicity would come at the cost of no longer being able to implement highly customised policies.

A possible alternative way of reducing complexity would involve a re-design of the *reads* data structure itself, making it capable of holding extra details of the operations that the collection has carried out. With sufficient extra details, it may be possible to support customised policies purely by hooking the serialisability check itself. For example, the *reads* record for a single dictionary key could show whether the key has been explicitly requested, or examined in a search but rejected, or examined in an enumeration. A *Strict* check would then consider all keys that have been examined to any degree, whereas a *Specific* check would care only about those that had requested explicitly. We have not yet tried working through this example to a sufficient level to crystallise what *reads* annotations would be needed to support what variations in policy.

Having specified how the consistency check should be performed, the next step for a programmer is to specify what can be done if the check happens to fail. Without further ado. . .

4.5 Customised handling of consistency failures

As was mentioned in Section 2, the default implementation of the `commit` of a child world into its parent consists of two parts:

1. Every object that has *reads* records in the child world carries out a consistency (serialisability) check against the current state in the parent. If this fails, the object throws an exception.
2. If no exception was thrown, every object that has *writes* records in the child world is told to merge its changes into the parent, and afterwards the child world resets its update trackers. Otherwise nothing is changed.

This is the simplest—and harshest—implementation of `commit`: if even a single object in the child world finds that it fails the consistency check, the whole `commit` is immediately abandoned.

As was pointed out in Section 3.2, even this simple pass/fail model can be put to good use. There we described how a simple transaction-handling system could be implemented by assigning each transaction's computations to a specially spawned world, then if the `commit` of a given transaction world turns out to fail, the system just discards that world and tries the transaction again later with a new one.

Seeking to enrich the model, here we consider how one could introduce opportunities for world-dependent applications to customise the handling of `commit`, so that a single consistency failure is no longer taken as reason for abandoning the whole attempt. One desirable outcome would be if, through such customisations, an application suffering failures in the consistency checks of a subset of objects used in the child world could still salvage and transfer to the parent some of the other changes.

The key to this gentler form of handling is to allow for some form of fix-up when objects fail their checks. For the Worlds/Squeak implementation we can make use of Squeak's powerful exception handling mechanism. When activated, an exception handler has access to the chain of context objects (stack frames) that led to its activation, and has response options that include resuming the computation from where the exception was thrown, even supplying a new value for the exception-throwing expression. For example:

```
| div |
div := 0.
[(3/div) * 30] on: ZeroDivide do: [:ex | div := 1. ex resume: 3]
"=> 90"
```

The failed division is caught, some arbitrary fix-up work is done (setting `div` to a non-zero value), then the code is resumed with a new value for the fixed expression.

To use this facility to fix up failed `commit` cases, we imagine the use of exception objects with instance variables that let the handler discover the world being committed, the object and slot being checked, and the mismatching values that triggered the exception. The two-part `commit` shown above would be expanded to three parts:

1. (as before) Every object that has *reads* records in the child world carries out a consistency check against the current state in the parent. When an object fails this test, an exception is thrown.
2. For each exception thrown, an attempt is made to fix up the child world so that usable updates can still be committed. If some exception is not amenable to fix-up, the `commit` will be abandoned after all.
3. If the `commit` has not been abandoned, every object that (still) has *writes* records in the child world is told to merge its changes into the parent, and afterwards the child world resets its update trackers.

The inserted part 2 is where a programmer can specify a commit-exception handler suited to the application. The simplest form of handler specification would be as follows:

```
[w commit] on: SerializabilityCheckFailed do: [:ex | ex resume: true].
```

The handler here will be activated once for each object that throws an exception. It ignores the exception, passing the value `true` back to the consistency-check loop to make it move on to the next object in its list (note that the writer of an exception handler needs to have at least basic knowledge of how the serialisability check works). The effect is a complete disabling of the serialisability check for this particular `commit`.

A more sophisticated exception handler could use information included in the exception to fix up the child world in a way that will ensure that only valid updates are eventually written to the parent in part 3 of the `commit`. Here is a case that borrows from the example shown in Figure 1:

```
w := thisWorld sprout.
meal := WIdentityDictionary new.
meal at: #soup put: 'gazpacho'.
meal at: #main put: 'trout'.

w eval: [
  (meal at: #main) = 'lobster' ifFalse: [ meal at: #soup put: 'bisque' ].
  self calcCaloriesIn: meal.
  ].

(*** some other code that may change the items within meal ***)

[ w commit ] on: SerializabilityCheckFailed do: [ :ex |
  ex targetObject == meal
    ifTrue: [
      w forgetWriteOfSlot: #soup in: meal.
      w forgetReadsFrom: meal.
      w eval: [ self calcCaloriesIn: meal ].
      ex resume: true.
      ]
    ifFalse: [ ex pass ].
  ].
```

In this example, the child-world evaluation reads some information from the parent and perhaps makes a change, then invokes the method `calcCaloriesIn` that we can assume stores a calorie total somewhere in the application's data (for example, as a further field in `meal`). The exception handler supplied for `commit` checks whether the source of the exception is the `meal` object: if so, the response is to abandon any update that might have been made to the #soup item, to clear any *reads* records for the meal as a whole, and to re-calculate the calories. The serialisability check is then resumed. Note that this handler does not attempt to fix exceptions arising in relation to any object other than `meal`; instead it re-throws them (using the `pass` method), which will cause the `commit` to be abandoned if there is no other handler to catch them.

As suggested by this example, we envisage the existence of fix-up methods that adjust the child world by (a) altering or removing *reads* records, or (b) altering or removing *writes* records. We feel it would not make sense to let the fix-up have a direct effect on the parent world, given the rule that an abandoned `commit` must leave the parent unchanged. Indeed, for this same reason it would also be preferable for individual fix-ups to have no direct effect on the child world either until the `commit`'s go-ahead is confirmed. One way to achieve this would be if all proposed fix-ups are performed in an independent copy of the child world, created for this purpose. Only if all exceptions have been dealt with satisfactorily will that world be told to merge its changes into the parent, and the original child then told to clear its update trackers.

Another possible use for the serialisability check and its potential fix-up is in preventing the accidental creation of zombie worlds, as discussed in Section 2. One change proposed by Alex Warth and others to reduce the risk of creating zombies is to precede every child-world `eval` with an automatic serialisability check. This would catch situations in which the child's execution is about to resume even though values read previously by the child have now changed in the parent. Supplying an application-specific handler for exceptions thrown during this check would allow the programmer to fix up the child world first, perhaps by telling the world to forget some entries in its *reads* records. Alternatively the handler could ignore the exception, on the grounds that it does not matter in this application (for example, because the child is never going to be asked to commit).

Of course, any customised exception handling—especially if accompanied by fix-up—is overriding the system's standard safety protections. Responsibility for ensuring the safety of the affected application falls to the writer of the exception handler, who must therefore have detailed knowledge of how that application works; there is no one-size-fits-all way of writing such a handler. How to provide good support from the Worlds infrastructure for writing powerful yet safe exception handlers is another tricky but rewarding direction for future study.

Onwards and upwards!

5 Acknowledgements

We give deep thanks to all at VPRI for the discussion and debate, past and ongoing—especially Alex Warth, Takashi Yamamiya, Ted Kaehler, and Alan Kay.

References

[1] A. Lienhard, J. Fierz and O. Nierstrasz. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns: Proceedings of TOOLS EUROPE 2009*, Zurich, Switzerland. Springer, 272–288. 2009.

[2] A. Lunzer and K. Hornbæk. Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios. *ACM Trans. Comput.-Human Interact.* 14, 4, Article 17 (January 2008), 1–44.

[3] A. Lunzer and K. Hornbæk. Subjunctive interfaces for the Web. Chapter in Cypher, A., Dontcheva, M., Lau, T. and Nichols, J.W. (eds.) *No Code Required: Giving Users Tools to Transform the Web*, Morgan Kaufman, 267–285. 2010.

[4] Y. Ohshima, B. Freudenberg, A. Lunzer and T. Kaehler. A Report on KScript and KSWorld. VPRI Research Note-2012-008, 2012.

[5] Y. Ohshima, A. Lunzer, B. Freudenberg and T. Kaehler. Making Applications in KSWorld. VPRI Research Memo M2013-003, 2013.

[6] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: Controlling the Scope of Side Effects. In *Proc. 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, Lancaster, UK. LNCS 6813, 179–203. 2011.