Programming with Python PastSet

PastSet was first introduced as an interprocess communication paradigm in Anshus in 1992. The paradigm resembles that of Linda, but with some significant differences. Tuples are generated dynamically based on tuple templates that may also be generated dynamically. A tuple template is an ordered set of types. Tuples based on a particular template has an ordered set of variables matching the types in the template. Each type in a tuple template has an associated value-space, or dimension, describing the set of all possible values for that type in this template. Taken together, the types in a template spawns a space encompassing all conceivable type-value combinations for tuples based on that template. A tuple with all singular values represents a singular point in this space.

As with Linda, PastSet supports writing (called move) tuples into tuple-space and reading (called observe) tuples that reside in tuple space. Contrary to Linda's in operation, PastSet observe does not remove tuples from tuple space, the tuple that is observed is marked as observed but remains in the PastSet so that it can be read again if specified so directly. No mechanism is provided to remove individual tuples from PastSet. All PastSet operations return only when the operation has completed, or an error has been detected, no asynchronous calls exist.

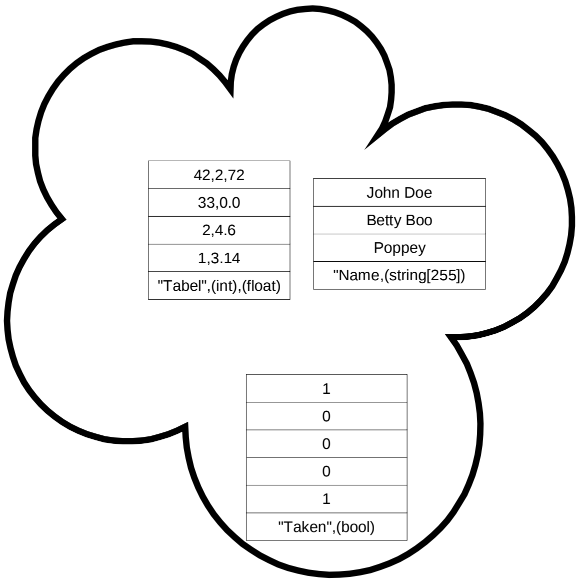
In PastSet, each set of tuples based on identical templates is denoted an element of PastSet. An element may be seen as representing a trace of interprocess communications in the multidimensional space spawned by the tuple template. PastSet preserves the causal order among all operations on tuples based on the same or identical templates. There is no ordering between tuples of different elements. Tuples that match the same type, but which the programmer does not wish to place in the same element can be differentiated by an initial flag.

In effect, PastSet keeps a sequentially ordered log of all tuples of the same or identical templates that have existed in the system. This also allows the processes to re-read previously read tuples.

It is the intention that the added semantics of PastSet will allow programmers to more easily create parallel programs that are not limited to the traditional 'bag of tasks' type.

Two pointers First and Last are associated with each element in PastSet. First refers to the elements oldest unobserved tuple. Last refers to the tuple most recently moved into the element. A parameter, DeltaValue, associated with each element in PastSet defines the maximum number of tuples allowed between First and Last for that element. A process may change DeltaValue at any time. The move and observe operators update First and Last, and obey the restrictions imposed by DeltaValue for each element in PastSet.

Functionality is provided to truncate PastSet on a per element basis, permanently removing all tuples that are older than a given tuple



pyPastSet

pyPastSet is a reimplementation of PastSet in Python, it is not written with optimal speed in mind but rather ease-of-use and true portability. pyPastSet maintains a mostly PastSet-like notation but some changes have been made to make pyPastSet more pythonesque, i.e. elements are objects in pyPastSet and operations on elements are methods to that object.

1.1 PastSet

Before using PastSet the application needs to connect to PastSet.

```
import pastset
pset = pastset.PastSet()
```

Example 1: Getting access to PastSet.

1.2 The Enter-operator

Before accessing PastSet, processes must issue at least one *enter*-operation specifying a tuple template that will be used. In addition to specifying a tuple template, the *enter*-operation also establishes a binding between the template and a unique element in PastSet. If no identical template has been specified for any previously executed *enter*-operation, the binding is established with a new element created in PastSet. If an identical template has been specified for a previously executed *enter*-operation, the binding is established to the previously created element.

```
(“Test”, (int), (float))
(“Test”, (int), (float))
```

Example 2: Two matching tuple templates

```
(“Test”, (int), (float))
(“Test”, (int), (complex))
```

Example 3: Two tuple templates that do not match

The tuple-template may contain a tag that is used to differentiate between otherwise similar templates. An appropriately complex tag may be used as a security mechanism to prevent unauthorized access to the element, much like using a large, sparse name-space. In addition to the tag, the parameters for the *enter*-operator are the tuple including the tag, and an identifier for a set of filter functions that will be described in a later section. The *enter*-operator returns a element object that identifies the element and is used when accessing the element from that point on. If no existing element fits the tuple-template, a new element of PastSet is created.

```
my_element = pset.enter((“Test”, int, float))
my_element = pset.enter((“Test”, int, float), my_filter)
```

Example 4: The PastSet API entry for EnterDimension

1.3 The Move operator

The *move*-operator is used to add tuples to PastSet. It is semantically similar to the ‘out’ operator in Linda, but has a different syntax identifying the operation’s target element. The *move*-operator takes an element object and a tuple as parameters.

The *move*-operator returns after the operation has completed. The *move* operation blocks unless it can complete without violating the limit defined by *DeltaValue*. A process blocked on *move*, remains blocked until one of two things happens. Either another process executes an *observe* operation increasing the *First* pointer by one, allowing one blocked *move* operation to complete. Alternatively another process may increase *DeltaValue* allowing one or more blocked *move*s to complete. A blocked *move* operation is terminated if the element is deleted from Past-Set; see the *axe*-operator below.

```
my_element.move((42, 3.14))
```

Example 5: The PastSet API entry for Move - note that the flag need not be passed for the Move operator

1.4 The Observe operator

The *observe*-operator reads a tuple value from an element, and makes it available to the process that issued the *observe*. *Observe* may be called with or without an index. The optional parameter is an absolute number that specifies which tuple to read.

If no tuple number is specified, the *observe* operator reads the value of the tuple pointed to by the *First* pointer, and atomically increases the pointer by one. If *First* > *Last*, the process blocks until another process issues a *move* onto the element. This situation occurs when an *observe*-operation is issued and there are no unobserved tuples in the element.

If a tuple number is specified, that specific tuple value is read. If the tuple does not exist, the process will block until it eventually does, if the tuple has been truncated an error will be returned. Reading specific tuples does not alter the *First* pointer, even if the specified tuple is the one pointed to by the *First* pointer. If the specified tuple no longer exists, see the *axe* operator below, an Expired exception is raised.

The *observe*-operator does not directly compare to either of Linda’s operators; in and read. *Observe* differs from in, since the *observe*d tuple remains in PastSet available for later *observe* operations. It differs from read, by not always being completely transparent.

```
new_tuple = my_element.observe() # returns the oldest unobserved tuple
new_tuple = my_element.observe(42) # returns tuple number 42
```

Example 6: The PastSet API entries for Observe

1.5 The X-function

Any element may be associated with one X-function. X-functions are function pairs, X-in and X-out, which are applied as tuples are moved or observed. When a *move* operation is executed tuple data is filtered through the X-in function and then stored according to the X-in result. When an *observe*-operation is issued, the X-out function is called with the input tuple as parameter, X-out returns one specific tuple.

X-functions may potentially perform any semantic operation on the tuple, but are primarily intended to supply the following features: alternatives to sequential ordering, pattern matching, and tuple manipulation. Alternative ordering may be used for achieving any number of orderings of tuples, i.e. by a real-time time-stamp in the tuple, by sorting, or for supporting priorities in 'bag of job' type applications.

Pattern matching as supplied by Linda can also be supplied via an X-function, as the X-in function can build a lookup table, i.e., a hash table, and the X-out function can then perform pattern-matching using regular database algorithms.

General tuple manipulation may be used for a number of purposes; in the extreme most of the applications functionality may be implemented using X-functions. An example of such tuple manipulation could be a graphical application where *move*s and *observe*s to an element work on bitmapped frames for a video stream. An X-in function could mpeg compress frames as they are *move*d to PastSet, and an X-out function could decompress frames as they are *observe*d.

1.6 The Axe and DelElement operators

The *axe*-operator provides a coarse mechanism for truncating elements by marking for deletion "old" tuples in any specified element. *Axe* called on an element takes as argument the absolute number of the newest tuple to mark for deletion. At the completion of *axe*, this tuple and all older tuples in this element are marked for deletion. The actual deletion of tuples and freeing of storage space may be done asynchronously to the completion of *axe* and may chose a lower index than the specified as the first one to be eliminated. Thus *axe* cannot be used as a way of removing tuples that may not be seen any more, it is purely a measure to preserve memory.

DelElement is used to remove any single element in PastSet, freeing its storage for other uses. *DelElement* removes the element at the time it is called, so that after a *DelElement* operation returns, the element identifier is no longer valid, and en *EnterDimension* on an identical tuple template will return a new, empty, element.

```
my_element.axe(index)

pset.delelement(my_element)
```

Example 7: The PastSet API entries for the Axe and DelElement operators

1.7 The First, Last and Delta operators

PastSet allows the programmer to read the values of the *First* and *Last* pointers of any element, and to read or modify the *DeltaValue*. *First* and *Last* called on an element returns the respective index. Delta is also bound to elements, and may take as an optional parameter the new *DeltaValue*. A decrease of *DeltaValue* below the current *Last-First* is readily accepted. Future *move*-operations are tested for completion against the new *DeltaValue*, and are possibly blocked accordingly. Increases to *DeltaValue* allow a corresponding number of blocked *move* operations to complete. Blocked *move* operations are unblocked in the same order that they were blocked.

```
index = my_element.first()

index = my_element.last()

size = my_element.delta()

new_size = my_element.delta(42)
```

Example 8: The PastSet API entries for the First, Last and DeltaValue operators

1.8 Spawn

Spawn is used to start new processes on the compute nodes participating. They are launched from the available PastSet clients running on the nodes. Spawn takes two arguments, one is the name of the application (filename) that the new process should start executing, the other parameter is a list of arguments to that function.

```
pset.spawn(application, arguments)
```

Example 9: The PastSet API for Spawn