Arrays in SHIM: A Proposal

Smridh Thapar, Olivier Tardieu, and Stephen A. Edwards
Columbia University
st2385@columbia.edu, {tardieu,sedwards}@cs.columbia.edu

Abstract

The use of multiprocessor configurations over uniprocessor is rapidly increasing to exploit parallelism instead of frequency scaling for better compute capacity. The multiprocessor architectures being developed will have a major impact on existing software. Current languages provide facilities for concurrent and distributed programming, but are prone to races and non-determinism. SHIM, a deterministic concurrent language, guarantees the behavior of its programs are independent of the scheduling of concurrent operations. The language currently supports atomic arrays only, i.e., parts of arrays cannot be sent to concurrent processes for evaluation (and edition). In this report, we propose a way to add non-atomic arrays to SHIM and describe the semantics that should be considered while allowing concurrent processes to edit parts of the same array.

1 Introduction

In a concurrent framework, it is very important to define the rules and restrictions that concurrent processes must follow to perform a task deterministically. In this report, we discuss these rules for arrays in a concurrent setting along with an array extension to the concurrent SHIM framework.

Arrays in SHIM might be spread over a distributed environment with their parts being handled by different processes. Concurrent updates to the same array might lead to inconsistency and thereby make the language non-deterministic. To avoid this, rules need to be defined for parallel processing with arrays without compromising the power or flexibility of the language.

The SHIM language currently supports atomic arrays only, that is, arrays in SHIM cannot be spread over different processes. This requires sending the entire array across processes even if only a single element of the array needs to be sent. Also, atomic arrays prohibit concurrent updates to different parts of the same array. We carefully examine these situations and propose a design construct for adding non-atomic arrays to the language.

2 Related Work

SHIM already defines rules for sharing and synchronizing base type variables between parallel processes. SHIM systems consist of sequential processes that communicate using rendezvous through point-to-point communication channels. SHIM systems are therefore delay-insensitive and deterministic for the same reasons as Kahn's networks, which they resemble by design, but are simpler to schedule and require only bounded resources by adopting rendezvous-style communication inspired by Hoare's CSP. We propose our extension to SHIM with these key ideas in mind.

The enhanced compute capacity achieved with multiprocessor systems has lead to the development of various languages. X10 is an object-oriented programming language designed for programming of Non-Uniform Cluster Computing Systems. This report is inspired by the developments in the X10 language which has led to the proposal of non-atomic arrays for SHIM. X10 also defines an array sub-language that supports dense and sparse, distributed, multi-dimensional arrays. An X10 array object is a collection of multiple elements that can be indexed by multi-dimensional points that form the underlying index region for the array. An array's distribution specifies how its elements are distributed across multiple places in the PGAS. In SHIM, concurrent processes execute independently but they meet at specified synchronization points and communication takes place in a rendezvous fashion. The extension proposed in this report is more concerned with splitting arrays into parts, passing them to concurrent processes and synchronization of these processes with respect to arrays.

Another related work to this report is Guava, a dialect of Java that defines rules to statically guarantee that parallel threads access shared data only through synchronized methods. They identify classes into three categories: Monitors, which may be referenced from multiple threads, but whose methods are accessed serially; Values, which cannot be referenced and therefore are never shared; And Objects, which can have multiple references but only from within one thread, and therefore do not need to be synchronized. Arrays in guava can either be of type Value or Object. Thus arrays in Guava can be shared but only in one thread. Our focus in this report is to propose non-atomic arrays which can be shared over different processes over different locations.

3 Overview

This report deals with defining non-atomic arrays in SHIM and the set of operations that guarantee scheduling independent behavior. Parts of these non-atomic arrays will be handled by many concurrent processes and concurrent updates to the array should not allow inconsistencies.

To add such non-atomic arrays and ensure that no inconsistencies occur requires the addition of new constructs to the language. We will define these constructs and list the reason for various choices.

In the following section, we will propose the basic array constructs for SHIM. We list the different kinds of arrays and propose their declaration and initialization syntax. Later we explain a new construct for splitting arrays into smaller parts and discuss restrictions on the use of these parts in parallel processes.

Concurrent processes may need to synchronize on the different parts of the array during their execution. We discuss the synchronization syntax and explain the behavior of synchronization with arrays. Finally, we show two examples that illustrate the use of these extensions. The first example uses these non-atomic arrays to illustrate dynamic splitting and passing parts of an array to concurrent processes. The second example is related to synchronization of the parts of array between concurrent processes during their execution. It also explains how deadlock is avoided with serial send and receive operations between concurrent processes.

4 Array Declaration and Initialization

Determinism, the property of inevitable consequence of antecedent states, is a fundamental principle in the SHIM language. This leads to careful design issues in SHIM. One consequence is that SHIM doesn't allow pointers. Because of this reason we propose Java-like arrays. Arrays in this proposal are data structures that have operators that can be used with them. These arrays can be initialized at the time of declaration; we discuss this in detail below.

Array declaration requires the array type, its size and a unique name. The example below creates an array named *a* and type *int*. The array index ranges from 0 to $n-1$. *array-size* is an expression that does not need to be a compile-time constant.

type `[`*array-size*`]` *array-name*;

```
int[n] a;
```

Access to array elements can be done by using their index number preceded by the array name. Please note that index number will always range from 0 to n-1 where 'n' is the size of the array. If an invalid index is accessed an out-of-bound error will be generated.

array-name`[`*index*`]`

```
a[2]
```

The size of an array can be returned by placing the *.size* keyword after an array name. The *.size* operator is a read only operation and the size of the array cannot be changed after array initialization.

array-name`.size`

```
a.size
```

Arrays can be initialized during declaration. There are three ways to do so: Initialization with constants, by values from another array and by reference.

Initialization with a constant list has a similar syntax as C. The constant values are listed after the declaration in parenthesis separated by commas. The elements of the array are initialized with the values given and the size of the array is determined by the number of values listed. The size of the array specified should be equal to the number of elements in the constant list otherwise an error is generated.

type `[`*size*`]` *array-name* = { *const-list* }`;`

```
int [3] a = {25, 24, 45};
```

Initialization can also be done with values from another array. This enables to create a new array as a copy of another array. The example below creates a new array *m* whose size and values are taken from array *a*.

type `[]` *array-name* = *array-name* `;`

```
int[] m = a;
```

The size of the array is not defined wherever they are declared with initializations with other objects. The reason for this is to remove ambiguity and hence the size of the array is strictly not required when the array is being declared with initializations.

Initialization by reference creates a reference/alias for the existing array. Hence, no new memory allocation takes place; although there is an exception to this rule when it comes to initializing the array by reference from arrays which belong to processes on different locations. The example below creates a new array (name) *m* from array *a*.

type `[]` & *array-name* = *array-name* `;`

```
int[]& m = a;
```

5 Splitting Arrays

In a concurrent setup an array might be updated by many processes. To avoid the consistency and determinism issues related to this problem we require an explicit array split before the array can be passed in parts to different processes.

We choose to split the array by size, that is, by defining the size of each part. Although a split of an array can be performed by specifying the ranges (low and high values) for each part but the reason for choosing size over ranges is to reduce the complexity involved while checking the access conflicts as ranges can have conflicts when the limits are calculated at run time. Hence we consider the following syntax:

type `[]` $\&_{\text{opt}}$ { *id-list* } =
    `split(` *array-name* `,` *break-points* `);`

Thus, *split* is a function returning a tuple and the elements of this tuple are the names are the new arrays. It takes an array and a set of break points as input and returns the sub-arrays (or parts of arrays) as the outputs which are further bonded with the array names provided in the list. Another decision made regarding the split operation was to have a functional notation for the split. The reason for selecting this syntax is that the split operation is unique and is a function returning a tuple. The functional notation makes it intuitive to the user that the split operation takes different types of arguments (an array, and integer break points) and returns a tuple consisting of a set of arrays.

To make things efficient we confine an array split operation to allow a split into exactly two pieces. Thus the syntax becomes:

type `[]` $\&_{\text{opt}}$ `{` *id* `,` *id* `}` =
    `split(` *array-name* `,` *break-point* `);`

In the syntax above, ampersand (&) symbol is optional. This means that and array split can be categorized into to groups: split by value and split by reference. Split by value creates new arrays with new memory allocations and copies the value from the original array whereas split by reference creates new array names as reference to different parts of the original array.

Consider the split by value example below, it creates two arrays m and n and copies the elements 0 to $b-1$ of array a to m and b to a.size -1 to n.

```
int[] {m, n} = split(a, b);
```

In contrast, the following example illustrates split by reference. It creates two new array names (only references/alias) in which m references elements 0 to $b-1$ of array a and n refers to elements b to a.size -1 of array a.

```
int[]& {m, n} = split(a, b);
```

6 Arrays and Function Declaration

SHIM allows function arguments to be passed in two ways: by value or by reference. Similarly, we propose passing arrays in two ways: array arguments passed by value or by reference.

Functions that accept an array argument by value will get a copy of the array from the calling function and any change made to such an array will not be reflected in the original array. Below is an example of a function that takes an array argument by value.

```
void myFunc(int[] new_array)
```

The above syntax is very different from usual C/Java syntax. In C/Java this kind of syntax is used to pass arrays by reference since they do not allow a straight way to pass arrays by value. But we propose a different kind of syntax and allow arrays to be passed either by value or by reference. Following contrast with the syntax of passing arrays by reference will make things clearer.

Functions that accept reference to an array as an argument will get a reference to the original array and any changes made to the array will be reflected in the original array as well. Below is an example declaration of a function that has an array argument by reference.

```
void myFunc(int[]& new_array)
```

The syntax of passing arrays by reference is much like passing normal variables by reference, that is, the array arguments that include ampersand (&) in there declaration are passed by reference. To sum up, the argument variables (even arrays) those are declared with ampersand (&) are passed by reference and other are passed by value.

Again, you can notice in the examples above that the size of the array is forbidden when the array is being declared as an argument. This is to prevent the array size mismatch between the arrays passed from the calling function. The size operator can be used in the function definition to determine the actual number of elements in it.

Below illustrates the passing of arrays to functions.

```
void f(int[]& a) {              // a = {1,1,1}
  for(int i=0; i<a.size; i++)
    a[i] = 2;
}                               // a = {2,2,2}

void g(int[] a) {               // a = {1,1,1}
  for(int i=0; i<a.size; i++)
    a[i] = 3;
}                               // a = {3,3,3}

void h(int[]& a) {              // a = {1,1}
  for(int i=0; i<a.size; i++)
    a[i] = 4;
}                               // a = {4,4}

void main() {
  int[5] arr = {1,1,1,1,1};
  int bp = 3;
  int[]& {m, n} =
        split(arr, bp);         // int [3]m, [2]n
  f(m); par g(m); par h(n);
}                               // arr = {2,2,2,4,4}
```

In the above example, the main function has an array of integers named *arr* of size 5. The split operation breaks the array in two parts at index 2. Thus, array m gets index 0 to 2 of array *arr* and array n gets index 3 to 4 of array *arr*. Note that the size of arrays m and n will be 3 and 2 respectively and their incides will range from 0 to 2 and 0 to 1 respectively. After the split, the *par* statement allows the three children *f(m)*, *g(m)* and *h(n)* to run in parallel. Arrays m and n are passed by reference to function f and h respectively. Hence, any change made to the arrays in these functions will be reflected in *main* as well. And array m is passed to function g by value. Any change made this array by function g would be reflected in g only and not in *main*.

After all the parallel processes end and main resumes, the value of the array *a* is $\{2,2,2,4,4\}$. This is because changes made to the array *a* by function *g* are totally discarded when *g* terminates. But the changes made to the their respective arrays *a* by function *f* and *h* are reflected the original arrays *m* and *n* even after their lifetime.

In SHIM, a variable can not be passed by reference to more than one parallel process. In other words, a variable can be passed to as many processes in parallel by value but to only one process by reference. The reason for this will be clear from the following example.

```
void f(int[]& a);
void g(int[] a);

void main() {
  int[5] arr = {1,3,4,2,5};
  int[]& {m, n} = split(arr, 3);

  f(m); par f(n);   // OK
  f(m); par g(m);   // OK
  f(m); par f(m);   // Error: Same array
                    // cannot be passed by
                    // reference twice
  g(m); par g(m);   // OK
  f(m); par f(arr); // Error: m is an alias of
                    // arr and both cannot be
                    // reference in parallel
  f(m); par g(arr); // OK
  f(arr); par g(m); // OK
}
```

In the above example, the third parallel function call *f(m); par f(m);* is invalid because function *f* takes an array by reference but this parallel function call violates the rule that the same variable (or array) cannot be passed by reference to more the one parallel process.

Also in the above example, consider the fifth parallel function call *f(m); par f(arr);*. Array *m* references a part of array *arr* hence both of them hold common elements (also, *m* is an alias of *arr* in a way). Hence, we cannot pass such arrays by reference together in parallel. All other statements are valid.

The reason for the restriction—the same variable cannot be passed by reference to more than one parallel process—is that when more than one parallel process is allowed to change the same variable, inconsistencies may occur. To understand this, you can assume that a process accepting an object by reference has write access to it and a read access to an object accepted by value. And no two parallel processes can have write access to the same object.

7 Arrays and Synchronization

In SHIM, all variables that are declared as function arguments can be synchronized with other processes with the keyword 'next' preceding their name.

`next` *variable-name*`;`

A send or a receive operation depends on the declaration. A send action is taken if the argument variable has been declared as reference and a receive action otherwise.

The *next* operation works in the very similar fashion for arrays. A function accepting an array by reference will be allowed to change its content and will be sending out the array on *next* operations. And, a function accepting an array by value will be receiving the array on the *next* operations from other parallel processes that hold that array by reference. The example below illustrates this behavior.

```
void f(int[]& arr) {            // arr = {1,1,1}
  for(int i=0; i<arr.size; i++)
    arr[i] = 2;
                                // arr = {2,2,2}
  next arr;                     // Send 'arr'
}                               // arr = {2,2,2}

void g(int[] arr) {             // arr = {1,1,1}
  for(int i=0; i<arr.size; i++)
    arr[i] = 3;
                                // arr = {3,3,3}
  next arr;                     // Recieve 'arr'
}                               // arr = {2,2,2}

void main() {
  int[3] arr = {1,1,1};
  f(arr); par g(arr);
}                               // arr = {2,2,2}
```

In the above example, main function passes the array *arr* to two processes *f* and *g* by reference and value respectively. Thus, a *next* operation on *arr* in function *f* sends its *arr* content and a *next* operation on *arr* in function *f* receives the contents of *arr*. Notice in the above example, that after synchronization the content of *arr* changes to $\{2,2,2\}$ in function *g*.

During synchronization, a process waits for all the parallel processes that hold the same variable that it is synchronizing upon. If there is no process to synchronize, the *next* operation is bypassed. The behavior is same with *next* on arrays as well, that is, a process in which an array was declared as an argument will wait for all the parallel processes which have that array, while synchronizing upon that array.

We mentioned before that a variable can not be passed by reference to more than one parallel process. Another reason for this restriction, in the context of synchronization, is that it becomes unclear that which process sends the variable to the receiving processes. If there are two processes sending the same data to a third process, problems may arise.

Consider an example where we want two parallel processes to operate on the same array but make changes only to the different parts of it. Also the two processes may require the other halves, to which they do not have write access to, with them for reading those values. To solve this problem we will need to split the array in two halves, pass each half by reference to each process, and pass the other halves to each process by value. Please consider the example below:

```
void f(int[]& a, int[] b) {
  next a; // Send array 'a'
  next b; // Recieve array 'b'
}

void g(int[] a, int[]& b){
```

```
  next a; // Recieve array 'a'
  next b; // Send array 'b'
}

void main() {
  int[6] arr = {1,3,4,2,5,8};
  int[]& {m, n} = split(arr, arr.size/2 );

  f(m, n); par g(m, n);
}
```

In the above example, the array *arr* is split in two parts: *m* and *n*. Now, function *f* gets *m* by reference and *n* by values and the function *g* gets *n* by reference and *m* by value. Further, when function *f* performs the *next* operation on *a* (alias of *m*) it sends its values to function *g* to read them. And when function *f* performs the *next* operation on *b* (alias of *n*) it waits to read its latest values from function *g*.

8 Examples

8.1 Merge Sort

Merge sort is a classical sorting algorithm. Here we will add parallelism to the merge sorting algorithm and re-write it in SHIM and the extension proposed so far. This program uses the parallel constructs of SHIM, the array constructs mentioned in this proposal and the rules defined to pass them to functions.

```
void mergeSort(int[] &a) {

  if(a.size>2) {
    // Split, if array has more than 2 elements
    int[]& {m, n} = split(a, a.size/2);
    // Recurs with each part in parallel
    mergeSort(m); par mergeSort(n);
  }

  // Merge Start
  int[a.size] temp;
  int i=0, j=a.size/2, tmp_pos=0;

  while ((i < a.size/2) && (j < a.size)) {
    if (a[i] <= a[j])
      temp[tmp_pos++] = a[i++];

    else
      temp[tmp_pos++] = a[j++];
  }

  while (i < a.size/2)
    temp[tmp_pos++] = a[i++];

  while (j < a.size)
    temp[tmp_pos++] = a[j++];

  for (i=0; i < a.size; i++)
    a[i] = temp[i];
}

void main() {
  int[10] a = {1,4,6,4,6,3,7,3,5,2};
  mergeSort(a);
}
```

The above merge sort program takes an array, splits it in two and passes each part to two parallel processes. Each part is sorted and returned back. And thus, in a recursive way the above example sorts the array with parallel execution.

8.2 Counter

This example exploits the use of synchronization provided by SHIM and the extension of synchronization for arrays proposed in this report. The example is meant to explain the synchronization behavior of the language and is not an alternative proposal for the implementation of a counter.

In this example we take an array of four elements. Since we are implementing a binary counter, each element of the array will represent a bit of the number. Hence, we implement a four-bit binary counter.

Following is the code for the counter. The output follows the code further followed by the explanation of the example. Note, that the *print* function, as given in this example, is neither the part of the SHIM specification nor of the proposal in this report. It is present only to give an idea of what values the counter code generates.

```
// Counter function
void counter(int[]& x, int[] y) {
  while(1) {
    int flag = 1;
    // Loop to check if all elements are 1
    for(int i=0; i<y.size; i++)
      if(y[i] == 0)
        flag = 0;

    // Switch 'x[0]' if all 'y' are 1
    if(flag == 1)
      x[0] = 1 - x[0];

    next x; // Send array 'x'
    next y; // Receive array 'y'
  }
}

// Invert Process, to switch between 0 and 1
void invert(int[]& z) {
  while (1) {
    z[0] = 1 - z[0]; // Switch value
    next z;          // Send array 'z'
  }
}

void printArr(int[] x) {
  while (1) {
    print(x);
    next x;    // Receive array 'x'
  }
}

void main() {
  int[4] a = {0,0,0,0};
  int[]& {b,c} = split(a, 1);  // int [1]b, [3]c
  int[]& {d,e} = split(c, 1);  // int [1]d, [2]e
  int[]& {f,g} = split(e, 1);  // int [1]f, [1]g

  printArr(a); par
  counter(b,c); par
  counter(d,e); par
  counter(f,g); par
  invert(g);
}
```

The output is

```
0000  0001  0010  0011  0100  0101  0110  0111
1000  1001  1010  1011  1100  1101  1110  1111
```

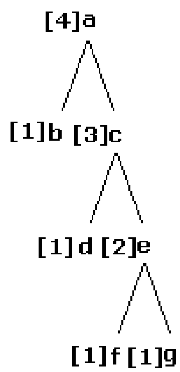

Figure 1: Splitting arrays

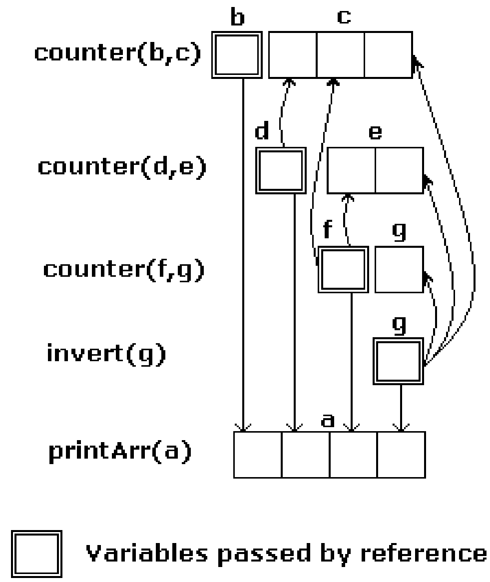

Figure 2: A synchronization example

The *main* function of the code has an array *a* of length four. Each element of this array represent a bit of the array. We split this array into two parts: *b* holds the first element of *a* and *c* holds the remaining 3 elements. We further split array *c* into *d* and *e* in the similar way. Figure 1 shows the split of arrays in our example.

An array will only share an element with its parent or its children, recursively. It can be noticed from the figure that which arrays share elements. For example, array *g* will share its element with arrays *e*, *c* and *a* but not with *f*, *d* and *b*. And array, *c* shares some of its elements with arrays *a*, *d*, *e*, *f* and *g* but none with *b*.

We split the array such that arrays *b*, *d*, *f* and *g* each hold a single element and together cover the entire array *a*. Each of these elements is passed by reference to the 4 parallel processes, that is, three instances of function *counter* and one instance of function *invert*. Also, these processes are passed the arrays on their right (in the figure or in the split) by value. Note that no element of the original array is passed by reference to more than one parallel process. Otherwise, an error would have been generated as explained before.

Now, each of the four processes run their code and will wait for synchronization when a *next* statement is reached. During synchronization the processes which holds the variable (to be synchronized upon) by reference will send the variable and the other processes will receive. The synchronization takes place in a rendezvous fashion and process will send a variable (which it holds with reference) to all the other processes (which hold that variable by value). Figure 2 illustrates the synchronization.

As mentioned the process holding a variables by reference will send out its value to all the processes participating in the synchronization over the same variable. For example, array *g* has common elements with arrays *e* and *c* and hence, the process *invert(g)* (which hold *g* by reference) will send out value of variable *g* to all the other 3 processes which can be seen in the figure. Similarly, other arrays (and its elements) will be synchronized.

The *invert(h)* process switches the value of its element between zero and one during each iteration before synchronization and the *counter* processes check if all the elements of its array *y* (which is held by value) are equal to one. If they are, then the *counter* process changes the only element of its array *x* (which is held by reference) to 1 and this change is propagated to all other concerned processes during synchronization at the end of the iteration.

It is important to observe the synchronization behavior in the *counter* function. The statement *next x* (a send statement) precedes the *next y* (a receive statement). SHIM is prone to deadlocks in situations where serial send and receive statements executed. Careful observation is needed to be given to such situations. The *counter* function used in the example may seem to deadlock but it doesn't and we will explain this by observing the synchronization in steps.

Consider the synchronization figure above. Function *invert(g)* will send its array to function *counter(f,g)*. But, *counter(f,g)* cannot receive *g* until it sends out *f*. And in turn, *counter(f,g)* cannot send *f* until the function *counter(b,c)* sends *b*. So, the first synchronization that takes place in a given iteration will be between the function *counter(b,c)* and *printArr(a)*. After *counter(b,c)* sends *b* it will receive elements of *c* from functions *counter(d,e)*, *counter(f,g)*, and *invert(g)*. After this, the function *counter(d,e)* receives *e* from *counter(f,g)*, and *invert(g)*. And in the end, function *counter(f,g)* receives *g* from *invert(g)*.

Thus, the processes will execute in parallel without a deadlock and counter output is printed by *printArr(a)* in each iteration.

9 Grammar

Here we list the grammar for SHIM with the syntax proposed for the extension of non-atomic arrays.

$$
\begin{array}{rl}
e & ::= L \mid V \mid V\ \texttt{[}\ \texttt{e}\ \texttt{]} \mid V\ \texttt{.size} \mid op_1\ e \mid e\ op_2\ e \mid \texttt{(}\ e\ \texttt{)} \\
s & ::= V\ \texttt{=}\ e\texttt{;} \mid V\ \texttt{[}\ e\ \texttt{]}\ \texttt{=}\ e\texttt{;} \mid P\texttt{(}\ (V(\texttt{,}V)^*)?\ \texttt{);} \\
 & \quad \mid \texttt{\{}\ b^*\ \texttt{\}} \mid \texttt{if (}\ e\ \texttt{)}\ s\ \texttt{else}\ s \mid \texttt{while (}\ e\ \texttt{)}\ s \\
 & \quad \mid s\ \texttt{par}\ s \mid \texttt{next}\ V\texttt{;} \mid \texttt{try}\ s\ \texttt{catch(}\ E\ \texttt{)}\ s \mid \texttt{throw}\ E\texttt{;} \\
b & ::= T\ V\texttt{;} \mid T\ \texttt{[}\ e\ \texttt{]}\ V\ (\ \texttt{= \{}\ L\ (\texttt{,}L)^*\ \texttt{\}}\ )?\ \texttt{;} \\
 & \quad \mid T\ \texttt{[]}\ (\&)?\ V\ \texttt{=}\ V\ \texttt{;} \\
 & \quad \mid T\ \texttt{[]}\ (\&)?\ \texttt{\{}\ V\ \texttt{,}\ V\ \texttt{\}}\ \texttt{= split(}\ V\ \texttt{,}\ e\ \texttt{);} \mid s \\
d & ::= T\ V \mid T\ \&V \mid T\ \texttt{[]}\ (\&)?\ V \\
p & ::= \texttt{void}\ P\texttt{(}\ (d(\texttt{,}d)^*)?\ \texttt{)}\ \texttt{\{}\ b^*\ \texttt{\}} \\
m & ::= p^*\ \ \texttt{void main()}\ \texttt{\{}\ b^*\ \texttt{\}}
\end{array}
$$

Here, e denotes expressions, s statements, b blocks, d declarations, p a program, and m a main function. L denotes literals, T types (e.g., `int`, `void`), E exceptions, V variables, and P procedures. `par` binds most tightly.

The grammar above adds the functionality of declaration of new arrays, initializing them, splitting them and accessing the elements of arrays in SHIM. With this extension we add non-atomic arrays while preserving the main ideas of SHIM being a deterministic concurrent language.

10 Conclusion

The proposed extension to SHIM is an attempt to add arrays to its concurrent framework. For concurrent programming languages like SHIM, scheduling independence should be an important concern because the output of the program may vary in a concurrent setup depending upon the scheduling choices made by the system. We proposed rules to split an array into smaller parts, pass them between various processes and synchronize upon these array parts during their execution.

We provide the *split* construct as a result of which concurrent processes are able update to different parts of the array simultaneously. We also define restrictions on passing the parts of an array to parallel processes which guarantees consistency while allowing concurrent updation of the array.

Synchronization in serial steps to send and receive variable might lead to deadlock and we explain that such situations need careful evaluation in SHIM. It can be said, that we avoid deadlocks with this proposal when synchronizing upon the array by splitting the array which in turn breaks cycles in the synchronization graph. Although, this claim remains to be proven.

Implemention of this proposal remains future work. Also, further work needs to be done on this extension to allow the split and synchronization of multi-dimensional arrays.