Universiteit Leiden
The Netherlands

Asynchronous Programming in the Abstract Behavioural Specification Language
Azadbakht, K.

Citation
Azadbakht, K. (2019, December 11). *Asynchronous Programming in the Abstract Behavioural Specification Language*. Retrieved from https://hdl.handle.net/1887/81818

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/81818

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

Leiden University Repository

The handle http://hdl.handle.net/1887/81818 holds various files of this Leiden University dissertation.

Author: Azadbakht, K.
Title: Asynchronous Programming in the Abstract Behavioural Specification Language
Issue Date: 2019-12-11

Chapter 2

Preferential Attachment on Multicore Systems

2.1 Introduction

Social networks in the real world appear in various domains such as, among others, friendship, communication, collaboration and citation networks. Social networks demonstrate nontrivial structural features, such as power-law degree distributions, that distinguish them from random graphs. There exist various network generation models that synthesize artificial graphs that capture properties of real-world social networks. Some existing network generative models are the Erdos-Renyi (ER) [33] model of random graphs, the Watts-Strogatz (WS) [75] model of Small-world networks, and the Barabasi-Albert model of scale-free networks. Among these models, Barabasi-Albert model, which is based on Preferential Attachment [18], is one of the most commonly used models to produce artificial networks, because of its explanatory power, conceptual simplicity, and interesting mathematical properties [73]. The need for efficient and scalable methods of network generation is frequently mentioned in the literature, particularly for the preferential attachment process [4, 9, 19, 23, 41, 58–60, 73, 79]. Scalable implementations are essential since massive networks are important; there are fundamental differences between the structure of small and massive networks even if they are generated according to the same model, and there are many patterns that emerge only in massive networks [56]. Analysis of the large-scale networks is of importance in many areas, e.g. data-mining, network sciences, physics, and social sciences [16]. The property that we have focused on in this chapter is the degree of the nodes and by preferential attachment (PA) we mean degree-based preferential attachment. In PA-based generation of the networks, each node is introduced to the existing graph preferentially based on the degrees of the existing nodes, i.e., the more the degree of an existing node, the higher the probability of choosing it as the target of a new connection.

The PA-based parallel and distributed versions of generating the scale-free graphs are based on a partitioning of the nodes and a parallel process for each partition which adds edges to its nodes. The edges are generated by random selection of target nodes. The data structure prescribed in the *Copy Model* [54] guarantees that the selection of the target is done consistently, e.g., the probability distribution of selecting the target nodes in the parallel version should remain the same as the distribution in the sequential one. However, from the point of view of the control flow, the following main problem arises: random selection of the target node requires synchronization between the parallel processes. The process that hosts the randomly selected target node has possibly not determined the node yet. However the process hosting the source node must be informed about the target node in the future once it is determined.

random selection requires synchronization between the parallel processes, i.e., the target nodes are not resolved yet. and the need for conflict resolution, namely, the selection of a node which has already been selected as a target of the given source node.

The main contribution of this chapter is a high-level Actor-based model for the PA-based generation of networks which avoids the use of low-level intricate synchronization mechanisms. A key feature of the Actor-based model itself, so-called *cooperative scheduling*, however, poses a major challenge to its implementation. In this chapter, we discuss the scalability of a multicore implementation based on Haskell which manages cooperative scheduling by exploiting the high-level and first-class concept of continuations [62]. Continuations provide the means to “pause” (part of) the program’s execution, and programmatically switch to another execution context; the paused computation can be later resumed. Thus, continuations can faithfully implement cooperative scheduling in a high-level, language-builtin manner.

The rest of the chapter is organized as follows. The description of the Actor-based modeling framework which is used to model the PA-based generation of massive networks is given in section 2.2. Section 2.3 elaborates on parallelizing the PA model. Section 2.4 mentions the related works. Finally we conclude in section 2.5.

2.2 The Modeling Framework

We propose an Actor-based modeling framework that supports concurrency and synchronization mechanisms for concurrent objects. It extends the ABS language and, apart from the ABS functional layer which includes algebraic data types and pattern matching, it additionally features global arrays as a mutable data structure shared among objects. This extension fits well in the multicore setting to decrease the amount of costly message passing, and also to simplify the model. In general, this feature can cause complicated and hard-to-verify programs. Therefore the model only allows using this feature in a disciplined manner, which restricts the array to initially *unresolved* slots with single-write access (also known as *promises* in languages like Haskell and Scala[1]), to avoid race conditions. In this chapter, and chapter 3 and 4 the initial value 0 denotes the unresolved state of a slot.

2.3 Parallel Model of the PA

In this section we present the solution for the PA problem utilizing the idea of active objects cooperating in a multicore setting. For the solution we adopt the *copy model*, introduced in [54]. We first introduce the main data structure of the proposed approach which is based on the graph representation in *copy model*. Next we present the basic synchronization and communication mechanism underlying our solution and its advantages over existing solutions.

2.3.1 The Graph Representation

We introduce one shared array, arr, as the main data structure that holds the representation of the graph. The array consists of the edges of the graph. Each (i, j) where $i, j > 0$ and $j = i + 1$, and $j \ mod \ 2 = 0$ shows an edge in the graph between $arr[i]$ and $arr[j]$ (Figure 2.1(a)).

According to the PA, each node is added to the existing graph via a constant number of edges (referred to as m) targeting *distinct* nodes. There is also an initial clique, a complete graph with the size of $m0$ where ($m0 \geqslant m$), which is stored at the beginning of the array. Therefore the size of the array is calculated based on the number of nodes, num, and the number of edges that connect each new node to the existing distinct nodes, m. The connections of a new node are established via a probability distribution of the degrees of the nodes in the existing graph, that is, the more the degree of the existing node, the more the probability of choosing it as the target. For instance, if the node n is the new node to be added to the graph with the existing graph with $[1..n-1]$ nodes then, according to equation 2.1, the probability distribution of choosing the existing nodes is $[p_1..p_{n-1}]$. ($deg(i)$ gives the degree of the node i in the existing graph)

$$p_i = \frac{deg(i)}{\sum_{j=1}^{n-1} deg(j)} \qquad\qquad \sum_{i=1}^{n-1} p_i = 1 \tag{2.1}$$

As mentioned, the connections for the new node should be distinct. Therefore if a duplicate happens the approach retries to make a new connection until all the connections are distinct. This graph representation provides the above mentioned probability distribution since the number of occurrences of each node in the array is equal to its degree. Figure 2.1(b) represents the position of node n in the graph array, where $m = 3$. In order to add node n to the existing graph containing $n - 1$ nodes, with the assumption that $m = 3$, targets are selected randomly from the slots

[1]`https://docs.scala-lang.org/sips/completed/futures-promises.html`

that are located previous to the node n (with the principle shown in Figure 2.2). It is obvious that self-loop cannot happen, i.e., an edge whose source and target are the same. Figure 2.1(c) illustrates an optimization on the array so that the array only contains the targets of the edges since the sources for each node are calculable. The array is half size as the one in Figure 2.1(b). Each slot in the array can have one of two states: *resolved* or *unresolved*. In the former case it contains the node number which is greater than zero, and in the latter it contains zero.

A sequential solution for the generation of such graphs consists of processing the array from left to right to resolve all the slots. The parallel alternative, on the other hand, is to have multiple active objects processing partitions of the array in parallel. As shown in the following equations, we distinguish between the following uses of indices. At the lowest level we have the indices of the slots. The next level is the id of the nodes. Each node contains a sequence of slots. Finally at the top level we have the id of the partitions. Each partition contains nodes and consequently slots. In the proposed approach the partitions satisfy the following equations which express that the sets of indices of the partitions are mutually disjoint (equation 2.3); that their union is equal to the whole array (equation 2.2); furthermore, the sequence of slots of each node must be placed in one partition (equation 2.4) so that one active object resolves the new node and race conditions are avoided for the checking duplicates:

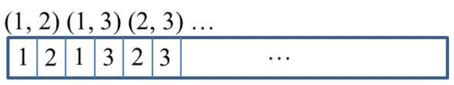
$$\bigcup_{i=1}^{w} par_i = G \tag{2.2}$$

$$\forall (1 \leq (i \neq j) \leq w). par_i \cap par_j = \emptyset \tag{2.3}$$

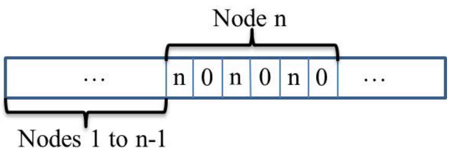
$$\forall i, j \in G.(node(i) = node(j)) \rightarrow (par(i) = par(j)) \tag{2.4}$$

where G is the global set containing all the indices of the shared array, w holds the number of partitions, par_i is the set which holds the indices of the ith partition of the array, $node(i)$ is a function that returns the node id to which the slot of the array with index i belongs, and $par(i)$ is a function that returns the partition id to which the index i belongs. Note that indices that belong to a specific node differ from the occurrences of that specific node in the array. The former indices are the slots that represents the edges that are created during introducing the *new* node to the graph, which its size is constant (denoted by m), while the latter changes during the graph generation.

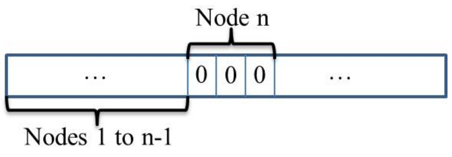
There are different approaches to partition the array so that they hold the above equations, such as Consecutive and Round Robin Node Partitioning (CSP and RRP respectively). As it is shown in [4], RRP is more efficient and it is observed a better load balancing among processors as well as less unresolved chains of dependencies which leads to less computational overhead. Therefore we have utilized RRP to partition the array among active objects.



(a) The array which represents the graph



(b) The nth node and its connections to the existing graph with $n-1$ nodes



(c) A memory usage optimization to the array based on *copy model*

Figure 2.1: The array representing the graph

2.3.2 Synchronization of Chains of Unresolved Dependencies

Each active object only resolves (i.e. writes to) the slots which belong to its own partition . Nevertheless it can read all the slots throughout the array. In the parallel solution, an active object may select a slot as the target which is not resolved yet since either the other active object responsible for the target slot has not yet processed it or the target slot may wait for another target itself (see dependency chains in figure 2.2). The way waiting for unresolved slots is managed is crucial for the complexity of the model and its scalability. Next we describe the two main approaches to deal with unresolved dependencies (Figure 2.3):

Synchronization by communication: Active object A processes its own partition of the array and for each randomly selected, unresolved slot it sends a request to the object B responsible for the target. When object B processes the request, it checks whether the slot is resolved. If it is not then it stores the information of the request (e.g. the sender id, the slot requiring the value of the target) in a corresponding data structure. Because B is the only object which writes to the target slot when it

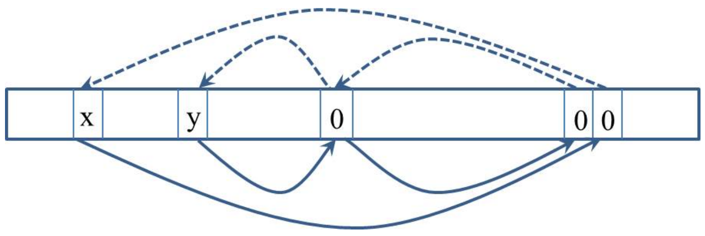
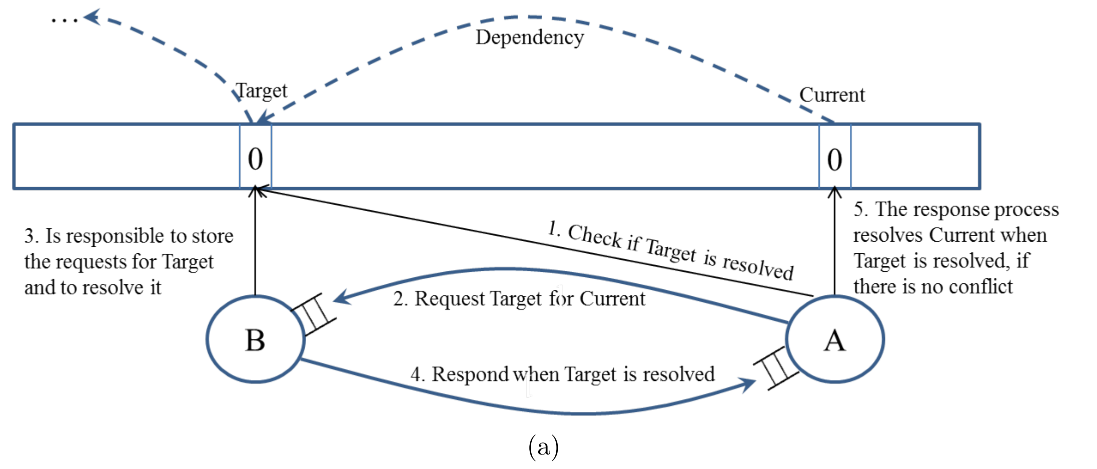


Figure 2.2: An example of the general sketch of dependencies (right to left) and computations (left to right)


...
Dependency
Target
Current
0
0
3. Is responsible to store the requests for Target and to resolve it
1. Check if Target is resolved
5. The response process resolves Current when Target is resolved, if there is no conflict
2. Request Target for Current
B
A
4. Respond when Target is resolved
(a)

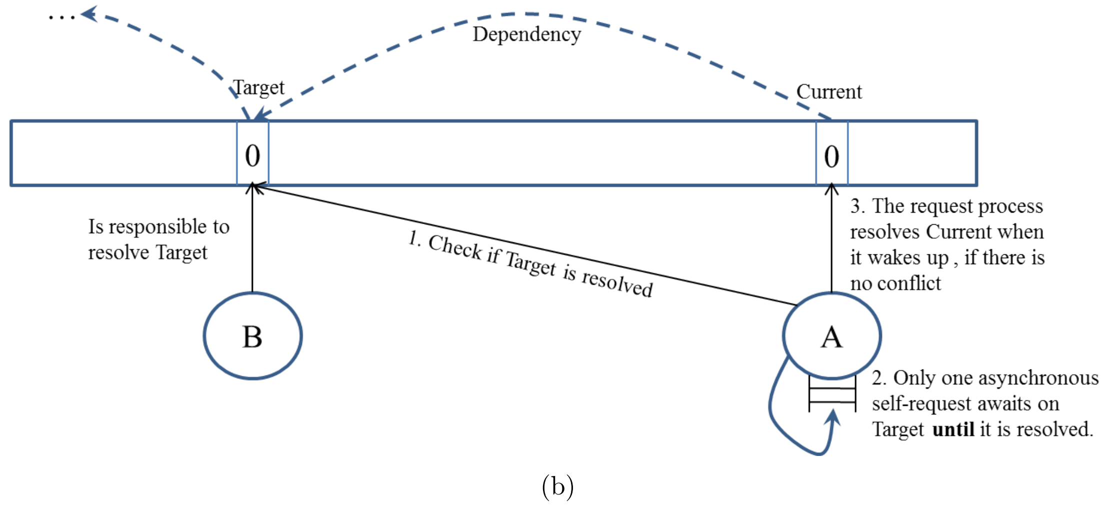

...
Dependency
Target
Current
0
0
Is responsible to resolve Target
1. Check if Target is resolved
3. The request process resolves Current when it wakes up , if there is no conflict
B
A
2. Only one asynchronous self-request awaits on Target **until** it is resolved.
(b)

Figure 2.3: Two different solutions for the PA problem (the second one is the proposed approach)

is resolved, it suffices that B answers all the stored requests waiting for the resolved target by broadcasting the value of the slot. As such this approach exploits the wait-notify pattern rather than busy-wait polling, and it can be efficient depending on how the programmer implements the data structure. However, this approach involves a low-level user-defined management of the requests through the explicit user-defined implementation of the storage and retrieval of the requests. Note that in this approach there are *exactly* two messages that have to be passed for each request for an unresolved slot (Figure 2.3(a)).

Synchronization by Cooperative Scheduling: Active object A processes its own partition of the array and for each unresolved randomly selected slot it sends an asynchronous *self* request called "request" for the target value. When object A schedules and processes the request "request" it checks whether the slot is resolved or not (by a Boolean condition) and if not it *awaits* on this condition. This means that the request process is *suspended*. It is notified when the boolean condition evaluates to *true* and stored back in the object's queue of active processes (Figure 2.3(b)). This approach also avoids busy-waiting and follows the wait-notify pattern. However, the key feature in this approach is the use of cooperative scheduling in which the executing process of an active object can release conditionally the control cooperatively so that another process from the queue of the object can be executed. The continuation of the process which has conditionally released the control will be stored into a separate queue of suspended processes. These processes are stored again in the object's queue for execution when they are notified. The Haskell implementation of this mechanism takes care of the low-level storage, execution and suspension of the processes generated by asynchronous messages. The ABS code itself, see below, remains high-level by means of its programming abstractions describing asynchronous messaging and conditional release of control.

2.3.3 The Actor-based Model of PA

The main part of the encoding of our proposed approach is depicted as a pseudo-code in Figure 2.4. The full implementation of the model is provided online[2]. The worker objects are active objects which resolve their corresponding partitions. To this aim, each worker goes through its own partition and it checks a randomly selected target for each of its slots (note that m denotes the number of connections, or slots in the array, per node). Since we use the optimized array representation (shown in Figure 2.1(c)), half of the array that corresponds to sources of the edges are not part of the array and the values (i.e., node numbers) are calculable from the index. However those indices can be targeted. The auxiliary function $result(arr, target)$ checks the target index. If it is calculable, then it calculates and returns the value without referring to the array. Otherwise, the value of the index should be retrieved from the array. In such case, if the target slot is already resolved then the worker takes the value and resolves the slot of the `current` index in case there is no conflict. If it is not resolved yet then it calls the *request* method asynchronously. The *request* method awaits on the target until it is resolved. Then it uses the value of the resolved target to resolve the current slot, if there is no duplicate. In case of a duplicate, the algorithm selects another target randomly in the same range as the previous one.

Note that the calls to the request method in lines 8 and 22 are asynchronous (denoted by exclamation mark) and synchronous (denoted by dot) respectively. The asynchronous call is introduced so as to spawn one process per each unresolved dependency. In the synchronous call, however, there is no need to spawn a new process since the current process is already introduced for the corresponding unresolved dependency. Note that suspension of such a process thus involves in general an entire call stack, which poses one of the major challenges to the implementation of ABS, but which is dealt with in Haskell by the high-level and first-class concept

[2]`https://github.com/kazadbakht/PA/blob/master/src/ParProRR.abs`

of continuations (described in more detail below).

```
1:  Each active object $O$ executes the following in parallel
2:  run(...) : **void**
3:  **for each** Node $i$ in the partition **do**
4:      **for** $j = 1$ *to* $m$ **do**
5:          $target \leftarrow$ random$[1..(i-1)2m]$
6:          $current = (i-1)m + j$
7:          **if** $result(arr, target) = 0$ **then**
8:              **this** ! $request(current, target)$
9:          **else if** $duplicate(result(arr, target), current)$ **then**
10:             $j = j - 1$                         ▷ Repeat for the current slot $j$
11:         **else**
12:             $arr[current] = result(arr, target)$                ▷ Resolved
13:
14:
15: request($target : Int, current : Int$) : **void**
16: **await** $(result(arr, target) \neq 0)$
17:                                     ▷ At this point the target is resolved
18: $value = result(arr, target)$
19: **if** $duplicate(value, current)$ **then**
20:     $target =$ random$[1..(target/m)2m]$
21:                                 ▷ Calculate the target for the current again
22:     **this**.$request(target, current)$
23: **else**
24:     $arr[current] = value$                                  ▷ Resolved
25:
26:
27: duplicate($target : Int, current : Int$) : $Boolean$
28: **for each** $i$ in (indices of the node to which $current$ belongs) **do**
29:     **if** $arr[i] == value$ **then**
30:         **return** True
31: **return** False
```

Figure 2.4: The sketch of the proposed approach

2.4 Related Work

There exist some attempts to develop efficient implementations of the PA model [4, 9, 19, 41, 58, 59, 73, 79]. Some existing works focus on more efficient implementations of the sequential version [9, 19, 73]. Such methods propose the utilization of data-structures that are efficient with respect to memory consumption and time complexity. Few existing methods are based on a parallel implementation of the PA model [4, 59, 79], among which some methods [59, 79] are based on a version of the PA model which does not satisfy its basic criteria (i.e., consistency with the original model). The approach in [4] requires complex synchronization and communication management and generates considerable overhead of message passing. This stems from that this latter approach is not developed for a multicore setting but for a distributed one. However our focus is to have a high-level parallel implementation of the original PA model utilizing the computational power of multicore architectures [12].

2.5 Conclusion and Future Work

We showed that the PA-based generation of networks allows a high-level multicore implementation using the ABS language and its Haskell backend that supports cooperative multitasking via continuations and multicore parallelism via its lightweight threads. An experimental validation of a scalable distributed implementation of our model based on Haskell is presented in [12].

Future work will be dedicated toward optimizations of the Haskell runtime system for the ABS. Other work of interest is to formally restrict the use of shared data structures in the ABS to ensure encapsulation. One particular approach is to extend the compositional proof-theory of concurrent objects [31] with foot-prints [28] which capture write accesses to the shared data structures and which can be used to express disjointness of these write accesses.