On The Nature of Cooperative Scheduling in Active Objects

Vlad Serbanescu
Centrum Wiskunde and Informatica
Amsterdam, Netherlands
vlad.serbanescu@cwi.nl

Frank de Boer
Centrum Wiskunde and Informatica
Amsterdam, Netherlands
f.s.de.boer@cwi.nl

ABSTRACT

Active objects interact via asynchronous messages which specify method invocations. In contrast to the run to completion mode of method execution, mechanisms for suspending the execution of a method allow an active object to schedule cooperatively its methods in a co-routine manner. In this paper, we show how cooperative scheduling can be reduced to a run to completion mode of execution. We do so by a formal translation using a guarded command language for describing the execution of method bodies.

KEYWORDS

Active objects, cooperative scheduling, run to completion, semantics, correctness

ACM Reference Format:
Vlad Serbanescu and Frank de Boer. 2020. On The Nature of Cooperative Scheduling in Active Objects. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3341105.3373896

1 INTRODUCTION

Active objects provide a powerful conceptual model of distributed systems (see [1] for a survey of active object languages). Active objects support a "programming to interfaces" discipline by a strict encapsulation of their local state and communication via asynchronous method calls. Asynchronous method calls generate messages which are stored in the (usually FIFO) buffer of the callee. In the basic, pure asynchronous model of active objects (as described for example by the Rebeca active object language [10]) methods are executed in a run to completion mode.

Various extensions of this basic model exist which support additional synchronization mechanisms (again, see [1]). For example, the Abstract Behavioral Specification (ABS, for short) language ([7]), supports a mechanism for synchronizing on return values by so-called futures, and a mechanism for cooperative scheduling of method invocations by a single active object.

A future is dynamically generated by an asynchronous call of a method which defines a return type, and is used as a reference to the return value which can be read by the so-called **get** operation. Such an operation blocks the execution in case the return value has not yet been produced.

Cooperative scheduling of method invocations by a single active object is enabled in ABS by so-called **await** statements. A Boolean await statement suspends the execution of the current method invocation, and allows the active object to schedule other, enabled method invocations stored in its process queue. Await statements which involve a future suspend the execution of the method invocation until the corresponding return value has been produced.

Cooperative scheduling in ABS by means of await statements provides a powerful abstraction which supports a controlled co-routine manner of execution of the method invocations by an active object. A key feature of the execution of methods in ABS is that it does *not* provide an explicit statement for resuming a suspended method. Methods are only rescheduled for execution by the underlying scheduler. This implicit behaviour by the underlying scheduler [12] allows for an important improvement of the program quality and avoids the error-prone usage of explicit resumption, e.g., resuming a routine twice in Scala [11] raises an exception.

In this paper we investigate the expressive power of cooperative scheduling in ABS. We show that the powerful abstraction of cooperative scheduling in ABS can in fact be modeled by a run to completion model GAC (Guarded ACtor) of active objects which features a guarded-command language ([4]) for the description of the method bodies. The formal translation of ABS into GAC is given by an intermediate language ABS-SPAWN which uses an explicit spawn operation to model the execution of await statements. In the GAC language the operation of spawning local processes can be modeled directly by asynchronous self-calls.

Plan of the paper. In the next section we first introduce informally the ABS language, and illustrate the use of cooperative scheduling by an example. In Section 3 we then discuss a new operational semantics of the ABS language, which is particularly suited for proving correctness of the translation. The semantics of the operation of spawning local processes is described in Section 4, and a formal translation is given of ABS which shows how to model cooperative scheduling by spawning local processes. In Section 5 we finally introduce the guarded command language for active objects which allows to model the spawning of local processes directly by asynchronous self calls.

2 ABSTRACT BEHAVIOURAL SPECIFICATION LANGUAGE (ABS)

This section describes the main features of the ABS language: asynchronous communication together with fine-grained suspension and resumption of the control flow in a method.

SAC'20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373896

We first introduce these main features by an example. Listing 1 models a worker pool which allows the parallel execution of asynchronous tasks. A `WorkerPool` actor contains a set of `Worker` references (line 3) that is used to call methods asynchronously as defined in the latter class. The method `sendWork` is defined with an immediate suspension point based on a Boolean condition that the set of workers is not empty (line 6). The suspension mechanism underlying the await statement allows to schedule any update of the set of worker actors by a call of the method `finished`. Such an update will then allow the scheduling of the execution of the rest of the block in `sendWork`.

Listing 1: The Worker Pool Actor/Class

```
1  class WorkerPool(){
2      // initialization omitted for brevity
3      Set<Worker> workers;
4
5      Result sendWork() {
6          await !(emptySet(workers));
7          Worker w = take(workers);
8          workers = remove(workers, w);
9          Fut<Result> f = w ! doWork();
10         await f?;
11         Result result = f.get;
12         return result;
13     }
14
15     Unit finished(Worker w) {
16         workers = insertElement(workers, w);
17     }
18 }
```

As such, an asynchronous invocation of the method `sendWork` would suspend if the Boolean condition is not met, and would implicitly resume once the state of the worker set is satisfied. This implicit behaviour is defined in the operational semantics of ABS (detailed in Section 3). Furthermore it would execute in the same context (thread) as it initially started, adhering to the actor semantics. Upon resumption, the rest of the control flow is followed where a reference of type `Worker` is selected from the set and is used to asynchronously call the `doWork` method (line 9). A unique future is assigned to this call and stores the completion status and result value of this call. To control the state of the worker set, the `WorkerPool` actor defines a method `finished` that adds an available `Worker w`, passed as an argument, back into the set (lines 15 and 16). Note that due to the imposed actor semantics, the access to the worker set is thread-safe, as the field may not be accessed outside the `WorkerPool`. Inside the actor there may only be at most one method running on its associated thread.

Listing 2: The Worker Actor/Class

```
1  class Worker(WorkerPool p) implements Worker{
2
3      Result doWork(){
4          Result r;
5          // computation
6          p ! finished(this);
7          return r;
8      }
9  }
```

Listing 2 contains the `Worker` class. The `Worker` actor is defined with an instance variable that references its associated worker pool `p` (line 1). The method `doWork` sketches a method that performs a certain computation and returns its result (line 7). The return instruction is preceded by an asynchronous call to the method `finished` of `p` (line 6).

Getting back to `WorkerPool` class in Listing 1, it first uses a statement which awaits on a Boolean condition. The await statement also has a second form, **await** `f?` (line 10) which suspends the executing method invocation and resumes it based on the completion status of the future `f`. The method can then be rescheduled when the control flow corresponding to `f` has computed the return value. In contrast to that, futures in ABS can also part of an expression `f.get` (line 12 in Listing 1) that blocks all the method invocations of an actor until the return value has been computed. In particular, the statement on line 12 can never be blocking as the preceding await always ensures `f` is complete.

The await construct for futures can also be used in a high-level abbreviation as on line 1 of Listing 3. This "one-line" construct suspends execution of an asynchronous invocation and assigns its result once the generated future has completed and computed return value. It is a sugar syntax for lines 5-7.

Listing 3: ABS Await sugared syntax

```
1  Result result = await w ! doWork();
2
3  //can be expanded to
4
5  Fut<Result> f = w ! doWork();
6  await f?;
7  Result result = f.get;
```

For technical convenience, in this paper we assume that all communication between actors is done asynchronously. Only synchronous self calls are allowed. For example, the functionality of obtaining a worker can be isolated in a separate method `getWorker` defined in the same class like in Listing 4. The actor can then make a synchronous self call to this method like in line 7. This type of control flow where a synchronous self call is suspended gives rise to suspension of an entire call stack. In our example, suspension that results from the await statement in line 2 creates a call stack which consists of a top frame that holds the suspended synchronous self call followed by the bottom frame which is the continuation of the asynchronous method invocation of `sendWork`(the block of code that starts on line 8 after the self call returns).

Listing 4: Synchronous Call in ABS

```
1  Worker getWorker(){
2      await !(emptySet(workers));
3      Worker w = take(workers);
4  }
5
```

```
6   Result sendWork() {
7       Worker w = this.getWorker();
8       workers = remove(workers, w);
9       Fut<Result> f = w ! doWork();
10      await f?;
11      Result result = f.get;
12      return f;
13  }
```

In contrast to multi-threading in Java, ABS imposes that such call stacks are not interleaved and executed in any random order. Within an actor only one call stack can execute and it runs until it is either completed or suspended by an await instruction.

An ABS program consists of a set of classes, and each class consists of a set of method definitions. Each method body is assumed to end with a **return** statement even if the result type is void. In this paper we abstract from the nominal type system of ABS and its functional layer, and focus on the control flow of ABS programs. Figure 1 presents the formal syntax of ABS statements which are used to describe the method bodies. The expression e denotes a local side-effect free expression (that is, its evaluation only depends on the local state of the actor and does not affect this local state). For the purpose of this paper we can abstract from its syntax (which in general involves the functional layer of ABS). For notational convenience we assume that every method call (asynchronous or synchronous self call) returns a value. We assume these values typed according to the type system of ABS.

Further we restrict a guard g of an await statement by either a local side-effect free Boolean condition b or a single a future variable. It is not difficult to see that this restriction does not restrict the expressive power since any await statement on a guard which consists of a Boolean condition and a set $\{y_1, \ldots, y_n\}$ of futures can be implemented by a sequential composition:

```
1   await y1?; ...; await yn?; await b;
```

because a future is single-write shared data (note that the await on the Boolean condition should indeed be executed last).

S	::=	ϵ	empty statement
	|	$x = e$	basic assignment
	|	$y = x!m(\bar{e})$	asynchronous method call
	|	$y = m(\bar{e})$	synchronous method call
	|	$x =$ **new** $C(\bar{e})$	object creation
	|	**await** g	await statement
	|	$x = y$.**get**	get statement
	|	**if** b $\{S\}$ **else** $\{S\}$	conditional statement
	|	$S; S$	sequential composition
	|	**return** e	return statement

Figure 1: Syntax for ABS statements.

For technical convenience, we also abstract from the so-called Concurrent Object Groups (COG) as provided by the ABS language. However, it is not difficult to generalize the main result of this paper to the language including COG's. More importantly, it should be noted that the syntax does not include the usual **while** statement. A detailed discussion of the challenges of translating await statements occurring in the body of a while statement will be presented in section 4. Note however that the while statement can be modeled by tail recursion, using synchronous self calls.

3 ABS OPERATIONAL SEMANTICS

This section presents a different approach to the semantics of the ABS language using variable renaming of local variables instead of local environments. This allows for a simple definition of a process as the statement to be executed, which in turn allows for a transparent way of modeling cooperative scheduling. In the ABS language values include values of the primitive built-in types, references to object identities and identities of futures.

We assume given an ABS program P where object configurations are of the form (σ, S, Q):

- σ assigns values to the instance variables (fields) of the class (we treat the keyword **this** as a distinguished instance variable identifying the object) and all the fresh variables generated for the local variables of the different method invocations. For any side-effect free expression e (including Boolean conditions b) we denote by $\sigma(e)$ the value of e in σ.
- S represents the current statement of the active process that is run by the actor denoted by σ(**this**).
- Q is a set of statements which represent suspended processes.

We define a global configuration G as a pair (F, O) where F is a partial function which assigns to each future identity f in its domain a value $F(f)$ and O is a set of configurations (as defined above). By $F(f) = \bot$ we denote that the future f has not been completed yet. For handling the completion of futures by return statements, we introduce an implicit formal parameter **dest** which holds the value returned by the method invocation. In the rules below we assume some mechanism for generating fresh variables. Generating fresh variables is needed to distinguish between the **dest** variable of each method and also to avoid name clashes when renaming local variables. To make the use of the **dest** variable explicit, we replace every **return** statement in a method body with the auxiliary statement **return** e *to* **dest**.

The following rule describes the operation semantics of an assignment.

Assignment Rule.

$$(F, \{(\sigma, x = e; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = \sigma(e)], S, Q)\} \cup O)$$

Here and in the sequel we denote by $\sigma[x = v]$ the update of σ which assigns the value v to the variable x.

Asynchronous Invocation Rule. The following rule describes the semantics of an asynchronous method call.

$$(F, \{(\sigma, y = x!m(\bar{e}); S, Q), (\sigma', S', Q')\} \cup O) \rightarrow (F[f = \bot], \{(\sigma[y = f], S, Q), (\sigma'', S', Q'')\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \bot]$ denotes the function which results from extending the domain of F by assigning $\bot$ to f)
- $\sigma(x) = \sigma'$(**this**).
- S' is the current statement of the active process run by the target object x (the callee in $x!m(\bar{e})$)

- Q'' extends Q' with the body of method m where all the formal parameters (including the distinguished variable **dest**) are replaced by fresh (that is, not in use in (σ', S', Q')) variables.
- σ'' results from assigning the values of the actual parameters $\sigma(\bar{e})$ to the corresponding fresh local variables. Additionally $\sigma''(\mathbf{dest'}) = f$ where **dest'** is the fresh local variable corresponding to the destiny variable **dest**.

Synchronous Self Call Rule.

$$(F, \{(\sigma, x = m(\bar{e}); S, Q)\} \cup O) \rightarrow$$

$$(F[f = \bot], \{(\sigma', S'; x = f.\mathbf{get}; S, Q)\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \bot]$ denotes the function which results from extending the domain of F by assigning $\bot$ to f).
- f is the future that will hold the result of the asynchronous method invocation m.
- S' is obtained by renaming the local variables in the body of method m (as above, including the variable **dest**) by fresh variables and σ' assigns to these fresh variables the values of the actual parameters $\sigma(\bar{e})$. Additionally $\sigma'(\mathbf{dest'}) = f$ where **dest'** is the fresh local variable corresponding to the destiny variable **dest**. This translation of the body replaces the return statement with an auxiliary statement **return** *e to* **dest**. Freshness is defined as a variable not in use by any statement in S' and Q' .

Note that we thus use simple inlining which works because we introduce fresh variables for the formal parameters of methods. We use a future in order to get a uniform semantics for returning a value for both synchronous calls and asynchronous calls. The get operation will always be enabled (because of the assumption that any method body will end with a return statement), but it is used here instead of an await because we want the process to proceed, as otherwise we would have a release point which breaks the call stack.

In the case of a method which is declared void then the syntax would be **return null** *to* **dest**. By means of this convention, every suspended statement is uniquely identified by its destiny variable,so that we can model Q as a set..

Object Instantiation Rule.

$$(F, \{(\sigma, x = \mathbf{new}\ C(\bar{e}); S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma[x = o], S, Q), (\sigma', S', \emptyset)\} \cup O)$$

where:

- o is a fresh object identity (not appearing as value of a variable in the initial configuration).
- S' is the constructor method body.
- $\sigma'(this) = o$ and σ' assigns to the formal parameters of the constructor method the values $\sigma(\bar{e})$.

Note that each object configuration assigns a new object identity to the instance variable *this*. This explains the usage of union in object configurations.

Conditional Statement Rule. The conditional statement has the following two rules.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$
$$(F, \{(\sigma, S_1; S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$
$$(F, \{(\sigma, S_2; S, Q)\} \cup O)$$

where $\sigma(b) = false$.

Return Rule.

$(F, \{\sigma, \mathbf{return}\ e\ to\ \mathbf{dest'}, Q)\} \cup O) \rightarrow (F[f = \sigma(e)], \{(\sigma, \epsilon, Q)\} \cup O)$

where $f = \sigma(\mathbf{dest'})$

Get Rule.

$(F, \{(\sigma, x = y.\mathbf{get}; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = F(\sigma(y))], S, Q)\} \cup O)$

where $F(\sigma(y)) \neq \bot$.

Await Rule.

$(F, \{(\sigma, \mathbf{await}\ g; S, Q)\} \cup O) \rightarrow (F, \{(\sigma, \epsilon, \{\mathbf{await}\ g; S\} \cup Q)\} \cup O)$

where ϵ represents the empty statement, denoting that the current executing statement has ended. This rule "blindly" suspends the current statement without evaluating the guard. The evaluation of the guards will be performed in the context of the scheduling rule below.

Scheduling Rule. The following rules schedule enabled **await** statements of a Boolean and a future variables respectively. For suspended statements that start with an **await** we have the following two rules.

$$(F, \{(\sigma, \epsilon, \{\mathbf{await}\ b; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$

$$(F, \{(\sigma, \epsilon, \{\mathbf{await}\ y; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable such that $F(\sigma(y)) \neq \bot$.

For any other suspended statement that is in Q, e.g., that resulted from an asynchronous call, we have the following rule:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

4 ABS-SPAWN

In this section we introduce the ABS-SPAWN language which is obtained from the ABS language discussed above by replacing the await statement with a statement **spawn**(g, S), the so-called **spawn** statement, for spawning a new local process that executes the statement S. In ABS-SPAWN method invocations are thus executed in a run-to-completion mode.

For the operational semantics of the ABS-SPAWN language we introduce the run-time syntax construct $(g \rightarrow S)$ that represents a suspended statement (S) that is guarded by an enabling condition (g). We thus make a distinction between the statement that spawns a process and resulting generated suspended process. The guard in $(g \rightarrow S)$ is the enabling condition for scheduling the corresponding statement S for execution. For its semantics we used the same

notions for a global configuration and object configuration, as introduced for the semantics of the ABS language, The semantics of the ABS-SPAWN language results from the semantics of ABS by replacing the **Await Rule** with the rule **Spawning Subtasks** and changing the **Scheduling Rule**, as described above.

Spawning Subtasks. Spawning a sub-task simply consists of adding a corresponding statement with an enabling condition to the set Q of suspended processes:

$$(F, \{(\sigma, \textbf{spawn}(g, S); S', Q)\} \cup O) \rightarrow (F, \{(\sigma, S', \{(g \rightarrow S)\} \cup Q)\} \cup O)$$

Scheduling Rule. The following rules describe the scheduling of an enabled suspended task. The first two rules are for statement suspended by **await**.

$$(F, \{(\sigma, \epsilon, \{(b \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \epsilon, \{(y \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable and $F(\sigma(y)) \neq \perp$.

The last rule is for statements that are suspended as a result of an asynchronous invocation and is the same as in the ABS operational semantics:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

Translating ABS into ABS-SPAWN. We next introduce a formal translation from ABS programs into ABS-SPAWN programs. This translation is applied to every class in the ABS program. For each class, every method body is viewed as a sequential composition of the first instruction followed by its (sequential) continuation and translated accordingly.

$$\begin{aligned}
T(\epsilon) &:= \epsilon \\
T(x = e;\ S) &:= x = e; T(S) \\
T(\textbf{await}\ g;\ S) &:= \textbf{spawn}(g,\ T(S)\ ) \\
T(\textbf{if}\ b\ \{S_1\}\ \textbf{else}\ \{S_2\}; S) &:= \textbf{if}\ b\ \{T(S_1;\ S)\}\ \textbf{else}\ \{T(S_2;\ S)\} \\
T(y\ =\ x!m(\bar{e});\ S) &:= y = x!m(\bar{e});\ T(S) \\
T(y\ =\ m(\bar{e});\ S) &:= y\ =\ m(\bar{e});\ T(S) \\
T(x\ =\ \textbf{new}\ C(\bar{e}); S) &:= x = \textbf{new}\ C(\bar{e});\ T(S) \\
T(x = y.\textbf{get};\ S) &:= x = y.\textbf{get};\ T(S) \\
T(\textbf{return}\ e;\ S) &:= \textbf{return}\ e;\ T(S)
\end{aligned}$$

Figure 2: Translation of ABS into ABS-SPAWN

The scheme is applied using a bottom-up approach starting at the level of statements using Figure 2. The scheme is then lifted to the level of method bodies. Finally a translation of a class simply consists of the translation of its method definitions.

In Figure 2 the empty statement is denoted by ϵ (we assume here the syntactical equivalence $S; \epsilon \equiv S$). The translation of an await construct with guard g followed by a (sequential) continuation S results simply in a spawn statement with two parameters: the guard g and the task representing the translation applied to the continuation ($T(S)$). A conditional statement is translated by "absorbing" the sequential continuation that follows into the two branches of the statement. This general pattern also would apply to, for example, the translation of the ABS case statement (or pattern matching statement) where the continuation has to capture for each possible pattern (P_i) both the block to be executed on that pattern branch (S_i) as well as the rest of the control flow that follows the statement (S). The translation thus captures the whole syntactic continuation that follows an await statement as the new task to be spawned. Therefore the translation of the method containing the await statement will terminate directly after having spawned the corresponding subtask, thus emulating an implicit suspension point.

The While Statement. We describe next the problem of translating a repetitive loop or the while statement. Intuitively, to capture the syntactic continuation that follows an await statement occurring in the body of the while statement, the translation could simply "unfold" the loop. However this would result in a recursive translation. Instead, we can model while statements by means of a tail recursive method. Note that such a method should capture in its formal parameters the execution context (that is, all the local variables used in the loop body).

Listing 5: While Loop in ABS

```
1  { List<Fut<Int>> futuresList = Nil;
2
3      //ABS code that fills the futuresList with
4      //futures resulting from asynchronous calls
5
6      this.sum=0;
7      while( !emptyList( futuresList ) ){
8          Fut<Int> f = head( futuresList );
9          await f?;
10         Int x = f.get;
11         this.sum = this.sum + x;
12         futuresList = tail( futuresList );
13     }
14     if( this.sum > 0 ){
15         //do work
16     }
17 }
```

To describe this in more detail, we look at an example in Listing 5 that computes the sum of numbers generated by asynchronous calls whose results are captured in a list of futures. In ABS, lists are part of the functional layer and all functions applied on them (`head, tail, emptyList`) are side-effect free. We note that in this particular program the variables `x, f` are local variables declared inside the repetitive loop, `futuresList` is a local variable defined in the method's body prior to the loop scope and `sum` is a class member variable.

Listing 6: Re-written While Loop in ABS using tail recursion

```
1  //new method
2  Unit m(List<Fut<Int>> futuresList){
```

```
3       if(!emptyList(futuresList)){
4           Fut<Int> f = head(futuresList);
5           await f?;
6           Int x = f.get;
7           this.sum = this.sum + x;
8           futuresList = tail(futuresList);
9           m(futuresList);
10      }
11  }
12
13  { //original method scope
14      this.sum=0;
15      m(futuresList);
16      if(this.sum > 0){
17          //do work
18      }
19  }
```

This repetitive loop can naturally be “unfolded" by defining a new method `m` with a formal parameter of type `List< Fut<Int>>`, as we observe it is the only local variable declared prior to the loop. This is shown in Listing 6. We can see that this way of “unfolding"" the loop works because the state of execution (in this case, the continuously processed list) is passed to the next call as formal parameters. Listing 7 then shows the translation of tail-recursive method modeling the while statement. Note that the syntactic continuation of the await statement is captured in this translation by the recursive call.

Listing 7: Translation Tail Recursion

```
1   Unit m(List<Fut<Int>> futuresList){
2       if(!emptyList(futuresList)){
3           Fut<Int> f = head(futuresList);
4           spawn( f?, {
5               Int x = f.get;
6               this.sum = this.sum + x;
7               futuresList = tail(futuresList);
8               m(futuresList)
9           } );
10      }
11  }
```

To conclude the presentation of ABS-SPAWN, we apply the translation scheme to the `WorkerPool` class written in ABS in Listing 1. The resulting code in ABS-SPAWN is illustrated in Listing 8.

Listing 8: The Worker Pool Class in ABS-SPAWN

```
1   class WorkerPool(){
2       Set<Worker> workers;
3
4       Result sendWork() {
5         spawn ( !( emptySet(workers) ) , {
6               Worker w = take(workers);
7               workers = remove(workers, w);
8               Fut<Result> f = w ! doWork();
9               spawn (f?, {
10                  Result result = f.get;
11                  return result;
12              } ) ;
13        } );
14      }
15
16      Unit finished(Worker w) {
17          workers = insertElement(workers, w);
18      }
19  }
```

Correctness of the ABS Translation. In order to show the correctness of the above translation of ABS programs into ABS-SPAWN programs, we use G to denote a global ABS configuration as well as ABS-SPAWN configurations. We introduce the notation:

$$G \rightarrow_{\text{abs}} G'$$

to differentiate between transitions in pure ABS and transitions in ABS-SPAWN which are denoted as:

$$G \rightarrow_{\text{abs-spawn}} G'$$

Let $T(G)$, for any global ABS configuration G, denote the result of applying the translation to all the *executing* ABS statements in G and translating any *suspended* **await** statement **await** g; S in G by $g \rightarrow T(S)$. We now can state the following theorem which states the correctness of the translation of **await** statements in ABS, the proof of which proceeds by a straightforward case analysis of the first instruction of an executing statement.

Theorem 4.1. *For any configuration G of an ABS program we have:*

$$G \rightarrow_{abs} G' \text{ iff } T(G) \rightarrow_{abs-spawn} T(G')$$

Proof. The proof proceeds by a case analysis of the transition rules. We treat the following main cases. We only need consider those statements that are affected by the translation, because for statements like the *assignment* and *empty* statement, the semantics of ABS coincides with that of ABS-SPAWN. We only consider the main case of translating the **await** statement, because the translation of the conditional statement is correct because of standard programming equivalences.

The proof is divided into two parts. The first part is presented by the diagram in Figure 3 and treats the translation of the **await** statement that appears as an instruction in a context Σ, that is, $\Sigma[(\sigma, S, Q)]$ describes a global configuration (F, O), with $(\sigma, S, Q) \in O$. The upper transition corresponds to the application of the **Await Rule**, the result of which, namely that the process **await** g; S is added to the suspended processes Q (of the executing active object), is denoted by the corresponding global configuration.

$$\Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)]$$

The lower transition results from the definition of the translation scheme to global configurations, and a corresponding application of the **Spawning Tasks** rule in ABS-SPAWN.

Conversely, the second part is presented by the diagram in Figure 4 and shows the correctness of translating the **await** statement

Figure 3: Execution of an Await Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \textbf{await } g;\ S, Q)] & \xrightarrow[\text{abs}]{} & \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \textbf{spawn}(\ g;\ T(S)), T(Q))] & \xrightarrow[\text{abs-spawn}]{} & T(\Sigma)[(\sigma, \{g \to T(S)\} \uplus T(Q))] \end{array}$$

Figure 4: Scheduling a Suspended Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] & \xrightarrow[\text{abs}]{} & \Sigma[(\sigma, S, Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \epsilon, \{g \to T(S)\} \uplus T(Q))] & \xrightarrow[\text{abs-spawn}]{} & T(\Sigma)[(\sigma, T(S), T(Q))] \end{array}$$

as part of a suspended process which conforms to the semantics of the **Scheduling Rules** in ABS and ABS-SPAWN, respectively.

5 THE GAC LANGUAGE

It is worthwhile to note that by the above translation scheme we can actually embed ABS in a language *without* any await statements (that allow for cooperative scheduling) by encoding $\textbf{spawn}(g, S)$ itself as an asynchronous self call of the form $\textbf{this}!m()$, where $m()$ is an unique method name with defining body $g \to S$.

In Figure 5 we introduce so-called guarded command statements (following [4]) as statements for describing the method bodies in ABS.

S	::=	ϵ	empty statement
	|	$x = e$	basic assignment
	|	$y = x!m(\bar{e})$	asynchronous method call
	|	$y = m(\bar{e})$	synchronous method call
	|	$x = \textbf{new } C(\bar{e})$	object creation
	|	$x = y.\textbf{get}$	get statement
	|	$(*)\Box_{i=1}^{n}\ g_i \to \{S_i\}$	guarded command
	|	$\textbf{case } e\ \{e \Rightarrow S\}$	case statement
	|	$S; S$	sequential composition
	|	$\textbf{return } e$	return statement

Figure 5: ABS guarded command statements.

The semantics of the statement $\Box_{i=1}^{n}\ g_i \to S_i$ consists of a non-deterministic selection of one of the statements S_i for which the associated guard g_i is enabled. It blocks the execution of the active object if none of the guards are enabled. A guard itself in the GAC language consists of a Boolean condition and a set of futures. Such a guard is enabled if the Boolean condition holds and all its futures are completed (that is, for all of them the return value has been produced). Its iterated version (indicated by the asterisk) consists of repeatedly executing the marked guarded choice as long as one of its guards is enabled. It terminates as soon as none of the guards is enabled. Formally, the semantics of the guarded command statements is described by the following rules (the semantics of the other statements are described as in the ABS semantics).

Guarded Choice Rule.

$$(F, \{(\sigma, \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

For its iterated version we have the following two transitions.

Iterated guarded Choice Rule.

$$(F, \{(\sigma, *\Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to$$
$$(F, \{(\sigma, S_j; *\Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ.

$$(F, \{(\sigma, *\Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to (F, \{(\sigma, S, Q)\} \cup O)$$

provided none of the g_j is enabled in σ and F.

Further, we have the following scheduling rules.

Scheduling Rules.

$$(F, \{(\sigma, \epsilon, \{\Box_{i=1}^{n}\ g_i \to \{S_i\}; S\} \cup Q)\} \cup O) \to (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

$$(F, \{(\sigma, \epsilon, \{*\Box_{i=1}^{n}\ g_i \to \{S_i\}; S\} \cup Q)\} \cup O) \to$$
$$(F, \{(\sigma, S_j; *\Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

The resulting GAC language thus follows a strict run to completion mode of execution of the methods by active objects, like the Rebeca language)[10]). Differently from the Rebeca language, it features guarded command statements which allow to associate an enabling condition with a suspended process. Note that such a suspension mechanism avoids modeling a suspended process $g \to S$ by a recursive method definition

$$m()\{\textbf{if } g\ \{S\}\ \textbf{else}\ \{\textbf{this}!m()\}\}$$

which involves busy waiting (and which assumes testing a future as a Boolean condition).

As an overall conclusion we illustrate in Figure 9 the translation of the `WorkerPool` into GAC. Note the need to wrap the statements of the guarded commands into separate methods. Thus these methods can be called asynchronously and stored as suspended messages into the queue of the `WorkerPool` until their guards are enabled. As such, execution of other enabled statements can continue without blocking the actor.

Listing 9: The Worker Pool Class in GAC

```
1  class WorkerPool(){
2      Set<Worker> workers;
3
4      Result sendWork() {
5        ! emptySet(workers) → this ! m1( workers );
6      }
7
8      Result m1(Set<Worker> workers){
9          Worker w = take(workers);
10         workers = remove(workers, w);
11         Fut<Result> f = w ! doWork();
12         f → this ! m2( f );
13     }
14
15     Result m2(Fut<Result> f){
16         Result result = f.get;
17         return result;
18     }
19
20     Unit finished(Worker w) {
21         workers = insertElement(workers, w);
22     }
23 }
```

6 CONCLUSION

This work arose out of the work [9] that involves implementing the ABS language in Java and Scala. In fact, the main Theorem 4.1 of this paper, which states the correctness of the translation of ABS into the ABS-SPAWN language, provides a main step in the proof of the correctness of the compiler that translates ABS into Java.

Quoting Felleisen in [5], there exists an abundance of informal claims on the relative expressive power of programming languages. In this paper however, we investigated in a formal way, that is, based on a formal operational semantics, the expressive power of cooperative scheduling as supported by the await statement of the ABS language.

The proposal of ABS-Spawn is used in the Java and Scala implementation and provides a workaround to suspending threads. The entire Java implementation [8] is based on replacing synchronous calls with asynchronous calls followed by an await (suspension) on the created implicit future. The implementation also includes an underlying scheduler which differentiates priorities between explicit futures created by the program and implicit futures [6] created by the replacement mechanism of synchronous calls.

We introduced the ABS-SPAWN language and its formal semantics which, instead of the await statement, features a statement for spawning local processes while maintaining a run to completion mode of execution of the methods, and provided a formal translation T which translates every ABS program P into an ABS-SPAWN program $T(P)$ such that $T(P)$ simulates P, and vice versa.

We further introduced the GAC language which features guarded command statements [4] as statements for describing the method bodies in ABS. The standard semantics of guarded statements is extended in ABS to their semantics as suspended processes. This allows to model the spawn statement of the ABS-SPAWN language itself directly by an asynchronous self-call, wrapping the corresponding guarded command in a method.

In [3], the expressive power of general Actor-based systems has been studied. Of interest is to extend that research into an investigation of the expressive power of the other main feature of the execution model underlying the ABS language, namely futures. Of particular interest is to investigate the expressive power of the non-blocking test operation on futures in the guards of the GAC language, along the lines of the seminal work of [2], which provides an in-depth study the expressive power of the the coordination primitives of the Linda language (asynchronous communication via a shared data space, read operation, non-blocking test operators on the shared space).

REFERENCES

[1] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5, Article 76 (Oct. 2017), 39 pages. https://doi.org/10.1145/3122848

[2] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. 2000. On the Expressiveness of Linda Coordination Primitives. *Inf. Comput.* 156, 1-2 (2000), 90–121. https://doi.org/10.1006/inco.1999.2823

[3] Frank S. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. 2014. Decidability Problems for Actor Systems. *Logical Methods in Computer Science* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:5)2014

[4] Edsger W Dijkstra. 1978. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*. Springer, 166–175.

[5] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35 – 75.

[6] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. 2019. Godot: All the Benefits of Implicit and Explicit Futures.

[7] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*. Springer, 142–164.

[8] Vlad Serbanescu, Frank S. de Boer, and Mohammad Mahdi Jaghoori. 2018. Actors with Coroutine Support in Java. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings*. 237–255.

[9] Vlad Serbanescu, Frank S. de Boer, and Mohammad Mahdi Jaghoori. 2018. ASCOOP: Actors in Scala with Cooperative Scheduling. In *2018 IEEE International Conference on Computational Science and Engineering, CSE 2018, Bucharest, Romania, October 29-31, 2018*. 19–28.

[10] Marjan Sirjani. 2007. Rebeca: Theory, Applications, and Tools. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–126.

[11] Enroute Storm. Accessed: 2019-09-26. Scala Coroutines. http://storm-enroute.com/coroutines/.

[12] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. 2012. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer* 14, 5 (01 Oct 2012), 567–588. https://doi.org/10.1007/s10009-012-0250-1