Multi-active Objects and their Applications (extended version).

Ludovic Henrio, Justine Rochas

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS
UMR 7271

Multi-active Objects and their Applications (extended version).

Ludovic Henrio, Justine Rochas
EQUIPE COMRED

Rapport de Recherche

November 2017

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) - UMR7271 - UNS CNRS
2000, route des Lucioles — Les Algorithmes - bât. Euclide B — 06900 Sophia Antipolis — France
http ://www.i3s.unice.fr

Membre de UNIVERSITÉ CÔTE D'AZUR

Multi-active Objects and their Applications (extended version).[1]

Ludovic Henrio[2], Justine Rochas[3]

EQUIPE COMRED
November 2017 - 63 pages

Abstract : In order to tackle the development of concurrent and distributed systems, the active object programming model provides a high-level abstraction to program concurrent behaviours. There exists already a variety of active object frameworks targeted at a large range of application domains : modelling, verification, efficient execution. However, among these frameworks, very few consider a multi-threaded execution of active objects. Introducing *controlled* parallelism within active objects enables overcoming some of their limitations. In this paper, we present a complete framework around the multi-active object programming model. We present it through ProActive, the Java library that offers multi-active objects, and through MultiASP, the programming language that allows the formalisation of our developments. We then show how to compile an active object language with cooperative multi-threading into multi-active objects. This paper also presents different use cases and the development support to illustrate the practical usability of our language. Formalisation of our work provides the programmer with guarantees on the behaviour of the multi-active object programming model and of the compiler.

Key-words : Programming languages, distributed systems, active objects.

1. This article is an extended version of an article published in Logical Methods in Computer Science.
2. Laboratoire I3S – CNRS – ludovic.henrio@cnrs.fr
3. Laboratoire I3S – Université Côte d'Azur – justine.rochas@unice.fr

MULTI-ACTIVE OBJECTS AND THEIR APPLICATIONS

LUDOVIC HENRIO AND JUSTINE ROCHAS

Université Côte dAzur, CNRS, I3S, France
e-mail address: ludovic.henrio@cnrs.fr

Université Côte dAzur, CNRS, I3S, France
e-mail address: justine.rochas@unice.fr

Abstract. In order to tackle the development of concurrent and distributed systems, the active object programming model provides a high-level abstraction to program concurrent behaviours. There exists already a variety of active object frameworks targeted at a large range of application domains: modelling, verification, efficient execution. However, among these frameworks, very few consider a multi-threaded execution of active objects. Introducing *controlled* parallelism within active objects enables overcoming some of their limitations. In this paper, we present a complete framework around the multi-active object programming model. We present it through ProActive, the Java library that offers multi-active objects, and through MultiASP, the programming language that allows the formalisation of our developments. We then show how to compile an active object language with cooperative multi-threading into multi-active objects. This paper also presents different use cases and the development support to illustrate the practical usability of our language. Formalisation of our work provides the programmer with guarantees on the behaviour of the multi-active object programming model and of the compiler.

1. Introduction

Systems and applications nowadays run in a concurrent and distributed manner. In general, existing programming models and languages lack adequate abstractions for handling the concurrency of parallel programs and for writing distributed applications. Two well-known categories of concurrency bugs are associated with synchronisation. On the one hand, deadlocks appear when there is a circular dependency between tasks. On the other hand, data races correspond to the fact that several threads access a shared resource without proper synchronisation. In object-oriented programming, allowing several threads of control to execute methods breaks the encapsulation property of objects. Industrial programming languages generally do not enforce by default a safe parallel execution nor offer a programmer-friendly way to program concurrent applications. In Actors and active object languages, an application is designed as a set of independent entities only communicating

1998 ACM Subject Classification:
D.3.2. Programming languages. Language Classifications. Concurrent, distributed, and parallel languages; D.3.3. Programming languages. Language Constructs and Features. Concurrent programming structures.
Key words and phrases: Programming languages, distributed systems, active objects.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

by passing messages. This approach makes the programming of distributed and concurrent systems easier.

The active object programming model [9] reconciles object-oriented programming and concurrency. Each active object has its own local memory and cannot access the local memory of other active objects. Objects can only exchange information through message passing, implemented with requests sent to active objects. This characteristic makes object accesses easier to trace, and thus it is easier to check their safety. Also, the absence of a shared memory between active objects makes them well-adapted to a distributed execution. Section 2 will give an overview of active-object languages.

Active object models however have limitations in terms of efficiency on multicore environments and of deadlocks. Concerning deadlocks, one solution is to allow the currently executing thread to be released and allow the current active object to handle another request. This approach is called cooperative multi-threading and is used in several languages like for example ABS (abstract behavioral specification) language [35].

However cooperative multi-threading relies on the programmer's expertise to place release points which is not always realistic. Additionally, cooperative multi-threading does not solve the problem of the inefficiency in distributed architectures made of several multicore machines. This leads us to the design of a new model based on active objects: multi-active objects [30]. This model enables local multi-threading inside an active object. In active object models, an activity is the unit of composition of the model: an activity is a set of entities evolving independently and asynchronously from other objects. In classical active-object models, an activity is a set of objects associated with a request queue and a single thread. In the multi-active object an activity is a single active object equipped with a request queue, it may serve one or several requests in parallel. This programming model relies on the notion of *compatibility between requests*, where two compatible requests can safely be run in parallel. *Groups* of methods are introduced to easily express compatibility between requests: two requests can run in parallel if they target methods belonging to two compatible groups. MultiASP [30], is a calculus that formalises the multi-active objects.

Concerning the definition of MultiASP, compared to [30], this article presents a new version of the operational semantics of the language, and includes an extension that deals with thread management; this article also presents a new debugging tool for multi-active objects. This article is a follow up of the article presented at Coordination 2016 [33]. Compared to [33], one contribution of this article is a full overview on the multi-active programming model including the different programming tools that we provide for this programming model. The strongest improvement featured by this article compared to [33] is the formalisation of the translation and the correctness of the translation from ABS programs into multi-active objects, and in particular the definition of the relationship between ABS configurations and their translation in MultiASP. A brief summary of the proof is presented in the body of the paper and the details of the proof can be found in the appendix. From a practical point of view, the multi-active object model is distributed as part of the ProActive Java library[1] that supports the distributed execution of active objects. The proof allows us to provide more precise conditions of applicability of the equivalence between ABS programs and their translation. We summarise below our main contributions.

- We introduce the multi-active object programming model, together with the library that supports it, ProActive, and with its formalisation language, MultiASP.

[1] `https://github.com/scale-proactive`

This language is also equipped with advanced features regarding the scheduling of requests: thread management and priority of execution. The operational semantics of MultiASP is presented in two parts: the base semantics of the calculus, and an extension dealing with thread management.

- We show the practical usability of the language in two ways. First, we illustrate the programming model with a case study based on a distributed peer-to-peer system. Second, we show a tool that displays the execution of multi-active objects and that helps in debugging and tuning multi-active object-based applications.
- We present an approach for encoding cooperative active objects into multi-active objects. We specify a backend translator and prove the equivalence between the ABS program and its translation in MultiASP[2].

Section 2 details the background of this paper, the active object programming model; it provides a comparison of active object languages and positions our contribution on multi-active objects. Section 3 introduces our multi-active object framework, its semantics, and the semantics of the thread management functionality. Section 4 presents the automatic translation of ABS programs into ProActive programs, shows the properties of the translation, and provides a discussion relating this work with the other ABS backends and other related works. Section 5 concludes this article.

2. Background: Active Objects

After a brief introduction to active objects and actors, this section provides an analysis of the characteristics that can be used to classify active-object languages. Then ASP, ABS, and Encore are presented in more details. These languages have been chosen for the following reasons: ASP is the language that we extend in our multi-active object model, ABS is the language for which we provide a backend in Section 4, and Encore features a few novel features including some controlled parallelism inside active objects. We refer to [9] for a precise description of other active object models including JCoBox, AmbientTalk, and Creol. The section concludes by a focus on the related works featuring some form of multi-threaded active objects.

2.1. Origins and Context.

The active object programming model, introduced in [38], has one global objective: facilitate the correct programming of concurrent entities. The active object paradigm derives from actors [1]. Actors are concurrent entities that communicate by posting messages to each other. Each actor has a mailbox and a thread that processes the messages of the mailbox. Only the processing of a message can have a side effect on the actor's state. Once a message is posted to an actor, the sender continues its execution, without knowing when the message will be processed. This way the interaction between actors is asynchronous, allowing the different entities to perform tasks in parallel.

Active objects are the object-oriented descendants of actors. Like an actor, each active object has an associated thread and we call *activity* a thread together with the objects that are accessible by this thread. Active objects communicate using asynchronous method invocations: when an object invokes a method of an active object, this creates a *request* that is posted in a mailbox (also called *request queue*) on the callee side. On the invoker

[2] The backend is available at: `https://bitbucket.org/justinerochas/absfrontend-to-proactive`

side, the execution continues while the request is being processed. On the callee side, the request is dropped in the request queue and waits until its turn comes to be executed.

Like in object-oriented programming, the invocations of methods on active objects can return a result. Since these invocations are asynchronous, their result cannot be known just after the invocation. To represent the expected result and to allow the invoker to continue its execution asynchronously, a placeholder is created for the result of the request. This place holder is called a *future*, which is a promise of response that will be later filled by the result of the request. A future is *resolved* when the value of the future is computed and available. Futures have seen their early days in MultiLisp [27] and ABCL/1 [46]. They have been formalised in ABCL/f [44], in [23], and, more recently, in a concurrent lambda calculus [39], and in the Creol language [17]. In summary, active objects and actors enforce decoupling of activities: each activity has its own memory space, manipulated by its own thread. This strict isolation of concurrent entities makes them locally safe and also suited to distributed systems.

2.2. **Design of Active Object Languages.** Existing implementations of the active object programming model make different design choices for four main characteristics that we have extracted and listed below, as answers to design questions.

How are objects associated to activities? We distinguish three different ways to map objects to threads/activities in active object languages:

Uniform object model: all objects are active objects with their own execution thread. All communications between them necessarily create requests. This model is simpler to formalise and reason about, but leads to scalability issues in practice.

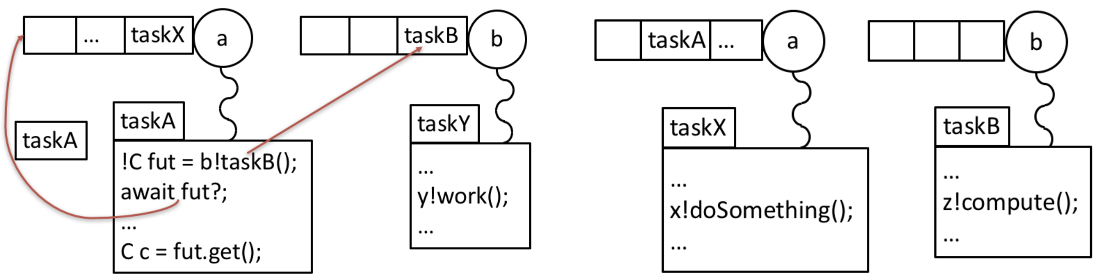
Creol [36, 17] and Rebeca [43] are uniform active object languages.

Non uniform object model: some objects are not active objects; they are called *passive* objects and are only accessible by one active object. This model is scalable as it requires less communication and less threads, but it is trickier to formalise and reason about because some of the objects are only locally reachable and an additional mechanism is necessary to transmit them between active objects. Reducing the number of activities also reduces the number of globally accessible references in the system, and thus enables the instantiation of a large number of objects. Non-uniform active objects reflect better the design of efficient distributed applications where many objects are created but only some of them are remotely accessible.

ASP [12], AmbientTalk [19, 15], and Joelle [14] are typical non-uniform active-object languages. Orleans [7] is a recent industrial active object language with a non-uniform active object model.

Object group model: an activity is made of a set of objects sharing an execution thread, but all objects can be invoked from another activity. This approach has a good scalability and propensity to formalisation, but the addressing of all objects in distributed settings is difficult to maintain because it requires a distributed referencing system able to handle a large number of objects.

ABS [35] and JCoBox [40] have concurrent object groups but JCoBox additionally allows data sharing for immutable objects. in JCoBox, the object groups are called coboxes, while they are called COGs (concurrent object groups) in ABS.



(A) Configuration 1. (B) Configuration 2.

FIGURE 1. Cooperative scheduling in Creol.

How are requests scheduled? We distinguish three request scheduling models:

Mono-threaded scheduling: within an active object, requests are executed sequentially without interleaving. This model is simple to reason about and has some strong properties but is the most prone to deadlock and potential inefficiency because one activity is blocked as soon as a synchronisation is required.

ASP [12], AmbientTalk [19, 15], and Rebeca [43] feature Mono-threaded scheduling but Rebeca and AmbientTalk perform no synchronisation and have no deadlock.

Cooperative scheduling: a running request can explicitly release the execution thread to let another request progress. Requests are not processed in parallel but they might interleave. While no data races are possible, the status of an object might be difficult to predict when a request is restored after a release point if the cooperatively scheduled requests modify the object state.

Creol [36, 17], JCoBox [40], and ABS [35] are typical examples of cooperative scheduling languages. Figure 1 illustrates cooperative scheduling, based on a Creol example. It shows two objects `a` and `b` each with a queue of awaiting tasks (rectangles). The currently executed task is shown below the object. In Figure 1a, object `a` does an asynchronous method invocation on object `b`, and then awaits for the result. As the future `fut` is not yet resolved when the `await` statement executes, this suspends the execution of the current task `TaskA` that returns to the pending task queue. At this point, another request for this object can start or resume; in the example a new active task is `TaskX` is started. Figure 1b shows a later configuration, after several steps of execution; `taskA` is still in the queue of object `a`, `taskX` is running. In object `b`, `TaskY` finished and the next task has started. Finally, when `TaskB` and `taskX` finish, `taskA` can resume and retrieve the value computed by `TaskB` with a blocking call to `.get()`, shown at the end of `taskA` in Figure 1a.

Multi-threaded scheduling: several threads are running in parallel in the same actor or active object. Either several requests can be served in parallel, or a data-level parallelism allows a request to be processed by several threads. In this context, an *activity* becomes a set of threads together with the objects they manipulate: there is one activity per active object or per object group. Consequently, data races are possible but only within an activity. An additional mechanism is necessary to control which threads can run in parallel. Parallelism provides efficiency at the expense of possible data-races. Somehow, similarly to cooperative multithreading, efficiency and expressiveness are gained at the expense of some possible incoherency in the object state. However, the two approaches offer different trade-offs: in cooperative multithreading incoherences can be limited by removing release points, while in multi-threaded scheduling incoherences can be limited by controlling which threads can run in parallel. On the other hand, multithreading enables parallelism internally to an active object which is more efficient in some cases like in a multicore setting.

MultiASP and parallel actor monitors [41] feature multi-threaded scheduling where several requests can be served in parallel by the same active object. Multi-threaded actors [5] feature multi-threading, but with each thread hosted by a different active-object (an actor contains several active objects in the terminology of [5]). Section 2.6 discusses in more details multithreading in active objects and actors.

How much is the programmer aware of asynchronous aspects? Some active object languages use a specific syntax for asynchronous method calls and a specific type for futures. This makes the programmer aware of where synchronisation occurs in the program. When the asynchronous invocation is explicit, there often exists a special type for future objects, and an operator to access the future's value. On the contrary, asynchronous method calls and futures can also be implicit: this enables transparency of distributed aspects, and facilitates the transmission of future references (sometimes called first-class futures [31]). In this case, there is almost no syntactic difference between a distributed program with asynchronous invocations and usual objects, consequently writing simple distributed programs is easier. On the other hand, explicit manipulation of futures allows the programmer to better control the program execution, but also requires a better programming expertise.

Creol [36, 17], JCoBox [40], ABS [35], and Encore [10] have explicit future types and explicit asynchronous invocations contrarily to ASP [12] where there is no specific type for futures or active objects.

How are handled the results of asynchronous method invocations? What distinguishes active objects from actors is the fact that communication between activities is performed by invocation of methods, according to the object's interface, and not through send and receive operations. This distinction in the terminology is also highlighted in Akka [45] where "actors" are distinguished from "typed actors" that are in fact active objects. The different languages propose different ways to handle method results:

- **No return value:** In some languages the methods of an active object cannot return any value. This is for example the case of Rebeca or Scala actors [26].
- **Futures:** The most classical approach is to use a future to represent the expected result of an asynchronous method call. When the returned value is needed, the program is interrupted until the value becomes available. The futures can be explicitly visible to the programmer like in ABS or transparent like in ASP. In the second case there is no need for a specific instruction to wait for the future, the current thread is automatically blocked when the value associated to a future is *needed*. This mechanism is called *wait-by-necessity*. In languages with cooperative scheduling, it is possible to let another request execute while the future is awaited (e.g. using the `await` statement in ABS).
- **Asynchronous futures:** Some languages use a future to represent the expected result of an asynchronous method call but provide no synchronisation on futures. Instead a continuation is triggered upon the resolution of the future. This continuation can be more or less explicit depending on the language. Akka and AmbientTalk feature asynchronous futures. For example, in AmbientTalk, a future access creates

g::=b | $x?$ | $g \wedge g'$ — guard
s::=`skip` | x=z | `suspend` | `await` g | `return` e | `if` e { s } `else` { s } | s ; s — statement
z::=e | e.m($\overline{e}$) | e!m($\overline{e}$) | `new` [`local`]C($\overline{e}$) | x.get — expression with side effect
e::=v | x | `this` | *arithmetic-bool-exp* — expression
v::=`null` | *primitive-val* — value

FIGURE 2. Syntax of the concurrent object layer of ABS (definition of method, class, and types omitted).

```
1   BankAccount ba = new BankAccount();
2   Transaction t = new local Transaction(ba);
3   WarningAgent wa = new WarningAgent();
4   Fut<Balance> bfut = ba!apply(t);
5   await bfut?;
6   Balance b = bfut.get;
7   wa!checkForAlerts(b);
8   b.commit()
```

LISTING 1. ABS program code.

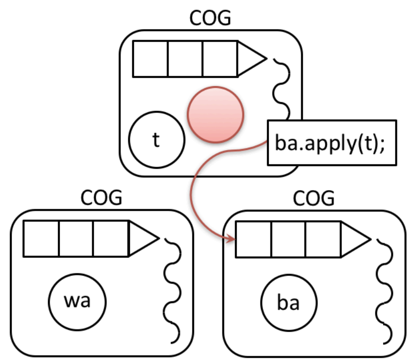


FIGURE 3. ABS program execution.

and asynchronous invocation that will be triggered after the future has been resolved. The event-based execution of the different activities makes sequences of actions more difficult to enforce. However, an AmbientTalk program is always partitioned into separate event handlers that maintain their own execution context, making the inversion of control easier to handle.

2.3. **Abstract Behavioural Specification.** *ABS* [35] is an object-oriented modelling language based on active objects. ABS takes its inspiration in CREOL for the cooperative scheduling and in JCOBOX for the object group model. It uses explicit asynchronous method calls and futures. ABS is intended for modelling and verification of distributed applications. The object group model of ABS is represented by Concurrent Object Groups (COGs). A COG manages a request queue and a set of tasks that have been created as a result of asynchronous method calls to any of the objects inside the COG. Inside a COG, only one task is active at any time. New objects can be instantiated in a new COG with the `new` keyword. In order to instantiate an object in the current COG, `new local` must be used instead. In the example, the transaction `t` is instantiated in the COG that is running the current (main) method. Contrarily to JCOBOX, ABS makes no difference on the object kind: all objects can be referenced from any COG and all objects can be invoked either synchronously or asynchronously. Listing 1 shows an ABS program that creates two new COGs and performs asynchronous method calls. Note the specific syntax (`!`) for asynchronous method invocation and the `await` instruction on Line 5 that releases the current task if the future stored in `bfut` is not yet available. Figure 3 pictures the sending of the `apply` request to the remote COG. In the illustration, COGs are large rectangles with round angles, request queues are depicted at the top of COGs, and objects are symbolised by circles. The concurrent object part of the syntax of ABS is shown in Figure 2. In this syntax, x range

```
BankAccount ba = PAActiveObject.newActive(BankAccount.class, null, node) ;
Transaction t = new Transaction(ba);
WarningAgent wa = PAActiveObject.newActive(WarningAgent.class, null, node);
Balance b = ba.apply(t); // t is deeply copied
wa.checkForAlerts(b); // if b is a future it is passed transparently
b.commit(); // a wait-by-necessity is possible here
```

LISTING 2. An example of ProActive program. *node* is not defined here.

over variable names, the overline notation ($\overline{e}$) is used for lists, *arithmetic-bool-exp* range over arithmetic and boolean expressions, and *primitive-val* stands for primitive (integer and boolean) values. The most significant statements and constructs have been described in the example. Note that field access is restricted to the current object (`this`).

ABS comes with numerous engines (see `http://abs-models.org/`) for verification of concurrent and distributed applications: a deadlock analyser [24], a resource consumption analyser [37], a termination and cost analyser COSTABS [3, 2], and a verifier for generic program properties KeY-ABS [20, 21]. In addition to verification tools, ABS tools also comprise a frontend compiler and several backends that translate ABS programs into Maude, Java, or Haskell [8]. This work is partially based on the Java backend.

2.4. **Asynchronous Sequential Processes.** ASP [13] is an active object language with a non uniform active object model; it is intended to be close to realistic implementations of distributed systems. ASP features *mono-threaded scheduling* and futures are implicitly created from asynchronous remote method calls. ASP features a wait-by-necessity behaviour upon access to an unresolved future. ASP has proven determinism properties and formalises object groups and software components [12]. Communications ensure causal ordering of requests. ProActive [6] is the Java library that enforces the ASP semantics. As the active object model of ASP is transparent, ProActive active objects follow as much as possible the syntax of standard Java. The only syntactic difference is the `newActive` primitive that is used to create an active object. Listing 2 shows the same example as in the ABS above, written in ProActive. Nevertheless, the semantics of the two programs is different since in ABS the future is always resolved when the `checkForAlerts` request is sent.

ProActive is intended for distribution, it forms a complete middleware that supports the deployment of applications on distributed infrastructures such as clusters, grids, and clouds. To this end, when an active object is created, it is registered in the Java RMI registry, RMI being the main communication layer used in ProActive. Consequently, passive objects are deeply copied when communicated between activities, the different copies are then handled independently and can be in an inconsistent state. The advantage of this approach is its scalability and its coherency with the mechanism of RMI, and consequently its practical effectiveness. Because ProActive is a Java API, it must be integrated with the standard Java programming language. Proxies are created to represent generalised pointers to futures and active objects; consequently futures and active object cannot be of primitive type and a method that returns a primitive type would be called synchronously.

2.5. **Encore.** Encore [10] has an active object programming model mixed with other parallel patterns. Encore is essentially based on a non-uniform object model but uses capacities to handle the concurrent access to passive objects, thus enabling shared references to passive objects. Encore uses cooperative scheduling of requests that is similar to Creol and ABS. Futures are explicitly typed and their value must be retrieved via a `get` construct. Encore natively features parallel constructs other than asynchronous method invocations. Internal parallelism can be spawned explicitly through `async` blocks, or implicitly through parallel combinators [22]. Encore unifies all its parallel constructs with the use of futures for handling asynchrony. A chaining operator $\rightsquigarrow$ can add a callback to a future, in order to execute it when the future is resolved. Thus Encore features both synchronisation on futures and asynchronous futures. Cooperative scheduling and future chaining mix explicit and automatic synchronisation, and save the programmer from the burden of precisely placing all the release points in the program.

2.6. **Multi-threaded Actors and Active Objects.** In this paper, we focus on multi-threaded active objects [30], that are characterised by a controlled parallelism within an activity. Contrarily to Encore, parallelism does not occur inside a request, but between requests, and under the control of the programmer since only requests tagged as compatible can run in parallel. Comparing multi-threaded scheduling to cooperative scheduling is more complex as the concurrency model is much different: on one hand cooperative scheduling prevents race conditions that might occur within multi-threaded active objects (e.g. if the wrong requests are declared compatible), but on the other hand multi-threaded active objects provide a parallel execution that is more efficient and allow the programmer to control which requests run in parallel. Section 4 shows how cooperative scheduling can be faithfully encoded with multi-threaded active objects and provides a deeper technical insight on the comparison between the two approaches.

In [29] the authors enable automatic parallel execution of requests inside actors. They use transactional memory to undo local side-effects that may be conflicting. However this work supports actors interacting only by asynchronous messages; it does not take into account any other form of synchronisation, like futures in active objects.

Other works introduced multi-threading inside actors or active objects. Parallel actors monitors [41] (PAM) was designed at the same time as MultiASP; it provides an interface to schedule the parallel service of requests in an actor. The framework is richer and allows the programmer to express any scheduling of request and fully control the request treatment. On the contrary, multi-active objects only allow the programmer to state which requests can be served in parallel, possibly depending on dynamic criteria. Multi-active objects feature a higher level of abstraction where the programmer is unable to reorder request or cancel some of them, overall multi-active objects preserve more guarantees of the original actor and active-object programming model. In fact, compatibility annotations could be used to generate a specific PAM scheduler that would simulate the behaviour of multi-active objects. More recently, *multi-threaded actors* have been proposed [5]. This approach is quite different from multi-active objects or PAM because each active object is mono-threaded but several active objects share the same request queue. The parallel treatment of requests in the same request queue is similar to the approach presented in this paper, but the fact that those requests are served by different actors makes a significant difference. On one side the approach of [5] ensures the absence of race-conditions but on the other side, it gives no guarantee on which object will serve which request. This approach is well adapted to stateless objects, but not adapted to stateful objects, as different requests can be handled by different objects. Indeed consider two requests targeting a multi-threaded actor, if the first one modifies the state of one of the actor's object and the second request is handled by a different object, then the state modification will not be visible by the second request.

3. The Multiactive Object Framework

This section presents the Multiactive object programming model, its formal model called MultiASP, and its implementation and support inside the ProActive library. Even if the works were realised independently, the concurrency and annotation system of multi-threaded active objects shares some similarities with JAC (Java annotations for concurrency) [28].

3.1. **Programming Model and Language.**

3.1.1. *Principles.* Multi-active objects enable local parallelism inside active objects. To this end, the programmer can annotate the class of an active object with information about concurrency by defining a compatibility relationship between requests. The principle is to allow two independent requests to execute in parallel, but prevent requests that could conflict on some resources to run concurrently.

In the RMI style of programming, every remote invocation to an object will be run in parallel with no synchronisation. As a result, data-races can happen on concurrently accessed resources. A classic approach to solve this problem is to protect concurrent executions with a lock, but this approach is too fine-grained to be scalable, and possibly error-prone. By nature, active objects materialise a much safer model where no inner concurrency is possible. However, we aim at a programming model that is locally concurrent and more flexible than the mono-threaded active objects, but more constrained and less error-prone than bare synchronisation of threads. The principle relies on the notion of *compatible requests.* Only requests that are declared compatible can run in parallel. To start serving a new request, a multi-active object first checks that it is compatible with all the currently served requests, and with the requests that should be served before[3].

Multi-active objects extend active objects by assigning each method to at most one group. These groups define a compatibility relationship between requests: only the requests that target methods belonging to compatible groups can be executed in parallel, whereas the others will be guaranteed not to run concurrently. Since the groups and their compatibilities are defined with annotations, the application logic is not mixed with concurrency features. Two groups should be made compatible if their methods do not access the same data, and if the methods of the two groups can be executed in any order. Compatibility is a commutative, non-transitive relation. A group may be declared to be compatible with itself. We base our multi-active object framework on active objects à la ASP. An active object can be turned into a multi-active object by applying the following design methodology:

- Without annotations, a multi-active object behaves identically to a mono-threaded active object, no race condition is possible, but no local parallelism is possible either. Methods assigned to no group are incompatible with all the methods: by default methods are conflicting, and no data-race is possible.
- If some parallelism is desired, for efficiency reasons or because of potential deadlocks, each remotely invocable method can be assigned to a group. Then compatibility between the groups can be defined based on the fields accessed by each method.

[3] Request service in ASP and ProActive follows a FIFO policy.

- If even more parallelism is desired, the programmer has two non-exclusive options: protect the access to the fields by a locking mechanism and declare more groups as compatible, or define a compatibility function that decides at runtime which requests are compatible, depending on the request parameters or on the object state.

In all cases, we assume that the programmer defines the groups and their compatibilities correctly. Dynamic checks or static analysis should be added to ensure that no race condition appear at runtime, but this is out of scope of this paper.

Multi-active objects provide a customisable trade-off between gaining some parallelism at the intra-object level, removing some deadlocks, and loosing some safety in terms of absence of data races (compared to classical active objects where the absence of data races is guaranteed). Compatibilities must be defined carefully to prevent safety problems.

3.1.2. *Multi-active objects in Practice.* ProActive offers an implementation of the multi-active object programming model as an extension of its active object implementation. A set of Java annotations can be used for the specification of multi-active object notions. The annotations are processed at runtime, which enables to decide dynamically on the compatibility of requests. The three main annotations are the following:

- A `@Group` annotation can be declared on top of a class to define a group of requests. Request in the same group share the same compatibility requirements.
- A `@MemberOf` annotation can be defined on top of a method definition, and specifies the group to which the method belongs.
- A `@Compatible` annotation can be used to declare the groups that are compatible, i.e. to declare the requests that can be run in parallel safely.

As an example, consider a distributed peer-to-peer system implemented with multi-active objects. The purpose of this application is to offer a high performance distributed system for data storage. Each peer is represented by a multi-active object that is instantiated from a `Peer` class. The `Peer` class is partially shown in Listing 3. A peer has operations that are related to the structure of the peer-to-peer system, as well as operations that are related to data management. In the annotation `DefineGroup`, four groups are defined (Lines 1 to 6). They correspond to the different concerns addressed by the class. All the groups except `broadcasting` are self compatible: they allow several requests of the same group to be run in parallel. Then compatibility rules are defined between those groups (in the annotation `DefineRules`), for example Line 8 declares that the methods of groups `gettersOnImmutable`, `broadcasting`, and `dataManagement` can be executed in parallel. One can notice that the `join` method does not belong to any group: a `join` request will always be executed alone, it is incompatible with all other requests. On the contrary requests on method `getIdentifier` can be executed in parallel with any other request.

3.1.3. MultiASP. MultiASP is the active object programming language that extends ASP for the support of multi-active objects. MultiASP formalises the multi-active objects that are implemented in ProActive, and allows us to reason on the execution of ProActive programs. A seminal version of MultiASP is given in [30]. This paper shows an updated version based on classes instead of object instances. In this paper, we also extend the operational semantics with advanced scheduling capabilities. MultiASP is an imperative programming language, whose syntax is inspired from object-oriented core languages resembling to Featherweight Java [34]. As can be seen in Figure 4, the syntax of

```java
@DefineGroups({
  @Group(name="gettersOnImmutable", selfCompatible=true),
  @Group(name="dataManagement", selfCompatible=true),
  @Group(name="broadcasting", selfCompatible=false),
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"gettersOnImmutable", "broadcasting", "dataManagement"}),
  @Compatible({"gettersOnImmutable", "monitoring"})
})
public class Peer implements Serializable, RunActive {
  private LongWrapper identifier;
  private Zone zone;

  @MemberOf("gettersOnImmutable")
  public LongWrapper getIdentifier() { ... }

  public BooleanWrapper join(Peer p, int dimension) { ... }

  @MemberOf("dataManagement")
  public AddResponse add(AddQuery query) { ... }

  @MemberOf("broadcasting")
  public void broadcast(Key constraint, RoutingPair message) { ... }
}
```

LISTING 3. The Peer class of the fault tolerant broadcast application.

$$
\begin{array}{rcll}
P & ::= & \overline{C}\ \{\overline{x}\ s\} & \text{program} \\
S & ::= & \mathtt{m}(\overline{x}) & \text{method signature} \\
C & ::= & \mathtt{class}\ \mathtt{C}(\overline{x})\ \{\overline{x}\ \overline{M}\} & \text{class} \\
M & ::= & S\{\overline{x}\ s\} & \text{method definition} \\
s & ::= & \mathtt{skip} \mid x = z \mid \mathtt{return}\ e \mid s\ ;\ s & \text{statement} \\
z & ::= & e \mid e.\mathtt{m}(\overline{e}) \mid \mathtt{new}\ \mathtt{C}(\overline{e}) \mid \mathtt{newActive}\ \mathtt{C}(\overline{e}) & \text{expression with side effects} \\
e & ::= & v \mid x \mid \mathtt{this} \mid \textit{arithmetic-bool-exp} & \text{expression} \\
v & ::= & \mathtt{null} \mid \textit{primitive-val} & \text{value}
\end{array}
$$

FIGURE 4. The class-based static syntax of MultiASP.

MULTIASP is also close to PROACTIVE programs. A program consists of a set of classes and one main method. Classes, methods, and statements are standard. In the syntax, x ranges over variable names, C over class names, and `m` over method names. We characterise a list of elements with the overlined notation. The list $\overline{x}$ denotes local variables when it appears in method bodies and object fields in class declarations. In MULTIASP, as in PROACTIVE, there are two ways to create an object: `new` creates a new object in the current activity (a passive object), and `newActive` creates a new active object. Also, no syntactic distinction exists between local and remote invocations, $e.\mathtt{m}(\overline{e})$ is the generic method invocation, that triggers an asynchronous method invocation if the targeted object

$$
\begin{array}{rcl}
cn & ::= & \overline{elem} \\
elem & ::= & \textsc{fut}(f,v,\sigma) \mid \textsc{fut}(f,\bot) \mid \textsc{act}(\alpha,o,\sigma,p,Rq) \\
v & ::= & o \mid \alpha \mid \texttt{null} \mid \textit{primitive-val} \\
Storable & ::= & [\overrightarrow{x \mapsto v}] \mid v \mid f \\
s & ::= & x = \bullet \mid \texttt{skip} \mid x = z \mid \texttt{return}\ e \mid s\ \texttt{;}\ s \\
z & ::= & e \mid e.\texttt{m}(\overline{e}) \mid \texttt{new C}(\overline{e}) \mid \texttt{newActive C}(\overline{e}) \\
e & ::= & v \mid x \mid \texttt{this} \mid \textit{arithmetic-bool-exp}
\end{array}
\qquad
\begin{array}{rcl}
E & ::= & \{\ell \mid s\} \\
F & ::= & E \mid E :: F \\
q & ::= & (f, \texttt{m}, \overline{v}) \\
p & ::= & \overrightarrow{q \mapsto F} \\
Rq & ::= & \varnothing \mid q :: Rq \\
\sigma & ::= & \overrightarrow{o \mapsto Storable} \\
\ell & ::= & \texttt{this} \mapsto v, \overrightarrow{x \mapsto v}
\end{array}
$$

FIGURE 5. The runtime syntax (e and z similar to the static syntax).

is active or a local synchronous method invocation if it is passive. Similarly, as synchronisation on futures is transparent and handled through wait-by-necessity, there is no particular syntax for interacting with a future. Variables in ASP refer either to a local variable of the current method or to a field of the current object. A special variable, `this`, enables access to the current object. The sequence operator is associative with a neutral skip element: a sequence of instructions can always be rewritten as $s; s'$, with s not a sequence.

The runtime syntax of MULTIASP is shown in Figure 5. The set of elements of a MULTIASP configuration *cn* are of two kinds: activities and future binders. We rely on three infinite sets: *object locations* in the local store, ranged over by $\{o,\ o',\ \ldots\}$; *active objects names*, ranged over by $\{\alpha,\ \beta,\ \ldots\}$; and *future names*, ranged over by $\{f,\ f',\ \ldots\}$. Additionally, the following terms are defined:

- There are two kinds of runtime values. *Simple values* (v) can be either values of the static syntax (`null`, *primitive-val*), the location of an object in the local store (o), or active object names (α). *Storable values* (*Storable*) are either objects, futures, or simple values. An object is a mapping[4] from field names to their values: $[\overrightarrow{x \mapsto v}]$.
- A local environment ℓ mapping local variables (including `this`) to simple values.
- A thread F is a stack of methods being executed, where each method execution E consists of a local environment and a statement s to execute. The first method of the stack is actually executing, the others have been put in the stack due to local synchronous method calls. A special statement $x = \bullet$ allows us to remember the current execution point when a new element is added to the stack.
- Activities are of the form $\textsc{act}(\alpha, o, \sigma, p, Rq)$. An activity contains terms that define:
  – α, the *name of the activity*.
  – o, the location of the *active object* in σ.
  – σ, a *local store* mapping object locations to storable values.
  – *Rq*, a FIFO *queue of requests*, awaiting to be served.
  – p, a set of *requests currently served*: a mapping, associating to each currently served request the corresponding thread F.
- Future binders are of two forms. The form $\textsc{fut}(f, \bot)$ means that the value of the future has not been computed yet: it is *unresolved*. The form $\textsc{fut}(f, v, \sigma)$ is used when the value of the future has been *resolved*. The future value can be either a primitive value or a reference to an active or passive object. If the future value references a passive object, then the piece of store σ defines the content of this object. As only active objects are remotely accessible, the part of the store referenced by

[4]We denote mappings by $\overrightarrow{\_ \mapsto \_}$, and use union $\cup$ (resp. disjoint union $\uplus$) over mappings. Mapping updates are written $\sigma[x \mapsto v]$, updating the value associated to x in σ. *dom* is the domain of a mapping. Additionally, for any vectors $\overline{y}$ and $\overline{v}$ of the same length, $[\overrightarrow{y \mapsto v}]$ maps the elements of one vector to the other.

$$
\begin{array}{lll}
[\![\mathit{primitive\text{-}val}]\!]_{(\sigma+\ell)} \triangleq \mathit{primitive\text{-}val} & & [\![\texttt{null}]\!]_{(\sigma+\ell)} \triangleq \texttt{null} \\
[\![\alpha]\!]_{(\sigma+\ell)} \triangleq \alpha & & \\
[\![e \oplus e']\!]_{(\sigma+\ell)} \triangleq [\![e]\!]_{(\sigma+\ell)} \oplus [\![e']\!]_{(\sigma+\ell)} & \text{if } [\![e]\!]_{(\sigma+\ell)} \text{ and } [\![e']\!]_{(\sigma+\ell)} \text{ are primitive values} & \\
[\![x]\!]_{(\sigma+\ell)} \triangleq [\![\ell(x)]\!]_{(\sigma+\ell)} & \text{if } x \in \mathrm{dom}(\ell) & \\
[\![x]\!]_{(\sigma+\ell)} \triangleq [\![\ell(\texttt{this})(x)]\!]_{(\sigma+\ell)} & \text{if } x \notin \mathrm{dom}(\ell) & \\
[\![o]\!]_{(\sigma+\ell)} \triangleq o & \text{if } \sigma(o) = f \mid [\overrightarrow{x \mapsto v}] & \\
[\![o]\!]_{(\sigma+\ell)} \triangleq [\![\sigma(o)]\!]_{(\sigma+\ell)} & \text{if } \sigma(o) = o' \mid \alpha \mid \texttt{null} \mid \mathit{primitive\text{-}val} &
\end{array}
$$

FIGURE 6. Evaluation function

this location must also be transmitted when the future's value is sent back to the caller. This step involves a serialisation mechanism explained below. Note that the store σ can contain references to other futures.

An object location is fresh if it does not exist in the store where it is added. A future or an activity name is fresh if it does not exist in the configuration, we use the following auxiliary functions:

- *fields*(C) returns the fields as defined in the declaration of the class named C.
- bind instantiates and initializes a method execution. If o is of class C and M is a method of C with the signature $m(\overline{y})$ and body $\{\overline{x}\ s\}$, then: $\mathrm{bind}(o, \mathtt{m}, \overline{v'}) = \{\overrightarrow{y \mapsto v'}, \overrightarrow{x \mapsto \texttt{null}}, \texttt{this} \mapsto o \mid s\}$
- $[\![e]\!]_{(\sigma+\ell)}$ returns the value of e by computing the arithmetic and boolean expressions and retrieving the values stored in σ or ℓ, this evaluation function is defined in Figure 6. $\oplus$ stands for any arithmetic or binary binary operator (unary operators can be expressed similarly). If one member of an arithmetic expression is an unresolved future, the function is undefined. $[\![e]\!]_{(\sigma+\ell)}$ returns a value and, if the value is a reference to a location in the store, it follows references recursively; it only returns a location if it points to an object[5]. $[\![\overline{e}]\!]_{(\sigma+\ell)}$ returns the tuple of values of $\overline{e}$.
- ready is a predicate that decides whether a request q is ready to be served. We will use $\mathrm{ready}(q, p, Rq)$ where p the requests currently served and Rq the requests that have been received before q; it is *true* if q is compatible with all requests in p and in Rq: $\mathrm{ready}(q, p, Rq) = (\forall q' \in (\mathrm{dom}(p) \cup Rq)\,.\mathit{compatible}(q, q'))$.

  In the semantics, compatibility is expressed between requests. This is expressive enough to encode the compatibility relations that can be written in ProActive, including compatibility depending on method parameters. It is also easy to extend it in order to define compatibility depending on object state. Compatibility on request can be derived from group compatibility by stating that two requests are compatible if they target methods belonging to compatible groups.
- Serialisation reflects the communication happening in Java RMI. All references to passive objects are serialised when communicated between activities, so that they are always handled locally. We formalise a serialisation algorithm that marks and copies the objects to serialise recursively. The function serialise takes a value v and a store σ and returns a sub-store of σ containing the serialisation of v. It is defined as the mapping verifying the following (co-inductive) constraints:

$$\mathrm{serialise}(o, \sigma) = (o \mapsto \sigma(o)) \cup \mathrm{serialise}(\sigma(o), \sigma) \qquad \mathrm{serialise}([\overrightarrow{x \mapsto v}], \sigma) = \textstyle\bigcup_{v' \in \overline{v}} \mathrm{serialise}(v', \sigma)$$
$$\mathrm{serialise}(f, \sigma) = \mathrm{serialise}(\alpha, \sigma) = \mathrm{serialise}(\texttt{null}, \sigma) = \mathrm{serialise}(\mathit{primitive\text{-}val}, \sigma) = \varnothing$$

[5]The way the store is built guarantees that expression evaluation terminates if the store is finite.

$$\textsc{Serve}\quad \frac{\text{ready}(q,p,Rq) \qquad q=(f,\mathtt{m},\overline{v}) \qquad \text{bind}(o,\mathtt{m},\overline{v})=\{\ell \mid s\}}{\textsc{act}(\alpha,o,\sigma,p,Rq::q::Rq') \to \textsc{act}(\alpha,o,\sigma,\{q\mapsto\{\ell|s\}\}\uplus p,Rq::Rq')}$$

$$\textsc{Assign-Local}\quad \frac{x\in\text{dom}(\ell) \qquad v=[\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha,o,\sigma,\{q\mapsto\{\ell \mid x=e;s\}::F\}\uplus p,Rq) \to \textsc{act}(\alpha,o,\sigma,\{q\mapsto\{\ell[x\mapsto v]\mid s\}::F\}\uplus p,Rq)}$$

$$\textsc{Assign-Field}\quad \frac{\ell(\mathtt{this})=o \qquad x\in\text{dom}(\sigma(o)) \qquad x\notin dom(\ell) \qquad \sigma'=\sigma[o\mapsto(\sigma(o)[x\mapsto[\![e]\!]_{(\sigma+\ell)}])]}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell \mid x=e;s\}::F\}\uplus p,Rq)\to\textsc{act}(\alpha,o_\alpha,\sigma',\{q\mapsto\{\ell\mid s\}::F\}\uplus p,Rq)}$$

$$\textsc{New-Object}\quad \frac{\mathit{fields}(\mathtt{C})=\overline{y} \qquad o \text{ fresh} \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad \sigma'=\sigma\cup\{o\mapsto[\overrightarrow{y\mapsto v}]\}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell\mid x=\mathtt{new\ C}(\overline{e});s\}::F\}\uplus p,Rq)\to\textsc{act}(\alpha,o_\alpha,\sigma',\{q\mapsto\{\ell\mid x=o;s\}::F\}\uplus p,Rq)}$$

$$\textsc{New-Active}\quad \frac{\mathit{fields}(\mathtt{C})=\overline{y} \qquad o,\beta \text{ fresh} \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad \sigma'=\{o\mapsto[\overrightarrow{y\mapsto v}]\}\cup\text{serialise}(\overline{v},\sigma)}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell\mid x=\mathtt{newActive\ C}(\overline{e});s\}::F\}\uplus p,Rq) \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell\mid x=\beta;s\}::F\}\uplus p,Rq)\ \textsc{act}(\beta,o,\sigma',\varnothing,\varnothing)}$$

$$\textsc{Invk-Active}\quad \frac{[\![e]\!]_{(\sigma+\ell)}=\beta \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad f,o \text{ fresh} \qquad \sigma_1=\sigma\cup\{o\mapsto f\} \qquad (\overline{v_r},\sigma_r)=\text{rename}_{\sigma'}(\overline{v},\text{serialise}(\overline{v},\sigma)) \qquad \sigma''=\sigma'\cup\sigma_r}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell\mid x=e.\mathtt{m}(\overline{e});s\}::F\}\uplus p,Rq)\ \textsc{act}(\beta,o_\beta,\sigma',p',Rq') \to \textsc{act}(\alpha,o_\alpha,\sigma_1,\{q\mapsto\{\ell\mid x=o;s\}::F\}\uplus p,Rq)\ \textsc{act}(\beta,o_\beta,\sigma'',p',Rq'::(f,m,\overline{v_r}))\ \textsc{fut}(f,\bot)}$$

$$\textsc{Invk-Active-Self}\quad \frac{[\![e]\!]_{(\sigma+\ell)}=\alpha \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad f,o \text{ fresh} \qquad \sigma_1=\sigma\uplus\{o\mapsto f\} \qquad (\overline{v_r},\sigma_r)=\text{rename}_{\sigma_1}(\text{serialise}(\overline{v},\sigma)) \qquad \sigma'=\sigma_r\cup\sigma_1}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell|x=e.\mathtt{m}(\overline{e});s\}::F\}\uplus p,Rq) \to \textsc{act}(\alpha,o_\alpha,\sigma',\{q\mapsto\{\ell\mid x=o;s\}::F\}\uplus p,Rq::(f,\mathtt{m},\overline{v_r}))\ \textsc{fut}(f,\bot)}$$

$$\textsc{Invk-Passive}\quad \frac{[\![e]\!]_{(\sigma+\ell)}=o \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad \text{bind}(o,m,\overline{v})=\{\ell'\mid s'\}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell\mid x=e.\mathtt{m}(\overline{e});s\}::F\}\uplus p,Rq)\to\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell'\mid s'\}::\{\ell\mid x=\bullet;s\}::F\}\uplus p,Rq)}$$

$$\textsc{Return-Local}\quad \frac{v=[\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha,o,\sigma,\{q\mapsto\{\ell\mid\mathtt{return}\ e;s_r\}::\{\ell'\mid x=\bullet;s\}::F\}\uplus p,Rq)\to\textsc{act}(\alpha,o,\sigma,\{q\mapsto\{\ell'\mid x=v;s\}::F\}\uplus p,Rq)}$$

$$\textsc{Return}\quad \frac{v=[\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha,o,\sigma,\{(f,m,\overline{v})\mapsto\{\ell\mid\mathtt{return}\ e;s_r\}\}\uplus p,Rq)\ \textsc{fut}(f,\bot) \to \textsc{act}(\alpha,o,\sigma,p,Rq)\ \textsc{fut}(f,v,\text{serialise}(v,\sigma))}$$

$$\textsc{Update}\quad \frac{\sigma(o)=f \qquad (v_r,\sigma_r)=\text{rename}_\sigma(v,\sigma') \qquad \sigma''=\sigma[o\mapsto v_r]\cup\sigma_r}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq)\ \textsc{fut}(f,v,\sigma') \to \textsc{act}(\alpha,o_\alpha,\sigma'',p,Rq)\ \textsc{fut}(f,v,\sigma')}$$

Figure 7. Semantics of MultiASP.

- The function $\text{rename}_\sigma(\overline{v},\sigma')$ renames the object locations appearing in σ' and $\overline{v}$, making them disjoint from the object locations of σ; it returns the renamed set of values $\overline{v'}$ and another store σ'', as a pair of the form $(\overline{v'},\sigma'')$.

Figure 7 shows the semantics of MultiASP as a transition relation between configurations. The rules only show the activities and futures involved in the reduction, the rest of the configuration is kept unchanged. Most of the rules are triggered depending on the shape of the first statement of an activity's thread. The reduction rules are described below:

- Serve picks the first ready request in the queue (i.e. compatible with executing requests and with older requests in the queue) and allocates a new thread to serve it. It fetches the method body and creates the execution context.
- Assign-Local assigns a value to a local variable. If the statement to be executed is an assignment of an expression that can be reduced to a value, then the mapping of local variables is updated accordingly.
- Assign-Field assigns a value to a field of the current object (the one pointed to by `this`). It is similar to the previous rule except that it modifies the local store.
- New-Object creates a new local object in the store at a fresh location, after evaluation of the object parameters. The new object is assigned to a field or to a local variable by one of the two rules above.
- New-Active creates a new activity that contains a new active object. It picks a fresh activity name, and assigns the serialised object parameters: the initial local store of the activity is the piece of store referenced by the parameters. The freshness of the location of the new active object ensures that it is not in the serialised store.
- Invk-Active performs an asynchronous remote method invocation on an active object. It creates a fresh future with undefined value. The arguments of the invocation are serialised and put in the store of the invoked activity, possibly renaming locations to avoid clashes. The special case $\alpha = \beta$ requires a trivial adaptation with the rule Invk-Active-Self.
- Invk-Passive performs a local synchronous method invocation. The method is retrieved and an execution context is created; the thread stack is extended with this execution context. The interrupted execution context is second in the stack; the result returned by the method will replace the $\bullet$ when it is computed. Note that the distinction between synchronous and asynchronous calls depends on the kind of the reference to the invoked object: an active object is invoked synchronously if it is invoked through its local reference.
- Return-Local handles the return value of local method invocation. It replaces the $\bullet$ in the second entry of the stack by the returned value. The return corresponds to a local invocation because it is not the only execution context in the stack.
- Return occurs when a request finishes. It stores the value computed by the request as a future value. Serialisation packs the objects referenced by the future value.
- Update updates a future reference with a resolved value. This is performed at any time when a future is referenced and the future value is resolved.

The local rules reflect a classical object oriented language: New-Object and Assign-Field modify the local store, and Invk-Passive and Return-Local affect the local execution context. The other rules deal with parallelism and communication. Given a program $P = \overline{\mathtt{C}}\ \{\, \overline{x}\ s \,\}$, executing P requires to create an initial configuration with a single activity that serves one request containing the main block s with local variables $\overline{x}$: $cn_0 = \text{ACT}(\alpha, o, \varnothing, \{q \mapsto \{\overrightarrow{x \mapsto \mathtt{null}}, \mathtt{this} \mapsto o \mid s\}\}, \varnothing)$. Classically $\to^*$ denotes the transitive closure of $\to$. In [30], it was proven that this semantics ensures that request services are scheduled such that *parallelism is maximised while preventing two incompatible requests from being served in parallel*. Safe parallelism can be formalised as follows

```
1 @DefineThreadConfig(threadPoolSize=2, hardLimit=false)
2 @Group(name="monitoring", selfCompatible=true, minThreads=1, maxThreads=2)
```

Listing 4. An example of annotations for thread management.

Theorem 3.1 (Safe parallelism)**.** *Any two requests served in parallel are compatible: if* cn_0 *is the initial configuration and* $cn_0 \to^* \textsc{act}(\alpha, o, \sigma, \{q \mapsto \{\ell \mid s\} :: F\} \uplus \{q' \mapsto \{\ell' \mid s'\} :: F'\} \uplus p, Rq)\ Q$ *with* $q = (f, \mathtt{m}, \overline{v})$*,* $q' = (f', \mathtt{m}', \overline{v}')$ *then* $compatible(q, q')$*.*

3.2. **Request Scheduling Extension.** The semantics presented above ensures that any ready request can be served immediately, which maximizes parallelism. While this property is strong and valuable, such a scheduling policy might reveal inefficient in practice and some additional mechanism is needed to control the number of threads running in parallel. To control the parallelism the programmer can limit the number of executing threads globally or per group. We detail below the annotations provided in ProActive for thread limitation and extend the semantics to model this aspect.

3.2.1. *Controlling thread creation in ProActive.* To customise scheduling inside multi-active objects we introduce scheduling annotations. The `@DefineThreadConfig` annotation allows the programmer to control two aspects:

- **The maximum number of threads used by a multi-active object.** This is the *thread limit* of the multi-active object. It prevents a potential thread explosion at runtime. This limit is checked whenever a thread is to be created or activated.
- **The kind of thread limit.** It can be of two kinds: either *soft* of *hard*. A *soft thread limit* counts, in the thread limit, only the threads that are currently active, i.e. that are not blocked in wait-by-necessity. On the contrary, a *hard thread limit* counts, in the thread limit, all the created threads, even the ones that are in wait-by-necessity.

The `@DefineThreadConfig` annotation is illustrated in Listing 4, Line 1. A multi-active object created with this annotation has two threads at maximum to process its requests. Since the configuration specifies a soft thread limit, additional threads can be created to compensate threads that are blocked in wait-by-necessity. The thread limit and the kind of thread limit can also be changed programmatically and dynamically through the API.

In addition to the global thread limitation that applies per multi-active object, a mechanism controls the number of threads allocated to a given group. This mechanism first reserves some threads for the processing of the requests of the group. Second, it specifies the maximum number of threads that can be used by the requests of the group at the same time. Since these specifications apply to groups of requests, we extend the `@Group` annotation with two optional parameters. Line 2 of Listing 4 shows a group definition where one thread is reserved for the group and this group cannot use more than two threads.

When the number of threads is limited, requests compete for thread resources. Due to the limited number of threads, there might exist requests ready for execution (because they fulfil compatibility conditions), but that cannot be executed because all threads of the multi-active object are busy. Such requests form the *ready queue* of a multi-active object. By default, the ready queue is, like the reception queue, ordered according to the order of reception. We introduce a priority mechanism that allows the programmer to

```
@DefinePriorities({
  @PriorityHierarchy({
    @PrioritySet({"gettersOnImmutable"}),
    @PrioritySet({"broadcasting", "dataManagement"}),
    @PrioritySet({"monitoring"})
  })
})
```

Listing 5. An example of priority annotations.

determine an order of importance in the execution of requests. Priorities applies on the ready request queue, and thus on compatible requests. We extend the set of multi-active object annotations with priority annotations that relate groups of requests. We represent the priority dependencies with a graph structure, that allows a partial ordering of requests. A `@PriorityHierarchy` annotation creates an ordering on groups. The order in which the groups are declared defines the priority dependencies, like `group1 > group2 > group3`. Several groups can belong to the same priority level. Listing 5 shows an example of priority annotations applied to the `Peer` class of Listing 3. It states that methods of the group `gettersOnImmutable` should be served in priority compared to other methods, `monitoring` methods have the lowest priority, and there is no order between the two other groups.

Overall, the advanced scheduling annotations for priorities and manipulation of threads offer a high-level specification mechanism to deal with fine-grained optimisation of multi-active object-based applications. The ProActive API also allows the programmer to switch between hard and soft thread limit at runtime, for the current active object.

3.2.2. *The semantics of request scheduling.* We formalise in MultiASP the management of threads in multi-active objects. Our approach is to introduce in the semantics additional qualifiers on activities and requests to record scheduling informations. First, we extend the syntax of MultiASP so that the thread limit mechanism can be programmatically changed between a *soft limit* and a *hard limit*. The syntax is extended with two new statements:

$$s \;::=\; \ldots \;|\; \mathbf{setLimitSoft} \;|\; \mathbf{setLimitHard}$$

Then, we extend the MultiASP semantics with request scheduling aspects. We suppose that the group of a request q can be retrieved through an auxiliary function $group(q)$. Additionally, a filtering operator $p\big|_g$ returns the requests from group g among the set of threads p. The thread limit of a group g can be retrieved with $\mathcal{L}_g$. To indicate the status of each thread, we qualify each of the currently served requests as either actively served: q_A, or passively served: q_P. Each entry in the current request queue is now either $q_A \mapsto F$ or $q_P \mapsto F$. Passively served requests are requests that have been blocked in wait-by-necessity. The auxiliary function $\text{Active}(p)$ returns the number of actively served requests in the set of threads p. Each activity has either a soft limit status written $\text{act}(\ldots)_S$, or a hard limit status written $\text{act}(\ldots)_H$. An activity has by default a soft limit status when it is created. sh is used as a variable ranging over S and H: $sh ::= S \,|\, H$. Finally, we modify the reduction rules of the operational semantics as follows:

- We add a rule Activate-Thread for activating a thread. It looks at the group of the considered request and checks if this group has not reached its thread limit.

The *sh* variable is used so that the kind of thread limit is kept unchanged.

Activate-Thread

$$\frac{group(q) = g \qquad \text{Active}(p\big|_g) < \mathcal{L}_g}{\textsc{act}(\alpha, o, \sigma, \{q_P \mapsto F\} \uplus p, Rq)_{sh} \to \textsc{act}(\alpha, o, \sigma, \{q_A \mapsto F\} \uplus p, Rq)_{sh}}$$

- Each rule allowing a thread to progress requires that the request processed by this thread is active. To this end, q is replaced by q_A in all MultiASP reduction rules except for Serve and Update.
- The Serve rule is only triggered if the thread limit of the request group is not reached, i.e. if: $\text{Active}(p\big|_{group(q)}) < \mathcal{L}_g$.
- We add two additional rules, Set-Hard-Limit and Set-Soft-Limit, for changing the kind of thread limit of an activity:

Set-Hard-Limit

$$\textsc{act}(\alpha, o, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitHard}; s\} :: F\} \uplus p, Rq)_{sh} \to \textsc{act}(\alpha, o, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_H$$

Set-Soft-Limit

$$\textsc{act}(\alpha, o, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitSoft}; s\} :: F\} \uplus p, Rq)_{sh} \to \textsc{act}(\alpha, o, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_S$$

- A wait-by-necessity occurs only in case of method invocation on a future, since field access is only allowed on the current object `this`. We add a rule Invk-Future that passivates the current thread when a method invocation is performed on a future that has not been updated locally yet. This rule is only applied for activities in *soft thread limit* and when a method is invoked on a reference to a future.

Invk-Future

$$\frac{[\![e]\!]_{(\sigma+\ell)} = o' \qquad \sigma(o') = f}{\textsc{act}(\alpha, o, \sigma, \{q_A \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S \to \textsc{act}(\alpha, o, \sigma, \{q_P \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S}$$

In summary, four rules are added to MultiASP semantics, and the terms for activities and requests are qualified with thread indicators to take into account advanced scheduling mechanisms of multi-active objects. Note that the thread limit is only checked upon thread activation, thus when switching from soft to hard limit, there may be interrupted passive threads in the set of currently executed requests. In other words, it is not always true that, when the current limit is hard, there is no passivated currently executed request. This corresponds to the behaviour of ProActive that checks thread limit only upon creation or activation of a thread. Though the semantics allows for activation/passivation loops for the same thread, in practice useless activations are avoided by monitoring the status of the futures and activating, by a notification mechanism, a thread that can progress.

3.3. **Development Support.** We conclude this section by presenting a visualisation tool dedicated to multi-active objects. Its purpose is to help the programmer understand the behaviour of his multi-active object applications and to debug concurrent programs easily.

3.3.1. *Viewing executions.* The multi-active object programming model is designed such that there is a minimum specification to be written by the programmer. However, multi-active objects are very good for programming systems that involve complex coordination of entities and massive parallelism. For these advanced cases, being able to observe the behaviour of the application is crucial, either to debug it or to improve its performance.

We present a debugger that offers a visualisation of multi-active object executions based on a post mortem analysis. The user of this tool can observe the application execution while

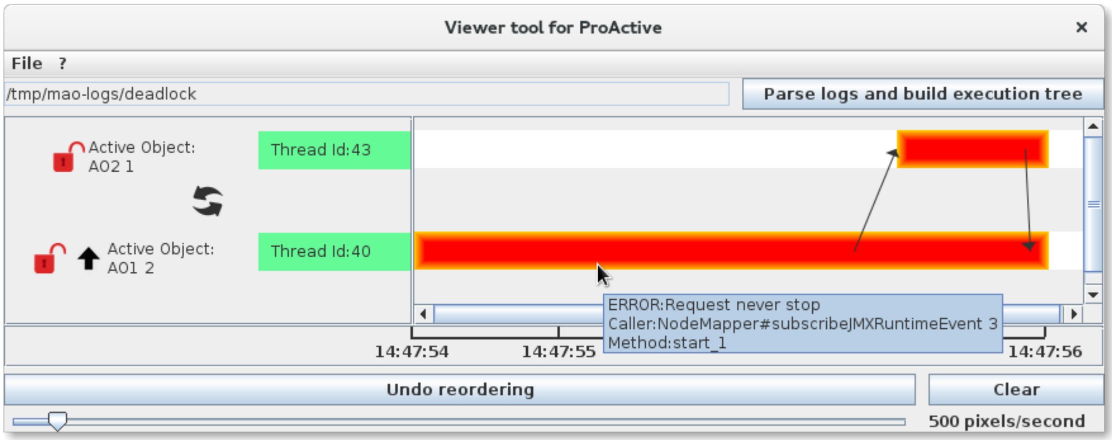


FIGURE 8. The debugger tool: deadlock scenario (screenshot).

being exposed to a view that corresponds to the notions he manipulates when programming. In the main frame, a thread is represented by a sequence of requests on a time line. A screenshot of the debugger tool is provided in Figure 8. In this example, two single-threaded multi-active objects are displayed. Arrows represent the request sending between multi-active objects. The length of an element is representative of its duration. A mouse-over on a request displays the name of the request, the identity of the sender, and the timestamp of its reception. The debugger highlights the compatibility of the selected request with respect to others. A precise listing of requests can also be viewed based on a particular timestamp, distinguishing the requests that are being executed, the ones in the queue, the completed ones, and the ones that will be received later.

3.3.2. *Use case.* In active objects, a deadlock is generally caused by a circular dependency of requests. With multi-active objects, a deadlock can be due to two constraints: a lack of compatibility or a lack of threads. We show in Figure 8 the debugger tool in a scenario where it helps the programmer to identify a deadlock in a multi-active object execution (red requests). The debugger produces in this case the 'request never ends' error, informing the user of a potentially faulty execution. The representation of communications shows that there is a circular request dependency between multi-active objects `First` and `Second`. For example, if the request received by `First` was not compatible with the request that executes, then it would never execute. Here the deadlock is instead due to a lack of threads: no more thread can be created. This case only happens if the soft thread limit is not activated, else another thread could compensate the thread that is in wait-by-necessity.

Besides deadlocks, the debugger also shows the sequence of actions that occurred in a particular execution, and thus help in spotting race conditions that can happen, typically the concurrent sending of requests to the same target object that would create non-deterministic behaviour. This tool is part of the ProActive framework.

4. Encoding of Cooperative Active Objects into Multi-active Objects

We present in this section a particular application of multi-active objects that consists in encoding the cooperative active object language ABS into the multi-active objects of

```
Class COG {
  UUID freshID()
  UUID register(Object x, UUID id)
  Object retrieve(UUID id)
  Object execute(UUID id, MethodName m, params) {
    w=this.retrieve(id); x=w.m(params); return x}
  Object execute_condition(UUID id, MethodName cm, params) {
    w=this.retrieve(id); x=w.cm(params); return x}
}
```

Listing 6. The COG class in MultiASP

MultiASP/ProActive. In other words, we present here a ProActive backend for ABS that translates ABS code to Java programs that use the ProActive library and offers a distributed execution of ABS programs. We also formalise and prove the correctness of the translation from ABS to MultiASP programs.

4.1. ProActive Backend and Translational Semantics.

One way to execute ABS programs is to use the Java backend. The Java backend translates ABS programs into Java programs according to ABS semantics, but without distribution support. The ProActive backend relies on the Java backend concerning the functional layer of the language, introducing a new translation for the object and concurrency layer, plus distributed notions.

First, to translate the object model of ABS into multi-active objects, we put several objects under the control of one active object, which fits the active object group model of ABS. Compared to translating each ABS object into a multi-active object, this solution requires less synchronisation between activities and is more scalable. Indeed to guarantee that a single thread executes in a set of active objects we would need to synchronise the different active objects and schedule their execution. Implementing such a scheduling pattern would not only be difficult and costly (each synchronisation should be performed either by a method invocation or by a future access) but also would not match the active object principles (loose coupling between active objects). In the translation, we introduce a `COG` class for representing ABS cogs. Only objects of the `COG` class are multi-active objects. The `COG` class in MultiASP is shown in Listing 6. `UUID` represent the type of object identifiers. The `COG` class has methods to store and retrieve locally the translated ABS objects, and to generically execute a method on them. A special method `execute_condition` is used to execute await conditions. It is a separate method because it does not have the same purpose as `execute` that serves external asynchronous requests; and the two methods follow different compatibility rules. We assign to every translated ABS object a unique identifier.

Second, all translated ABS classes are extended with two parameters: a *cog* parameter, storing the cog to which the object belongs, and an *id* parameter, storing the identifier of the object in that cog. The methods *cog*() and *myId*() return those two parameters. A dummy method *get*() that returns `null` is also added to each object, it is used to perform synchronisation on future objects.

Concerning statements, the translation only impacts the statements that deal with object creation, method invocation, or future manipulation. The translation of ABS statements and expressions into MultiASP is shown in Figure 9, and explained below. The code that is generated by the ProActive backend for ABS is similar to what is shown

$$\llbracket x = \texttt{new C}(\overline{e}) \rrbracket \triangleq \begin{array}[t]{l} newcog = \texttt{newActive COG}();\\ id = newcog.freshId();\\ no = \texttt{new C}(\overline{e}, newcog, id);\\ z = newcog.register(no, id);\\ x = no \end{array}$$

$$\llbracket x = \texttt{new local C}(\overline{e}) \rrbracket \triangleq \begin{array}[t]{l} t = \texttt{this}.cog();\\ id = t.freshId();\\ no = \texttt{new C}(\overline{e}, t, id);\\ z = t.register(no, id);\\ x = no \end{array}$$

$$\llbracket x = e!\texttt{m}(\overline{e}) \rrbracket \triangleq \begin{array}[t]{l} t = e.cog(); id = e.myId();\\ x = t.execute(id, \texttt{m}, \overline{e}) \end{array}$$

$$\llbracket x = e.\texttt{m}(\overline{e}) \rrbracket \triangleq \begin{array}[t]{l} t = e.cog(); b = \texttt{this}.cog();\\ \texttt{if}(t == b) \quad \{x = e.\texttt{m}(\overline{e})\}\\ \texttt{else}\ \{\ id = e.myId();\\ \quad x = t.execute(id, \texttt{m}, \overline{e});\\ \quad \textbf{setLimitHard};\\ \quad w = x.get();\\ \quad \textbf{setLimitSoft}\} \end{array}$$

$$\llbracket x = e \rrbracket \triangleq x = e$$

$$\llbracket x = y.get \rrbracket \triangleq \begin{array}[t]{l} \textbf{setLimitHard};\\ w = y.get();\\ \textbf{setLimitSoft};\\ x = y \end{array}$$

$$\llbracket await\ g \rrbracket \triangleq \begin{array}[t]{l} \texttt{if}(\neg g) \quad \{\\ \quad t = \texttt{this}.cog();\\ \quad id = \texttt{this}.myId();\\ \quad z = t.execute\_condition(id, condition\_g, \overline{x});\\ \quad w = z.get()\ \} \end{array}$$

where $\overline{x}$ are the local variables used in g and g contains no future await of the form $y?$

$$\llbracket suspend \rrbracket \triangleq \begin{array}[t]{l} t = \texttt{this}.cog();\\ id = \texttt{this}.myId();\\ z = t.execute\_condition(id, condition\_True, \overline{x});\\ w = z.get() \end{array}$$

$$\llbracket await\ x? \rrbracket \triangleq w = x.get()$$

$$\llbracket await\ g \wedge g' \rrbracket \triangleq \llbracket await\ g \rrbracket; \llbracket await\ g' \rrbracket$$

FIGURE 9. Translational semantics from ABS to MULTIASP

in MULTIASP. We organise the description as follows. First we describe the storage and access to objects, this explains the translation of the two `new` statements. Then, we define the translation of method invocations and particularly of asynchronous method calls. Section 4.1.3 defines synchronisation on futures and cooperative scheduling aspects. Requests are split into three groups: one for executing requests, one for allocating fresh object identifiers, and a third one for registering objects in a *cog*. Groups and compatibility are presented in Section 4.1.4. Section 4.1.5 describes `await` statement on boolean expressions.

4.1.1. *Object addressing.* To allow all objects to be accessible like in ABS, we use a two-level reference system. Each COG is accessible by a global reference and each ABS object is accessible inside its COG through a local identifier. The pair (COG, identifier) is a unique reference for all translated ABS objects and allows the runtime to retrieve any object.

We translate the ABS `new` statement ($\llbracket x = \texttt{new C}(\overline{e}) \rrbracket$), that creates a new object in a new COG, with the instantiation of a COG multi-active object and of the new object in MULTIASP. First, the COG multi-active object is created with the `newActive` primitive. Thus, further method invocations on the *newcog* variable will be asynchronous remote method calls. Then, a fresh identifier is retrieved from the new COG, to create the new object that is stored in a reserved temporary local variable *no*. The new object is a passive object that has a reference to its COG. Then, the new object is referenced in the new COG through the `register` method. Since the new COG is a remote multi-active object, the new object is copied when it is transmitted to the new COG through the `register` invocation. Finally, the `x` variable references the new object that is ready to be used.

For objects that are created with `new local` in ABS ($\llbracket x = \texttt{new local}\ C(\overline{e}) \rrbracket$), the translation in MULTIASP is similar to the translation of the ABS `new` statement, but the local COG is retrieved instead of creating a new one. No new activity is created here and the object is registered locally, without copy.

4.1.2. *Method invocation.* When an asynchronous method call is performed in ABS ($e!\mathtt{m}(\overline{e})$), in MultiASP a remote method invocation is executed. Because of the hierarchical object referencing, we first retrieve the cog of the translated object that is invoked in ABS, as well as the identifier of the object. Then, we perform a generic method call (implicitly asynchronous) named `execute`, on the retrieved cog. The `execute` method of the `COG` class looks up the targeted object through its identifier and runs on it the given method by reflection, with the translated parameters. In MultiASP, an asynchronous method call entails a copy of the invocation parameters. Consequently, several copies of the same translated ABS object exist in MultiASP, whereas only one copy of each object exists in ABS. This is not a problem because only one of these copies is registered and when an object's copy is used in the translated program (i.e. a method is invoked on it[6]), the same process is always applied: the invocation is forwarded to the cog that manages the only registered object. Thus, in the end, only this registered object is manipulated by all the invocations made on the object's copies. Consequently, the behaviour by reference of ABS (and other similar active object languages) can be simulated with the behaviour by copy of MultiASP. The same mechanism is applied to handle the translation of future updates. Also, since the copies of an object are only used to retrieve the registered object, all transmitted objects between cog can be lightweight: they only need to embed the reference to the `COG` and the identifier of the only registered object. In the ProActive backend, we tune the serialisation mechanism so that only those fields are transmitted between activities, saving memory and bandwidth in the execution of the translated program .

For the case of synchronous method calls, in ABS ($!e.\mathtt{m}(\overline{e})$), we distinguish two cases, like in the ABS semantics. Either the call is local and an execution context is pushed in the stack, or the call is remote and we perform an asynchronous method invocation and immediately wait the associated future, blocking the current activity thread (see below).

4.1.3. *Cooperative scheduling.* First, we consider the translation of ABS `await` statements on future variables ($[\![await\ x?]\!]$). In ABS, an await statement on an unresolved future makes the current request release the execution thread. In order to stop the execution of the request in the multi-active object-based translation, we trigger a wait-by-necessity. To this end, we call the dummy *get()* method so that a wait-by-necessity is automatically triggered if the future is unresolved. However, in MultiASP, a wait-by-necessity blocks the thread and does not release it. To encode cooperative scheduling we configure the `COG` class with the following annotations configuring its internal scheduling. Since in the translation, all remote method invocations go through the `execute` method of the `COG` class, we put this method in a multi-active object group. We declare this group self compatible so that several `execute` methods can be scheduled on several threads of the multi-active object at the same time. Then, we set a thread limit of one thread and we use a soft thread limit:

```
@Group(name="scheduling", selfCompatible=true)
@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
```

This configuration first allows a single execute method to run at any time, so that there cannot be more than one running thread in a cog multi-active object. The soft thread limit also allows a thread to process an `execute` request while another `execute` request is passivated, waiting for a future. Finally, as the translation of the await statement triggers

[6]Field access outside the current object is not allowed in ABS and ASP.

a wait-by-necessity, this setting ensures exactly the synchronisation of multi-active object threads that simulates the cooperative scheduling of ABS through the `await` statement.

Second, we consider the translation of ABS `get` statements ($[\![x = y.get]\!]$). In ABS, a `get` statement retrieves a future value, blocking the execution thread if necessary. Compared to `await`, another thread cannot continue executing while waiting for a future through a `get` statement. In the translation, we temporarily go from a soft thread limit to a hard thread limit so that no other thread can start while the current thread is waiting for the future to be resolved. To this end, we use the **setLimitSoft** and **setLimitHard** statements introduced in Section 3.2. In the translation, a **setLimitHard** statement precedes the call to the *get()* method, so that no other thread starts or resumes while performing the `get` operation. Then, the soft thread limit is restored.

4.1.4. *Groups and compatibility.* In general, to simulate the scheduling of ABS executions, we create different method groups in the `COG` class, and each group has its own thread limit:

$$group(freshId) = g_1 \qquad group(execute) = g_2 \qquad group(register) = g_3$$
$$\mathcal{L}_{g_1} = 1 \qquad \mathcal{L}_{g_2} = 1 \qquad \mathcal{L}_{g_3} = \infty$$

The compatibility relationship is defined such that[7]:

$$compatible(q, q')\ \underline{\text{iff}} \left(\begin{array}{l} (q = (\_, freshId, \_) \ \Rightarrow\ q' \neq (\_, freshId, \_)) \ \wedge \\ (\nexists id, x, \mathtt{m}, \overline{e}.\ q = (\_, register, [x, id]) \wedge q' = (\_, execute, [id, \mathtt{m}, \overline{e}])) \end{array} \right)$$

Group g_1 encapsulates *freshId* requests. These requests cannot execute in parallel safely, so g_1 is not self compatible. Group g_2 gathers *execute* requests (i.e. ABS requests). It is limited to one thread to comply with the threading model of ABS, and the requests are self compatible to enable interleaving. Group g_3 contains *register* requests that are self compatible and that have no thread limit. Concerning compatibility between groups, g_1 is compatible with other groups. Concerning g_3 and g_2, their compatibility is defined dynamically such that an *execute* request and a *register* request are compatible if they do not affect the same identifier. `execute_condition` methods are compatible with everything.

4.1.5. `await` *on conditional expressions.* Regarding the translation of the `await` *on conditions* and of the `suspend` statement of ABS, we realise it by the means of an additional multi-active object group and of a special configuration of its thread limit. First, note that futures are single-valued assignments, and once they are available they will remain available. Second, as defined in [25], the ABS `await` statement accepts only monotonic guards and conjunctive composition. Because of monotonicity, the translation of conjunctive ABS conditional guards ($[\![await\ g \wedge g']\!]$) can be performed sequentially. Concerning futures, this can be done by calls to the *get()* method of the translated ABS objects (the activity is in soft limit at this point). Then, in order to translate ABS conditional guards ($[\![await\ g]\!]$), for each guard g, we generate a method *condition_g* that takes as parameters the variables $\overline{x}$ used in g. A condition evaluation g is defined as follows:

$$condition\_g(\overline{x}) = \mathtt{while}(\neg g)\ skip; \mathtt{return\ \ null}$$

In more recent semantics of ABS, guard are not monotonic; to encode this semantics, we would have to check again the boolean part of the condition when the thread is resumed (at this point no other thread can interfere with the object state). Each conjunctive guard

[7] We use _ as a wildcard: $q = (\_, \mathtt{m}, \_)$ means that there is a future and a set of parameters such that q has this form, in other words "q is a request on method m.

should be re-ordered so that it starts by a set of guards on futures (monotonic by nature), and finishes with a boolean expression guard. This boolean part of the await would be the only non-monotonic part that should be checked again when the service thread is resumed.

We translate the ABS `suspend` statement ($[\![suspend]\!]$) similarly to conditional guards, but with a condition that is always true. We define an *execute_condition* method in the `COG` class that generically executes generated condition methods. The *execute_condition* method has its own group with an infinite thread limit because any number of conditions can evaluate in parallel (those requests are compatible with all the other requests):

$$group(execute\_condition) = g_4 \quad \mathcal{L}_{g_4} = \infty$$

4.1.6. *Distribution.* In addition to the translation of active object paradigms, the ABS compiler is slightly modified in order to enable distributed execution of ABS programs. We also make some other adaptations in the generated Java classes to ensure an efficient distributed execution.

Serialisation. The main challenge when moving objects from one memory space to another is to reshape them so that they can be transmitted on the network medium. A serialised version of an object must be created by the sender so that the object graph can be rebuilt from the serialised version by the recipient. As ProActive is based on Java RMI, all objects that are part of a remote method call (i.e. parameters and return values) must implement the Java `Serializable` interface, otherwise a distributed execution throws an exception. Thus, the generated Java classes implement this interface.

Copy Optimisation. To minimise copy overhead, in the ProActive translation we declare the translated class fields with the `transient` Java keyword, which prevents them from being embedded in a serialised version of an object (they are replaced by `null`). The fields that receive a value when the object is created go through a customised serialisation mechanism: they are copied only the first time they are serialised (i.e. when the object is copied into its hosting cog). After this initial copy, we only copy the object identifier and the reference to its cog. We could also represent all the objects as generalised references and store the object's state in a dedicated place. However the translation of local synchronous calls to the original object would become highly inefficient (compared to the current Java method invocation). This would also make the formalisation more complex.

Deployment. Any distributed program needs a deployment specification mechanism in order to place pieces of the program on different machines. ProActive embeds a deployment descriptor that is based on XML configuration files, where the programmer declares physical machines to be mapped to virtual nodes. Such virtual nodes can then be used in the program in order to deploy an active object on them. In our case, as the ProActive program is generated, we have to raise the node specification mechanism at the level of the ABS program. We slightly modified the ABS syntax (and parser) to allow the programmer to specify the name of the node on which a new cog must be deployed. The ProActive backend then links this node name to the deployment descriptor of ProActive. Now, an ABS `new` statement is optionally followed by a string that identifies a node, as follows:

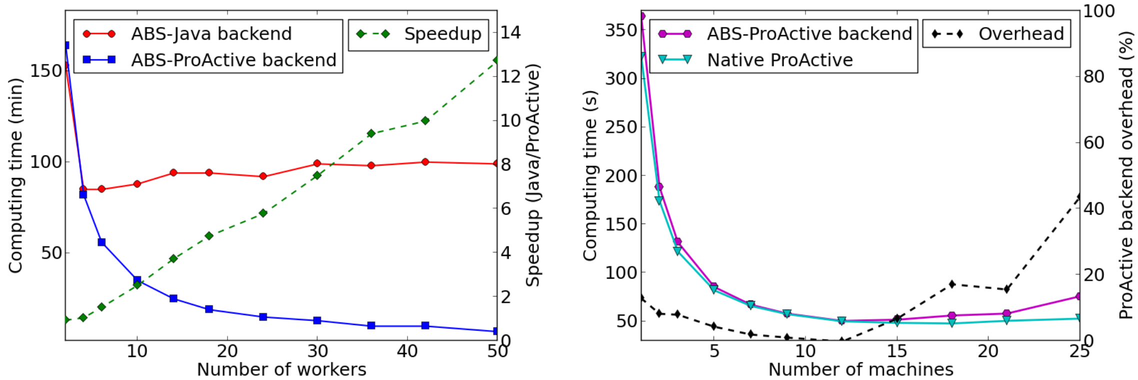
```
Server server = new "mynode" Server();
```

During translation, the ProActive node object corresponding to this node identifier is retrieved, and given as parameter of the `newActive` primitive to deploy the active object on this node. For this specification to work, a simple descriptor file similar to the following one must be created and attached to the ABS program:

```
<GCMDeployment>
  <hosts id="mynode" hostCapacity="1"/>
  <sshGroup hostList="172.16.254.1"/>
</GCMDeployment>
```

In this example, the virtual node `"mynode"` is mapped to the machine with the IP address mentioned in the `hostList` attribute. DNS names can be specified here as well. Many other deployment options can also be defined [6]. If several machines are specified in the `hostList`, then the ProActive backend picks one machine in the list each time a new `COG` is created, in a round robin manner. If no node is specified in ABS, then the new cog is created on the same machine in a different JVM, enforcing a strict isolation of cogs.

4.1.7. *Wrap up and applicability.* In conclusion, by carefully setting multi-active object annotations, and by adapting to distribution requirements, we are able to execute ABS programs in a distributed setting by using ProActive. This translation is automatically handled by the ProActive backend for ABS. The approach presented here and instantiated in the case of ABS and MultiASP could be generalized and applied to other active object languages, and systematically provide a way to deploy and run most active object languages. The effort to port our result to other active object languages depends on the target language. To get an efficient translation of different active object models into distributed multi-active objects, one needs to answer the questions related to the object model and to the scheduling model of the active object language. Most of all, one must carefully consider the location of objects: "Should objects be grouped to preserve the performance of the application? If yes, how and under which control?". When it comes to distributing active objects over several memory spaces, the only scalable solution that we have seen is to address the objects hierarchically. This strategy is easily applicable to any active object language based on object groups. For example, adapting this work to JCoBox should raise no technical difficulty. The most challenging aspect is that in JCoBox the objects share a globally accessible and immutable memory. In this case, the global memory could be translated into an active object that holds all immutable objects: since they are immutable, communicating them by copy is correct. In the case of uniform active object languages, like Creol, creating one active object per translated object handles straightforwardly the translation but limits scalability. The best approach is to group several objects behind a same active object for performance reasons, like building abstract object groups that resemble ABS and JCoBox. In the case of Encore, objects are already separated into active and passive objects, which makes the translation of the object model easier. Once the distributed organisation of objects has been defined, then preserving the semantics of the source language relies on a precise interleaving of local threads, which is possible thanks to the various threading controls offered by multi-active objects. Simulating policies different from cooperative scheduling is also possible with multi-active objects. For example, the transposition of the ProActive backend to AmbientTalk could seem tricky on the scheduling aspect, due to the existence of callbacks. However, a callback on a future can still be considered as a request that is immediately executed in parallel, but starts by a wait-by-necessity on the adequate future.



(A) Against existing solution (time in minutes) (B) Against native code (time in seconds)

FIGURE 10. Execution time of DNA-matching ABS application

4.2. **Experimental Evaluation.** We evaluate the ProActive backend for ABS on several ABS applications and we compare the execution of the ProActive program generated by the ProActive backend to the execution of the program generated by the Java backend for ABS. We first test four example programs given in the ABS tool suite. These examples involve a bank account program that consists of 167 lines of ABS and that creates 3 cogs, a leader election algorithm over a ring (62 lines, 4 cogs), a chat application (324 lines, 5 cogs), and a deadlock example that hangs by circular dependencies between activities (69 lines, 2 cogs). For all examples, we observe that the behaviour of the program translated with the ProActive backend is the same as the one translated with the Java backend. Thus, the scheduling policy enforced in ProActive faithfully respects the one of the reference implementation. These examples run in a few milliseconds, thus they are inadequate for performance analysis of distributed executions.

We focus the rest of the experiments on an application that requires more computational resources: the pattern matching of a DNA sequence in a database. We implement a MapReduce programming approach [18] with one cog per worker. We search a pattern of 250 bytes in a database of 5MB of DNA sequences. We compute the global execution time when varying the number of workers. When executing the program given by the Java backend, we execute it on a single machine; when using the ProActive backend, we deploy two multi-active objects (i.e. two workers) per machine. We use a cluster of the Grid5000 platform [11], where machines have 2 CPUs of 2.6GHz with 2 cores, and 8GB of memory.

Figure 10a shows the execution time of both ProActive and Java translations of the ABS application from 2 to 50 workers. The ProActive backend scales as expected. The execution of the Java program reaches an optimal degree of parallelism for 4 workers (the number of cores of the machine) and then cannot benefit from higher parallelism. With the ProActive backend and 25 machines, the application completes in 19 minutes.

We further evaluate whether the ProActive backend is effective compared to a program that would be directly written in ProActive. Figure 10b compares the execution time of the two implementations of the DNA matching application: the first one is the generated program of the ProActive backend for ABS, and the second one is a hand-written version of the application in ProActive, i.e. executing according to ProActive semantics. Here, the program generated by the ProActive backend has been slightly modified:

we have manually replaced the translation of functional ABS types (integers, booleans, lists, and maps) with the corresponding standard Java types. Indeed, the implementation of the functional layer of ABS in Java is not as efficient as primitive Java types. Considering that our point is to evaluate the additional communication cost of the ProActive translation, we do not take into account the implementation of ABS types in Java. The performance is approximately 30 times better depending on the implementation of types. This experiment can be split into two parts: when the number of machines grows up to 12 some speed-up can be noticed in both implementations, while above 12 machines the time spent in communication becomes too important compared to the computation time, and no significant performance improvement is obtained when adding more machines. While the amount of computation performed by each peer is significant, the overhead introduced by the ProActive backend is rather low compared to a native version of the application, since it is overall under 10%. At a larger scale, the generated version is characterised by additional communications to access intermediate objects and thus the translated application suffers even more from the time wasted in communication; it becomes less efficient.

To conclude, with the ProActive backend, one can run efficiently high-performance distributed application written in ABS and benefit from distribution compared to the Java backend. We conducted experiments with a new backend for Java 8 on the same applications and reached a similar conclusion. This comparison is based on a "high-performance computing" application. An application relying on more thread interleaving, and coordination of interleaving threads would probably be less efficient in the ProActive backend.

4.3. **Formal Study of the Translation.** This section investigates the equivalence between ABS programs and their translation in MultiASP. We present two theorems stating under which conditions the ABS program and its MultiASP translation are equivalent. More precisely, we have one theorem for each direction of the simulation. We achieve the proofs of the theorems in terms of weak simulation in the two cases. We formally prove that, except from some reasonable differences between the two languages, the translated program behaves exactly like the original one. The differences are mostly due to the distributed nature of the execution, and to the fact that futures have a different semantics in the two languages. It would have been possible to overcome those differences by providing an encoding that is more artificial and less shallow (adding intermediate objects or threads) but this would have prevented us from drawing conclusions about the similarities and differences between the languages. Figure 11 recalls the runtime syntax of ABS. It defines the structure of ABS runtime configurations. Compared to MultiASP configurations, the structure of objects and futures are relatively similar except that each object has an entry in the configuration and thus active objects and futures do not have a local store; additionally request queues consist of already instantiated method bodies. A *cog* element is used to denote which object of each cog is the active one.

The semantics of ABS is shown in Figure 12, it uses a function bind and an evaluation function $[\![\,]\!]$ similar to the ones of MultiASP. It uses two reserved variables in the local stores: *destiny* stores the identifier of the future for which the current computation is performed; *cont* stores the identifier of the future of the request that was interrupted in order to perform a local synchronous method invocation; this request should be recovered upon reaching the `return` statement. The rest of the semantics is quite similar to [24]. The asynchronous invocation enqueues a new request and creates a new futures. Two rules exist for creating an object inside the same *cog*, or inside a newly created one. When reaching an

$$
\begin{array}{rcl}
cn & ::= & \epsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, a, p, \overline{q}) \mid \mathit{cog}(c, \mathit{act}) \mid cn\ cn \\
p & ::= & q \mid \texttt{idle} \\
q & ::= & \{l \mid s\} \\
v & ::= & o \mid f \mid \texttt{null} \mid \mathit{primitive\text{-}val}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{act} & ::= & o \mid \varepsilon \\
\mathit{val} & ::= & v \mid \bot \\
a & ::= & \overline{x \mapsto v}
\end{array}
$$

FIGURE 11. Runtime syntax of ABS.

`await` statement that releases the current thread, the interrupted request is re-scheduled: it is inserted at any point in the request queue.[8] This semantics ensures that the order of incoming requests is kept unchanged while the interrupted ones are re-schedule at any point in the future. This encodes the FIFO ordering of service requests discussed below.

Compared to the semantics of ABS appearing in [24], the following changes have been made: requests are organised in queues to enforce FIFO ordering of requests (see below); *return* statements terminate the execution of the current method like in mainstream programming languages; continuation is handled as a reserved variable, like *destiny* and not as a special statements, this was necessary because of the new semantics for *return*. Our objective is to establish a correspondence between the configurations of ABS and the ones of the program translated in MULTIASP. The proofs of lemmas and theorems presented in this section can be found in appendix.

To reason on the equivalence between ABS and MULTIASP configurations, we adopt the following conventions on object locations and activity names. We identify activity names of MULTIASP with COG names (ranged over by α, β). Also, object locations are valid only locally in MULTIASP and globally in ABS, their equivalent global reference is the pair (activity name, identifier). We suppose that each object name in ABS is of the form i_α where α is the name of the COG where the object is created, and where i is unique locally in activity α. The semantics then allows us to choose the MULTIASP identifier of the created object such that it is equal to the desired location "i".

4.3.1. *Restrictions of the translation.* The proof of correctness of the translation is valid under four specific restrictions. We detail them below.

(1) **Causal ordering of requests** (applies in both directions of the equivalence). MULTIASP ensures causal ordering of communications with a rendez-vous that precedes all asynchronous method calls: the request *is dropped* in the remote request queue *synchronously*. This brief synchronisation does not exist in ABS where requests can arrive in any order. The difference between the modes of message transmission being unrelated to our study, we constrain the semantics of ABS so that it uses the same communication timing as MULTIASP. Thus in the semantics of Figure 12, the message sending and reception rule of the original ABS is replaced by the RENDEZ-VOUS-COMM rule.
(2) **FIFO service of requests** (applies in both directions of the equivalence). In MULTIASP, while thread activation can happen in any order, the order in which requests are served is FIFO by default instead of the non-deterministic activation of a thread featured by ABS semantics. In both the Java backend and the PROACTIVE backend, activation and request service are FIFO, although PROACTIVE supports the definition of different policies through multi-active object annotations [32]. Consequently, we only reason on executions that enforce a FIFO policy, i.e. executions

[8] $\mathit{Schedule}(p, \overline{q})$ is a function that inserts p at any position inside $\overline{q}$

(Skip)
$$ob(o,a,\{l \mid \texttt{skip};s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l \mid s\},\overline{q})$$

(Assign-Local)
$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^A_{(a+l)}}{ob(o,a,\{l \mid x = e;s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l[x \mapsto v] \mid s\},\overline{q})}$$

(Assign-Field)
$$\frac{x \in \mathrm{dom}(a) \setminus \mathrm{dom}(l) \quad v = [\![e]\!]^A_{(a+l)}}{ob(o,a,\{l \mid x = e;s\},\overline{q}) \overset{A}{\to} ob(o,a[x \mapsto v],\{l \mid s\},\overline{q})}$$

(Cond-True)
$$\frac{\texttt{true} = [\![e]\!]^A_{(a+l)}}{ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l \mid s_1;s\},\overline{q})}$$

(Cond-False)
$$\frac{\texttt{false} = [\![e]\!]^A_{(a+l)}}{ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l \mid s_2;s\},\overline{q})}$$

(Await-True)
$$\frac{f = [\![x]\!]^A_{(a+l)} \quad v \neq \bot}{ob(o,a,\{l \mid \texttt{await}\ x\,?;s\},\overline{q})\ \mathit{fut}(f,v) \overset{A}{\to} ob(o,a,\{l \mid s\},\overline{q})\ \mathit{fut}(f,v)}$$

(Await-False)
$$\frac{f = [\![x]\!]^A_{(a+l)} \quad \mathit{Schedule}(\{l \mid \texttt{await}\ x\,?;s\},\overline{q}) = \overline{q'}}{ob(o,a,\{l \mid \texttt{await}\ x\,?;s\},\overline{q})\ \mathit{fut}(f,\bot) \overset{A}{\to} ob(o,a,\texttt{idle},\overline{q'})\ \mathit{fut}(f,\bot)}$$

(Release-Cog)
$$ob(o,a,\texttt{idle},\overline{q})\ \mathit{cog}(c,o) \overset{A}{\to} ob(o,a,\texttt{idle},\overline{q})\ \mathit{cog}(c,\epsilon)$$

(Activate)
$$\frac{c = a(\mathrm{cog})}{ob(o,a,\texttt{idle},\{l \mid s\} :: \overline{q})\ \mathit{cog}(c,\epsilon) \overset{A}{\to} ob(o,a,\{l \mid s\},\overline{q})\ \mathit{cog}(c,o)}$$

(Read-Fut)
$$\frac{f = [\![e]\!]^A_{(a+l)} \quad v \neq \bot}{ob(o,a,\{l \mid x = e.\texttt{get};s\},\overline{q})\ \mathit{fut}(f,v) \overset{A}{\to} ob(o,a,\{l \mid x = v;s\},\overline{q})\ \mathit{fut}(f,v)}$$

(New-Object)
$$\frac{o' = \mathrm{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad a' = [\overline{x \mapsto v}, \mathrm{cog} \mapsto c]}{ob(o,a,\{l \mid x = \texttt{new local C}(\overline{e});s\},\overline{q})\ \mathit{cog}(c,o) \overset{A}{\to} ob(o,a,\{l \mid x = o';s\},\overline{q})\ \mathit{cog}(c,o)\ ob(o',a',\texttt{idle},\varnothing)}$$

(New-Cog-Object)
$$\frac{c' = \mathrm{fresh}(\,) \quad o' = \mathrm{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad a' = [\overline{x \mapsto v}, \mathrm{cog} \mapsto c']}{ob(o,a,\{l \mid x = \texttt{new C}(\overline{e});s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l \mid x = o';s\},\overline{q})\ ob(o',a',\texttt{idle},\varnothing) \quad \mathit{cog}(c',\epsilon)}$$

(Rendez-vous-Comm)
$$\frac{f = \mathrm{fresh}(\,) \quad o' = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad q'' = \mathrm{bind}(o',f,m,\overline{v},\texttt{class}(o'))}{ob(o,a,\{l \mid x = e!\texttt{m}(\overline{e});s\},\overline{q})\ ob(o',a',p,\overline{q'}) \overset{A}{\to} ob(o,a,\{l \mid x = f;s\},\overline{q})\ ob(o',a',p,\overline{q'} :: q'')\ \mathit{fut}(f,\bot)}$$

(Context)
$$\frac{cn \overset{A}{\to} cn'}{cn\ cn'' \overset{A}{\to} cn'\ cn''}$$

(Cog-Sync-Call)
$$\frac{o' = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad f = \mathrm{fresh}(\,) \qquad c = a'(\mathrm{cog}) \quad f' = l(\mathrm{destiny}) \qquad \{l' \mid s'\} = \mathrm{bind}(o',f,\texttt{m},\overline{v},\mathit{class}(o'))}{ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e});s\},\overline{q})\ ob(o',a',\texttt{idle},\overline{q'})\ \mathit{cog}(c,o) \overset{A}{\to} ob(o,a,\texttt{idle},\overline{q} :: \{l \mid \texttt{await}\ f?;x = f.\texttt{get};s\})\ \mathit{fut}(f,\bot)\ ob(o',a',\{l'[\mathit{cont} \mapsto f'] \mid s'\},\overline{q'})\ \mathit{cog}(c,o')}$$

(Cog-Sync-Return-Sched)
$$\frac{v = [\![e]\!]^A_{(a+l)} \quad c = a'(\mathrm{cog}) \qquad f = l'(\mathrm{destiny}) \quad l(\texttt{cont}) = f}{ob(o,a,\{l \mid \texttt{return}\ e;s\},\overline{q})\ \mathit{cog}(c,o)\ ob(o',a',\texttt{idle},\overline{q'} :: \{l' \mid s\} :: \overline{q''})\ \mathit{fut}(f,\bot) \overset{A}{\to} ob(o,a,\texttt{idle},\overline{q})\ \mathit{cog}(c,o')\ ob(o',a',\{l' \mid s\},\overline{q'} :: \overline{q''})\ \mathit{fut}(f,v)}$$

(Self-Sync-Call)
$$\frac{f' = l(\mathrm{destiny}) \quad o = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \qquad f = \mathrm{fresh}(\,) \quad \{l' \mid s'\} = \mathrm{bind}(o,f,\texttt{m},\overline{v},\mathit{class}(o))}{ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e});s\},\overline{q}) \overset{A}{\to} ob(o,a,\{l'[\mathit{cont} \mapsto f'] \mid s'\},\overline{q} :: \{l \mid \texttt{await}\ f?;x = f.\texttt{get};s\})\ \mathit{fut}(f,\bot)}$$

(Rem-Sync-Call)
$$\frac{o' = [\![e]\!]^A_{(a+l)} \quad f = \mathrm{fresh}(\,) \quad a(\mathrm{cog}) \neq a'(\mathrm{cog}) \qquad q'' = \mathrm{bind}(o',f,m,\overline{v},\texttt{class}(o'))}{ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e});s\},\overline{q})\ ob(o',a',p,\overline{q'}) \overset{A}{\to} ob(o,a,\{l \mid f = e!\texttt{m}(\overline{e});x = f.\texttt{get};s\},\overline{q})\ ob(o',a',p,\overline{q'} :: q'')\ \mathit{fut}(f,\bot)}$$

(Return)
$$\frac{v = [\![e]\!]^A_{(a+l)} \quad f = l(\mathrm{destiny}) \quad \mathit{cont} \notin \mathrm{dom}(l)}{ob(o,a,\{l \mid \texttt{return}\ e;s\},\overline{q})\ \mathit{fut}(f,\bot) \overset{A}{\to} ob(o,a,\texttt{idle},\overline{q})\ \mathit{fut}(f,v)}$$

(Self-Sync-Return-Sched)
$$\frac{v = [\![e]\!]^A_{(a+l)} \quad f = l'(\mathrm{destiny}) \quad l(\texttt{cont}) = f}{ob(o,a,\{l \mid \texttt{return}\ e;s\},\overline{q} :: \{l' \mid s\} :: \overline{q'})\ \mathit{fut}(f,\bot) \overset{A}{\to} ob(o,a,\{l' \mid s\},\overline{q} :: \overline{q'})\ \mathit{fut}(f,v)}$$

Figure 12. Semantics of ABS.

that serve requests and resume them in a FIFO order. The semantics of Figure 12 enforces such a FIFO service of incoming requests, while interleaving the service of interrupted requests in a non-deterministic manner.

(3) **Absence of 'futures of futures'** (applies in the direction from ABS to MultiASP). The semantics of futures is very different in the two languages and must be handled carefully. Firstly, a variable holding a pointer to a future object in MultiASP is equivalent to the same variable holding directly the future reference in ABS (several subsequent references might be followed this way). This is because the MultiASP semantics use additional locations to handle transparent futures. Secondly, the equivalence can follow future references in ABS. This is because future update occurs transparently in MultiASP while in ABS they only happen upon a `get` operation. Thus in the MultiASP configuration, more future updates might have been performed compared to the ABS configuration. However following (future or location) references is not always sufficient. Indeed, consider the ABS configurations (i) $fut(f, f')$ $fut(f', \bot)$ and the configuration (ii) $fut(f, \bot)$; they are observationally different, whereas in MultiASP they are not because any synchronisation on the future f will synchronise on the future f' too. Consequently we restrict ourselves to *programs that do not create futures of futures*, i.e. that cannot create terms of the form $fut(f, f')$ in any reachable configuration. The transparency of futures and of future updates creates an intrinsic an unavoidable difference between the two active object languages. However, this is not a major restriction on expressiveness because it is still possible to have a wrapper for futures values: a future value that is an object containing a future. Such a wrapper could be created by the translation but this would break the one-to-one mapping between ABS and MultiASP futures.

(4) **Forward-based distributed future updates** (applies in the direction from MultiASP to ABS). The distributed future update mechanism of MultiASP cannot be strictly faithfully represented in ABS. The problem arises when futures are transmitted between activities, for example when a future is a parameter of a request sent to another activity. In this case in MultiASP, two proxies for the same future will exist in the store of the two activities, whereas only one centralised future exists in ABS. This situation can create intermediate states in MultiASP where the future value is available but not completely propagated to all the locations where the future has been transmitted. This can create behaviours that are not possible to reach in ABS, because a future update is atomically made available to all activities referencing the future. This difference is only observable when the flow of method invocations races with the flow of future updates. Enforcing an atomic future update in MultiASP would directly solve the issue but this is not realistic in a distributed setting. The solution we advocate for handling the potential inconsistency in future resolution is related to the future update mechanism of ProActive: the value of a future is transmitted between activities along the same chain as the future was transmitted originally. We extend the causally-ordered communications to the transmission of future values (instead of just having causally ordered request sending). This prevents any observable inconsistency in the future updates and simulates faithfully the ABS behaviour. This solution does not require many additional synchronisations; it is reasonable and relevant in a distributed context.

Restrictions (1) and (2) already exist in the Java backend for ABS. And we can notice that these differences are more related to request scheduling policies and to communication channels than to the nature of the two active object languages. Also, restriction (4) is related to distributed systems, and any distributed translation would have to deal with some inconsistency in the observability of a future result. In the end, restriction (3) is the only one that gives a real insight on the differences of the two active object languages, because it is related to way the futures are designed and handled in the languages.

4.3.2. *Equivalence relation.* We define an equivalence relation $\approx$ between MultiASP and ABS terms. This equivalence relation aims at proving that any single step of one calculus can be simulated by a sequence of steps in the other. The notion of observational equivalence is a bit similar to the proof in [16]. In particular, we can use the same observation notion: processes are observed based on remote method invocations. The equivalence relation $\approx$ consists of three parts: equivalence of values, equivalence of statements, and equivalence of configurations. In the following, we use the notation cn for an ABS configuration and $\underline{cn}$ for a MultiASP configuration.

Definition 4.1 (Equivalence of values)**.** $\approx_\sigma^{cn}$ is an equivalence relation between values (or between a value and a storable), in the context of a MultiASP store σ and of an ABS configuration cn, such that:

$$v \approx_\sigma^{cn} v \qquad\qquad f \approx_\sigma^{cn} f \qquad\qquad i\_\alpha \approx_\sigma^{cn} [cog \mapsto \alpha, Id \mapsto i, \overline{x \mapsto v'}]$$

$$\frac{v \approx_\sigma^{cn} \sigma(o)}{v \approx_\sigma^{cn} o} \qquad\qquad \frac{fut(f, v') \in cn \qquad v' \approx_\sigma^{cn} v}{f \approx_\sigma^{cn} v}$$

Runtime values in ABS are either object references, future references, or primitive values. The equivalence relation $\approx_\sigma^{cn}$ specifies that two values or futures are equivalent if they are the same. An object is only characterised by its identifier and its cog name. The two last cases are more interesting and reflect the difference between the future update mechanisms of ABS and MultiASP. First, the equivalence can follow as many local indirections in the MultiASP store as necessary. Second, the equivalence can follow future references in ABS, because a future can be updated transparently in MultiASP while in ABS the explicit future read has not occurred yet.

Definition 4.2 (Equivalence of statements)**.**
$s \approx_{(\sigma+\ell')}^{cn} s'$ $\underline{\text{iff}}$ either $[\![s]\!] = s'$ or $s = (x = v; s_1) \ \wedge\ s' = (x = e; [\![s_1]\!])$ *with* $v \approx_\sigma^{cn} [\![e]\!]_{(\sigma+\ell')}$.

Two MultiASP and ABS statements are equivalent if one is the translation of the other, or if both start with an assignment of equivalent values to the same variable, followed by an ABS statement on one side and its translation on the MultiASP side.

Finally, we define an equivalence between ABS and MultiASP configurations; note that the following definition only deals with MultiASP configurations that are obtained when evaluating the translation ob an ABS program.

Definition 4.3 (Equivalence of configurations)**.** The ABS configuration cn and the MultiASP configuration $\underline{cn}$ are equivalent, written $cn \approx \underline{cn}$, iff the three condition of Figure 13 hold.

In more details, the equivalence of Figure 13 globally considers three cases:

– The first five lines deal with equivalence of cogs. This case compares both activity content and activity requests on the ABS and MultiASP sides:

- To compare objects (Lines `1a-1c`), we rely on the fact that activities have the same name in ABS and in MultiASP. Each ABS object ob must correspond to one MultiASP object in the equivalent activity α. The object equivalent to ob must (i)

(1a) $\forall \alpha.\quad \exists a.\, cog(\alpha,a) \in cn \ \underline{\text{iff}}\ \exists o_\alpha,\sigma,p,Rq.\, act(\alpha,o_\alpha,\sigma,p,Rq) \in \underline{cn}\ \underline{\text{with}}\ \forall i.$

(1b) $\exists \overline{v},p',\overline{q}.ob(i\_\alpha,\overrightarrow{x\mapsto v},p',\overline{q}) \in cn\ \underline{\text{iff}}\ \exists o,\overline{v'}.\,\sigma(o)=[cog\mapsto\alpha, myId\mapsto i, \overrightarrow{x\mapsto v'}]\ \underline{\text{with}}$

(1c) $\overline{v} \approx^{cn}_{\sigma} \overline{v'}\ \wedge$

(1d) $\begin{pmatrix}\exists l,s.\,p'=\{l|s\}\ \underline{\text{iff}}\ \exists f,i,\mathtt{m},\overline{v''},\ell',s',\ell'',s''.\big((f,execute,i,\mathtt{m},\overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \\ \wedge\ \ell'(\mathtt{this})=o\big)\ \underline{\text{with}}\ \forall x\in dom(l)\backslash destiny.\, l(x)\approx^{cn}_{\sigma}\ell'(x) \wedge l(destiny)=f) \wedge s \approx^{cn}_{\sigma+\ell'} s'\end{pmatrix} \wedge$

(1e) $\forall f.\begin{pmatrix}\big(\exists l,s.(\{l|s\}\in\overline{q}\wedge l(destiny)=f)\ \underline{\text{iff}} \\ \exists i,\mathtt{m},\overline{v''},\ell',s',\ell'',s''.\Big(\big((f,execute,i,\mathtt{m},\overline{v''})_P\mapsto\{\ell'|s'\}::\{\ell''|s''\}\in p\wedge\ell'(\mathtt{this})=o\big) \\ \vee\ \big((f,execute,i,\mathtt{m},\overline{v''})\in Rq\wedge o_\alpha.retrieve(i)=o\wedge \mathrm{bind}(o,\mathtt{m},\overline{v''})=\{\ell'|s'\}\big)\Big) \\ \underline{\text{with}}\quad (\forall x\in\mathrm{dom}(l)\backslash\{destiny\}.\, l(x)\approx^{cn}_{\sigma}\ell'(x))\wedge s\approx^{cn}_{\sigma+\ell'} s'\big)\end{pmatrix}$

(2) $\forall f.\quad \exists v.fut(f,v)\in cn\ \underline{\text{iff}}\ \exists v',\sigma.\,(\textsc{fut}(f,v',\sigma)\in\underline{cn}\wedge \mathrm{Method}(f)=execute)\ \underline{\text{with}}\ v\approx^{cn}_{\sigma} v'$

(3) $\forall f.\quad fut(f,\bot)\in cn\ \underline{\text{iff}}\ \textsc{fut}(f,\bot)\in\underline{cn}\wedge\mathrm{Method}(f)=execute$

FIGURE 13. Equivalence between ABS and MULTIASP configurations. Method(f) returns the method of the request corresponding to future f. We use the notation: $\exists y.\,P$ iff $\exists x.\,Q$ with R, meaning: $(\exists y.\,P$ iff $\exists x.\,Q)\wedge \forall x,y.\,P\wedge Q\Rightarrow R$. This allows R to refer to x and y.[9]

be in the store, (ii) reference α as its COG, (iii) have i as identifier (the corresponding ABS identifier is i_α), and (iv) have the other fields equivalent to the ones of *ob*.

- To compare the requests, we compare the tasks that exist in ABS *ob* terms to the tasks that exist in the corresponding MULTIASP *act* terms. We consider two cases:

  (1) Concerning the active task (Line 1d): the single active task of *ob* in ABS (in p') must have exactly one equivalent active task in MULTIASP (in p). In MULTIASP, this task must have two elements in the current stack[10]: the call to the COG (the *execute* call) and the stacked call that redirected to the targeted object o, where o is equivalent to *ob*. In addition, values of local variables must be equivalent, except *destiny* that must correspond to the future of the MULTIASP request. Finally, the current thread of the two tasks must be equivalent according to the equivalence on statement.
  (2) Concerning the inactive tasks (Line 1e), two cases are possible. Either the task has already started and has been interrupted: it is passive and the comparison is similar to the active task case. Or the task has not started yet: there must be a corresponding entry in the request queue *Rq* of the MULTIASP active object α; additionally (i) computed futures must be equivalent, (ii) invoked objects must be equivalent, and (iii) the method body built by *bind* must be equivalent to the ABS task. It is important to notice that the order of the request queue *Rq* of the MULTIASP object is the same as the part of the ABS queue $\overline{q}$ that corresponds to the second case, i.e. not yet started requests.

– Line 2 deals with the equivalence of resolved futures. A future value in MULTIASP refers to its local store. Two resolved futures are equivalent if their values are equivalent. In

[9]By abuse of notation and for readability, we flatten the arguments of the request and write $(f,execute,i,\mathtt{m},\overline{v''})$ instead of $(f,execute,(i::\mathtt{m}::\overline{v''}))$.

[10]The proof only deals with asynchronous invocations, and thus the stack never contains more than two elements. The synchronous invocations are identical to the Java backend and can be studied in this context.

MultiASP, only futures from *execute* method calls are considered, because they represent the applicative method calls.
– Line 3 states that the same futures must be unresolved in both configurations.

Overall, the association of the equivalence of values (Definition 4.1), the equivalence of statements (Definition 4.2), and the equivalence of ABS and MultiASP configurations (Definition 4.3), forms the global equivalence relation $\approx$. We rely on equivalence $\approx$ to prove that our translation of ABS programs into MultiASP is correct.

4.3.3. *Preliminary Lemmas.* Before stating the two theorems ensuring the correctness of the translation, we provide some basic properties of ABS and of the equivalence relation.

Lemma 4.1 (Activated object in ABS). *In an* ABS *configuration, if an object ob has a non-idle active request, then there exists a* cog *in which ob is the current active object.*

$$ob(i\_\alpha, \overrightarrow{x \mapsto v}, p, \overline{q}) \in cn \wedge p \neq \textbf{idle} \Rightarrow cog(\alpha, i\_\alpha) \in cn$$

Lemma 4.2 (Equivalence of values). $v \approx_{\sigma}^{cn} v' \Rightarrow v \approx_{\sigma}^{cn} [\![v']\!]$

Lemma 4.3 (Equivalence of evaluation functions). *Let cn be an* ABS *configuration and suppose* $cn \approx \underline{cn}$. *Let* $ob(o\_\alpha, a, \{l|s\}, \overline{q}) \in cn$. *By definition of* $\approx$, *there exists a single activity* $\text{ACT}(\alpha, o, \sigma, p, Rq) \in \underline{cn}$, *with* $\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, a']$ *and* $(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\mathtt{this}) = o$. *For any* ABS *expression e we have:* $[\![e]\!]_{a+l}^{A} \approx_{\sigma}^{cn} [\![e]\!]_{\sigma+\ell'}$

The serialisation mechanism, and the renaming of local references, are crucial points of difference between ABS and MultiASP. Lemma 4.4 deals with these aspects.

Lemma 4.4 (Equivalence after serialisation and renaming).

$$v \approx_{\sigma}^{cn} v' \wedge \sigma' = serialise(\sigma, v') \Rightarrow v \approx_{\sigma'}^{cn} v'$$
$$\overline{v} \approx_{\sigma}^{cn} \overline{v'} \wedge (\overline{v''}, \sigma') = rename_{\sigma}(\overline{v'}, \sigma) \Rightarrow \overline{v} \approx_{\sigma'}^{cn} \overline{v''}$$

Besides the lemmas above, to reason on the properties of the translation, we rely on the fact that all ABS objects are locally registered, in MultiASP, in the active object encoding their cog, and we define an invariant for that:

Invariant Reg. *For every activity* α *such that* o_α *is the location of the active object of activity* α *in its store* σ_α, *if the current task is an invocation to* $o_\alpha.retrieve(i)$, *then this invocation succeeds because the object has been registered first. The invocation returns some object* o' *such that* $i\_\alpha \approx_{\sigma}^{cn} o'$.

4.3.4. *Properties of the translation.* To prove the correctness of the translation from ABS to MultiASP, we prove two theorems. The two theorems exactly specify under which conditions each semantics simulates the other. We first define formally the initial configuration. Let $P = \overline{IC}\ \{\overline{x}\ s\}$ be an ABS program, let cn_0 be the corresponding initial ABS configuration: $ob(start, \varnothing, p, \varnothing)$, where the process p is the activation of the program main block: $p = \{\overrightarrow{x \mapsto \mathtt{null}}|s\}$. $[\![P]\!]$ is the MultiASP program obtained by translation. Its initial MultiASP configuration is: $\text{ACT}(\alpha_0, o, \sigma_0[o \mapsto \varnothing], q_0 \mapsto \{\overrightarrow{x \mapsto \mathtt{null}}|[\![s]\!]\}, \varnothing)$. It is easy to see that this initial configuration has the same behaviour as
$\text{ACT}(\alpha_0, o, \sigma_0[o \mapsto \varnothing, start \mapsto \varnothing], (f, execute, start, \mathtt{m}) \mapsto \{\overrightarrow{x \mapsto \mathtt{null}}|[\![s]\!]\} :: \{\varnothing|x = \bullet\}, \varnothing)$.

We denote this new MultiASP initial configuration $\underline{cn_0}$ and notice that $cn_0 \approx \underline{cn_0}$.

Theorem 4.5 (ABS to MultiASP)**.** *The translation simulates all* ABS *executions with FIFO policy and rendez-vous communications, provided that no future value is a reference to another future.*

$$cn_0 \xrightarrow{\mathrm{A}}{}^{*} cn \wedge \nexists f, f'.\, fut(f, f') \in cn \;\Rightarrow\; \exists \underline{cn}.\, \underline{cn_0} \to^{*} \underline{cn} \;\wedge\; cn \approx \underline{cn}$$

Theorem 4.6 (MultiASP to ABS)**.** *Any reduction of the* MultiASP *translation corresponds to a valid* ABS *execution.*

$$\underline{cn_0} \to^{*} \underline{cn} \Rightarrow \exists cn.\, cn_0 \xrightarrow{\mathrm{A}}{}^{*} cn \wedge cn \approx \underline{cn}$$

The proof of the first theorem can be found in appendix, this proof is a classical case analysis on the ABS reduction rule used for evaluating the current configuration (we do an induction on the reduction). Table 1 summarises the most informative part of the proof of Theorem 4.5. It shows which MultiASP rule (second column) is used to simulate which ABS rule (first column. The third column shows additional non-observable steps that are introduced in the MultiASP translation. These additional steps are needed to reach an equivalent configuration but they are always local and they never introduce concurrency. They typically deal with object registration, access to intermediate objects, or scheduling inside the *cog* object. As expected, the proof case for the Read-Fut rule uses the fact that "futures of futures" are forbidden. Lemmas 4.3 and 4.4 are used frequently to ensure the equivalence of the configurations, and in particular the equivalence between a direct object reference of ABS and the local MultiASP object representing an object registered in another *cog*.

A detailed proof sketch for Theorem 4.6 can also be found in appendix. In this proof the equivalence relation ≈ has to be adapted because of the existence of additional (deterministic) statements in the translation. Table 2 summarises this second proof by showing which ABS rule is used to simulate which MultiASP rule. Again, in some cases, a few (non-observed) additional ABS rules are applied to reach an equivalent configuration.

4.4. **Discussion on the Translation and its Properties.** In the translation, ABS requests, cogs, and futures respectively match MultiASP requests, active objects, and futures. For each ABS object there exist several copies of this object in MultiASP; all copies share the same value for the field cog and identifier, but only the copy that is hosted in the right cog/activity is equivalent to the ABS object. This forms a shallow translation: applicative requests, futures, active objects, and object fields are mapped faithfully by the same notion in MultiASP as in ABS. The main difference is that all requests transit by the cog active object that acts as a local scheduler; "execute" requests have no counterpart in ABS. In the equivalence relation, the objects are identified by their identifier and their cog name, and the equivalence can follow futures. The equivalence between requests distinguishes two cases. First, for active tasks, there is a single active task per cog in ABS and it must correspond to the single active thread serving an *execute* request in MultiASP. Second, inactive tasks in ABS correspond either to passive requests being currently interrupted or to requests that have not been served yet in MultiASP. For each request that has started its execution, the second element in the stack of method calls corresponds to the invoked request, and the equivalence of executed statements, of local variables, and of the corresponding future is verified.

In both directions, we prove a weak simulation relation with additional non-observable steps dealing with the intermediate structures created by the translation. Additional steps

ABS rule	**MultiASP rule**	**Additional MultiASP rules**
Assign-Local	Assign-Local	–
Assign-Field	Assign-Field	–
Await-True	–	Update / – , Invk-Passive, Return-Local, Assign-Local-Tmp
Await-False	Invk-Future	–
Release-Cog	–	–
Activate	–	Activate-Thread / (Serve, Invk-Passive, Return-Local, Assign-Local-Tmp)
Read-Fut	–	Set-Hard-Limit, Update / – , Invk-Passive, Return-Local, Set-Soft-Limit, Assign-Local-Tmp
New-Object	New-Object	Invk-Passive, Assign-Local-Tmp Return-Local
New-Cog-Object	New-Object	New-Active,Assign-Local-Tmp, Invk-Active-Meta, Return
Rendez-vous-Comm	Invk-Active	Invk-Passive, Return-Passive, Assign-Local-Tmp
Return	Return	Return-Local, Assign-Local-Tmp

Table 1. Summary of the simulation of ABS in MultiASP. Assign-Local-Tmp is Assign-Local on a variable introduced by the translation. Invk-Active-Meta is Invk-Active on a method that is not *execute*.

MultiASP rule	**ABS rule**	**Additional ABS rules**
Assign-Local	Assign-Local	–
Assign-Field	Assign-Field	–
Invk-Future	Await-False	Release-Cog
Invk-Passive	–	–/Read-Fut/Await-True
Activate-Thread	Activate	–
Serve	Activate	–
Invk-Active	Rendez-vous-Comm	–
New-Object in an activity with no ABS object	New-Cog-Object	–
New-Object in a non-empty activity	New-Object	–
Return	Return	–
Others	–	–

Table 2. Summary table of the simulation of MultiASP in ABS.

also ensure equivalence concerning future updates. However, our results are stronger than the standard guarantees given by a weak simulation because the added steps do not introduce concurrency, the silent actions are always confluent. Overall, most of the properties provided by ABS tools like absence of deadlock or resource consumption are guaranteed to be preserved by our translation. The most striking example of an observable reduction in ABS that is not observable in MultiASP is the update of a future with its computed value. Indeed, the transparent creation and update of futures creates an intrinsic difference between the two programming languages. This is why, in the first theorem, we exclude the possibility for a future value to be itself a future. Also, concerning assignments, only the ones concerning ABS local variables can be observed in MultiASP, assignments of temporary variables introduced by the translation have no counterpart in ABS. Concerning asynchronous method invocations, only the applicative ABS requests are observable.

Identifying the differences of observability between active object languages gives a significant insight on their design and differences. We could observe the crucial differences between the language because we chose a faithful translation that matches most of the elements of ABS and MultiASP configurations in a one-to-one way. The divergent notions of the two languages can be spotted easily thanks to our shallow translation.

4.5. **Related Works and Discussion on the Execution of Active Object Programs.** Using a backend allows to decouple the language for specifying and verifying the program from the execution language, and makes it possible to explore easily new programming paradigms. Such a design enables writing programs and proving properties on a high-level language, and relying on massively used platforms for implementation.

A closely related ongoing work [42] aims at implementing the ABS semantics with the paradigms of Java 8, using a lightweight thread continuation mechanism (representative of cooperative scheduling). This work makes a particular focus on efficiency, as opposed to the seminal Java backend for ABS. Preliminary results showed that the Java 8 backend for ABS scales much better than the existing Java backendconcerning the number of threads co-allocated on the same machine. However, to the best of our knowledge, this backend is still under development and does not support distributed execution.

In parallel with this work, ABS has been extended with *deployment components* [8] which are specific objects representing the locations where new cogs must be deployed. Deployment components are specific objects of the ABS language. They are used to reason on the deployment of the application: they represent an abstraction of the deployment location and enable the reasoning on the distributed nature of the application. Instead we use *virtual nodes* expressing deployment in the ABS syntax similarly to the ProActive approach. Those two contrary design decisions reveal the difference of nature between the two languages concerning deployment. The distribution system of ABS is meant to reason on object distribution. On the contrary, ProActive virtual nodes are meant to be used in conjunction with external deployment tools. ProActive virtual nodes are binders to entries in a deployment descriptor file. The expressiveness in terms of distribution of ProActive is closer to languages like Akka where deployment locations are strings. Deployment components can be manipulated and stored like any other object in ABS and cannot be translated simply into ProActive virtual nodes that are only strings pointing to a specific entry in a deployment file. It would be possible to use some specific MultiASP active objects to represent deployment components, but the relation with the normal *cog* objects would need to be specified both in practice, involving more development, and from a theoretical point of view, a new equivalence relation would be needed.

The Haskell backend [8] for ABS, also focuses on a distributed execution of ABS programs. As opposed to Java-based backends, lightweight thread continuations are natively available in Haskell, which makes the Haskell backend for ABS very efficient even with a high degree of local parallelism: much more cogs can be hosted on the same machine than with programs generated by the ProActive backend. In this work, the ABS compiler is extended to integrate the notion of deployment components within the ABS language. The Haskell backend for ABS also has support for garbage collection of distributed objects, built on top of the Haskell garbage collector. In ProActive, activation of distributed garbage collection is optional, but it is less crucial since all passive objects are garbage collected by the JVM. Nevertheless, to the best of our knowledge, the ProActive backend is the only backend that generates a distributed and executable code formally proven to be correct with respect to the ABS semantics.

Our backend suffers from several performance limitations inherent to the approach and to the conceptual differences between the languages. First, in some cases, many threads might be blocked in a wait-by-necessity state and this could be improved in several ways. Currently, the ProActive library reuses a blocked thread to serve a new request if it is sure that the re-used thread cannot be re-activated while the new request is served: a request re-uses a blocked thread if this request will resolve the future on which the thread is blocked. In other cases, the nature of Java (non-preemptive with heavy threads) prevents us from doing further optimisations without relying on a heavier compilation phase, e.g. using continuations, and without breaking the direct correspondence between the two languages, and the simplicity of the object language translation. Another approach could consist in creating more active objects, i.e. one per ABS object, but, as discussed in the introduction of this section, this would make the synchronisation between the different objects tricky. Finally, the `await` statement on a boolean expression relies on a busy-wait polling which can be highly inefficient, a wait-notify translation could be more efficient provided the monitoring of field mutation is done efficiently [4]. Wait-notify is already used to implement wait-by-necessity in ProActive, it is thus used by `await` operations on futures. We however do not use wait-notify for the computation of boolean expressions because this would require to explicitly do a wait (upon evaluation of the expression) and a notify (upon modification of the related variables). Wait and notify are not ASP primitives and do not follow the active-object paradigm, we cannot use them in the current approach.

These limitations illustrate the different approaches of the two languages: ProActive library was designed to compose distributed applications made of heavy threads with relatively few synchronisation points. ABS was designed to model parallel applications with many synchronisations and no problem of distribution. ProActive shines for implementing distributed applications typical of high-performance computing: intensive computations that feature complex synchronisations but not many threads and synchronisation points per machine. Consequently our backend is very efficient for applications that do not do many cooperative thread-release or many conditional awaits. Indeed, we showed that such HPC applications can be run efficiently with the ProActive backend.

5. Conclusion

This paper presents a framework for multi-active objects, featuring programmer-friendly concurrent programming, advanced request scheduling mechanisms, and development support. We give the guidelines of the programming model usage through practical applications. The implementation of multi-active objects given by ProActive offers a set of annotations that allow the programmer to control the multi-threaded execution of active objects. The formalisation of the ProActive library in the MultiASP programming language provides an operational semantics to reason on multi-active object executions. Priorities and thread management, combined with multi-active object compatibilities, reveal to be convenient to encode scheduling patterns. This article presents the ProActive backend for ABS. This backend automatically transforms ABS models into distributed applications. We formalise the translation and prove its correctness. We establish an equivalence relation between ABS and MultiASP configurations, and we prove two theorems that corroborate the correctness of the translation, illustrating the conceptual differences between the two languages.

Overall, we attach a particular interest to three objectives: usability, correctness, and performance of our framework. We address the complete spectrum of the multi-active object programming model, from design to execution. Testing our developments in realistic settings assesses the efficiency of our implementation. On the other hand, our work is formalised and we highlight the properties of our model. Thus, we believe that we reinforce the guarantees offered by the multi-active object programming model, on which the programmer can rely.

In the future, we want to investigate the verification of compatibility annotations, but also the dynamic tuning of thread management aspects where the thread scheduler could adjust at runtime the number of threads allocated depending on the status of the machine and of the active object.

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). We would like to thank the reviewers of this article for their thorough work and constructive comments.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static analyzer for concurrent objects. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, pages 562–567. Springer Berlin Heidelberg, 2014.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A cost and termination analyzer for ABS. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 151–154, New York, NY, USA, 2012. ACM.

[4] K. Azadbakht, N. Bezirgiannis, and F. S. de Boer. Distributed network generation based on preferential attachment in ABS. In *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*, pages 103–115, 2017.

[5] K. Azadbakht, F. S. de Boer, and V. Serbanescu. Multi-threaded actors. In M. Bartoletti, L. Henrio, S. Knight, and H. Torres-Vieira, editors, *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, volume 223 of *EPTCS*, pages 51–66, 2016.

[6] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid, pages 205–229. Springer, London, 2006.

[7] P. A. Bernstein and S. Bykov. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing*, 20(5):71–75, 2016.

[8] N. Bezirgiannis and F. Boer. ABS: A high-level modeling language for cloud-aware programming. In M. R. Freivalds, G. Engels, and B. Catania, editors, *SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science.*, pages 433–444, Berlin, Heidelberg, 2016. Springer.

[9] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, Oct. 2017.

[10] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In M. Bernardo and B. E. Johnsen, editors, *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy.*, pages 1–56, Cham, 2015. Springer International Publishing.

[11] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing*, pages 8 pp.–, Nov 2005.

[12] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer Publishing Company, Incorporated, 1st edition, 2005.

[13] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134, New York, NY, USA, 2004. ACM.

[14] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the*, pages 3–12, Nov 2007.

[16] M. Dam and K. Palmskog. Location-independent routing in process network overlays. *Serv. Oriented Comput. Appl.*, 9(3-4):285–309, Sept. 2015.

[17] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 316–330, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[19] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In P. A. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany.*, pages 517–526, Cham, 2015. Springer International Publishing.

[21] C. C. Din, S. L. Tapia Tarifa, R. Hähnle, and E. B. Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In M. Butler, S. Conchon, and F. Zaïdi, editors, *Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France.*, pages 217–233, Cham, 2015. Springer International Publishing.

[22] K. Fernandez-Reyes, D. Clarke, and D. S. McCain. ParT: an asynchronous parallel abstraction. In *Coordination Models and Languages, 18th International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Crete, Greece, June 6-9, 2016. Proceedings*, 2016.

[23] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, Jan. 1999.

[24] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *Software & Systems Modeling*, pages 1–36, 2015.

[25] R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects: 11th International Symposium, Bertinoro, Italy, 2012, Revised Lectures*, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, Feb. 2009.

[27] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[28] M. Haustein and K.-P. Löhr. JAC: declarative Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[29] Y. Hayduk, A. Sobe, D. Harmanci, P. Marlier, and P. Felber. *Speculative Concurrent Processing with Transactional Memory in the Actor Model*. Springer International Publishing, Cham, 2013.

[30] L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In R. Nicola and C. Julien, editors, *Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques.*, pages 90–104. Springer Berlin Heidelberg, 2013.

[31] L. Henrio, M. U. Khan, N. Ranaldo, and E. Zimeo. First class futures: Specification and implementation of update strategies. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 295–303, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] L. Henrio and J. Rochas. Declarative scheduling for active objects. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1339–1344, New York, NY, USA, 2014. ACM.

[33] L. Henrio and J. Rochas. From modelling to systematic deployment of distributed active objects. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 208–226, 2016.

[34] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[35] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of the 9th International Conference on Formal Methods for Components and Objects*, FMCO'10, pages 142–164, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, Nov. 2006.

[37] E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67 – 91, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011).

[38] R. G. Lavender and D. C. Schmidt. Pattern languages of program design 2. chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[39] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, Nov. 2006.

[40] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] C. Scholliers, E. Tanter, and W. De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 2014.

[42] V. Serbanescu, K. Azadbakht, F. de Boer, C. Nagarajagowda, and B. Nobakht. A design pattern for optimizations in data intensive applications using ABS and Java 8. *Concurrency and Computation: Practice and Experience*, 28(2):374–385, 2016. cpe.3480.

[43] M. Sirjani, Movaghar, and M. R. Mousavi. Compositional verification of an object-based model for reactive systems. In *Computer Science of Iran Computer Conf., 11th Intl. Conf., CSICC, Tehran, Iran.* 2001.

[44] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.

[45] D. Wyatt. *Akka Concurrency*. Artima, 2013.

[46] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. In *Proceedings of the conference on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM.

Appendix

The following appendix contains the proofs of the lemmas and theorems stated in the article.

Appendix A. Proofs of Lemmas

Lemma 4.1, page 34 (Correspondence of activated objects). In an ABS configuration, if an object ob has an active request that is not `idle`, then there exists a COG in which ob is the current active object.

$$ob(i\_\alpha, \overrightarrow{x \mapsto v}, p, \overline{q}) \in cn \wedge p \neq \texttt{idle} \Rightarrow cog(\alpha, i\_\alpha) \in cn$$

Proof sketch for Lemma 4.1. The proof is done by induction on the ABS reduction rules. □

Lemma 4.2, page 34 (Equivalence of values).

$$v \approx^{cn}_{\sigma} v' \Rightarrow v \approx^{cn}_{\sigma} [\![v']\!]$$

Proof of Lemma 4.2. We prove this lemma by recurrence on the proof of $v \approx^{cn}_{\sigma} v'$ (Definition 4.1). This corresponds to a recurrence on the number of indirections of references that are followed to evaluate v'. We consider three cases:

- If v is a primitive value (case $v \approx^{cn}_{\sigma} v$), then $v' = v$ and the equivalence $v \approx^{cn}_{\sigma} v$ is direct because $v = [\![v]\!]$.
- If v is not a primitive value ($\dfrac{v \approx^{cn}_{\sigma} \sigma(o)}{v \approx^{cn}_{\sigma} o}$), then $v' = o$ and $v \approx^{cn}_{\sigma} \sigma(o)$. Therefore, we need to handle three cases:
  - $\sigma(o) = f$. In this case, since $[\![o]\!] = o$ by the evaluation function of Figure 6, then $v \approx^{cn}_{\sigma} o$.
  - $\sigma(o) = [\overrightarrow{x \mapsto v}]$. In this case, since the evaluation of o is also equal to o, we also have $v \approx^{cn}_{\sigma} o$.
  - $\sigma(o) = v''$. In this case, since $v \approx^{cn}_{\sigma} v''$, we have $v \approx^{cn}_{\sigma} [\![v'']\!]$ by recurrence hypothesis. Then $v \approx^{cn}_{\sigma} [\![o]\!] = [\![\sigma(o)]\!] = [\![v'']\!]$ by the evaluation function of Figure 6.
- The other cases of Definition 4.1 are not applicable here, because they are not a valid MultiASP value (e.g. a future is not a MultiASP value).

□

Lemma 4.3, page 34 (Equivalence of evaluation functions). Let cn be an ABS configuration and suppose $cn \approx \underline{cn}$. Let $ob(o\_\alpha, a, \{l|s\}, \overline{q}) \in cn$. By definition of $\approx$, there exists a single activity $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$, with $\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, a']$ and $(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\mathtt{this}) = o$. For any ABS expression e we have:

$$[\![e]\!]^{A}_{a+l} \approx^{cn}_{\sigma} [\![e]\!]_{\sigma+\ell'}$$

Proof of Lemma 4.3. We only consider here the case where x is a variable, because the other cases are not different from what is done in the Java backend for ABS. In particular, the evaluation of arithmetic expressions relies on the equivalence of variables and on the fact that the evaluation of the arithmetic expression itself is the same.

We prove the equivalence of evaluation functions on variables by case analysis.

- Either the variable belongs to the fields of the object $ob(o\_\alpha, a, \{l|s\}, \overline{q})$, and not to the local variables: $a(x) = v \wedge v \notin \text{dom}(l)$. In this case, we have $ob(o\_\alpha, a, \{l|s\}, \overline{q}) \in cn$ with $x \mapsto v \in a$. By definition of equivalence, we have $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$, $\exists (f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in dom(l) \backslash destiny$, $l(x) = v \approx_\sigma^{cn} \ell'(x) = v'$ by last assertion of Line `1e` of equivalence condition, and $\sigma(o) = [mycog \mapsto \alpha, myId \mapsto i, a']$ by Line `1b`. Also by definition of equivalence, $a'(x) = v'$ and $v \approx_\sigma^{cn} v'$, because σ is the store of α. We have $[\![x]\!]_{\sigma+\ell'} = [\![v']\!]$ by the definition of the evaluation function of Figure 6, with $\ell'(\mathtt{this}) = o$ by the equivalence condition of Figure 13 (because $x \notin \text{dom}(l)$ and hence, $x \notin \text{dom}(\ell')$). Thus we have $v \approx_\sigma^{cn} [\![x]\!]_{\sigma+\ell'} = [\![v']\!]_{\sigma+\ell'}$ by Lemma 4.2.
- Or the variable belongs to local variables: $l(x) = v$. In this case, we have $ob(o\_\alpha, a, \{l|s\}, \overline{q}) \in cn$ with $l(x) = v$ and, by definition of equivalence, $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$, $\exists (f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in \text{dom}(l) \backslash destiny$, $l(x) = v \approx_\sigma^{cn} \ell'(x) = v'$ by last condition of Line `1e` of equivalence relation. Finally, $[\![x]\!]_{a+l}^A \approx_\sigma^{cn} \ell'(x)$ and by Lemma 4.2: $[\![x]\!]_{a+l}^A \approx_\sigma^{cn} [\![\ell'(x)]\!]_{\sigma+\ell'}$.

□

Lemma 4.4, page 34 (Equivalence after serialisation and renaming).

$$v \approx_\sigma^{cn} v' \wedge \sigma' = \text{serialise}(\sigma, v') \Rightarrow v \approx_{\sigma'}^{cn} v'$$
$$\overline{v} \approx_\sigma^{cn} \overline{v'} \wedge (\overline{v''}, \sigma') = \text{rename}_\sigma(\overline{v'}, \sigma) \Rightarrow \overline{v} \approx_{\sigma'}^{cn} \overline{v''}$$

Proof sketch for Lemma 4.4. The proof relies mostly on the definition of equivalence ≈. Concerning serialisation, the main difference is the fact that we check the equivalence in a smaller (or equal) store. However, this store is self-contained and every reference that could be followed by the equivalence always exist in the serialised store. Concerning renaming, the proof is trivial as only the case where the references are followed is changed and the renaming ensures the equivalence of the reached values. □

Invariant Reg, page 34. *For every activity α such that o_α is the location of the active object of activity α in its store σ_α, if the current task is an invocation to $o_\alpha.retrieve(i)$, then this invocation succeeds because the object has been registered first. The invocation returns some object o' such that $i\_\alpha \approx_\sigma^{cn} o'$.*

Proof of Invariant Reg. To prove the **Invariant Reg** we rely on the translational semantics provided in Figure 9. Suppose that a call to $o_\alpha.retrieve(i)$ is performed. We want to show that this call succeeds and returns an object equivalent to i_α. If we have such a call, it means that we are in the body of an $execute(id, \mathtt{m}, \overline{parameters})$ call, with $id = i$. Since the *execute* method is a reserved method name, we know that we are executing a remote method call, corresponding to an expression: $e.\mathtt{m}(\overline{parameters})$ with $e.cog() = \alpha$ and $e.myId() = id$. Also, we have e such that $[\![e]\!]_{\sigma_\beta+\ell} = o$ and $\sigma_\beta(o) = [cog \mapsto \alpha, myId \mapsto id, \cdots]$. Then, we know that we cannot have method calls on the temporary variable *no* that is used to temporarily store a newly created object until it is registered to its COG, because this variable is only known in the instantiation thread and no method call is performed on it. Thus, since e cannot be *no*, it is necessarily a variable x that has been assigned after a call to $t.register(no)$

(where $t \mapsto \alpha$) that makes the COG t aware of the new object referenced by no. Consequently, the call to *register* precedes the remote call $e.\mathtt{m}(\overline{parameters})$[11]. Now, we have two cases.

- Either the *register* call is local because it comes with a new local object instantiation (with the `new local` ABS keyword). In this case, since the *register* call is synchronous, it is necessarily finished before $e.\mathtt{m}(\overline{parameters})$ is invoked, so the later $o_\alpha.retrieve(i)$ succeeds naturally.
- Or the *register* call is remote because it comes with a new remote object instantiation (with the `new` ABS keyword). In this case, since the *register* call is asynchronous, it may happen that the *register* and the *execute* requests are both awaiting in the queue. However, thanks to causal ordering and to the FIFO service policy featured by MultiASP, we are guaranteed that the *register* request is in the queue before the *execute* request, and that the *register* request will be served first. In addition, considering the multi-active compatibilities defined for the groups of *register* and *execute*, we see that an incompatibility is raised between the two requests because they access the same identifier. Therefore, the $o_\alpha.retrieve(i)$ request cannot be served before the *register* request finishes, and thus, the $o_\alpha.retrieve(i)$ that is performed in the *execute* request succeeds.

Finally, in both cases we have: $i\_\alpha \approx_\sigma^{cn} o'$. □

Appendix B. Proofs of Theorems

B.1. From ABS to MultiASP.

Theorem 4.5, page 35 (ABS to MultiASP). The translation simulates all ABS executions with FIFO policy and rendez-vous communications, provided that no future value is a future reference.

$$cn_0 \overset{A}{\to}^* cn \wedge \nexists f, f'. fut(f, f') \in cn \;\Rightarrow\; \exists \underline{cn}.\, \underline{cn_0} \to^* \underline{cn} \;\wedge\; cn \approx \underline{cn}$$

Proof of Theorem 4.5. Firstly, we look at the starting configuration. Let $P = \overline{IC}\ \{\overline{x}\ s\}$ be an ABS program, let cn_0 be the initial configuration corresponding to this ABS program: $ob(start, \varnothing, p, \varnothing)$, where the process p corresponds to the activation of the program's main block: $p = \{l|s\}$. The initial MultiASP configuration corresponding to program $[\![P]\!]$ is: $\textsc{act}(\alpha_0, o, [o \mapsto \varnothing], q_0 \mapsto \{l|[\![s]\!]\}, \varnothing)$. It is easy to see that this initial configuration has the same behaviour as: $\textsc{act}(\alpha_0, o, [o \mapsto \varnothing, start \mapsto [cog \mapsto \alpha_0, myId \mapsto 0],\ (f, execute, start, \mathtt{m}) \mapsto \{l|[\![s]\!]\} :: \{\varnothing|x = \bullet\}, \varnothing)$. We denote this new MultiASP initial configuration $\underline{cn_0}$. This way, we have the ABS initial configuration cn_0 that is equivalent, according to $\approx$, to the MultiASP configuration $\underline{cn_0}$: $cn_0 \approx \underline{cn_0}$.

Secondly, Theorem 4.5 is proven by induction on the ABS reduction rules. It relies on the definition of equivalence $\approx$ shown in Figure 13, and on the following induction step: If $cn_0 \overset{A}{\to}^* cn$, $cn \approx \underline{cn}$, $cn \overset{A}{\to} cn'$, and $\nexists f, f'. fut(f, f') \in cn'$, then $\exists \underline{cn'}.\ \underline{cn} \to^* \underline{cn'} \;\wedge\; cn' \approx \underline{cn'}$. First, note that in the definition of equivalence between values and between statements ($\approx_\sigma^{cn}$), the ABS configuration cn is only used to track futures. Thus, in all the cases except Return, $\approx_\sigma^{cn}$ and $\approx_\sigma^{cn'}$ are the same. Each case is concluded by arguments on observability. We are interested in observing remote method invocations, return of asynchronous method calls, assignments to field variables, and assignments to local variables that were not introduced by

[11]We also use the fact that a pointer could not be forged.

the translation into MultiASP. Other reduction rules are considered to be not observable and will be used transparently as many times as necessary to simulate any ABS reduction. There is a discussion about silent transitions and matching between rules at the end of the proof.

1) **Case of the [Assign-Local] ABS rule.**

We first consider the ABS Assign-Local rule, to show that we can observe equivalent configurations on ABS and MultiASP side after reduction. In this rule, a particular ABS configuration cn leads to a configuration cn', noted $cn \xrightarrow{A} cn'$, as follows:

$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^{A}_{(a+l)}}{ob(i\_\alpha, a, \{l \mid x = e; s\}, \overline{q}) \xrightarrow{A} ob(i\_\alpha, a, \{l[x \mapsto v] \mid s\}, \overline{q})}$$

Suppose that Cn is also related to a MultiASP configuration $\underline{cn}$ by the definition of equivalence, i.e., $cn \approx \underline{cn}$. Then, we want to show that a configuration $\underline{cn'}$, equivalent to cn', can be obtained by MultiASP semantics from $\underline{cn}$. In the case of Assign-Local rule, it means that a local variable x must exist in $\underline{cn'}$ and that its value after assignment must be equivalent to the value assigned to it in ABS. To begin with, by Lemma 4.1 there must be a cog in the ABS starting configuration where i_α is the current active object: $\exists cog(\alpha, i\_\alpha) \in cn$. Then, by definition of the equivalence relation $\approx$, we have the following key points:

- First, there exists a corresponding MultiASP activity in the MultiASP starting configuration: $\exists o_\alpha, \sigma, p, Rq.\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$.
- Second, this activity has an active request initiated from an *execute* method call, and maps to a current statement to execute: $\exists f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''.((f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p$ with $(x = e; s) \approx^{cn}_{\sigma} s'$ and $\forall x \in \mathrm{dom}(l) \smallsetminus destiny.l(x) \approx^{cn}_{\sigma} \ell(x)$. Thus, by the definition of equivalence between statements, either $s' = (x = e; [\![s]\!])$, or $e = v$ (because ABS statement is $x = v; s$) and $s' = (x = e'; [\![s]\!])$ with $v \approx^{cn}_{\sigma} [\![e']\!]_{\sigma+\ell'}$. In both cases, there is e' such that $s' = (x = e'; [\![s]\!])66$

Those key points allow us to find the starting MultiASP configuration $\underline{cn}$ and to apply the MultiASP Assign-Local reduction rule:

$$\frac{x \in \mathrm{dom}(\ell') \qquad v' = [\![e']\!]_{(\sigma+\ell')}}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'[x \mapsto v'] \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

We have now to verify that the obtained MultiASP configuration cn' is in relation $\approx$ with the ABS configuration $\underline{cn'}$. We only consider the terms that changed in the reduction. First, we must have at least as many corresponding terms in MultiASP as in ABS. In this case, we have $cog(\alpha, i\_\alpha) \in cn'$ on one hand, and $\textsc{act}(\alpha, o, \sigma, p'', Rq) \in \underline{cn'}$ on the other hand. Second, we must check equivalence of each element. We have on the ABS side the term $ob(i\_\alpha, a, \{l[x \mapsto v]|s\}, \overline{q})$ and on the MultiASP side the activity α which has in particular $q_A \mapsto \{\ell'[x \mapsto v']|[\![s]\!]\} :: \{\ell''|s''\}$ in the method execution stack, with $v' = [\![e']\!]_{\sigma+\ell'}$. Then, to have $v \approx^{cn'}_{\sigma} v'$ we must have $v \approx^{cn'}_{\sigma} [\![e']\!]_{\sigma+\ell'}$. Now, we have two cases:

(1) either $e' = e$ by definition and therefore, by Lemma 4.3: $[\![e]\!]^{A}_{a+l} \approx^{cn'}_{\sigma} [\![e]\!]_{\sigma+\ell'}$.
(2) or $v \approx^{cn'}_{\sigma} [\![e]\!]_{\sigma+\ell'}$ and $[\![e']\!]_{\sigma+\ell'} = v'$.

Both cases lead to the fact that the assigned values are equivalent. The rest of the elements of the activity are unchanged except the remaining statements, but in this case we have $s \approx^{cn'}_{\sigma+\ell'} [[s]]$ by definition of equivalence.

2) **Case of the [Assign-Field] ABS rule.**

Like the previous case, suppose $cn \xrightarrow{A} cn'$ with the Assign-Field ABS reduction rule:

$$\frac{x \in \mathrm{dom}(a) \smallsetminus \mathrm{dom}(l) \quad v = [[e]]^{A}_{(a+l)}}{ob(i\_\alpha, a, \{l \mid x = e; s\}, \overline{q}) \xrightarrow{A} ob(i\_\alpha, a[x \mapsto v], \{l \mid s\}, \overline{q})}$$

With the same strategy as the previous case, namely by having first $cn \approx \underline{cn}$, then $\underline{cn} \to \underline{cn'}$ and finally $cn' \approx \underline{cn'}$, we want to show that an object field x exists in $\underline{cn'}$ and that its value after assignment is equivalent to the value assigned to it in ABS. First, we have again by Lemma 4.1, $\exists cog(\alpha, i\_\alpha) \in cn$. Second, by definition of the equivalence relation $\approx$, we have:

- A corresponding MultiASP activity: $\exists o_\alpha, \sigma, p, Rq.\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$.
- Values in the store σ that are equivalent to the values contained in a: $\exists o, \overline{v'}.\sigma(o) = [\,cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$ where $\overline{v} \approx^{cn}_{\sigma} \overline{v'}$ and $\overrightarrow{x \mapsto v} = a$. Consequently, $x \in \mathrm{dom}(\sigma(o))$ because $x \in \mathrm{dom}(a)$.
- A statement currently executed that belongs to the current active request of the activity: $\exists f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''$ such that: $(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell' | s'\} :: \{\ell'' | s''\} \in p$ and $\ell'(\mathtt{this}) = o$ with $(x = e; s) \approx^{cn}_{\sigma} s'$. Like the previous case, either $s' = (x = e; [\![s]\!])$ and $e' = e$, or $e = v$ and $s' = (x = e'; [\![s]\!])$ with $v \approx^{cn}_{\sigma} [\![e']\!]_{\sigma+\ell'}$.

Then, this leads to the following MultiASP Assign-Field rule:

$$\frac{\ell'(\mathtt{this}) = o \qquad x \in \mathrm{dom}(\sigma(o)) \qquad x \notin \mathrm{dom}(\ell') \qquad \sigma' = \sigma[o \mapsto (\sigma(o)[x \mapsto [[e']]_{(\sigma+\ell')}])]}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \to \textsc{act}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

Like in the previous case, we must now check that $cn' \approx \underline{cn'}$. First, we have $cog(\alpha, i\_\alpha) \in cn'$ and $\textsc{act}(\alpha, o_\alpha, \sigma', p'', Rq) \in \underline{cn'}$, which are equivalent terms. Second, we have to compare the content of $ob(i\_\alpha, a[x \mapsto v], \{l \mid s\}, q)$ to the content of activity α with $\sigma(o) = [\,cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v}[x \mapsto [[e']]_{\sigma+\ell'}]]$. Then, we must have $\overline{v} \approx^{cn'}_{\sigma} \overline{v'}$, which is unchanged except for the particular value v that must be equivalent to the evaluation of e'. Then, similarly to the Assign-Local case, we have again $v \approx^{cn'}_{\sigma} [[e']]_{\sigma+\ell'}$. Finally, we have $s \approx^{cn'}_{\sigma+\ell'} [[s]]$ by the definition of equivalence $\approx$.

3) **Case of the [Await-True] ABS rule.**

We start from the ABS Await-True reduction rule, in which $cn \xrightarrow{A} cn'$:

$$\frac{f = [[x]]^{A}_{(a+l)} \quad v \neq \bot}{ob(i\_\alpha, a, \{l \mid \mathtt{await}\ x\,?; s\}, \overline{q})\ fut(f, v) \xrightarrow{A} ob(i\_\alpha, a, \{l \mid s\}, \overline{q})\ fut(f, v)}$$

From the configuration cn, we know that, by Lemma 4.1, there exists in ABS α such that $cog(\alpha, i\_\alpha) \in cn$. We also know, by equivalence relation $\approx$ and by translational semantics, that there exist in MultiASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$. Besides, there exist ℓ', s', ℓ'', s'' such that, by translational semantics: $\{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \in p$. Also, we know that there are v', σ' such that $\textsc{fut}(f, v', \sigma') \in \underline{cn}$, with $v \approx^{cn}_{\sigma'} v'$. Thus, we have the following MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.get(); [[s]]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, v', \sigma')$$

By Lemma 4.3, we have $[\![x]\!]^{A}_{(a+l)} \approx^{cn}_{\sigma} [\![x]\!]_{(\sigma+\ell')}$, and, by case analysis on the definition of $\approx^{cn}_{\sigma}$ (Definition 4.1), we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, two cases are possible in MULTIASP: either $\sigma(o) = f$ (fourth case and second case of the definition of $\approx^{cn}_{\sigma}$), or $\sigma(o) = v_f$ where $v \approx^{cn}_{\sigma} v_f$ (fourth case and fifth case of the definition of $\approx^{cn}_{\sigma}$):

- In the first case, where o points to a future, we first perform a future update on the MULTIASP configuration $\underline{cn}$ (the point here is to 'catch up' with the ABS execution). Thus, we have $\underline{cn} \to \underline{cn'}$ where in $\underline{cn'}$ activity α is: $\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)$ with $\sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r$ and $(v_r, \sigma_r) = \text{rename}_\sigma(v', \sigma')$. After that, the current local method call $(f.get())$ can be performed. Consequently, after applying a local invocation and a local assignment, we have a MULTIASP configuration $\underline{cn''}$, such that $\underline{cn'} \to^* \underline{cn''}$, as follows:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell'[w \mapsto v_r] \mid [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, v', \sigma')$$

  Now, we must compare cn' and $\underline{cn''}$ to prove that they are equivalent. The only change concerns the value pointed to by the variable x. Suppose x is a field of the current object (the case where x is a local variable is similar). To prove Line `1c` of the definition of equivalence (Figure 13), we must ensure that $f \approx^{cn'}_{\sigma''} v_r$. Indeed, by definition of $\approx^{cn'}_{\sigma''}$, we need to have $\sigma(o) = v_r$ and also $v \approx^{cn'}_{\sigma''} v_r$. In our case, this is true because first, $v \approx^{cn'}_{\sigma'} v'$ and second, by Lemma 4.4, we have $v \approx^{cn'}_{\sigma'} v' \,\wedge\, (v_r, \sigma_r) = \text{rename}_{\sigma'}(v', \sigma') \Rightarrow v \approx^{cn'}_{\sigma_r} v_r$ with $\sigma_r \subseteq \sigma''$. Regarding activity α, the elements to be considered are the local variables and the statements; the other elements did not change in the reduction. MULTIASP configuration $\underline{cn'}$ contains an additional local variable w that comes from the *get* invocation; this local variable is not used. The other local variables did not change, which preserves equivalence. Finally, the remaining statements in ABS and in MULTIASP are guaranteed to be equivalent by the fact that $s \approx^{cn'}_{\sigma''+\ell'} [\![s]\!]$, according to the definition of equivalence.
- In the second case, where o points to a value, the future has already been updated in the past. Consequently, no preliminary future update is necessary and the last steps of the first case can be performed in the same way, but directly with $\underline{cn}$ instead of $\underline{cn'}$ and with v_f instead of v_r.

4) **Case of the [Await-False] ABS rule.**

We start from the ABS AWAIT-FALSE reduction rule, where $cn \xrightarrow{A} cn'$:

$$\frac{f = [\![x]\!]^{A}_{(a+l)} \quad Schedule(\{l \mid \mathtt{await}\, x\,?; s\}, \overline{q}) = \overline{q'}}{ob(o, a, \{l \mid \mathtt{await}\, x\,?; s\}, \overline{q})\ fut(f, \bot) \xrightarrow{A} ob(o, a, \mathtt{idle}, \overline{q'})\ fut(f, \bot)}$$

By Lemma 4.1, we know that there exists in ABS α such that $cog(\alpha, i\_\alpha) \in cn$. By the definition of equivalence $\approx$, we also know that there exist in MULTIASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$. Furthermore, we have $\textsc{fut}(f, \bot) \in \underline{cn}$ and, by translational semantics, we have: $\exists \ell', \ell'', s''.\{q_A \mapsto \{\ell' \mid w = [\![x]\!].get(); [\![s]\!]\} :: \overline{\{\ell'' \mid s''\}}\} \in p$. Thus, we have the following MULTIASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)$$

By Lemma 4.3, we have $[\![x]\!]^{A}_{(a+l)} \approx^{cn}_{\sigma} [\![x]\!]_{(\sigma+\ell')}$. By case analysis on the definition of $\approx^{cn}_{\sigma}$, we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, only one case is possible in MULTIASP: $\sigma(o) = f$, according to the fourth and second case of the definition of $\approx^{cn}_{\sigma}$. Indeed, no other case is possible because the future has not been resolved yet. Thus, the current statement of activity α is in fact a local method call to $x.get()$, where $\sigma(o) = f$. Since f has not been resolved yet in MultiASP neither, the current request of activity α switches to a passive status (q_P) by the MultiASP rule Invk-Future[12]. Thus, we have the following MultiASP configuration $\underline{cn'}$, such that $\underline{cn} \to \underline{cn'}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)$$

Now we have to prove $cn' \approx \underline{cn'}$. In MultiASP, the only element that changed in activity α is the status of the current request, from q_A to q_P. This corresponds to the fact that the current task became `idle` in ABS. Indeed, in the definition of equivalence Figure 13, as soon as there is an ABS term $\{l \mid s\}$ in p, there must be a corresponding active thread in MultiASP (Line `1d`). Similarly, passive threads in MultiASP correspond to some of the requests that are in the queue in ABS. Thus, the passivation of the MultiASP thread corresponds exactly to the transfer of the task to the request queue in ABS. Then, the equivalence of the two tasks is trivial, because $\mathtt{await}\, x; s \approx_\sigma^{cn'} w = x.get(); [\![s]\!]$. Note that the ordering of the not-yet-served requests is unchanged, both on the ABS side and on the MultiASP side.

5) **Case of the [Release-Cog] ABS rule.**

We have the following ABS reduction $cn \xrightarrow{\text{A}} cn'$ with the Release-Cog rule:

$$ob(i\_\alpha, a, \mathtt{idle}, \overline{q})\ cog(\alpha, i\_\alpha) \xrightarrow{\text{A}} ob(i\_\alpha, a, \mathtt{idle}, \overline{q})\ cog(\alpha, \epsilon)$$

From configuration cn, by definition of equivalence $\approx$, we know that there exist in MultiASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq)$ is in the corresponding MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$. Since, in cn, the current task is `idle` and all other objects in COG α are `idle` too (we know that by Lemma 4.1), there exist in MultiASP no $f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p$. In other words, there is no active *execute* request in the set p of executed requests of activity α. In this case, we directly have $cn' \approx \underline{cn}$, because the only term that changed from cn to cn' is the *cog* term, and the state of *cog* terms is not relevant in the definition of the equivalence $\approx$.

6) **Case of the [Activate] ABS rule.**

We have the following ABS reduction $cn \xrightarrow{\text{A}} cn'$ with the Activate rule:

$$\frac{c = a(\text{cog})}{ob(o, a, \mathtt{idle}, \{l \mid s\} :: \overline{q})\ cog(c, \epsilon) \xrightarrow{\text{A}} ob(o, a, \{l \mid s\}, \overline{q})\ cog(c, o)}$$

From configuration cn, by definition of equivalence $\approx$, we know that there exist o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq)$ is in a corresponding MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$. We also know that there is no active *execute* request in the set p of executed requests of activity α, because in the ABS configuration cn, the current task is `idle`. As in ABS the $\{l \mid s\}$ request belongs to the pending requests of the ABS object i_α, we have, by definition of equivalence $\approx$ (Figure 13, Line `1e`, passive requests case), two possible cases (for the two sides of the disjunction of Line `1e`), detailed below:

[12]Recall also that the activity has by default a soft thread limit status, only restraining the number of threads with an active status.

- In the first case, a corresponding passive request in MultiASP is in the set of executed requests p: $\exists f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''.(f, \mathit{execute}, i, \mathtt{m}, \overline{v''})_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \ \wedge \ \ell'(\mathtt{this}) = o$, with o the location of the object corresponding to i_α[13]. The requests involved in this case can only be *execute* requests. As we have explained in the translational semantics from ABS to MultiASP, *execute* requests belong to group g_2 ($\mathit{group}(\mathit{execute}) = g_2$) and this group has a maximum thread limit of one thread ($\mathcal{L}_{g_2} = 1$). Basically, this means that only one *execute* request can be active at a time. Thus, since no other request is active in p, we have the condition: $\text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}$ that is verified. We can apply the Activate-Thread MultiASP rule to be in configuration $\underline{cn'}$, where $\underline{cn} \to \underline{cn'}$:

$$\frac{\mathit{group}(q') = g_2 \qquad \text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q'_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S\end{array}}$$

  Note that in the above reduction, we have $q' = (f, \mathit{execute}, i, m, \overline{v''})$. We now have to prove that $cn' \approx \underline{cn'}$. The only change in the reduction takes place in the set of executed requests p: there is one less *execute* request that is passive (one less request to match in $\approx$ in Line `1e` of Figure 13, but one more *execute* request that is active). We must compare the *execute* request with the current task in ABS, in Line `1d` of $\approx$. Proving the condition of Line `1d` from the first case of the disjunction of Line `1e` is trivial, because all the elements perfectly match.
- In the second case, a corresponding request in MultiASP is in the queue of activity α: $\exists \ell', s', i, \mathtt{m}, .\,(f, \mathit{execute}, i, \mathtt{m}, \overline{v''}) \in Rq \wedge o_\alpha.\mathit{retrieve}(i) = o \wedge \text{bind}(o, \mathtt{m}, \overline{v''}) = \{\ell'|s'\} \ \underline{\text{with}} \quad (\forall x \in \text{dom}(l)\backslash \mathit{destiny}.\, l(x) \approx_\sigma^{cn} \ell'(x)) \wedge s \approx_{\sigma+\ell'}^{cn} s')$. Let $q' = (f, \mathit{execute}, i, \mathtt{m}, \overline{v''})$ and $Rq = Rq_0 :: q' :: Rq_1$. Because of the FIFO semantics of ABS, and because the definition of $\approx$ requires that the applicative requests are ordered in the same way in ABS and in MultiASP, we have that Rq_0 contains no *execute* request. As a consequence only *freshId* and *register* requests are in Rq_0. *freshId* requests are compatible with q' by definition but not *register* if the *register* targets the same object as the *execute* request, in this case the *register* request must be served first (this is also necessary to maintain the invariant **Reg**. In all cases, all requests in Rq_0 can be executed before serving *execute* and we can obtain a configuration $(\overline{gn()r})$ that is also compatible with the original ABS configuration $(gn(\approx)\overline{gn()r})$ and where q' can be served (i.e. enough requests of Rq_0 have been served so that the ready predicate is true). Thus, we can apply the MultiASP Serve reduction rule to be in configuration $\underline{cn'}$, where $\underline{cn_r} \to \underline{cn'}$:

$$\frac{\text{ready}(q', p, Rq_0) \quad q' = (f, \mathit{execute}, i, \mathtt{m}, \overline{v''}) \quad \text{bind}(o_\alpha, \mathit{execute}, \overline{v''}) = \{\ell'' \mid s''\}}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq'_0 :: q' :: Rq_1) \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid s''\}\} \uplus p'', Rq'_0 :: Rq_1)}$$

  Note that in the above reduction, s'' is the body of the execute method: $s'' = (w = \mathtt{this}.\mathit{retrieve}(i); x = w.\mathtt{m}(\overline{\mathit{params}}); \mathtt{return}\ x)$. Furthermore, we also have $o_\alpha.\mathit{retrieve}(i) = o$ thanks the the **Invariant Reg** and thus, after a local method call to *retrieve* and a local assignment, we can perform the main local method call $\mathtt{m}(\overline{\mathit{params}})$ on object o. We can then apply the MultiASP Invk-Passive reduction

[13]Formally, $\sigma(o) = [\mathit{cog} \mapsto \alpha, \mathit{myId} \mapsto i]$.

rule and obtain a last configuration $\underline{cn''}$, where $\underline{cn'} \to^* \underline{cn''}$, as follows:

$$\frac{[[w]]_{(\sigma+\ell'')} = o \qquad [[\overline{params}]]_{(\sigma+\ell'')} = \overline{v''} \qquad \text{bind}(o, \mathtt{m}, \overline{v''}) = \{\ell' \mid s'\}}{\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid x = w.\mathtt{m}(\overline{params}); return\ x\}\} \uplus p'', Rq) \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid x = \bullet; return\ x\}\} \uplus p'', Rq)\end{array}}$$

Finally, we have to prove that $cn' \approx \underline{cn''}$. To this end, there is one less request to match in the second case of the disjunction of Line `1e` of $\approx$, but the new request has to be compared in Line `1d` of $\approx$ (Figure 13). We have:
$\exists l, s. p' = \{l|s\}$ $\underline{\text{iff}}$ $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''$.
$\big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \land\ \ell'(\mathtt{this}) = o\big)$
$\underline{\text{with}}$ $\forall x \in dom(l) \backslash destiny.\, l(x) \approx^{cn}_{\sigma} \ell'(x) \land l(destiny) = f) \land s \approx^{cn}_{\sigma+\ell'} s'$
Indeed, the request we constructed matches those conditions: each condition is either a direct consequence of the applied reduction rules or was ensured by the initial conditions on the request that was in the queue. Besides, $\text{bind}(o, \mathtt{m}, \overline{v''}) = \{\ell' \mid s'\}$ implies that $\ell'(\mathtt{this}) = o$, which verifies the last condition above.

7) **Case of the [Read-Fut] ABS rule.**

We have the following ABS reduction $cn \xrightarrow{A} cn'$ with the Read-Fut rule:

$$\frac{f = [\![e]\!]^A_{(a+l)} \qquad v \neq \bot}{ob(i\_\alpha, a, \{l \mid x = e.\mathtt{get}; s\}, \overline{q})\ fut(f, v) \xrightarrow{A} ob(i\_\alpha, a, \{l \mid x = v; s\}, \overline{q})\ fut(f, v)}$$

To begin with, from the ABS configuration cn, we know that, by Lemma 4.1, there exists in ABS α such that $cog(\alpha, i\_\alpha) \in cn$. By the definition of equivalence $\approx$, there exist in MultiASP o_α, σ, p', Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{cn}$. There also exist ℓ', s', ℓ'', s'' such that $\{q_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\}\} \in p'$. By definition of equivalence on statements, we have $(x = e.\mathtt{get}; s) \approx^{cn}_{\sigma} s'$. By translational semantics, we have $s' = [\![x = e.\mathtt{get}; s]\!]$. Consequently, we have $s' = (\textbf{setLimitHard}; w = e.get(); \textbf{setLimitSoft}; x = e); [\![s]\!])$. Thus, we have the following MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell | \textbf{setLimitHard}; w = e.get(); \textbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \textsc{fut}(f, v', \sigma')\end{array}$$

Note that we have $v \approx^{cn}_{\sigma'} v'$ by definition of equivalence $\approx$. And by Lemma 4.3, we have $[\![e]\!]^A_{a+l} \approx^{cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$. Now, we can reduce the configuration $\underline{cn}$ to consume the **setLimitHard** statement with the MultiASP Set-Hard-Limit rule, we obtain the configuration $\underline{cn'}$, where $\underline{cn} \to \underline{cn'}$, as follows:

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitHard}; w = e.get(); \textbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid w = e.get(); \textbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_H\end{array}$$

Note that in the above reduction, we have: $[\![e]\!]^A_{a+l} = f$ in ABS and $[\![e]\!]^A_{a+l} \approx^{cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$ by Definition 4.2. Now, two cases are possible:

- In the first case, the future has not been updated yet, and we have: $[\![e]\!]_{\sigma+\ell} = o$ and $\sigma(o) = f$. Then, a future update can occur through the MultiASP Update reduction rule, this leads us to the configuration $\underline{cn''}$, where $\underline{cn'} \to \underline{cn''}$, as follows:

$$\frac{\sigma(o) = f \qquad (v_r, \sigma_r) = \text{rename}_\sigma(v', \sigma') \qquad \sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r}{\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq)\ \textsc{fut}(f, v', \sigma') \to \textsc{act}(\alpha, o_\alpha, \sigma'', p', Rq)\ \textsc{fut}(f, v', \sigma')}$$

Note that in the above reduction, we have: $p' = \{\ell | w = e.get(); \textbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\} \uplus p$. At this point, the $get()$ method call on object o can be performed and this method call succeeds because the future has been updated. Additionally, thanks to Lemma 4.4, $v \approx^{cn}_{\sigma'} v'$ implies that $v \approx^{cn}_{\sigma''} v_r$. In other words, the value of the future after serialisation is preserved. Here, we use the fact that we excluded the particular executions where the value of a future is a future, so since v is not a future, then neither v' nor v_r can point to a future. Finally, after applying some local reduction rules (a local method invocation, a local assignment changing ℓ to ℓ' and a change in the kind of thread limit), we end up in the following MULTIASP configuration $\underline{cn_3}$, where $\underline{cn''} \rightarrow^* \underline{cn_3}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell'|x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \quad \textsc{fut}(f, v', \sigma')$$

Finally, we have to prove that $cn' \approx \underline{cn_3}$. We have indeed $v \approx^{cn'}_{\sigma''} v'$ and we also have: $(x = v; s) \approx^{cn'}_{\sigma''} (x = e; [\![s]\!])$, because $[\![e]\!]_{\sigma''+\ell'} = v_r$ and $v \approx^{cn'}_{\sigma''} v_r$. Lastly, the other elements need not being considered because they did not change.

- In the second case, the future has already been updated, and we have: $[\![e]\!]_{\sigma+\ell} = v'$. By Lemma 4.3, we also have: $f \approx^{cn}_{\sigma} v'$. Then, performing *get* on e has no visible effect, and after applying several local MULTIASP reduction rules (like in the first case), we end up in a configuration $\underline{Cn''}$, where $\underline{cn'} \rightarrow \underline{cn''}$, as follows:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'|x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \quad \textsc{fut}(f, v', \sigma')$$

This final configuration is similar to the one of the first case. The only difference is that no future update was needed and thus the local store is unchanged. For the other elements, we can prove $cn' \approx \underline{cn''}$ similarly to the case above.

8) **Case of the [New-Object] ABS rule.**

We start from the NEW-OBJECT rule in which $cn \xrightarrow{\text{A}} cn'$ as follows:

$$\frac{i'\_\alpha = \text{fresh}(\texttt{C}) \quad \textit{fields}(\texttt{C}) = \overline{x} \quad \overline{v} = [[\overline{e}]]^A_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \alpha]}{\begin{array}{c} ob(i\_\alpha, a, \{l \mid x = \texttt{new local C}(\overline{e}); s\}, \overline{q})\ cog(\alpha, i\_\alpha) \\ \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid x = i'\_\alpha; s\}, \overline{q})\ cog(\alpha, i\_\alpha)\ ob(i'\_\alpha, a', \texttt{idle}, \varnothing) \end{array}}$$

By definition of equivalence $\approx$, there is a MULTIASP activity in $\underline{cn}$ corresponding to $cog(\alpha, i\_\alpha)$: $\exists o_\alpha, \sigma, p', Rq.\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq)$. By translational semantics, p' contains the statements that create an equivalent new object in MULTIASP: $p' = \{q_A \mapsto \{\ell \mid t = \texttt{this}.cog(); id = t.freshId(); no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [[s]]\} :: E \uplus p''\}$. By definition of equivalence $\approx$, there is a MULTIASP object in $\underline{cn}$ corresponding to $ob(i\_\alpha, a, \cdots)$: $\exists o, \overline{v'}.\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$, and this object maps to the current local variable $\texttt{this}$: $\ell(\texttt{this}) = o$. In summary, we have the following MULTIASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma[o \mapsto [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]], \{q_A \mapsto \{\ell[\texttt{this} \mapsto o] \mid t = \texttt{this}.cog(); \\ id = t.freshId(); no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [[s]]\} :: E \uplus p''\}, Rq)\end{array}$$

Then, $\underline{cn}$ can be reduced by evaluating all local reductions of α until a configuration $\underline{cn'}$ has the object instantiation as current statement. These reduction steps only execute local assignments and local (synchronous) method calls fetching the COG and the identifier of the invoked object. In particular, $\underline{cn} \rightarrow^* \underline{cn'}$ where $\underline{cn'}$ contains the activity α with the current thread executing the object instantiation: $q_A \mapsto \{\ell' \mid no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [[s]]\}$, with $\ell' = \ell[t \mapsto \alpha, id \mapsto i']$[14]. This configuration $\underline{cn'}$ can be

[14]Note that i', the same identifier as the fresh identifier allocated by ABS, can be chosen as a fresh identifier by the method *freshId*. This is due to the definition of equivalence between objects: if i' was not

reduced to a configuration $\underline{cn''}$, such that $\underline{cn'} \rightarrow \underline{cn''}$ by the MultiASP New-Object rule, as follows:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{x} \quad o' \text{ fresh} \quad \sigma' = \sigma \cup \{o' \mapsto [\mathit{cog} \mapsto \alpha, \mathit{myId} \mapsto i', \overrightarrow{x = v''}]\} \quad [[\overline{e}]]_{(\sigma+\ell')} = \overline{v''}}{\underline{cn'} \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid no = o'; z = t.\mathit{register}(no, id); x = no; [[s]]\} :: E\} \uplus p'', Rq)}$$

The configuration $\underline{cn''}$ has now the activity α containing the new object. This object has a proper identifier and points to the right activity α (it points to its COG). The *register* method call is then evaluated. Its effect is handled by the **Invariant Reg**, which makes sure that the object is available to the system after this call. The **Invariant Reg** is verified here. This leads us to the final configuration $\underline{cn'''}$ such that $\underline{cn''} \rightarrow^* \underline{cn'''}$, that contains the following activity:

$$\text{ACT}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell'' \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq)$$

Note that in the above reduction, we have: $\ell'' = \ell'[no \mapsto o']$. Now we have to prove $cn' \approx \underline{cn'''}$. Typically, we have two objects of the same COG affected in ABS (i_α and i'_α), and one activity affected in MultiASP by the reduction. We focus on the state of the two modified elements below:

- Let us first consider i'_α. We have a fresh object i'_α in ABS that must be equivalent to the fresh object o' in the store of activity α in MultiASP. By Definition 4.3 of $\approx$, Line `1b`, we have the additional fields of the new object that point to the COG (activity α) and to the identifier i', matching the ABS identifier i'_α. The equivalence of the other fields of the fresh objects (in ABS and in MultiASP) is obtained by Lemma 4.3. The newly created object is `idle` in ABS with no pending request. Consequently, Line `1d` and Line `1e` of Figure 13 are trivially verified.
- Secondly, let us consider i_α. The currently served request is the only element that changed, so only Line `1d` of Figure 13 has to be checked. The set of local variables did not change, except in MultiASP where the set of local variables contains more variables. Finally, we have to check the equivalence of the remaining statements. We fall in the second case of the definition of the equivalence of statements: $s(x = i'\_\alpha; s) \approx_{\sigma}^{cn'} (x = no; [\![s]\!])$ with $i'\_\alpha \approx_{\sigma}^{cn'} [\![no]\!]_{(\sigma'+\ell'')}$. Additionally, we have: $[\![no]\!]_{(\sigma'+\ell'')} = o'$ and we have already shown the equivalence between i'_α and o'.

9) **Case of the [New-Cog-Object] ABS rule.**

We start from the New-Cog-Object rule where $cn \xrightarrow{\text{A}} cn'$, as follows:

$$\frac{\beta = \text{fresh}(\,) \quad i'\_\beta = \text{fresh}(\mathtt{C}) \quad \mathit{fields}(\mathtt{C}) = \overline{x} \quad \overline{v} = [[\overline{e}]]^{A}_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \beta]}{\mathit{ob}(i\_\alpha, a, \{l \mid x = \mathtt{new}\ \mathtt{C}(\overline{e}); s\}, \overline{q}) \xrightarrow{\text{A}} \mathit{ob}(i\_\alpha, a, \{l \mid x = i'\_\beta; s\}, \overline{q}) \quad \mathit{ob}(i'\_\beta, a', \mathtt{idle}, \varnothing) \quad \mathit{cog}(\beta, \epsilon)}$$

From the ABS configuration cn, we know that by Lemma 4.1, $\exists \alpha.\, \mathit{cog}(\alpha, i\_\alpha) \in cn$. Also, by definition of equivalence $\approx$, $\exists o_\alpha, \sigma, p', Rq.\text{ACT}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{cn}$ and $\exists f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''.(f, \mathit{execute}, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p'$ and $(x = \mathtt{new}\ \mathtt{C}(\overline{e}); s) \approx_{\sigma+\ell'}^{cn} s'$. By definition of the equivalence on statements, and by translational semantics, we have: $(x = \mathtt{new}\ \mathtt{C}(\overline{e}); s) \approx_{\sigma+\ell'}^{cn} s'$ implies $s' = [\![x = \mathtt{new}\ \mathtt{C}(\overline{e}); s]\!]$. Thus, by definition of the translation:

$$s' = (\mathit{newcog} = \mathit{newActive}\ \text{COG}(); \mathit{id} = \mathit{newcog.freshId}(); \mathit{no} = \mathtt{new}\ \mathtt{C}'(\overline{e}, \mathit{newcog}, \mathit{id}); \\ z = \mathit{newcog.register}(\mathit{no}, \mathit{id}); x = \mathit{no}; [\![s]\!])$$

fresh in α, i'_α would already be an existing ABS object and could not be chosen as a fresh ABS object identifier.

All of these elements allow us to define the MULTIASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{\mathtt{q}_A \mapsto \{\ell' \mid newcog = newActive\ \textsc{cog}(); id = newcog.freshId();$$
$$no = \mathtt{new}\ \mathtt{C}'(\overline{e}, newcog, id); z = newcog.register(no, (id)); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq)$$

The MULTIASP configuration $\underline{cn}$ can be reduced with the NEW-ACTIVE rule to the configuration $\underline{cn'}$, such that $\underline{cn} \to \underline{cn'}$:

$$\frac{\beta, o_\beta \text{ fresh} \qquad \sigma' = \{o_\beta \mapsto [\overrightarrow{\overline{x_{\textsc{cog}} = v_{\textsc{cog}}}}]\} \cup \text{serialise}(\overline{v_{\textsc{cog}}}, \sigma) \qquad \overset{fields(\textsc{cog}) = \overline{x_{\textsc{cog}}}}{} \qquad [\![\overline{e_{\textsc{cog}}}]\!]_{(\sigma+\ell')} = \overline{v_{\textsc{cog}}}}{\begin{array}{c}\underline{cn} \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid newcog = \beta; id = newcog.freshId(); \\ no = \mathtt{new}\ \mathtt{C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing)\end{array}}$$

Note that we can pick β as a fresh activity name because, by definition of $\approx$, as β is free in ABS, it is also free in MULTIASP. Now, we can reduce $\underline{cn'}$ to a configuration $\underline{Cn''}$ by two local assignments and a remote invocation (to get a fresh identifier), until the current statement is the new object instantiation:

$$\textsc{act}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell''' \mid no = \mathtt{new}\ \mathtt{C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\}$$
$$:: \{\ell'' \mid s''\}\} \uplus p'', Rq) \qquad \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \qquad \textsc{fut}(f, \bot)$$

Note that in the above reduction, we have: $\ell''' = \ell'[newcog \mapsto \beta, id \mapsto i']$ where i' is a fresh id in the new cog[15]. On $\underline{cn}''$, we can apply the NEW-OBJECT MULTIASP rule to obtain the configuration $\underline{cn_3}$ such that $\underline{cn}'' \to \underline{cn_3}$, as follows:

$$\frac{fields(\mathtt{C}) = \overline{x} \qquad o \text{ fresh} \qquad \sigma'' = \sigma'_\alpha \cup \{o \mapsto [cog \mapsto \beta, myId \mapsto f, \overrightarrow{\overline{x = v''}}]\} \qquad [\![\overline{e}]\!]_{(\sigma+\ell')} = \overline{v''}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma'_\alpha, \\ \{q_A \mapsto \{\ell''' \mid no = \mathtt{new}\ \mathtt{C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing) \\ \to \textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell''' \mid no = o; z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing)\end{array}}$$

Then, the final steps consist in reducing the local assignment and evaluating the *register* remote method invocation. This brings us to the final configuration $\underline{cn_4}$, where $\underline{cn_3} \to^* \underline{cn_4}$, which contains the following MULTIASP terms:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell_4 \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma_3, \varnothing, (f', register, o', i'))$$

$$\textsc{fut}(f, i', \varnothing) \quad \textsc{fut}(f', \bot)$$

Note that in the above reduction, we have: $\ell_4 = \ell'''[no \mapsto o]$. In particular, $\sigma_3(o') = [cog \mapsto \beta, myId \mapsto i', \overline{x \mapsto v_2}]$ characterise the serialisation of o. Now we have to prove that $cn' \approx \underline{cn_4}$. Two activities in MULTIASP have to be considered: activity α, that has been modified during the reductions, and activity β, that has been introduced by the reductions. The two activities are checked against the equivalence $\approx$ below:

- The first MULTIASP activity (α) corresponds to *cog* α in ABS (that exists by Lemma 4.1), it contains now in its store a copy of the new object o corresponding to object i'_β in ABS. Objects o and i'_β are equivalent for two reasons. First, o

[15] There is a simplification in the proof here. The translation expressed in the paper performs a remote invocation to the new *cog*, obtains a future and the future is replaced by the fresh id. However, the implementation uses a primitive type for the identifiers, ensuring a synchronous call and preventing the creation of future, this could be encoded with an immediate `get`. In practice, in the proof we directly use i' the returned value; this is the closest to implementation, and is equivalent to the configuration obtained with `get`, modulo equivalence of values.

contains the required additional fields: *cog* that points the new activity β, and *myId* with value i'. Second, the other fields are only meaningful in activity β, thus we do not have to consider them in activity α. Regarding the current request, except z and *no* which are temporary variables introduced by the translation, the other local variables are unchanged. Second, the remaining statements fall in the second case of the definition of the equivalence of statements. Like in the case of [New-Object], we have: $s_{\text{ABS}} = (x = i'\_\beta; s) \;\wedge\; s_{\text{MultiASP}} = (x = no; [\![s]\!])$ with $i'\_\beta \approx^{cn'}_{\sigma''} [\![no]\!]_{(\sigma''+\ell_4)}$. Since we have: $[\![no]\!]_{(\sigma''+\ell_4)} = o$, and since we showed before that o is equivalent to i'_β, we can conclude that the remaining statements are equivalent.

- The second MultiASP activity (β) corresponds to the fresh *cog* β in ABS. The content of this activity reflects the fresh object o'. Concerning the object value, we must prove equivalence of object fields other than *cog* and *myId*. Here, we recall the object's fields that are meaningful are the ones that are hosted in β. Precisely, we have: $\overline{v} = [\![\overline{e}]\!]^A_{(a+l)}$ and we also have: $(\overline{v_2}, \sigma_3) = \text{rename}_{\sigma'}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \text{serialise}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \sigma''))$. By Lemma 4.3 and by Lemma 4.4, we have: $\overline{v} \approx^{cn'}_{\sigma_3} \overline{v_2}$. This verifies Line `1c` of the definition of equivalence $\approx$. In ABS, the new object is `idle` with no pending request. Line `1d` and Line `1e` of the definition of equivalence $\approx$ is verified, because no request is currently served in ABS, and accordingly, in MultiASP there is no *execute* request currently served.

Finally, we have two additional futures in MultiASP concerning the *freshId* and the *register* remote method calls. However, they are not considered in the equivalence because they do not correspond to an *execute* request (i.e. to an applicative request).

10) **Case of the [Rendez-vous-Comm] ABS rule.**

In this case, we only deal with communications that are not performed on the caller itself. Indeed, the self asynchronous method call is similar but requires a specific proof instance. We start from the ABS Rendez-vous-Comm reduction rule, where $cn \xrightarrow{\text{A}} cn'$, where $\alpha \neq \beta$:

$$\frac{f = \text{fresh}(\,) \quad o' = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad q'' = \text{bind}(o', f, m, \overline{v}, \texttt{class}(o'))}{\begin{array}{c} ob(o, a, \{l \mid x = e\,!\,\mathtt{m}(\overline{e}); s\}, \overline{q}) \;\; ob(o', a', p, \overline{q'}) \\ \xrightarrow{\text{A}} ob(o, a, \{l \mid x = f; s\}, \overline{q}) \;\; ob(o', a', p, \overline{q'} :: q'') \;\; fut(f, \bot)\end{array}}$$

First of all, by Lemma 4.1, $\exists \alpha.\ cog(\alpha, i\_\alpha) \in cn$ and $\exists \beta.\ cog(\beta, i'\_\beta) \in cn$. By definition of equivalence $\approx$ and of translational semantics, there exist $o_\alpha, \sigma_\alpha, p, Rq, \ell, \ell'', s'', o_\beta, \sigma_\beta, p', Rq$ such that the following terms belong to the MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell | t = e.cog(); id = e.myId(); x = t.execute(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{ACT}(\beta, o_\beta, \sigma_\beta, p', Rq')\end{array}$$

In the MultiASP configuration, there must be an object equivalent to i'_β resulting from the evaluation of e in activity α. This object has two copies: one in activity α and one in activity β, but the value of its applicative fields in activity α are meaningless (they are never used). Indeed, the copy of this object in activity α plays the role of a proxy to its meaningful counterpart in β. More formally, we have:

$$\begin{array}{l} o' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v'}]] \in \sigma_\alpha \text{ with } [\![e]\!]_{\sigma+\ell} = o' \\ o'' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v''}]] \in \sigma_\beta \text{ with } a' \approx_{\sigma_\beta} [\overrightarrow{x \mapsto v''}] \end{array}$$

The MultiASP configuration $\underline{cn}$ can be reduced using local rules until a configuration $\underline{cn'}$ contains the *execute* remote method invocation as current statement. More formally, we have $\underline{cn} \to^* \underline{cn'}$, where the current thread of activity α in $\underline{cn'}$ is: $q_A \mapsto \{\ell' | x = t.execute(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}$, with $\ell' = \ell[t \mapsto \beta, id \mapsto i']$. Then, we can reduce $\underline{cn'}$ to a configuration $\underline{cn''}$ by the MultiASP Invk-Active rule, such that $\underline{cn'} \to \underline{cn''}$, as follows:

$$\frac{[\![t]\!]_{(\sigma_\alpha + \ell')} = \beta \quad [\![\overline{e}]\!]_{(\sigma_\alpha + \ell')} = \overline{v} \qquad f, o_f \text{ fresh} \qquad (\overline{v_r}, \sigma_r) = \text{rename}_{\sigma_\beta}(\overline{v}, \text{serialise}(\overline{v}, \sigma_\alpha))}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell' \mid x = t.execute(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \qquad \textsc{act}(\beta, o_\beta, \sigma_\beta, p', Rq') \\ \to \textsc{act}(\alpha, o_\alpha, \sigma_\alpha[o_f \mapsto f], \{q_A \mapsto \{\ell' \mid x = o_f; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \textsc{act}(\beta, o_\beta, \sigma_\beta \cup \sigma_r, p', Rq' :: (f, execute, i', \mathtt{m}, \overline{v_r})) \qquad \textsc{fut}(f, \bot)\end{array}}$$

We finally have to prove that $cn' \approx \underline{cn''}$. First, we have as many terms that have changed in cn' as in $\underline{cn''}$: there are two *ob* and act terms and one added future in both configurations. Regarding the fresh future, it is the same in the two configurations: we conclude by Line `3` of the definition of ≈.

Regarding activity α, the set of local variables remains unchanged, except for temporary variables. The element that actually changed is the current statement. We have: $(x = f; s) \approx^{cn'}_{\sigma_\alpha} (x = o_f; [\![s]\!])$ by the second case of the definition of the equivalence of statements. For that, we need to have: $f \approx^{cn'}_{\sigma_\alpha} [\![o_f]\!]_{\sigma_\alpha + \ell'}$, which is true because $[\![o_f]\!]_{\sigma_\alpha + \ell'} = o_f$ and because we have: $\sigma_\alpha(o_f) = f$. Thus, we have: $f \approx^{cn'}_{\sigma_\alpha} \sigma(o_f)$ by definition of ≈. We conclude the equivalence between activity and *cog* α using Line `1d` of the definition of equivalence ≈.

Regarding activity β, a new pending request is created in the MultiASP configuration $\underline{cn''}$, corresponding to the new inactive thread in the ABS configuration cn'. According to Line `1e` of the definition of equivalence ≈, we have: $\{l|s\} \in \overline{q'} :: q'' \wedge l(destiny) = f$, with $\{l|s\} = \text{bind}(i'\_\beta, f, \mathtt{m}, \overline{v}, \mathtt{class}(i'\_\beta))$. Then, there is only one case to consider because we know that the request is not served yet. Thus, what we exactly have to prove is the following:

$$\begin{array}{l}\exists i', \mathtt{m}, \overline{v_r}, \ell_\beta, s'.(f, execute, i', \mathtt{m}, \overline{v_r}) \in Rq \wedge o_\beta.retrieve(i') = o \wedge \text{bind}(o, \mathtt{m}, \overline{v_r}) = \{\ell_\beta | s'\} \\ \underline{\text{with}} \quad \forall x \in \text{dom}(l) \backslash destiny. \, l(x) \approx^{cn'}_{\sigma_\beta} \ell_\beta(x) \wedge s \approx^{cn'}_{\sigma_\beta + \ell_\beta} s'\end{array}$$

Firstly, we have indeed the new request in the queue of activity β in the MultiASP configuration $\underline{cn''}$. Then, we need to ensure that: $o_\beta.retrieve(i') = o$, which is guaranteed by the **Invariant Reg**. On the other hand, the result of the auxiliary function bind in MultiASP is similar to the one that exists in ABS. The local variables on the ABS side also appear on the MultiASP side equivalently, except for the two following points:

- There is no *destiny* variable in MultiASP. However, this is taken into account by the definition of equivalence ≈, which compares the *destiny* variable with the future of the corresponding MultiASP request.
- The transmitted parameters $\overline{v_r}$ are the copies of the method parameters in ABS. Ensuring equivalence between request parameters in this case is handled by Lemma 4.3 for obtaining the equivalence of the emitted values, and by Lemma 4.4 to ensure that equivalence still holds after the values being serialised, sent to β and renamed.

One must additionally notice that the order of (non-applicative) requests is the same in the two configurations (both requests are appended at the end of the pending request queue). WE do not discuss the *cont* local variable here because it is mostly related to synchronous local method invocations.

11) **Case of the [Return] ABS rule.**

We start from the following Return ABS reduction rule, where $Cn \xrightarrow{A} Cn'$, as follows:

$$\frac{v = [\![e]\!]^{A}_{(a+l)} \quad f = l(\text{destiny}) \quad cont \notin \text{dom}(l)}{ob(o, a, \{l \mid \texttt{return}\ e; s\}, \overline{q})\ fut(f, \bot) \quad \xrightarrow{A} ob(o, a, \texttt{idle}, \overline{q})\ fut(f, v)}$$

From the starting ABS configuration, we have, by Lemma 4.1, $\exists cog(\alpha, i\_\alpha) \in cn$. By definition of equivalence $\approx$, $\exists o_\alpha, \sigma, p, Rq$ such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$, and $\exists f, i, \mathtt{m}, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{l'' \mid s''\} \in p$, and $\texttt{return}\ e; s \approx^{cn}_{\sigma+\ell'} s'$. By the definition of the equivalence of statements, we have: $s' = (\texttt{return}\ e; [\![s]\!])$, since the first statement is not in the form of $x = e; s$. Also, by Line 3 of the equivalence $\approx$, we have: $\textsc{fut}(f, \bot) \in \underline{cn}$. Furthermore, we know that in our MultiASP translation, a method call on an object is always wrapped in an *execute* method call on an active object (by definition of $\approx$). Therefore, we have $s'' = (x = \bullet; \texttt{return}\ x)$. All of these elements allow us to find the following MultiASP configuration $\underline{cn}$, where $cn \approx \underline{cn}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid \texttt{return}\ e; [\![s]\!]\} :: \{\ell'' \mid x = \bullet; \texttt{return}\ x\}\} \uplus p', Rq) \qquad \textsc{fut}(f, \bot)$$

From $\underline{cn}$, we can apply the Return-Local MultiASP reduction rule to reach the configuration $\underline{cn'}$, such that $\underline{cn} \to \underline{cn'}$, as follows:

$$\frac{v' = [\![e]\!]_{\sigma+\ell'}}{\underline{cn} \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'' \mid x = v'; \texttt{return}\ x\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)}$$

Then, after applying the Assign-Local MultiASP reduction rule on $\underline{cn'}$, we are in the configuration $\underline{cn''}$, such that $\underline{cn'} \to \underline{cn''}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''[x \mapsto v'] \mid \texttt{return}\ x\}\} \uplus p', Rq) \qquad \textsc{fut}(f, \bot)$$

We now denote $\ell_3 = \ell''[x \mapsto v']$. From $\underline{cn''}$, we can apply the Return MultiASP reduction rule to finally end up in the configuration $\underline{cn'''}$, such that $\underline{cn''} \to \underline{cn'''}$, as follows:

$$\frac{[\![x]\!]_{\sigma+\ell_3} = v'}{\underline{cn''} \to \textsc{act}(\alpha, o_\alpha, \sigma, p', Rq) \quad \textsc{fut}(f, v', \text{serialise}(v', \sigma))}$$

We further denote: $\sigma_s = serialise(v', \sigma)$, and we recall that: $[\![e]\!]_{\sigma+\ell'} = v'$. Now, we have to prove that $cn' \approx \underline{cn'''}$. Regarding the resolved future, we have the future term f both in the ABS and MultiASP configurations. Secondly, we have to ensure that: $v \approx^{cn'}_{\sigma_s} v'$ and to this end, we have to ensure that: $v \approx^{cn'}_{\sigma} [\![e]\!]_{\sigma+\ell'}$. Indeed, by Lemma 4.3, we have: $[\![e]\!]^{A}_{a+l} \approx^{cn'}_{\sigma} [\![e]\!]_{\sigma+\ell'}$. Thus, we have: $v \approx^{cn'}_{\sigma} v'$ and, by Lemma 4.4, we have: $v \approx^{cn'}_{\sigma_s} v'$. Regarding activity α, the request corresponding to future f does not exist any more, which matches with the fact that the ABS current task becomes `idle`: in the definition of equivalence $\approx$, there is one less request to compare in Line `1d`. Finally, the other elements of the activity did not change, thus preserving their equivalence.

Note that the *cont* dedicated variable has not been used because we restricted the proof to asynchronous method invocations. It would reveal useful if the proof was also dealing with local synchronous method calls.

12) **Case of the [Skip], [Cond-True], [Cond-False], [Context], [Cog-Sync-Call], [Self-Sync-Call], [Cog-Sync-Return-Sched], [Self-Sync-Return-Sched] and [Rem-Sync-Call] ABS rules.**

To finish, none of these rules are considered in the proof of Theorem 4.5, for different reasons. We review the reasons for not treating each of them below.

Firstly, in practice Skip, Cond-True, Cond-False and Context are kept unchanged from the translation provided by the Java backend for ABS to implement the ProActive backend for ABS. Consequently, their translation in an imperative programming language without remote method invocations and futures is already supposed to be sound and correct, and is trivial because it does not involve asynchronous remote method calls and futures. Furthermore, in the case of the ProActive backend, the `await` statement on conjunctive guards and boolean expressions is not handled using these rules. However, none of these four rules raises a particular difficulty because they only involve local computation.

Secondly, all the rules dealing with local synchronous method calls, namely Cog-Sync-Call, Self-Sync-Call, Cog-Sync-Return-Sched and Self-Sync-Return-Sched are not considered in the proof. Indeed, the effect of these rules send back the current task in queue and schedule it right away with the statement that triggered the call to mimic a synchronous treatment. Instead, we rely on a classical stack of method calls. The practical translation of those aspects is again unchanged from the Java backend for ABS.

Thirdly, in the case of the rule for remote synchronous method calls, namely Rem-Sync-Call, the ABS rule is a composition of a remote asynchronous method call and a future read; in the translation, we directly inline this composition. Thus, the proof for this rule simply inlines the two cases of Rendez-vous-Comm and Read-Fut.

In conclusion, all reduction rules of the ABS semantics ensure that an equivalent final configuration is reachable with the MultiASP translation. Consequently, the translation simulates all ABS executions with FIFO policy and rendez-vous communications, provided that no future value is a future reference, which completes the proof of Theorem 4.5. □

Overall, we could only prove a weak simulation relation between a program and its translation. Indeed, we have considered some of the ABS and some of the MultiASP rules as *silent actions*. On one hand, Some ABS rules are strictly associated to one MultiASP rule, like for strong simulation. But on the other hand, for some other ABS rules there is an associated MultiASP rule, but also some additional rules that are needed to reach the equivalent configuration: these latter rules are considered as silent actions. Also, some ABS rules do not strictly correspond to any MultiASP rule; in this case either the equivalence is just maintained, or some not observable transitions must be achieved (silent actions). Table 3 summarises our results on the simulation and on observability of ABS and MultiASP rules. It shows, for each ABS rule, the main MultiASP rule that simulates it, and, if necessary, the additional silent actions that must be performed. The '–' denotes the case where there is no corresponding rule. The '/' indicates a choice between two rules depending on a criteria detailed in the proof but not in the table. Further comments can be made about Table 3.

Firstly, we can notice that an additional Invk-Passive and Assign-Local-Tmp MultiASP rules are quite omnipresent. Indeed, we exhibited before that our translation introduces additional communications and temporary variables. We can notice that, if we ignore the introduction of those 'harmless' rules (in terms of what we are interested in observing), most of the important ABS rules can be simulated by a single MultiASP rule. Secondly, we can simulate the Await-False ABS rule with the Invk-Future MultiASP rule, but the Await-True ABS rule has no MultiASP equivalent. Then, for example it would also have been possible to consider the Invk-Future

ABS rule	**MultiASP rule**	**Additional MultiASP rules**
Assign-Local	Assign-Local	–
Assign-Field	Assign-Field	–
Await-True	–	Update / – , Invk-Passive, Return-Local Assign-Local-Tmp
Await-False	Invk-Future	–
Release-Cog	–	–
Activate	–	Activate-Thread / (Serve, Invk-Passive,Return-Local, Assign-Local-Tmp)
Read-Fut	–	Set-Hard-Limit, Update / – , Invk-Passive, Return-Local, Set-Soft-Limit Assign-Local-Tmp
New-Object	New-Object	Invk-Passive Assign-Local-Tmp Return-Local
New-Cog-Object	New-Object	New-Active, Assign-Local-Tmp, Invk-Active-Meta, Return
Rendez-vous-Comm	Invk-Active	Invk-Passive, Return-Passive, Assign-Local-Tmp
Return	Return	Return-Local Assign-Local-Tmp

Table 3. Summary table of the simulation of ABS in MultiASP. Assign-Local-Tmp represents an Assign-Local on a variable introduced by the translation instead of ABS local variables. In the same way, Invk-Active-Meta means that it is like an Invk-Active but on a method that is not the *execute* method.

MultiASP rule as a silent action for the Await-False ABS rule. Thirdly, the New-Object and New-Cog-Object ABS rules are both simulated by the New-Object MultiASP rule, but we can easily distinguish the two cases. Indeed, the New-Object MultiASP rule can distinguish whether the translated ABS cog is empty or not. More precisely, the New-Cog-Object ABS rule is simulated by a New-Object MultiASP on an empty cog, while the New-Object ABS rule is simulated by a New-Object MultiASP rule on a cog that is not empty. From another point of view, the New-Active MultiASP rule could be the observable rule (and not a silent action) to simulate the New-Cog-Object ABS rule instead of the New-Object MultiASP rule. However, doing so would mean that we create an (active) object in MultiASP that has no equivalent object in ABS. This is why it is more natural to trace the New-Object rules.

For the simulation we introduced the Invk-Active-Meta MultiASP rule, that is a silent action because it only applies to meta-requests that do not exist in ABS, like the *register* and *freshId* requests. Symmetrically, the visible INVK-ACTIVE MULTIASP rule, that simulates the RENDEZ-VOUS-COMM ABS rule, only corresponds to *execute* requests which are the ones that are observable in ABS executions. Again, as we can easily distinguish the requests of the INVK-ACTIVE and INVK-ACTIVE-META rules, then there is no ambiguity in the simulation. A similar remark can be made about the ASSIGN-LOCAL and ASSIGN-LOCAL-TMP MULTIASP rules. The latter is for handling the temporary variables that are introduced by the simulation. Finally, we can now easily list the MULTIASP and ABS rules that are involved but not observable in the simulation from ABS to MULTIASP:

- The silent MULTIASP actions are: NEW-ACTIVE, SERVE, INVK-PASSIVE, RETURN-LOCAL, UPDATE, ACTIVATE-THREAD, SET-HARD-LIMIT, SET-SOFT-LIMIT, INVK-ACTIVE-META, and (ASSIGN-LOCAL-TMP). The other MULTIASP rules are the ones that are observable in the simulation.
- The silent ABS actions are: AWAIT-TRUE, RELEASE-COG, ACTIVATE, READ-FUT. It is interesting to note that these rules are related to the manipulation of futures and of execution threads in ABS, and that are manipulated differently, or not at all, in MULTIASP. For example, a single MULTIASP rule is necessary for an object to release a thread whether two rules are necessary in ABS.

B.2. **From MultiASP to ABS. Theorem 4.6**, page 35 (MULTIASP to ABS). Any reduction of the MULTIASP translation corresponds to a valid ABS execution.

$\underline{cn_0} \to^* \underline{cn} \Rightarrow \exists cn.\, cn_0 \overset{\text{A}}{\to}{}^* cn \wedge cn \approx \underline{cn}$

Proof sketch of Theorem 4.6. This proof sketch verifies that any MULTIASP translation of an ABS program corresponds to an ABS execution that is possible. In this direction, the main difficulty is that we introduced additional steps in the reduction. However, for the proof we use the fact that the two active object languages have few concurrent rules: method invocation and future awaiting, and since there is only a single active thread at a time, it prevents local concurrency. Consequently, we know that any sequence of additional MULTIASP actions never performs a wait-by-necessity, and runs until the end of the sequence without any observable interruption. Overall, only thread activation, thread passivation, method invocation, and future update can create interleavings. We rely on this knowledge for the proof.

Proving this direction of the simulation is complex, since the translated code is more operational. In particular, intermediate states in MULTIASP must be attached to the right state in ABS. Here, what must be observed is that the translation of an ABS primitive mostly involves a single action that impacts the state of the equivalent ABS program. For example, for remote invocations, solely the *execute* requests impact the request queue of the recipient, and has an effect on the equivalence relation. The principle is to focus only on these actions and to allow the translation of statements to do the same. For example, in the case of the *execute* remote method invocations, assignments to intermediate variables are ignored in the equivalence, and all the states that precede the execution of the remote method invocation are considered equivalent to the same ABS state. This is mostly handled by ignoring assignments to variables that are not part of the ABS code. A particular other issue is that there is an intermediate state when the request is served and the associated ABS method is not started yet. This case must also be considered in the notion of equivalence. A 'just started' request is equivalent to having the request still in the queue. Finally, it is also important to notice that the theorem no more needs a restriction on future values

$$s \approx^{cn}_{(\sigma+\ell)} [\![s]\!] \qquad \frac{s \approx^{cn}_{(\sigma+\ell)} (\textbf{setLimitHard}; s')}{s \approx^{cn}_{(\sigma+\ell)} s'}$$

$$\frac{s \approx^{Cn}_{(\sigma+\ell)} (\mathit{tmp}{=}z; s') \qquad \mathit{tmp} \text{ is a local variable introduced by the translation}}{s \approx^{cn}_{(\sigma+\ell)} s'}$$

$$\frac{s \approx^{cn}_{(\sigma+\ell)} s' \qquad \mathit{tmp} \text{ is a local variable introduced by the translation}}{s \approx^{cn}_{(\sigma+\ell)} (\mathit{tmp}{=}z; s')}$$

$$\frac{s \approx^{cn}_{(\sigma+\ell+(\mathit{tmp}\mapsto[\![e]\!]_{(\sigma+\ell)}))} s' \qquad \mathit{tmp} \text{ is a local variable introduced by the translation}}{s \approx^{cn}_{(\sigma+\ell)} (\mathit{tmp}{=}e; s')}$$

$$\frac{s = (x = v; s_1) \qquad s' = (x = e; [\![s_1]\!]) \qquad v \approx^{cn}_{\sigma} [\![e]\!]_{(\sigma+\ell)}}{s \approx^{cn}_{(\sigma+\ell)} s'}$$

FIGURE 14. Refined equivalence of statements for the proof of Theorem 4.6.

because when a future access is possible in MULTIASP, it can be faithfully simulated in ABS. Indeed, when a wait-by-necessity succeeds in MULTIASP, this must correspond in ABS to enough explicit future access instructions to access the data; this data must be available because of the equivalence between configurations.

The goal here is not to do the exhaustive simulation, like for the proof of Theorem 4.5, mostly because the technical details will be massively redundant. However, this theorem ensures that no additional behaviours can be introduced by the translation. We provide here the most important points of the proof: the refinement of the equivalence relation $\approx$, and a summary of the equivalence between reductions.

1) **Adapting the equivalence relation.**

To prove that all the MULTIASP executions faithfully respect the ABS semantics, the equivalence relation must take into account the fact that several rules are used in MULTIASP to simulate a single ABS rule. In practice, it is easy to identify the case where the same statement is triggered both in MULTIASP and in ABS, while other MULTIASP statements are preparing the main statement being executed, e.g. fetching the identifier and the COG of the targeted object for a method invocation. Thus, we define a new equivalence relation on statements, as shown in Figure 14.

The first and the last rule of the equivalence are unchanged for the proof of Theorem 4.6, we add new rules. The first one discards **setLimitHard** instructions: the thread that have just performed a **setLimitHard** is considered as still in the same state even if one statement is missing. Similarly, the assignments to temporary variables can be added or removed, and when it is removed one can use the assigned value (but only if it is a simple expression). This is useful to relate, in MULTIASP, the temporary variable *no* and use it in the remaining because the statement: $no = o; s; x = no$ in MULTIASP is equivalent to: $x = o$ in ABS.

Two other modifications must be made regarding the equivalence of MULTIASP and ABS configurations (we refer to the equivalence definition of Figure 13). Firstly, considering the direction from ABS to MULTIASP, we have as many COG in ABS as active objects in MULTIASP. However, in ABS, when a COG is created, it is populated with at least one applicative object, whereas in the translation of the `new` ABS keyword, the instruction that instantiates an active object (the `newActive` MultiASP keyword) is separated from the instantiation of the new object that populates the cog in ABS. Even if, in any case, the thread cannot be interrupted when doing these steps, this particular moment of the execution is a point where the equivalence relation is not verified. Such a situation is ruled out by considering, in Line `1a` of Figure 13, only the activities that contain at least one object. In other words, the activities that are only populated with a cog active object in MultiASP are not considered to be part of the configuration yet. This can easily be formulated by checking the content of the local store of the activity: if σ only contains the cog object and no object corresponding to an ABS object, then the activity should not be considered in the equivalence. Specifically:

Line `1a` *of Figure 13:* $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{cn}$
has the additional condition: σ *has more than one entry.*

Secondly, there is a state when the request has started to be served in MultiASP but the corresponding ABS method has not started to be executed (precisely, when the stack corresponding to this request has a single entry). In this case, the corresponding thread that is currently executed is necessarily the single active thread. Such a request that has not started its applicative service must be considered as the only active request but must be handled similarly to a requests still in queue. Specifically:

Line `1d` *of Figure 13:* $\big((f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\mathtt{this}) = o\big)$
should be replaced by (with one more alternative case):

$$\big((f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\mathtt{this}) = o\ \vee$$
$$(f, execute, i, \mathtt{m}, \overline{v''})_A \mapsto \{\ell''|s''\} \in p \wedge o_\alpha.retrieve(i) = o \wedge \text{bind}(o, \mathtt{m}, \overline{v''}) = \{\ell'|s'\}\big)$$

Modulo these minor changes in $\approx$, the proof of Theorem 4.6 is similar to the proof of Theorem 4.5, but tedious and without novel aspects. Consequently, we only provide the principles of the proof and summarise the results on the simulation in Table 4.

2) **Proof principles.**

One of the most important point to consider is the handling of potentially inconsistent future updates. Recall that the problem is due to distributed first class futures: a MultiASP program can have several copies of a future, whereas only one such future exists in ABS. If we had an atomic future update in MultiASP, then the equivalence is trivial: when an Invk-Future rule in MultiASP is observed, an Await-False in ABS can be exactly observed. Instead, we ensure an execution schedule that is equivalent to atomic future update. To this end, we rely on the causal ordering of communications of MultiASP. Indeed, in between the update of the original copy of a future and the update of the last copy of this future, many events can occur. However, by ensuring causal ordering relationship of all communication events (i.e. requests and future updates), none of these events are related to the usage of this future's value, because otherwise they would necessarily occur after the considered future update. Consequently, the execution ordering is observationally equivalent to an atomic future broadcast (only unrelated events can be interleaved).

We rely on Table 4 for further commenting the simulation of the MultiASP translation with ABS executions. We review below each of the MultiASP rules. First the Assign-Local and Assign-Field MultiASP rules raise no particular comment, except that we can safely ignore them for the assignment of intermediate variables introduced by the translation. The Invk-Future MultiASP rule only activates if the current thread limit is a soft thread limit, which only happens in the translation of an ABS *await* statement, and

MultiASP rule	**ABS rule**	**Additional ABS rules**
Assign-Local	Assign-Local	–
Assign-Local-Tmp	–	–
Assign-Field	Assign-Field	–
Invk-Future	Await-False	Release-Cog
Invk-Passive	–	–/Read-Fut/Await-True
Activate-Thread	Activate	–
Serve	Activate	–
Invk-Active	Rendez-vous-Comm	–
Invk-Active-Meta	–	–
New-Object in an activity with no ABS object	New-Cog-Object	–
New-Object in a non-empty activity	New-Object	–
New-Active	–	–
Return-Local	–	–
Return	Return	–
Update	–	–
Set-Soft-Limit	–	–
Set-Hard-Limit	–	–

Table 4. Summary table of the simulation of MultiASP in ABS.

solely when the future is not resolved. Indeed, no additional future can be awaited in ABS. Additionally, the cog is released in ABS, which is a silent action that cannot be observed in MultiASP, note that this action is deterministic in the sense that an object that is idle can only release the *cog*lock. In the case of the Invk-Passive MultiASP rule, two cases are possible. The corresponding ABS rule can be the Read-Fut rule if the invoked method is *get*() and the current limit is a hard limit. Alternatively, the corresponding ABS rule can be the Await-True rule if the invoked method is *get*() and if the current thread limit is a soft thread limit. Lastly, the Invk-Passive MultiASP rule can also be not observable in ABS for all the requests that are introduced by the translation. Here, we use the fact that we do not consider in the proof the ABS local method invocations, else some specific cases should be added to handle local invocations of applicative methods. Then, the distinction between the Activate-Thread and Serve MultiASP rules is made depending on the status of the request in MultiASP: whether it has started to be served or not. Such a distinction does not exist in ABS. The Invk-Active MultiASP reductions that involve an *execute* request exactly match the Rendez-vous-Comm rule. Here, the preliminary steps and the additional steps performed in the translation are handled by the equivalence relation. Those steps also ensure that the object exists in ABS, and that it can be accessed through its cog. Obviously, the Invk-Active reductions of other methods (that we label Invk-Active-Meta) have no corresponding reduction in ABS. The New-Object MultiASP rule can either correspond to the instantiation of an object in the same cog (the New-Object ABS rule) or in a new cog (the New-Cog-Object ABS rule) if the store of the MultiASP activity in which the object is instantiated is empty (if it only contains the COG object). The RETURN MULTIASP rule is exactly matched with the RETURN ABS rule, except that in MULTIASP the request must be an *execute* request.

Finally, we can now easily list the ABS and MULTIASP rules that are involved but not observable in the simulation from MULTIASP to ABS:

- The silent ABS actions are: AWAIT-TRUE, READ-FUT, RELEASE-COG.
- The silent MULTIASP actions are: NEW-ACTIVE, INVK-PASSIVE, RETURN-LOCAL, UPDATE, SET-HARD-LIMIT, SET-SOFT-LIMIT, INVK-ACTIVE-META, and ASSIGN-LOCAL-TMP. We can note that except the INVK-PASSIVE MULTIASP rule, none of the aforementioned rules needs an additional reduction on the ABS side: the reduced configuration is always equivalent to the corresponding ABS configuration.

□