# Data-flow Explicit Futures

Ludovic Henrio



LABORATOIRE


INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS
UMR 7271

# Data-flow Explicit Futures

*Henrio Ludovic*
EQUIPE COMRED

Rapport de Recherche

04-2018

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

# Data-flow Explicit Futures

Henrio Ludovic[1]

EQUIPE COMRED
04-2018 - 42 pages

**Abstract :** A future is a place-holder for a value being computed, and we generally say that a future is *resolved* when the associated value is computed. In existing languages futures are either implicit, if there is no syntactic or typing distinction between futures and non-future values, or explicit when futures are typed by a parametric type and dedicated functions exist for manipulating futures. One contribution of this article is to advocate a new form of future, named *data-flow explicit futures*, with specific typing rules that do not use classical parametric types. The new futures allow at the same time code reuse and the possibility for recursive functions to return futures like with implicit futures, and let the programmer declare which values are futures and where synchronisation occurs, like with explicit futures. We prove that the obtained programming model is as expressive as implicit futures but exhibits a different behaviour compared to explicit futures. The second and main contribution of this article is an exhaustive overview and precise comparison of the semantics of futures in existing programming languages. We show that, when classifying future implementations, the most distinctive aspect is the nature of the synchronisation not the explicitness of the future declaration.

**Key-words** : Programming languages, Concurrency and Distribution, Synchronisation, Futures, Typing.

[1]Laboratoire I3S – CNRS – <ludovic.henrio@cnrs.fr>

# 1 Introduction

In programming languages, a future [6] is an entity representing the result of an ongoing computation. It is generally used to launch a sub-task in parallel with the current task and later retrieve the result computed by the sub-task. Sometimes a distinction is made between the future resolution, i.e. the availability of the result, and its *update*, i.e. the replacement of a reference to a future by the computed value. This article compares the use of futures in different programming languages and suggests a new paradigm for the design and implementation of futures. While the major contribution of this article is a simple twist in the way futures are designed and implemented, this contribution is justified by a formal description of the model, its properties, and a formal comparison with the two classical alternative design choices for futures.

This article first provides an overview of the different forms of future synchronisations and future creations that exist. The main distinction between existing future paradigms depends on whether the programmer is exposed to the notion of future. In some languages futures are not visible by the programmer and are pure runtime entities, whereas most of modern languages with futures use a parametric type to explicitly type futures, and the programmer can synchronise the execution of some statements with the end of the computation represented by the future. This synchronisation can either be a classical strict synchronisation blocking the execution, or registering a continuation to be executed upon availability of the future. Explicitly typed futures feature a crucial advantage and a crucial drawback. On the positive side the explicit access to future exposes the programmer to the potential synchronisation points in the program, and the potential delays that the programs can suffer, waiting for a result. On the other side, the use of parametric types for futures limits code re-usability: it is impossible to write a piece of code that would manipulate either a future or a value, performing only the future synchronisation if necessary. This strictness in the type system has major consequences concerning expressiveness, for example it is impossible to write a function that returns, either a value or the future that will be computed by another function, or at least not in a natural and simple way. Thus, writing a recursive function that calls itself in a parallel manner is generally artificially difficult, even for terminal recursion. Concerning implicit futures, they feature the complementary view: code re-usability is maximal as there is no syntactic difference between futures and standard objects, synchronisation occurs optimally when the result is needed, but the associated transparency makes the programmer unaware of synchronisation points, and complicates the detection of deadlocks. This article uses a toy example based a terminal recursive factorial function to illustrate these limitations.

Whereas previous studies [1, 7] focus on whether futures are explicit to distinguish several models, this article demonstrates that the crucial distinction between the different future constructs is rather the way synchronisation is performed. This becomes particularly visible when a parallel computation returns the result of another parallel computation. With implicit futures the synchronisation is data-driven: any future access will recursively wait for the termination of as many tasks as necessary to obtain a result. With explicit futures, there should be one synchronisation statement per recursive task: each synchronisation statement waits for the execution of a single statement that is supposed to return the result. To illustrate this, we introduce *data-flow explicit futures* that provide a compromise between usual explicit and implicit futures.

Actors [2, 31] and active object [7] languages are based on asynchronous communications between mono-threaded entities and massively use futures to represent replies to asynchronous

messages. We illustrate our proposal on a simple active object language but the approach is generalisable to other languages using futures. We exemplify future usage with a single simple and classical future usage primitive (`get`) that performs a strict synchronisation and discuss other future access primitives as an extension of this work.

**Proposal: Data-flow-synchronised explicit futures.** We designed a future paradigm providing the following characteristics:

**Explicit futures** Futures are explicitly typed and the point of access to a future is explicit in the program. The programmer is exposed to the points of synchronisation that occur in his/her program.

**Dafa-flow synchronisation** The language we propose has a simple synchronisation operation that is data-flow oriented: after a single synchronisation, the programmer is guaranteed to obtain a usable value: contrarily to explicit futures, it is never needed to perform a `get` on the result of another `get` operation.

**Dedicated type system** Our proposal comes with a simple type system for futures. This type system overcomes the limitations of the parametric types used in most of the type systems for futures.

**Code reuse** In our model, the same variable can receive either a future or a real value, provided it is declared of type future (which here means "potentially future"). For example, if the programmer declares that one method parameter might contain a future, this method can be invoked either with a future or with a normal value. In particular this allows us to easily write terminal recursive methods with parallel delegation.

**Approach: Behavioural equivalence and expressiveness.** In this article we are interested in the expressive and synchronisation power of futures. For this reason we design three languages only differing in the typing and synchronisation on futures. We investigate the existence of encodings from one language to the other without introducing powerful intermediate entities. Our objective is to understand if the synchronisation featured by one kind of futures can be simulated by the other kind, and not if one could simulate one kind of futures by adding additional objects and threads. In practice, the translations that we define in Section 4 only introduce additional variables and statements. Technically, the behavioural equivalence that we prove in this section is a branching bisimulation.

**Contribution and organisation.** This article presents:

- An overview of the different semantics for futures in programming languages, in Section 2.
- A proposal for a new kind of futures: explicit futures with data-flow synchronisation. The proposal is expressed in a minimalist active object language called `DeF`for data-flow explicit futures. Section 3 describes the semantics of this language, inspired from other semantics of active object languages, and its type system, mostly standard except concerning the typing of futures.
- The proof that data-flow explicit futures are as expressive as implicit futures. To this aim, Section 4 first expresses the semantics and typing of a language similar to `DeF` except that futures are implicit, it is called `IF`(implicit futures). Then we show how, only adding a few statements and variables, we can translate `DeF` programs into `IF` and conversely; we then prove that the translation preserves the behaviour.
- Evidences of the differences between traditional explicit futures and `DeF` in Section 5. We design a simple language with explicit futures, called `EF`, and show that the semantics of `EF` is different from `DeF`: in both directions, we provide counter-examples of programs

that cannot be translated faithfully from one language into the other.

Our main objective in this paper is not necessarily to advocate one kind of future semantics but to understand precisely and technically the differences between the different future kinds. For this we introduce a new future construct that is an intermediate between the existing ones, and that allow us to precisely and formally compare the semantics of explicit and implicit future types, and their synchronisation mechanisms. We however believe that the future paradigm of `DeF` is also a valuable compromise between the existing approaches, and constitutes a solid contribution of this article. Additionally, this article studies briefly another compromise between explicit and implicit futures: `DeFs` is a language similar to `DeF` except that it does not feature code reuse, instead it ensures that all `get` statements are necessary, and will perform a synchronisation. `DeFs` is only slightly different from `DeF` and will be presented in Section 3.3. It is compared with the other languages in Section 6.1.

## 2 A Brief Survey of Futures in Programming Languages

**The origins of futures.** A future is a programming abstraction that has been introduced by Baker and Hewitt [6]. It has then been used in programming languages like MultiLisp [19]. A future is not only a place-holder for a value being computed, it also provides naturally some form of synchronisation that allows some instructions to be executed when the result is computed. The synchronisation mechanism provided by futures is closely related with the flow of data in the programs. In MultiLisp, the future construct creates a thread and returns a future, the future can be manipulated by operations like assignment that do not need a real value, but the program would *automatically block* when trying to use a future for an operation that would require the future value (e.g. an addition). In MultiLisp, futures are implicit in the sense that they do not have a specific type and that there is no specific instruction for accessing a future but there is an explicit statement for creating them. The program only blocks when the value computed by another thread is necessary. Typed futures appeared with ABCL/f [30] in order to represent the result of asynchronous method invocations, i.e. methods invocations that are performed in parallel with the code that triggered the method.

The first work on formalisation by semantic rules of Futures appeared in [14, 15] and was intended at program optimisation. This work focused on the futures of MultiLisp, that are explicitly created but implicitly accessed. The authors "compile" a program with futures into a low-level program that does explicitly retrieves futures, and then optimise the number of necessary future retrievals. In a similar vein, $\lambda(fut)$ is a concurrent lambda calculus with futures with cells and handles. In [27], the authors define a semantics for this calculus, and two type systems. Futures in $\lambda(fut)$ are explicitly created, similarly to MultiLisp. Alice ML [26] can be considered as an implementation of $\lambda(fut)$.

One notion related to futures is the concept of *promises*. In general, a promise a future with an additional handler that can be used to resolve the future. In other words, there is an object that can be transmitted and then used to fill the future, this is the case for example in Akka promises. The advantage is that the resolution of the future is not anymore tight to a given funciton or process. The difficulty then is to ensure that a promise is resolved exactly once [32].

**Futures for actors and active objects.** An Actor is a mono-threaded entity that communicate with other actors by asynchronous message sending. An actor performs a single instruction at a time. The absence of multi-threading inside an actor and the fact that each

data is handled by a single actor prevents any form of data-race. However race-conditions still exist, typically when two actors send a message to the same third one, or when choosing the next message to be processed. Active objects unify the notion of Actor and objects, they give to each actor an object type and replace message passing by asynchronous method invocations: active objects communicate by calling methods on other active objects, asynchronously.

ABCL/f paved the way for the appearance of active object languages and the use of futures in these languages. In active object languages, to handle the asynchronous result of a method invocation, a future is used. First active object languages include like Eiffel// [3], ProActive [4], and Creol [23, 24].

ProActive [4] is a Java library for active objects; in ProActive futures are implicit like in MultiLisp, they are implemented with proxies that hide the future and provide an object interface similar to a standard object except that any access requiring the object's value (e.g. to call a method) may trigger a blocking synchronisation. The ASP calculus [10] formalises the Java library, and in particular allowed the proof of results of partial confluence, i.e. confluence under some restrictions. Indeed, futures are single-assignment entities; and the result of the computation is not sensible to the moment when futures are resolved provided they are accessed by a blocking synchronisation. Then, in a distributed setting, ProActive provides several future update mechanisms [20] featuring lazy and eager strategies to transport future values to their utilisation point.

Creol [24, 23] is also an active object language but with a non-blocking wait on future. A future is accessed with an *await* statement that enables *cooperative multi-threading* based on future availability: the execution of the current method is interrupted when reaching *await*, and only continued when the future is resolved. In the meantime other methods can be executed, which creates interleaving but prevents blocking the actor execution.

De Boer, Clarke, and Johnsen then provided in [11] the first richer version of future manipulation through cooperative scheduling and provided a compositional proof theory for the language. The future manipulation primitives of this language have been then used, almost unchanged, in JCobox [28], ABS [22], and Encore [8] three more recent active object languages. In those different works, futures are now explicitly typed with a parametric type of the form `Fut`$<T>$ and can be accessed in various ways (see below).

At the same time another active object language, AmbientTalk, was developed; it features a quite different semantics for future access. AmbientTalk [12] is based on the E Programming Language [25] which implements an actor model with a communicating event-loop. Futures in AmbientTalk are fully asynchronous: calls on futures trigger an asynchronous invocation that will be scheduled when the future is available. A special construct *when-becomes-catch* is used to define the continuations to be executed when the future is resolved.

In a more industrial setting, futures were introduced in Java in 2004 and used in one of the standard Java libraries for concurrent programming. A parametric type is used for future variables which are explicitly retrieved by a *get* primitive [17]. These simple futures are explicitly created and have a blocking access. Akka [18, 31] is a scalable library for implementing actors on top of Java and Scala. In Akka, futures are massively used, either to allow actor messages to return a value, or more automatically in the messages of the *typed actors*[2]. Akka also uses a parametric type for futures. Futures can be created explicitly with a *future* construct that creates a new thread.

**The different forms of future access.** Originally, futures were designed as synchronisa-

[2]Akka's typed actors are some kind of active objects.

tion entities, the first way to use futures is to block the thread that is trying to use a future. When futures are created transparently for the programmer, it is possible to enable the transmission of futures between entities without synchronisation; in this case the synchronisation occurs only when the future value is strictly needed. This synchronisation paradigm is called *wait-by-necessity*. Also in similar settings, futures that can be manipulated as any standard object of the languages are sometimes called *first-class futures*. In MultiLisp and ASP, future synchronisation is transparent and automatic.

Other languages provide future manipulation primitives; starting from touch/peek in ABCL/f, the advent of typed futures allowed the definition of richer and better checked future manipulation statements but also changed the synchronisation pattern, switching from a synchronisation on the availability of data to the synchronisation on a given instruction responsible for fulfilling the future. For example, a `get` statement in ABS is resolved by a corresponding `return` statement.

In some languages like Creol or AmbientTalk, futures can only be accessed asynchronously, i.e. the constructs for manipulating a future only allows the programmer to register some piece of code that will be executed when the future is resolved. In Creol, the principle is to interrupt the execution of the current thread while waiting for the future. This breaks the code sequentiality but enables more parallelism and can solve deadlock situations. In AmbientTalk, such asynchronous continuation can also be expressed but additionally future access can trigger an asynchronous method invocation that will be scheduled asynchronously after the current method is finished.

Nowadays, most languages with explicit futures provide different future access primitives. JCobox, Encore, and ABS let the programmer choose between a cooperative scheduling access using *await*, or a strict synchronisation preventing thread interleaving, using a *get* primitive. Interestingly, Encore also provides a way to trigger asynchronous method invocation on future resolution called *future chaining*, similarly to AmbientTalk.

Akka has a distinguished policy concerning future access. Blocking future access is possible in Akka (using *Await.result* or *Await.ready*, not to be confused with the await of ABS!). Instead, asynchronous future accesses should be preferred according to the documentation, like in AmbientTalk, by triggering a method invocation upon future availability. The underlying principle is to use *reactive programming*, both for messages and for future updates. Akka future manipulation comes with several advanced features such as ordering of asynchronous future accesses, failure handling (a future can be determined as a success or a failure, i.e. an exception), response time-out, etc.

Another interesting industrial use of futures is the Javascript language. Javascript features promises that are accessed asynchronously, similarly to Akka futures. The `then` operation can follow many resolve and thus the futures of Javascript are a form of data-flow synchronisation, similarly to ASP. Javascript promises are of course untyped, and thus somehow implicitly transmitted, while they are explicitly created and accessed. Somehow, Javascript promises complement nicely the set of existing future implementations: it is a form of *data-flow synchronisation asynchronous futures with explicit synchronisation and implicit typing*.

**Complex synchronisation patterns with futures.** It is worth mentioning that several languages provide primitive implementing more complex synchronisation patterns. For example, Encore can use futures to coordinate parallel computations [13], some operators gather several futures, or perform computation pipelining. In ProActive, a group of futures represents the result of a group communication, enabling SPMD (single program, multiple data)

computation with active objects [5]. Akka also provides methods for pipelining, iterating, and combining several future computations.

## Discussion

We now discuss the different design choices that exist in the use of futures and present their advantages and drawbacks. To illustrate these advantages, we refer to the 5 programs in Figure 1 implementing the same example in four languages. The figure illustrates with a minimal example the synchronisation aspects related to tail-recursive functions and the potential code reuse. It expresses a terminal recursive version of a factorial function, invoked twice from a `main` method, the second invocation passes the result of the first one to illustrate the transmission of futures as method parameters. We describe in this section the examples intuitively, a more precise description will be provided in the rest of the article, when the semantics of each language is formally defined. The four languages used in the example are our new language `DeF`, `IF` a language with implicit futures similar to `ASP`, and `EF` a language with explicit futures similar to Creol or ABS, plus a variant of `EF` with the primitive *await* that allows cooperative scheduling. The last example also uses `EF` but solves the deadlock of the third listing by introducing additional objects.

**Synchronous vs asynchronous future access.** A first important design choice about future access is whether the access to a future should be a synchronous or an asynchronous operation. The different forms of future access have been discussed above. With purely asynchronous future access there is no way to synchronise on a future at a given line of code. This has the great advantage to provide a deadlock-free programming model at the expense of an inversion of control forcing the programmer to face complex interleaving in code execution. On the contrary synchronous future access allows the programmer to write complete pieces of codes (e.g. the handling of a request) that run to completion if no deadlock appears. In Akka both accesses are possible but Akka advocates not to use blocking access to prevent deadlocks. Asynchronous access is adapted to reactive programming based on events, whereas synchronous future access provides more guarantees in terms of determinacy of results and is better adapted to the manipulation of stateful objects [10].

More advanced features like checking whether a future has been updated or waiting for the first update among a group of futures can introduce sources of non-determinacy more difficult to mitigate than asynchronous access.

**Explicit or implicit futures; factorial running example.** One crucial design choice about futures is whether the creation and use of futures should be explicit, with a dedicated type or syntax, or implicit, manipulated like any other object of the language. In most of the languages with futures, variables that can contain futures are given a parametric future type of the form `Fut<T>`, and some specific primitives must be used to create and manipulate futures. For example in ABS, ! performs an asynchronous method call, returns a future, and this future can be accessed by *await* or *get* primitives. Figure 1.d shows typical usage of futures in ABS; note the await that occurs line 8: it retrieves the future but allows another instance of the `fact` method to run in the meantime if the future is not resolved. Line 14 performs a `get` and retrieves the future value without letting the execution of another method be interleaved at this point. Figure 1.c shows a similar example in a language that has no `await` primitive. In this case a deadlock occurs during an execution of `get` because the active object that is stuck in the `get` statement needs to run another method to resolve the future, and each active object is mono-threaded. There are two important things to notice. First,

due to the use of parametric types, the method `fact` must perform a synchronisation on Line 8 because the return statement expects a value of type `int` even if the result is not used at this point. Second, because of the explicit synchronisation, the programmer knows that synchronisation (and potentially deadlocks) necessarily occurs during a `get` statement. This makes debugging easier, but also requires the programmer to place the synchronisation himself/herself, sometimes not at the most efficient place, like in this example.

The advantages of explicit futures is the control given to the programmer, and the fact that synchronisation points are explicit, and thus it is easier for the programmer to identify potential deadlocks. Also explicit futures allow for complex operations on futures like co-operative multi-threading or future chaining whereas, with implicit futures, only one default semantics is given to future access (in practice additional primitives are often provided but using them becomes complex).

On the other side, ASP is a typical example of a language with implicit futures. In ASP, there is no specific future type: a future that will be filled by a value of type $T$ also has type $T$. A method call can be either synchronous or asynchronous depending on the object that is invoked at runtime, and there is no primitive for accessing futures: an operation that requires a concrete value triggers a synchronisation. Typically, a synchronisation occurs upon method invocation on a future. In ProActive, implicit futures are implemented with proxies that encapsulate the behaviour of the future and receive the computed value. This is shown in Figure 1.b where variables that can contain futures are not statically identified, there is no type "future", and there is no synchronisation statement. We will see in Section 5 that a slightly modified program could (unexpectedly) deadlock on line 4.

A major advantages of implicit futures is that parts of the program can remain oblivious of whether they operate on regular values or on futures. Consequently, implicit future enable *wait-by-necessity* where the program is blocked only when a value is really needed, and futures can be transmitted *transparently* between program entities (methods, objects, etc.). Indeed, with implicit futures; a method that has been written for manipulating a regular object can as well receive a future.

Figure 1.a shows the same example written in `DeF`(i.e. with data-flow explicit futures), where synchronisation is explicit and occurs at the place written by the programmer but no synchronisation is needed inside `fact` because the type system is better adapted to the concept of future. A single `get` operation is required and only in the main block, it synchronises on the received parameter that can be a future. The deadlock mentioned above could also occur at line 4 but this time it corresponds to a `get` statement and is thus easier to spot. Finally, Figure 1.e illustrates a faithful encoding with explicit futures of the program 1.a, showing that additional objects and threads are necessary to encode data-flow synchronisation of futures in `EF`. Figure 1 also highlight the difference in the possibility to re-use code in each of future model: in `DeF` and `IF`, the `fact` method can be invoked with either a future or an integer as first parameter, it is not the case in the other languages.

**Data-flow vs. control-flow synchronisation; a new future paradigm.** The different synchronisation patterns between explicit and implicit futures have been first highlighted in [21] where the authors provide a backend for ABS implemented in ProActive, and the only difference in the behaviour is due to the nature of futures. Indeed synchronisation on an explicit future waits for the execution of the corresponding `return` statement whereas synchronisation on an implicit future waits for the availability of some useful data. This is only visible when a future contains another future, this is why such configurations were excluded

a) DeF(Fut«» is the constructor of future types)

```
Act{
 Fut«Int» fact(Fut«Int» nf, Int r){
  Fut«Int» y; Int n;
  n=get nf;
  if (n == 1) return r
  else {
   r = r*n;
   y = this.fact(n-1,r);
   return y }}
//MAIN
{ Act a,b; Fut«Int» y,w; Int z;
 a=new Act(); b=new Act();
 y = a.fact(3,1);
 w = b.fact(y,1);
 z = get w }
```

b) IF (similar to ASP)

```
Act{
 Int fact(Int n, Int r){
  Int y;
  if (n == 1) return r
  else {
   r = r*n;
   y = this.fact(n-1,r);
   return y }}
//MAIN
{ Act a,b; Int y,w;
 a=new Act(); b=new Act();
 y = a.fact(3,1);
 w = b.fact(y,1) }
```

c) EF (explicit futures with parametric type)

```
Act{
 Int fact(Int n, Int r){
  Fut<Int> x ;  Int m ;
  if (n==1)  return r
  else {
   r = r*n;
   x = this.fact(n-1,r);
   m = get x; //DEADLOCK
   return m }}
//MAIN
{ Act a,b; Fut<int> y,w; Int z;
 a=new  Act(); b=new  Act();
 y = a.fact(3,1);
 z = get y;       // earlier synchro
 w = b.fact(z,1);
 z = get w  }
```

d) EF with await (similar to ABS)

```
Act{
 Int fact(Int n, Int r){
  Fut<Int> x ;  Int m ;
  if (n==1)  return r
  else {
   r = r*n;
   x = this.fact(n-1,r);
   m = await x;
   return m }}
//MAIN
{ Act a,b; Fut<int> y,w; Int z;
 a=new  Act(); b=new  Act();
 y = a.fact(3,1);
 z = get y;
 w = b.fact(z,1);
 z = get w  }
```

e) EF with additional active object)

```
FutProxy{
 Int Unfold(Fut<Int> x) {
  Int y;
  y = get x;
  return y }
 Int MakeFuture(Int x) {
  return x
 }
}
Act{
 Fut<Int> fact(Int n, Int r){
  Fut<Int> x, m ; FutProxy fp;
  fp=new FutProxy();//creates
      Active Object for future
      management
  if (n==1) {
   m=fp.MakeFuture(r)
   return m }
  else {
   r = r*n;
   x = this.fact(n-1,r);
   m = fp.Unfold(x);
   return m }}
//MAIN
{ Act a,b; Fut<Fut<Int>> t;
    Fut<int> y,w; Int z;
 a=new  Act(); b=new  Act();
 t = a.fact(3,1);
 y = get t; z = get y;
 t = b.fact(z,1);
 w = get t; z = get w  }
```

Figure 1: A factorial example expressed in the different languages presented in this article.

in [21]. A stronger evidence that the two synchronisation patterns are fundamentally different is shown in [16] where the authors highlight that data-flow synchronisation can potentially wait for an arbitrary number of `return` statements, whereas with futures *à la* ABS each synchronisation waits for a single `return`.

This article explains these results on a more streamlined set of languages, only differing in the way futures are handled. We show that, even if previous results have highlighted the different semantics between explicit and implicit futures, the true distinctive feature between these approach is whether the synchronisation is data-flow or control-flow oriented. Implicit futures comes with a data-flow synchronisation but the synchronisation of explicit futures is based on the *control-flow*, it synchronises with the execution of the statement that fills the future. As a consequence one type of synchronisation cannot be statically encoded into the other one without creating additional objects or a complex control structure. To prove this aspect, we design an intermediate future construct that is at the same time explicit and with a data-flow synchronisation.

`DeF` is not only a useful tool to compare the expressiveness of the different languages, it also provides a valuable compromise between the existing approaches for futures. `DeF` provides explicit future type and explicit synchronisation points, helping the programmer to understand synchronisations and deadlocks. It also features data-flow driven synchronisation avoiding unnecessary `get` operations, and makes it possible to write pieces of code that are oblivious of whether they operate on regular values or on futures. We will prove that the obtained synchronisation patterns are similar to the ones of implicit futures, while making the presence of futures and their synchronisation explicit.

# 3 `DeF`: A Language for Explicit Futures with Data-flow Synchronisation

In this section we propose a core language for uniform active objects with explicit futures equipped with a data-flow oriented synchronisation. The model could be viewed as a core version of ABS with only a `get` primitive to access objects. The syntax is somehow inspired from the one of ABS. The language is called `DeF` for data-flow synchronised explicit futures. Except from the modelling of futures, we tried to adopt a design as simple as possible, but still modelling the principles of existing actor and active object languages.

**Notations.** $\overline{T}$ denotes a list of elements $T$, unless stated otherwise this list is ordered. In the syntax $x, y, u$ range over variable names, $m$ over method names, $\alpha, \beta$ range over active object identifiers, $f$ over future identifiers, and `Act` over class names. The set of binary operators on values is represented by an abstract operator $\oplus$, it replaces all the classical operations on integer and booleans. Mappings are denoted $[\overline{x} \mapsto \overline{a}]$ which builds a map from the two lists $\overline{x}$ and $\overline{a}$ of identical length, $m[x \mapsto a]$ updates a map, associating the value $a$ to the entry $x$, and $+$ merges two maps (taking values in the rightmost one in case of conflict). $\overline{q}\#q$ (resp. $q\#\overline{q}$) is the FIFO enqueue (resp. dequeue) operation.

## 3.1 Syntax and Semantics

Figure 2 shows the syntax of our language, including the type definitions discussed in the next section. A program $P$ is made of a set of classes named `Act`, each having a set of fields and a set of methods, plus a main method. A method $M$ has a name $m$ a set of parameters and

$$
\begin{array}{rcll}
P & ::= & \overline{\texttt{Act}\{\overline{T\ x}\ \ \overline{M}\}}\ \{\overline{T\ x}\ s\} & \text{program} \\
M & ::= & T\ \texttt{m}(\overline{T\ x})\ \{\overline{T\ x}\ s\} & \text{method} \\
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\,s\,\}\ \texttt{else}\ \{\,s\,\} \mid s\ ;\ s \mid \texttt{return}\ v & \text{statements} \\
z & ::= & e \mid v.\texttt{m}(\overline{v}) \mid \texttt{new Act}(\overline{v}) \mid \texttt{get}\ v & \text{right-hand-side of assignments} \\
e & ::= & v \mid v \oplus v & \text{expressions} \\
v & ::= & x \mid \texttt{null} \mid \textit{integer-and-boolean-values} & \text{atoms} \\
B & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Act} & \text{basic type} \\
T & ::= & B \mid \texttt{Fut}\ll B\gg & \text{Type}
\end{array}
$$

Figure 2: Static syntax of DeF.

a body, made of a set of local variables and a statement. Types and terms are standard of object languages except that `new` creates an active object, `get` accesses a future, and $v.m(\overline{v})$ performs a method invocation on an active object and thus systematically creates a future as will be shown in the semantics and the type system. The type constructor for future is $\texttt{Fut}\ll T\gg$, we chose a notation different from the standard $\texttt{Fut}<T>$ of ABS or Akka. We adopt this notation to syntactically show that a future type is not a standard parametrised type, but follows specific typing rules. Sequence is denoted ; and is associative with a neutral element `skip`. Consequently, each statement that is not `skip` can be rewritten as $s; s'$ with $s$ neither `skip` nor a sequence. $\oplus$ denotes the (standard) operations on integers and booleans. Several design choices had to be made in DeF, even if they are orthogonal to the subject of the paper we discuss them briefly below:

- For simplicity, we suppose that local variables and fields have disjoint names.
- We specify a service of requests in FIFO order like in ASP or Rebeca [29]. Another service policy could be specified. This choice is not related to the scope of the paper; we choose FIFO service because it is supported by many actor and active object implementations, probably because this makes programming of several interaction patterns easier.
- We define a sub-typing relation that only compares future and non-future types. Adding an additional sub-typing relation (e.g. based on the class of objects) raises no issue but is outside the scope of our study.
- In the design of a programming model for active objects, a crucial choice is to decide whether all objects are active or only some of them are. DeF is a uniform active object language, where all objects are active and all invocations are asynchronous and create a future. Many recent active object languages either have a non-uniform model where some (passive) objects can only be accessed synchronously by a single active objects, or *concurrent object groups* where several objects share the same execution thread. The interested reader is referred to [7] for a complete description of active object models.
  This aspect is also mostly orthogonal to the subject of our study. However if some objects can be invoked locally, a syntactic distinction between synchronous and asynchronous invocation might be necessary to identify the points of creation of futures. For example ABS and Creol use ! to identify asynchronous method invocations that create future. Note that the syntactic identification of future creation points is less crucial in DeF than in ABS because the sub-typing between futures and not futures allows us to perform a synchronous invocation when an asynchronous one is syntactically expected.

$$
\begin{array}{rcll}
cn & ::= & \overline{\alpha(a,p,\overline{q})}\ \overline{f(\bot)}\ \overline{f(w)} & \text{configuration} \\
p & ::= & \varnothing \mid q : \{\ell | s\} & \text{currentrequestservice} \\
q & ::= & (f, m, \overline{w}) & \text{request} \\
w & ::= & x \mid \alpha \mid f \mid \texttt{null} \mid \textit{integer-values} & \text{runtime values} \\
\ell & ::= & [\overline{x} \mapsto \overline{w}] & \text{local store} \\
a & ::= & [\overline{x} \mapsto \overline{w}] & \text{object fields} \\
e & ::= & w \mid v \oplus v & \text{expressions now can have runtime values} \\
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\, s \,\}\ \texttt{else}\ \{\, s \,\} \mid s\ \texttt{;}\ s \mid \texttt{return}\ v & \text{statements} \\
z & ::= & e \mid v.\texttt{m}(\overline{v}) \mid \texttt{new Act}(\overline{v}) \mid \texttt{get}\ w & \text{expressions with side effects}
\end{array}
$$

Figure 3: Runtime Syntax of DeF.

The operational semantics of DeF is shown in Figure 4; it expresses a small-step reduction semantics as a transition relationship between runtime configurations. The syntax of configurations and runtime terms is defined in Figure 3, statements are the same as in the static syntax except that they can contain runtime values like reference to an object or a future (inside assignment or get statement). A configuration is an *unordered* set of active objects and futures. Each active object is of the form $\alpha(a, p, \overline{q})$ where $\alpha$ is the active object identifier, $a$ stores the value of object fields, $p$ is the request currently served, and $\overline{q}$ a list of pending requests. The configuration also contains futures that can be either unresolved $\bot$ or resolved by a value. A request $q$ is characterised by the corresponding future $f$, the invoked method $m$, and a set of invocation parameters $\overline{w}$. The currently served request is either empty $\varnothing$ or made of the request identity $q$, and a pair $\{\ell \mid s\}$ containing a local environment $\ell$, and the statement to be evaluated.

The semantics uses the following auxiliary functions. The bind operator creates an execution context in order to evaluate a method. If the object $\alpha$ is of type Act, and m is defined in Act, i.e. Act{..$T$ m($\overline{T\ x}$) {$\overline{T\ y}$ $s$}..} is one class of the program $P$, then[3]: $\text{bind}(\alpha, (f, m, \overline{w})) \triangleq \{\, [\, \texttt{this} \mapsto \alpha,\ \overline{x} \mapsto \overline{w}\, ] \mid s \,\}$

To deal with assignment, we use a dedicated operator for updating the current fields or local variables: $(a + \ell)[x \mapsto w] = a' + \ell' \iff$ $a' = a[x \mapsto w]$ and $\ell' = \ell$, if $x \in \text{dom}(a)$, $a' = a$ and $\ell' = \ell[x \mapsto w]$, else

The semantics of a DeF program features the classical elements of active object programming [9, 22], the stateful aspects of the language are expressed as accesses to either local variables ($\ell$) or object fields ($a$). The three first rules of the semantics define an evaluation operator $[\![e]\!]_{a+\ell}$ that evaluates an expression. It is important to note that $[\![e]\!]_{a+\ell} = w$ implies that $w$ can only be an object or future name, null, or an integer or boolean value. The semantics of Figure 4 contains the following rules. Assign deals with assignment to either local variables or object fields. New creates a new active object at a fresh location $\beta$. Method invocation Invk enqueues a request in the target active object and systematically creates an undefined future $f$, the reference to the future can then be used (stored) by the invoker $\alpha$. The rule Invk-Self deals with the particular case where the target is the invoking object. Return evaluates a return statement and resolves the corresponding futures, finishing a request

[3]It is not necessary to initialize the local variables in the local environment because of the way the update operation on store and object fields is defined.

$$\frac{w \text{ is not a variable}}{[\![w]\!]_\ell \;=\; w} \qquad \frac{x \in \mathrm{dom}(\ell)}{[\![x]\!]_\ell \;=\; \ell(x)} \qquad \frac{[\![v]\!]_\ell = k \qquad [\![v']\!]_\ell = k'}{[\![v \oplus v']\!]_\ell \;=\; k \oplus k'}$$

Assign
$$\frac{[\![e]\!]_{a+\ell} = w \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, q : \{\ell \mid x = e \;;\; s\}, \overline{q'}) \to \alpha(a', q : \{\ell' \mid s\}, \overline{q'})}$$

Context
$$\frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}$$

New
$$\frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \;\text{ fresh} \qquad \overline{y} = \mathit{fields}(\texttt{Act})}{\alpha(a, q{:}\{\ell \mid x = \texttt{new Act}(\overline{v}) \;;\; s\}, \overline{q'}) \to \alpha(a, q{:}\{\ell \mid x = \beta \;;\; s\}, \overline{q'})\ \ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)}$$

Invk
$$\frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \qquad f \;\text{ fresh}}{\alpha(a, q{:}\{\ell | x{=}v.\texttt{m}(\overline{v}); s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}) \to \alpha(a, q{:}\{\ell | x{=}f; s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}\#(f, m, \overline{w}))\ f(\bot)}$$

Invk-Self
$$\frac{[\![v]\!]_{a+\ell} = \alpha \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad f \;\text{ fresh}}{\alpha(a, q : \{\ell \mid x = v.\texttt{m}(\overline{v}) \;;\; s\}, \overline{q'}) \to \alpha(a, q : \{\ell \mid x = f \;;\; s\}, \overline{q'}\#(f, m, \overline{w}))\ f(\bot)}$$

Return
$$\frac{[\![v]\!]_{a+\ell} = w}{\alpha(a, (f, m, \overline{w}) : \{\ell \mid \texttt{return}\ v\}, \overline{q})\ f(\bot) \to \alpha(a, \varnothing, \overline{q})\ f(w)}$$

Serve
$$\frac{\mathrm{bind}(\alpha, q) = \{l|s\}}{\alpha(a, \varnothing, q\#\overline{q'}) \to \alpha(a, q : \{l|s\}, \overline{q'})}$$

Get-Update
$$\frac{[\![w]\!]_{a+\ell} = f}{\alpha(a, q : \{\ell \mid y = \texttt{get}\ w \;;\; s\}, \overline{q'})\ f(w') \to \alpha(a, q : \{\ell \mid y = \texttt{get}\ w' \;;\; s\}, \overline{q'})\ f(w')}$$

Get-Resolved
$$\frac{[\![w]\!]_{a+\ell} = w' \qquad \nexists f.w' = f}{\alpha(a, q : \{\ell \mid y = \texttt{get}\ w \;;\; s\}, \overline{q'}) \to \alpha(a, q : \{\ell \mid y = w' \;;\; s\}, \overline{q'})}$$

If-True
$$\frac{[\![v]\!]_{a+\ell} = \texttt{true}}{\alpha(a, q : \{\ell \mid \texttt{if}\ v\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ ;\ s\}, \overline{q'}) \to \alpha(a, q : \{\ell \mid s_1 \;;\; s\}, \overline{q'})}$$

If-False
$$\frac{[\![v]\!]_{a+\ell} \neq \texttt{true}}{\alpha(a, q : \{\ell \mid \texttt{if}\ v\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ ;\ s\}\}, \overline{q'}) \to \alpha(a, q : \{\ell \mid s_2 \;;\; s\}\}, \overline{q'})}$$

Figure 4: Semantics of DeF.

service so that a new request can be served. Serve occurs when no request is being served, it dequeues a request and starts its execution. These rules ensure a strict single-threaded execution of each request one after the other. The most original and interesting aspect of the semantics is the two rules that deal with the `get` statement: Get-Update fetches the value associated to a future but this value is kept under the `get` statement, this rule is applied repetitively until the rule Get-Resolved is applicable, i.e. until the value inside `get` is not a reference to a future, at this point the `get` statement is removed. This way the `get` statement always returns a usable value. Note that a `get` can perfectly be called on a value that is not a future, in which case it has no effect.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{T-Var}) \qquad \frac{}{\Gamma \vdash \texttt{null} : Act}\ (\text{T-Null}) \qquad \frac{\Gamma \vdash z : T}{\Gamma \vdash z : \texttt{Fut}\ll T\gg}\ (\text{T-Subtype}) \qquad \frac{\Gamma(x) = T \qquad \Gamma \vdash e : T}{\Gamma \vdash x = e}\ (\text{T-Assign})$$

$$\frac{\mathit{fields}(\texttt{Act}) = \overline{T}\ \overline{x} \qquad \Gamma \vdash \overline{v} : \overline{T}}{\Gamma \vdash \texttt{new Act}(\overline{v}) : \texttt{Act}}\ (\text{T-New}) \qquad \frac{\Gamma \vdash v : T \qquad \Gamma \vdash v' : T' \qquad \oplus : T \times T' \to T''}{\Gamma \vdash v \oplus v' : T''}\ (\text{T-Expression})$$

$$\frac{\Gamma \vdash v : \texttt{Fut}\ll B\gg}{\Gamma \vdash \texttt{get}\ v : B}\ (\text{T-Get}) \qquad \frac{\Gamma \vdash v : \texttt{Act} \qquad \Gamma(\texttt{Act})(\texttt{m}) = \overline{T} \to T' \qquad \Gamma \vdash \overline{v} : \overline{T} \qquad \text{if } T' \text{ is of the form } \texttt{Fut}\ll B\gg \text{ then } T'' = T' \text{ else } T'' = \texttt{Fut}\ll T'\gg}{\Gamma \vdash v.\texttt{m}(\overline{v}) : T''}\ (\text{T-Invk})$$

$$\frac{\Gamma \vdash v : \texttt{Bool} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ v\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}}\ (\text{T-If}) \qquad \frac{\Gamma \vdash e : T \qquad \Gamma(\texttt{Act})(\texttt{m}) = \overline{T} \to T}{\Gamma \vdash \texttt{return}\ e}\ (\text{T-Return})$$

$$\frac{\Gamma \vdash s \qquad \Gamma \vdash s'}{\Gamma \vdash s\ ;\ s'}\ (\text{T-Seq}) \qquad \frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}][\overline{x'} \mapsto \overline{T'}] \qquad \Gamma' \vdash s}{\Gamma \vdash T''\ \texttt{m}\ (\overline{T\ x})\{\overline{T'\ x'}\ \ s\}}\ (\text{T-Method}) \qquad \frac{}{\Gamma \vdash \texttt{skip}}\ (\text{T-Skip})$$

$$\frac{\Gamma[\overline{x'} \mapsto \overline{T'}] \vdash s \qquad \forall \texttt{Act}\{\overline{T\ x}, \overline{M}\} \in \overline{\texttt{Act}\{\overline{T\ x}, \overline{M}\}} \cdot \forall M \in \overline{M} \cdot \Gamma[\overline{x} \mapsto \overline{T}] \vdash M}{\Gamma \vdash \overline{\texttt{Act}\{\overline{T\ x}, \overline{M}\}}\ \{\overline{T'\ x'}\ \ s\}}\ (\text{T-Program})$$

Figure 5: Type system (each operator $\oplus$ has a predefined signature).

The *initial configuration* for running a DeF program $\overline{\texttt{Act}\{\overline{T\ x}\ \ \overline{M}\}}\ \{\overline{T\ x}\ s\}$ consists of a single object serving a single request with body defined by the main method, the identity of the request is useless as no other object will fetch the result: $\alpha(\varnothing, (f, m, \varnothing) : \{\varnothing | s\}, \varnothing)$

### 3.2 A Type System for DeF

We define a simple type system for DeF (the syntax of types is in Figure 2). Most of the type system is standard, but the typing of the new future type requires some specific rules. In DeF a standard value may always be used when a future is expected, and this must be reflected in the type system. In other words, $\texttt{Fut}\ll T\gg$ is not a parametrised type but tags the type $T$ with the information that the term *may be* a future. On the contrary, $T$ is the type of an entity that is not a future. In DeF, it is not possible to write a type of the form $\texttt{Fut}\ll\texttt{Fut}\ll T\gg\gg$; indeed get unfolds as many future accesses as necessary to obtain a real value. Somehow, $\texttt{Fut}\ll T\gg$ would correspond, in a parametric type system for futures, to any positive or null number of embedded future construct $\texttt{fut}^* < T >$ .

The type checking rules are defined in Figure 5. The sub-typing rule T-Subtype states that a non-future term can always be considered as of type future. The rule T-Get guarantees that the type of the term obtained after a get is a basic type, and thus the result of a get

can be used directly. The other non-trivial rule is the typing of method invocation (T-Invk). This rule tags as "future" the result of the method invocation, if the method is declared to return a basic type we add the `fut` tag, but if the method is declared to return a future (e.g. because the `return` statement returns the result of another method invocation) then no tag is added; we do not have a `fut` type constructor contained in another one like one could have in ABS. The *initial typing environment* $\Gamma$, which types the program, associates to each class name a mapping from method names to method signatures. If `m` is a method of class `Act` defined as follow $T''$ `m` $(\overline{T\ x})\{\overline{T'\ x'}\ \ s\}$, we will have $\Gamma(\texttt{m}) = \overline{T} \to T''$.

This type system allows code reuse because a method that accepts a future can be invoked with a non-future value. The programmer can also write a recursive method that can either return a future or a standard value, it is sufficient to declare that this method potentially returns a future.

**Example.** Figure 1.a illustrates `DeF` and shows the possibility to return and to accept as parameter either an integer or a future. It also shows that it is easy to express tail-recursion because the typing of futures entails that a term tagged as future can require to unfold zero or many futures to access the value[4]. The synchronisation point is clearly marked with a *get* statement. The method `fact` first synchronises the integer passed as parameter, in case it is a future. We create two active objects $a$ and $b$ because if both invocations to `fact` were addressing the same active object, depending on request ordering a deadlock might occur (the active object would be blocked in the `get` statement at line 4).

**Properties of the type system.** Properties of the type system rely on the extension of the type system to runtime configurations. It is not possible to type configurations based on the only informations recorded in the semantics of Figure 4. Indeed, the main information missing is the type of active objects and futures. Even if the type of active object could potentially be inferred, it is impossible to infer the type of the futures because of the possibility to create a cycle of futures and obtain a configuration of the form $cn\ f(f)$. Indeed the ASP example creating a cycle of futures defined in [9] can be written in `DeF` (see Appendix 1 for the `DeF`version). Thus, to type configurations we annotate activities and futures with their types (we record the type of each object and each future at creation): $cn ::= \overline{\alpha_{\texttt{Act}}(a, p, \overline{q})}\ \overline{f_T(\bot)}\ \overline{f_T(w)}$. Except from this point, extending the type system to configurations is trivial. We write $\Gamma \vdash cn$ if the configuration $cn$ is well-typed in the environment $\Gamma$ (where $\Gamma$ is the initial typing environment defined above).

It is easy to check that `DeF` type system verifies *subject reduction*: if $\Gamma \vdash cn$ and $cn \to cn'$ then $\Gamma \vdash cn'$. For example the correct typing of the future value is ensured by the fact that the `return` statement is well-typed in the initial configuration (i.e. it has the return type of the method). This also ensures that the `get` statement is well-typed (accordingly to the future type and the return type of the method), and thus the Get-Update reduction rule does not change the type of the term under the `get` statement.

One can then notice that runtime values of type $B$ cannot be futures (see Figure 2). Consequently, concerning objects, variables of type `Act` either are `null` or point to an object of the right type. Thus, except for non-initialised objects, method invocations always succeed, and in particular: (1) each invocation creates a request on a method existing in the target object, and (2) the invoked object cannot be a future. Similar conclusions can be drawn for primitive types. This is in particular reflected in the property below that states that `DeF` has the desirable characterisation of blocked tasks: an object can only be blocked if it is idle, it is

[4]Note that the `fact` method must be given the return type `Fut`$\ll Int \gg$.

accessing `null`, or it is waiting for a future to be resolved.

**Property 1** (Blocked activity)**.** *For any configuration cn for any object $\alpha(a, p, \overline{q})$ of this configuration, A reduction rule involving the object $\alpha$ can always be applied except if:*

- *The object is* idle*: $p = \varnothing$ and $\overline{q} = \varnothing$; or*
- *The object* invokes `null`*: $p = \{\ell | x = v.m(\overline{v}); s\}$ and $[\![v]\!]_{a+\ell} =$ `null`; or*
- *The object is* waiting for a future resolution*: $p = \{\ell | x =$ `get` $v; s\}$ and there is a future identifier $f$ such that $[\![v]\!]_{a+\ell} = f$ and $f(\perp) \in cn$.*

This property can be seen as a soundness theorem ensuring that all accesses to asynchronous results are protected by a get. It would be easy and classical to prove a "no message not understood" theorem, from Property 1 and subject reduction the proof features no particular interest or difficulty.

### 3.3 `DeFs`: A variant of `DeF` without useless `get` operations

Because `DeF` is designed so that a method can receive as parameter an int or a future (provided the programmer takes care of the future case), we need to accept `get` $x$ statements where $x$ can be mapped to 3 or a future of value 3. As a consequence a `get` operation can perfectly lead to no synchronisation at all. It would be possible to design another future type system, call it `DeFs` a language with *data-flow explicit futures and strict synchronisation*. `DeFs` is identical to `DeF` except there is no future subtyping rule (T-Subtype). If the `Fut`$\ll T \gg$ construct of `DeF` can be informally understood as a `Fut*<T>` type, the `Fut`$\ll T \gg$ construct of `DeFs` can be seen as a `Fut+<T>`, meaning one or more imbricated futures. `DeFs` allows the definition of recursive asynchronous functions and ensure necessity of `get` operation: each `get` performs one or several synchronisations. This is formalised as follows:

**Property 2** (In `DeFs` every `get` is a synchronisation)**.** *In* `DeFs`*, any* `get` *operation performs a synchronisation. In other words, every* Get-Resolved *reduction is preceded by a* Get-Update *reduction on the same activity. More formally, for any initial configuration $cn_0$, for any reduction chain: $cn_0 \rightarrow^* cn_1 \xrightarrow{T} cn_2 \rightarrow^* cn_3 \overset{\text{Get-Resolved}}{\rightarrow} cn_4$ where $cn_3 \overset{\text{Get-Resolved}}{\rightarrow} cn_4$ occurs on $\alpha$ and $cn_2 \rightarrow^* cn_3$ are reductions not occurring on $\alpha$*[5]*, and $cn_1 \xrightarrow{T} cn_2$ occurs in $\alpha$, then $cn_1 \xrightarrow{T} cn_2$ is a synchronisation reduction and thus $T =$* Get-Update*.*

In this work, we favour code reusability instead of preventing useless get for software engineering reasons, this is why we mostly focus on `DeF`. However `DeFs` provides another compromise between `IF` and `EF`. `DeFs` has one additional property with no useless get, despite the absence of code reusability.

## 4 Comparison with Implicit Futures

In this section we describe a programming language with implicit futures. More precisely, we show the syntax and the semantics of a language similar to our proposal, only differing in the way futures are created and declared, i.e., a language with a semantics for futures similar to the one of ASP but keeping our simplistic formalisation of objects. We call this language `IF`

[5]Invocation Invk is considered to occur on the source activity because only the request queue of the destination is modified by the rule.

$$
\begin{array}{rcll}
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\, s \,\}\ \texttt{else}\ \{\, s \,\} \mid s\ ;\ s \mid \texttt{return}\ v & \text{statements} \\
T & ::= & B & \text{Type}
\end{array}
$$

Figure 6: The Syntax of IF and IF types (terms identical to DeF omitted).

$$
\textsc{Update}\ \frac{[\![x]\!]_{a+\ell} = f \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{\begin{array}{c}\alpha(a, q : \{\ell \mid s\}, \overline{q'}))\ f(w) \\ \rightarrow_I \alpha(a', q : \{\ell' \mid s\}, \overline{q'})\ f(w)\end{array}}
\qquad
(\text{T-Invk})\ \frac{\Gamma \vdash_I v : \texttt{Act} \qquad \Gamma(\texttt{Act})(\texttt{m}) = \overline{T} \to T' \qquad \Gamma \vdash_I \overline{v} : \overline{T}}{\Gamma \vdash_I v.\texttt{m}(\overline{v}) : T'}
$$

Figure 7: Semantics and type system of IF (the other rules are identical to DeF).

for "implicit futures", this language also has a data-flow synchronisation, since the programmer is not aware of the location of futures and can only trigger a synchronisation when trying to use a variable that holds a future. This calculus is a typical example of the notion of *wait-by-necessity* that consists in blocking the execution of a program only when a future value is absolutely needed: there is no way to wait for the resolution of a future except using the value that is stored in it. We show that both languages have a similar expressiveness as it is possible to translate one into the other in a quite natural way. The translation however highlights the particularities of the two languages.

### 4.1 Semantics of Implicit Futures (IF)

Compared to DeF, the syntax of IF has no get statement, and there is no future type, see Figure 6. The semantics of IF is denoted $\rightarrow_I$. It is the same as the one of DeF except that the two rules Get-Update and Get-Resolved are replaced by a single Update rule shown in Figure 7. This rule can be triggered at any point when a variable holds a reference to a future that has been resolved; it replaces the value of a variable holding the future reference by the future value. Note that this rule changes the local store of an activity in a transparent manner, but this makes sense in IF because there is no future type, and a variable holding a future reference is considered, by the programmer, as directly containing the value that is returned by the method invocation (after some time).

Typing is also simple; we denote $\vdash_I$ the new typing judgement. The rules T-Get and T-Subtype can be removed as they are not useful. Rule T-Invk is modified as shown in Figure 6. As there is no syntactic difference between futures and other values, the method invocation simply returns a basic type. Note that, except `null`, each term has a single type, it was not the case in DeF because of the sub-typing between future and non-future types. The initial configuration for an IF program is identical to the one for DeF.

**Example.** Figure 1.b shows an IF program for the factorial computation. Futures are not declared explicitly, and tail-recursive functions can be expressed easily. The Update rule will automatically transmit the computed result, in several steps, to the point of utilisation. The code reuse is even stronger than in DeF because the method can be written without stating that the parameter can be a future, and thus methods that were written in a sequential setting can be invoked with a future. The synchronisation automatically occurs as late as possible, in

$$[\![T\ \mathtt{m}(\overline{T\ x})\ \{\overline{T'\ x'}\ s\}]\!]_{\mathtt{IF}\to\mathtt{DeF}} \triangleq T\ \mathtt{m}(\overline{T\ x})\ \{\overline{T'\ x'}\{T\ y_T, T\ z_T | \exists v \in s.\ \vdash_I v : T\}\ s\}$$

$$[\![B]\!]_{\mathtt{IF}\to\mathtt{DeF}} \triangleq \mathtt{Fut}\ll B\gg$$

$$[\![\mathtt{if}\ v\ \{\ s\ \}\ \mathtt{else}\ \{\ s'\ \}]\!]_{\mathtt{IF}\to\mathtt{DeF}} \triangleq y_{\mathtt{Bool}} = \mathtt{get}\ v; \mathtt{if}\ y_{\mathtt{Bool}}\ \{\ s\ \}\ \mathtt{else}\ \{\ s'\ \}$$

$$\frac{\vdash_I v : T}{[\![x = v.m(\overline{v})]\!]_{\mathtt{IF}\to\mathtt{DeF}} \triangleq \begin{array}{l} y_T = \mathtt{get}\ v; \\ x = y_T.m(\overline{v}) \end{array}} \qquad \frac{\vdash_I v : T \qquad \vdash_I v' : T'}{[\![x = v \oplus v']\!]_{\mathtt{IF}\to\mathtt{DeF}} \triangleq \begin{array}{l} y_T = \mathtt{get}\ v; \\ z_{T'} = \mathtt{get}\ v'; \\ x = y_T \oplus z_{T'} \end{array}}$$

Figure 8: Translation from IF to DeF (other terms are unchanged).

this case, at the same point as in the Figure 1.a. The main drawback of this approach is that it can be difficult for the programmer to analyse synchronisation points and deadlocks. For example if $a = b$, both factorial invocations target the same object and the active object may be blocked at line 4 (when checking if a future equals 1). The programmer would probably not notice that this line could be a synchronisation point.

### 4.2 Encoding IF into DeF

In this section we show how to encode an IF program into a DeF program that has the same behaviour. The translation is relatively simple and highlights the fact that DeF makes explicit the synchronisation points. The principle of the translation is to suppose that every IF term can be a future, and to generate too many get statements, one at each point a statement uses a value. At runtime, most of those get statements will have no effect and be immediately resolved by the rule Get-Resolved. This is made possible because a term that is not a future can always replace a term that is supposed to be a future in DeF, we can thus suppose that all entities are futures even if most of them are not.

First, we adopt the following abuse of notation. We use $\vdash_I v : T$ to type the atom $v$ when placed in the adequate typing context $\Gamma$ without having to specify $\Gamma$ (it corresponds to the point of the program considered). We will use the same abusive notation for all the type judgements. Except for the adaptation explained below and the rules defined in Figure 8 an IF term is translated identically into a DeF term.

**Additional variables.** We first add to each method two local variables of each possible type: *For any $v$ appearing in the method body, if $\vdash_I v : T$ then two* fresh *variables $y_T$ and $z_T$ of type $T$ are declared in the method containing $v$.* They will be used as intermediate variables in the encoding. See first line of Figure 8.

**Typing translation.** In IF every variable can be a future. To reflect this fact, we translate every type into a type tagged as future (recall that $T$ and $B$ are identical in IF). See the second line of Figure 8. The only variables with a non-future types are the additional variables introduced above.

**Statement translation.** Only statements potentially triggering a wait-by-necessity, i.e. a transparent synchronisation on a future, need to be changed. For each such statement we add one (or two) get statement retrieving the future value in case the accessed variable contains

$$\frac{cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash a\,\mathcal{R}_1\,a_{\mathrm{D}} \qquad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash \ell\,\mathcal{R}_1\,NoTmp(\ell_{\mathrm{D}}) \qquad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash q\,\mathcal{R}_1\,q_{\mathrm{D}} \quad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash \overline{q'}\,\mathcal{R}_1\,\overline{q'_{\mathrm{D}}} \qquad cn, cn_{\mathrm{D}}, a+\ell, a_{\mathrm{D}}+\ell_{\mathrm{D}} \vdash s\,\mathcal{R}_1\,s_{\mathrm{D}} \qquad cn\,\mathcal{R}_1\,cn_{\mathrm{D}}}{\alpha(a, q:\{\ell|s\}, \overline{q'})\ cn \ \ \mathcal{R}_1 \ \ \alpha(a_{\mathrm{D}}, q_{\mathrm{D}}:\{\ell_{\mathrm{D}}|s_{\mathrm{D}}\}, \overline{q'_{\mathrm{D}}})\ cn_{\mathrm{D}}}$$

$$\frac{cn\,\mathcal{R}_1\,cn_{\mathrm{D}}}{f(\bot)\ cn\ \mathcal{R}_1\ f(\bot)\ cn_{\mathrm{D}}} \qquad \frac{f(w)\ cn, f(w_{\mathrm{D}})\ cn_{\mathrm{D}}, \varnothing, \varnothing \vdash w\,\mathcal{R}_1\,w_{\mathrm{D}} \quad cn\,\mathcal{R}_1\,cn_{\mathrm{D}}}{f(w)\ cn\ \mathcal{R}_1\ f(w_{\mathrm{D}})\ cn_{\mathrm{D}}} \qquad \frac{f(w) \in cn}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\,\mathcal{R}_1\,w_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash f\,\mathcal{R}_1\,w_{\mathrm{D}}}$$

$$\frac{f(w_{\mathrm{D}}) \in cn_{\mathrm{D}} \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\,\mathcal{R}_1\,w_{\mathrm{D}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\ \ \mathcal{R}_1\ \ f} \qquad \frac{}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ \ \mathcal{R}_1\ \ [\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}}}$$

$$\frac{\left([\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} u_T = \mathtt{get}\ v; \\ s' \end{array} \ \vee\ [\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} y_{T'} = \mathtt{get}\ v'; \\ u_T = \mathtt{get}\ v; \\ s' \end{array}\right) \quad s_{\mathrm{D}} = (u_T = \mathtt{get}\ w_{\mathrm{D}}; s') \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![v]\!]_\ell\,\mathcal{R}_1\,[\![w_{\mathrm{D}}]\!]_{\ell_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ \ \mathcal{R}_1\ \ s_{\mathrm{D}}}$$

$$\frac{\left([\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} u_T = \mathtt{get}\ v; \\ s' \end{array} \ \vee\ [\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} y_{T'} = \mathtt{get}\ v'; \\ u_T = \mathtt{get}\ v; \\ s' \end{array}\right) \quad s_{\mathrm{D}} = (u_T = w_{\mathrm{D}}; s') \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![v]\!]_\ell\,\mathcal{R}_1\,[\![w_{\mathrm{D}}]\!]_{\ell_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ \ \mathcal{R}_1\ \ s_{\mathrm{D}}}$$

$$\frac{\left([\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} u_T = \mathtt{get}\ v; \\ s_{\mathrm{D}} \end{array} \ \vee\ [\![s]\!]_{\mathtt{IF}\to\mathtt{DeF}} = \begin{array}{l} y_{T'} = \mathtt{get}\ v'; \\ u_T = \mathtt{get}\ v; \\ s_{\mathrm{D}} \end{array}\right) \quad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![v]\!]_\ell\,\mathcal{R}_1\,[\![u_T]\!]_{\ell_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ \ \mathcal{R}_1\ \ s_{\mathrm{D}}}$$

Figure 9: Equivalence between `IF` and `DeF` configurations. Term-by-term equivalence and syntactic equality omitted. $NoTmp(\ell) = \{[x \mapsto \ell(x)| \vdash x : B\}$

a future. See the three last cases of Figure 8.

**Correctness of the translation.** We prove here that the translation defined above preserves the behaviour of the original `IF` program. More precisely we prove that, considering some of the reduction rules as non-observable, there is a *branching bisimulation* between the execution of an `IF` program and the execution of its translation.

The reductions that cannot be observed faithfully are the following ones:

- The `IF` rule UPDATE: This rule happens automatically and transparently at any point in time whereas future update occurs upon need in `DeF`.
- symmetrically `DeF` rules GET-UPDATE and GET-RESOLVED only occur at a specific point in `DeF` and cannot be faithfully matched with the transparent future update of `IF`.

- finally, the assignment (Assign) of the variables $y_T$ introduced by the translation cannot be matched with any `IF` rule. Note that this concerns local variables that have a non-future type in the translation: $\vdash y_T : B$ for some basic type $B$.

Despite the non observability of these rules, the other reductions, like method invocations, are simulated faithfully, and thus the program features the correct behaviour. We let $\tau$ range over these non-observable transitions. Additionally, we label each reduction step by the name of the only applied rule that is not Context. By convention, in the rest of this section we use $t_{\text{D}}$ to range over terms of `DeF` and $t$ over terms of `IF`.

Figure 9 defines a relation $\mathcal{R}_1$ between a configuration reached when evaluating an `IF` program and a configuration reached when evaluating the translation in `DeF` of this program. The relation $\mathcal{R}_1$ is either of the form $cn\,\mathcal{R}_1\,cn_{\text{D}}$, checking the equivalence between an `IF` configuration $cn$ and a `DeF` configuration $cn_{\text{D}}$, or of the form $cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash t\,\mathcal{R}_1\,t_{\text{D}}$, checking the equivalence between a term $t$ (inside the `IF` configuration $cn$) and a term $t_{\text{D}}$ (inside the `DeF` configuration $cn_{\text{D}}$); in this case $\ell$ (resp. $\ell_{\text{D}}$) contain the values of the variables that can be referred by $t$ (resp. $t_{\text{D}}$). Term-by-term and syntactic equality are omitted in the figure (e.g., mappings are equivalent if the value associated to each variable is equivalent). The three first rules deal with equivalence of activities and futures; *temporary variables* introduced by the translation are ignored when comparing stores, the other elements are compared term-by-term. The next two rules handle *the following of future references*: in `IF` and in `DeF` the update of futures do not occur at the same moment, thus we allow the equivalence relation $\mathcal{R}_1$ to follow future references, both in `IF` and `DeF` (one should finally arrive to a common term to assert equivalence). The last four rules define *statement equivalence*; when one statement can be translated into several with explicit synchronisation, the execution in `DeF` can be at any intermediate step. We now state correctness of the translation.

**Theorem 1.** *[Correctness of the translation] $\mathcal{R}_1$ is a branching bisimulation between the operational semantics of the* `IF` *program $P$ and the operational semantics of the* `DeF` *program $[\![P]\!]_{\texttt{IF}\to\texttt{DeF}}$. The transitions* Update, Get-Update, Get-Resolved, *and* Assign *of the local intermediate variables are non-observable. More formally, with $R$ (resp. $\tau$) ranging over observable (resp. non-observable) rule names, if $cn\,\mathcal{R}_1\,cn_{\text{D}}$ then:*

$$cn \xrightarrow{\tau}_I{}^{*}\ cn' \implies cn'\,\mathcal{R}_1\,cn_{\text{D}} \qquad\qquad cn_{\text{D}} \xrightarrow{\tau}{}^{*}\ cn'_{\text{D}} \implies cn\,\mathcal{R}_1\,cn'_{\text{D}}$$

$$cn \xrightarrow{R}_I{}^{*}\ cn' \implies \exists cn'_{\text{D}}.\ cn_{\text{D}} \xrightarrow{\tau}{}^{*}\xrightarrow{R} cn'_{\text{D}} \wedge cn'\,\mathcal{R}_1\,cn'_{\text{D}}$$

$$cn_{\text{D}} \xrightarrow{R}{}^{*}\ cn'_{\text{D}} \implies \exists cn'.\ cn \xrightarrow{\tau}_I{}^{*}\xrightarrow{R}_I cn' \wedge cn'\,\mathcal{R}_1\,cn'_{\text{D}}$$

Appendix 2 details the proof. It consists of a classical case analysis on the applied rule, and relies on a few crucial lemmas concerning the equivalence relation. Those lemmas provide a more convenient characterisation of value equivalence (by following future references), and provide properties on the evaluation of expressions in equivalent terms.

**Note on the equivalence.** By construction, when there is a future in `DeF` this future can be updated in `IF` because update is transparent and can occur earlier than the explicit synchronisation whereas the synchronisation is automatically introduced just before using the value. A future in `IF` almost always correspond to a future in `DeF` but the converse is not true. The only case when a future in `IF` does not correspond to a future in `DeF` is between the application of the Get-Update rule and the use of the obtained value (triggering a wait-by-necessity in `IF`). Note that, if instead of branching bisimulation, weak bisimulation was sufficient, then we could simulate a Get-Update rule of `DeF` by an Update rule of `IF` and systematically have less futures in `IF` than in `DeF`, thus simplifying the definition of $\mathcal{R}_1$.

$$[\![\texttt{Fut}\ll B\gg]\!]_{\texttt{DeF}\to\texttt{IF}} \triangleq B \qquad \frac{\vdash v : \texttt{Fut}\ll\texttt{Act}\gg}{[\![x = \texttt{get}\ v]\!]_{\texttt{DeF}\to\texttt{IF}} \triangleq \texttt{if}\ (v == \mathit{nullAct})\ \{\, x = v \,\}\ \texttt{else}\ \{\, x = v \,\}}$$

$$\frac{\vdash v : \texttt{Fut}\ll\texttt{Int}\gg}{[\![x = \texttt{get}\ v]\!]_{\texttt{DeF}\to\texttt{IF}} \triangleq x = v + 0} \qquad \frac{\vdash v : \texttt{Fut}\ll\texttt{Bool}\gg}{[\![x = \texttt{get}\ v]\!]_{\texttt{DeF}\to\texttt{IF}} \triangleq x = v \wedge \mathit{True}}$$

Figure 10: Translation from DeF to IF (other terms are unchanged).

### 4.3 Encoding DeF into IF

This section defines an encoding from a DeF program into IF, maintaining the same semantics and creating the same objects and futures. The translation is shown in Figure 10.

**Typing.** Compared to DeF, in IF, the future type does not exist and thus should be removed in a translation.

**Synchronisation, statement translation.** There is no *get* operation, but the synchronisation happens automatically when needed. To maintain the same semantics we must ensure that the synchronisation occurs at the same place, for that we trigger a dummy computation using the future to simulate the get of DeF (see the last three rules of Figure 10). In case of an object, we suppose that in IF there is an == operation on objects that operates on non-future values. We also introduce a set of reserved variables *nullAct* for each object type Act, all these variables have value null. These variables could safely be replaced by the null value in practice but using reserved variables allows us to identify the operation of synchronisation on an object in the proof of bisimulation.

**Correctness of the translation.** Figure 11 shows a relation $\mathcal{R}_2$ that relates an IF configuration on the left with a DeF configuration on the right, where the IF configuration has been obtained by evaluating the translation of a DeF program as described above. Like in the preceding section $t_{\text{D}}$ denotes terms of DeF. The first three rules of the figure describe the comparison of activities and futures, ignoring additional variables introduced by the translation. The next two rules are similar to $\mathcal{R}_1$ and allows the equivalence to follow future references on both sides. The last three rules deal with the equivalence on statements and especially with the intermediate states reached when evaluating the translation of a get statement. The first one deals with direct translation. The last two rules deal with the intermediate states reached when evaluating a get statement. In both rules, there are four cases, three for the possible translation of get, ans one case about the assignment in IF of a non-future value, this case deals with the intermediate state reached when an object has been synchronised (after a comparison to *nullAct*) In the last rule, one premise checks that $w_{\text{D}}$ is not a future and ensures that $w_{\text{D}}$ is the result of a get operation; this is always true when the other premises are true and the evaluated program is the result of the translation but the hypothesis is used the proof of the branching bisimulation theorem below; it avoids proving a technical additional lemma related to these points in the program.

Concerning transitions that cannot be observed in the bisimulation relation, similarly to the other directions, future updates occur at different moments, and are thus considered non-observable. Additionally, reduction rules resulting of synchronisation operations introduced by the translation cannot be observed faithfully, this only concerns the evaluation of *if* statements

$$\frac{cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash a\,\mathcal{R}_2\,a_{\mathrm{D}} \qquad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash \mathit{NoTmp}(\ell)\,\mathcal{R}_2\,\ell_{\mathrm{D}} \qquad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash q\,\mathcal{R}_2\,q_{\mathrm{D}} \quad cn, cn_{\mathrm{D}}, \varnothing, \varnothing \vdash \overline{q'}\,\mathcal{R}_2\,\overline{q'_{\mathrm{D}}} \qquad cn, cn_{\mathrm{D}}, a+\ell, a_{\mathrm{D}}+\ell_{\mathrm{D}} \vdash s\,\mathcal{R}_2\,s_{\mathrm{D}} \qquad cn\,\mathcal{R}_2\,cn_{\mathrm{D}}}{\alpha(a, q:\{\ell|s\}, \overline{q'})\ cn \quad \mathcal{R}_2 \quad \alpha(a_{\mathrm{D}}, q_{\mathrm{D}}:\{\ell_{\mathrm{D}}|s_{\mathrm{D}}\}, \overline{q'_{\mathrm{D}}})\ cn_{\mathrm{D}}}$$

$$\frac{cn\,\mathcal{R}_2\,cn_{\mathrm{D}}}{f(\bot)\ cn \quad \mathcal{R}_2 \quad f(\bot)\ cn_{\mathrm{D}}} \qquad \frac{f(w)\ cn, f(w_{\mathrm{D}})\ cn_{\mathrm{D}}, \varnothing, \varnothing \vdash w\,\mathcal{R}_2\,w_{\mathrm{D}} \qquad cn\,\mathcal{R}_2\,cn_{\mathrm{D}}}{f(w)\ cn \quad \mathcal{R}_2 \quad f(w_{\mathrm{D}})\ cn_{\mathrm{D}}}$$

$$\frac{f(w) \in cn \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\,\mathcal{R}_2\,w_{\mathrm{D}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash f\ \mathcal{R}_2\ w_{\mathrm{D}}} \qquad \frac{f(w_{\mathrm{D}}) \in cn_{\mathrm{D}} \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\,\mathcal{R}_2\,w_{\mathrm{D}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash w\ \mathcal{R}_2\ f}$$

$$\frac{}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![s_{\mathrm{D}}]\!]_{\mathtt{DeF}\to\mathtt{IF}}\ \mathcal{R}_2\ s_{\mathrm{D}}}$$

$$\frac{\left(\begin{array}{c} s = (x = w{+}0) \vee s = (x = w \wedge \mathit{True}) \vee ((s = (x = w)) \wedge \nexists f.[\![w]\!]_\ell = f) \\ \vee s = (\mathtt{if}\ (w == \mathit{nullAct})\ \{\, x = w \,\}\ \mathtt{else}\ \{\, x = w \,\}) \end{array}\right) \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![w]\!]_\ell\,\mathcal{R}_2\,[\![w_{\mathrm{D}}]\!]_{\ell_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ ;\ [\![s_{\mathrm{D}}]\!]_{\mathtt{DeF}\to\mathtt{IF}}\ \mathcal{R}_2\ (x = \mathtt{get}\ w_{\mathrm{D}})\ ;\ s_{\mathrm{D}}}$$

$$\frac{\left(\begin{array}{c} s = (x = w + 0) \vee s = (x = w \wedge \mathit{True}) \vee ((s = (x = w)) \wedge \nexists f.[\![w]\!]_\ell = f) \\ \vee s = (\mathtt{if}\ (w == \mathit{nullAct})\ \{\, x = w \,\}\ \mathtt{else}\ \{\, x = w \,\}) \end{array}\right) \qquad \nexists f.[\![w_{\mathrm{D}}]\!]_{\ell_{\mathrm{D}}} = f \qquad cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash [\![w]\!]_\ell\,\mathcal{R}_2\,[\![w_{\mathrm{D}}]\!]_{\ell_{\mathrm{D}}}}{cn, cn_{\mathrm{D}}, \ell, \ell_{\mathrm{D}} \vdash s\ ;\ [\![s_{\mathrm{D}}]\!]_{\mathtt{DeF}\to\mathtt{IF}}\ \mathcal{R}_2\ (x = w_{\mathrm{D}})\ ;\ s_{\mathrm{D}}}$$

Figure 11: Equivalence between a configuration of a DeF program (on the right) and a configuration of its translation in IF(on the left). Term-by-term equivalence and syntactic equality are omitted. $\mathit{NoTmp}(\ell) = \{[x \mapsto \ell(x)| \vdash x : B \wedge B \neq \mathit{nullAct}\}$

where the condition is the comparison to a *nullAct* reserved variable.

**Theorem 2.** *[Branching bisimulation] The translation of an* DeF *program into* IF *behaves identically to the original one:* $\mathcal{R}_2$ *is a branching bisimulation between the operational semantics of the* DeF *program* $P$ *and the operational semantics of the* IF *program* $[\![P]\!]_{\mathtt{DeF}\to\mathtt{IF}}$. *The transitions* Update, Get-Update, Get-Resolved, *and* If-True *and* If-False *where the condition contain one nullAct variable. In other words: With* $R$ *ranging over observable rule names, and* $\tau$ *over non-observable rules, we have, if* $cn\,\mathcal{R}_2\,cn_{\mathrm{D}}$ *then:*

$$cn \xrightarrow{\tau}_I{}^{*}\ cn' \implies cn'\,\mathcal{R}_2\,cn_{\mathrm{D}} \qquad\qquad cn_{\mathrm{D}} \xrightarrow{\tau}{}^{*}\ cn'_{\mathrm{D}} \implies cn\,\mathcal{R}_2\,cn'_{\mathrm{D}}$$
$$cn \xrightarrow{R}_I{}^{*}\ cn' \implies \exists cn'_{\mathrm{D}}.\ cn_{\mathrm{D}} \xrightarrow{\tau}{}^{*}\xrightarrow{R} cn'_{\mathrm{D}} \wedge cn'\,\mathcal{R}_2\,cn'_{\mathrm{D}}$$
$$cn_{\mathrm{D}} \xrightarrow{R}{}^{*}\ cn'_{\mathrm{D}} \implies \exists cn'.\ cn \xrightarrow{\tau}_I{}^{*}\xrightarrow{R}_I cn' \wedge cn'\,\mathcal{R}_2\,cn'_{\mathrm{D}}$$

The proof of this theorem can be found in Appendix 3. It is done by a classical case analysis and relies on the fact that $\mathcal{R}_2$ verifies the same preliminary lemmas as $\mathcal{R}_1$. Like in the case of $\mathcal{R}_1$ the following of futures inside DeF configuration is only necessary to ensure branching bisimulation. It is only needed to ensure equivalence in well identified temporary configurations and could be avoided if weak bisimulation was sufficient.

### 4.4 Concluding Remarks

We proved that IF and DeF feature a similar behaviour. However, the two different future constructs are not controlled in the same way by the programmer. On one side, DeF requires the programmer to be more precise because some entities are tagged as futures and some not, on the other side, in IF any entity can be a future. This is visible in the translation from IF into DeF that consider all objects as potentially a future and adds a lot of get operations, but also in the other directions where synchronisation is enforced only at the specific points corresponding to a get operation. This increased precision would make the implementation of DeF futures more efficient, increase the accuracy of static analyses, and help the programmer identify potential deadlocks at the cost of additional instruction and exposure of the programmer to the notion of futures.

Concerning code reuse, both IF and DeF allow some form of code reuse in the sense that a piece of program can be written the same way independently of which variables contain a future or not. However, IF allows the programmer to reuse code written in a sequential setting without knowing that some variables might contain futures whereas, in DeF reusable code must take into account that some variable may contain futures (the programmer should perform the adequate *get*s). Thus more code is reusable in IF but the recycling code with additional unplanned synchronisation could easily lead to undesired synchronisations or deadlocks. In DeF the programmer has to plan the possible synchronisations on futures and is thus aware of the potential synchronisations when writing his/her code.

Finally, the translation from IF to DeF highlights the property that a single get operation is always sufficient to allow an operation accessing a future to succeed.

## 5 Comparison with Classical Explicit Futures

This section takes DeF as a basis and shows the minimal syntax and semantics changes to define a language with parametric typing of explicit futures called EF. Recall that this kind of explicit futures come with a *control-flow synchronisation* where a synchronisation statement waits for the resolution of a single future, and this synchronisation is released by the execution of a single *return* statement. In active object languages with explicit futures, the synchronisation on a future is resolved as soon as the corresponding request finishes, even if this request returned another future.

Futures of EF are similar to the ones of languages like Encore, Creol, ABS, Akka, or Java, even if the future creation and manipulation in these languages is generally richer than the simplified semantics proposed here. Indeed EF only has one primitive for accessing futures (get). We will additionally discuss the advantages of an await primitive, especially because it can simulate the behaviour of data-flow synchronised futures in some cases.

### 5.1 Semantics of Explicit Futures

The syntax of EF is the same as the one of DeF. Only the syntax of types is changed to allow several future type constructors to be nested. We denote Fut< > the parametric future type constructor. This allows typing a variable that holds a future that will itself be resolved into a future:

$$T \quad ::= \quad B \mid \texttt{Fut<}T\texttt{>} \qquad\qquad \text{Non-basic type}$$

$$\textsc{Get-Resolved}\quad \frac{[\![v]\!]_{a+\ell} = f}{\alpha(a, q : \{\ell \mid y = \texttt{get}\ v\ ;\ s\}, \overline{q'})\ f(w) \rightarrow_E \alpha(a, q : \{\ell \mid y = w\ ;\ s\}, \overline{q'})\ f(w)}$$

$$\textsc{(T-Get)}\quad \frac{\Gamma \vdash_E v : \texttt{Fut<}T\texttt{>}}{\Gamma \vdash_E \texttt{get}\ v : T} \qquad \textsc{(T-Invk)}\quad \frac{\Gamma \vdash_E v : \texttt{Act} \qquad \Gamma(\texttt{Act})(\texttt{m}) = \overline{T} \rightarrow T' \qquad \Gamma \vdash_E \overline{v} : \overline{T}}{\Gamma \vdash_E v.\texttt{m}(\overline{v}) : \texttt{Fut<}T'\texttt{>}}$$

Get-Update is removed T-Subtype is removed

Figure 12: Operational semantics and typing rules for EF(compared to DeF).

The semantics of EF is denoted $\rightarrow_E$. Its only difference compared to DeF is that a single rule is necessary for handling the get statement. Indeed, get unfolds one level of future indirection, and thus a future is resolved in one step, but in EF this future resolution can return another future. Contrarily to DeF one might have to do several consecutive get statements to access a single data, this depends on the type of the accessed value, for example if a term is of type Fut<Fut<Int>> then two consecutive get operations are needed to access the integer value. Figure 12 shows the small-step operational semantics and the typing rules for EF; rules identical to DeF are not repeated (note that two rules of DeF semantics and its type system strictly need to be removed). The type judgement for EF is denoted $\vdash_E$, the type system is simpler than the one of DeF. There is no T-Subtype rule for explicit futures because a value cannot be used where a future is expected in EF. T-Get is changed accordingly to the type syntax, a basic type is not necessarily obtained after the operation succeeds. The typing of the method invocation is simpler than in DeF because several Fut type constructors can be nested. It is thus sufficient to add one future type constructor upon each invocation.

## 5.2 Comparison with DeF

We study here the expressiveness of EF compared to DeF. Like in Section 4, we are interested in the expressiveness of the future construct only and thus we study whether it is possible to design translations between EF and DeF only relying on the addition of additional variables and statements. We show below that futures in DeF and EF are not equivalent and there are DeF programs that cannot be expressed in the same way in EF, and thus a faithful translation from DeF to EF would need to encode a future by introducing additional objects and methods. We also identify an EF program that cannot be simulated in DeF, except by instrumenting each future with additional information. We briefly explain how a more complex encoding could simulate data-flow synchronisation and encode terminal recursive methods in EF, but this is not in the scope of the paper because it relies on the introduction of additional constructs and threads and we do not formalise this translation or its properties.

**Encoding DeF futures into EF.** Somehow, the simplicity of the semantics and the type system of EF justifies the fact that the explicit future construct with parametric type is the most massively used. However the type system constrains the usage of futures comparatively to the data-flow oriented models (DeF and IF). This was already highlighted in the introduction of [16] where the authors proved that the existing deadlock analysis for ABS could not be reused to analyse implicit futures. The main reason for this impossibility is that, with implicit

futures, there is no static bound on the number of synchronisations performed at a single program point, and thus a translation would require to declare a type with an unbound number of enclosed `Fut<>` constructs.

The semantics and type system of `EF` allow us to highlight the fact that parametric future types limit code re-usability and do not allow tail-recursive parallel methods like `DeF` or `IF`. To illustrate this point we try to implement the factorial example of Figure 1.a in `EF`. The type system does not allow the method to return either an integer or a future: in `EF`, `fact(1,1)` should return a term of type `Int` but `fact(3,1)` should return a `Fut<Fut<Fut<Int>>>`. The simplest solution is to declare that the method returns an integer, and add a synchronisation line 8, before the `return` statement (to retrieve the integer value). This is shown in Figure 1.c. Unfortunately this program deadlocks because the `get` instruction that was added to ensure correct typing, and allow `fact` to return an `Int`, waits for the scheduling of another `fact` request in the same active object. This request cannot be executed because the active object is mono-threaded and blocked on the `get` statement. Additionally, there is no way to allow the `fact` method to potentially accept a future, and thus the programmer has to synchronise with the result of the first call before calling another `fact` method (line 14). An alternative solution is to create two `fact` methods with different signatures.

However, many languages with parametric types for futures allow cooperative multi-threading: there is a primitive called *await* in ABS that allows the current thread to be interrupted while a future is awaited. This breaks the property that in `EF`, a request runs to completion but allows to solve many deadlocks. Figure 1.d shows the same program as it would be implemented in `EF` augmented with an *await* primitive that releases the current thread and allow starting the execution of another method. This is a reasonable way to implement a tail-recursive factorial in ABS. This program does not deadlock, but contrarily to `DeF` and `IF` $n$ methods are interrupted in order to ensure the transmission of the result all the way back to the caller. This solution additionally requires the `fact` method to be scheduled again in order to transmit the result. Not only this semantics is a bit different, but also if another request is served but never finishes (e.g. a request enqueued during the execution of `fact(0)` that is scheduled before the *await* statement is released), this prevents the transmission of the result to the main method which is not the case in `EF`. Thus a systematic translation from `DeF` into `EF` cannot simply rely on *await*. It is possible to encode a similar awaiting task inside `EF` and without relying neither on the availability of the original active object nor on cooperative thread release. This can be performed by creating a new active object before each `return`, and return directly the future obtained by the call to a method `Unfold` as shown in Figure 1.e. Each active object of type `FutProxy` plays the same role as a future in `EF`. But then `fact` must return a `Fut<Fut<Int>>` that needs to be accessed by two successive `get`, line 26 and 28. `return r` must also be modified to return a future instead (lines 15,16). This example shows a possible way to return a future from a method in `EF` and a good sketch of a correct encoding from `DeF` to `EF` but it relies on additional active objects, and shows that `EF` futures are not sufficient to encode the semantics of `DeF` futures. Interestingly, the *future chaining* operation (`-->`) of Encore [8] can be used to encode the desired behaviour, in a much lighter and efficient way. Future chaining can be performed in attached mode (on the same thread as the active object) and have a similar behaviour to the use of `await` in Figure 1.d, or in detached mode because no race condition is possible and feature the same behaviour as the additional active object with the `Unfold` method (like in Figure 1.e).

Concerning code reuse and the possibility to provide a value where a future is expected, this could be encoded by adding intermediate objects that act as future proxies, but again

```
1  Act{
2   Fut<Int> foo(){  // in DeF: Fut«int» foo()
3    Fut<Int> z;     // in DeF: Fut«int» z;
4    z=this.loop();
5    return z
6   }
7   Int loop(){     // never terminates; in DeF: Fut«int» loop()
8    Fut<Int> x;    // in DeF: Fut«int» x;
9    x=this.loop();
10   get x;         // Deadlocks
11   return 0
12  }
13 }
14 //MAIN
15 { Act x; Int z; Fut<Fut<int>> y; // in DeF: Fut«int» y
16  x = new  Act();
17  y = x.foo();
18  z = get y      // Terminates in EF but not in DF
19 }
```

Figure 13: An EF program that has no equivalent in DeF (comments show the differences in DeF).

this requires the addition of new constructs to encode the semantics of DeF futures.

To summarise, adding additional objects that would be used instead of futures, it is possible to encode DeF futures in EF. However such an encoding would be complex, and very difficult to prove correct. In any case, we showed here that the two future constructs have a fundamentally different semantics and that translating DeF programs into EF would rely on other features of the languages (additional threads, objects, or primitives).

**Encoding EF futures into DeF.** An impossibility result also holds in the other direction: the data-flow nature of the synchronisation in DeF cannot simulate exactly the synchronisation that occurs in EF. Indeed, the synchronisation in EF is control-flow oriented: a `get` synchronises with the execution of a single `return` instruction whereas synchronisation in DeF is data-flow oriented: a `get` synchronises with the availability of some data, which can require the execution of an unknown number of return instructions. This was highlighted in [21] where the authors provide a systematic translation from ABS to multi-threaded active objects with implicit futures and prove that the translation is correct. The authors highlighted two weak simulations between the execution of an ABS program and the execution of its translation in multi-active objects with implicit futures. One of the limitations of the translation is the "*absence of futures of futures*" stating that the translation is not always correct if the value of a future is another future.

Figure 13 provides an example inspired from [21] that illustrates the impossibility to simulate faithfully explicit futures with data-flow synchronisation. It shows an EF program that terminates, but for which the similar DeF program (differences shown in comments) does not terminate. There is no way in DeF to check that the `foo` method terminated without checking that the `loop` method terminated. This is due to the fact that in EF it is possible to only check whether a single future has been resolved, i.e. a given method has finished whereas in DeF, a future access checks whether a real value (with a basic type) is accessible. In this case accessing the future results in a deadlock in DeF because the future referenced by $y$ is

in fact a reference to another future that is never resolved. More precisely, the execution of the program reaches a configurations (i) $fut(f, f')$ $fut(f', \bot)$ which can be distinguished from the configuration (ii) $fut(f, \bot)$ in `EF` whereas the two configurations are observationally identical in `DeF` (no future access primitive can distinguish the two configuration). The data-flow synchronisation of `DeF` is intrinsically different from the synchronisation of explicit futures. One can notice that this impossibility result is a bit artificial as it relies on an example that retrieves a result that cannot be used for a practical computation but it seems impossible to produce a non-artificial example here.

In [21], a solution to this limitation was suggested: "It is possible to have a wrapper for futures values: a future value that is an object containing a future." One can simulate the behaviour of explicit futures by introducing intermediate objects. Anyway, it is also not possible to encode all `EF` programs into `DeF` without adding intermediate structures.

# 6 Concluding Remarks

## 6.1 Comparing `DeFs` to the other languages

The counter-examples showing the intrinsic difference between `IF` and `DeF` also allow us to distinguish `DeFs` from the other languages. Concerning the return of futures and the encoding of recursive terminal functions, `DeFs` is similar to `DeF`, and thus cannot be translated into `EF`. Code reuse of `DeF` cannot be encoded into `DeFs`. If we consider the example of Figure 1, the same program in `DeFs` is similar to the `DeF` code (Figure 1.a), except that the function cannot accept a future and the same `get` operation as in Line 14 of Figure 1.c must be applied. This distinguishes `DeFs` from `DeF` and `IF`. Finally, the program of Figure 13 also allows us to distinguish `DeFs` from `EF`, because in `DeFs` the synchronisation is driven by the data-flow (the program does not terminate in `DeFs` neither).

## 6.2 Other synchronisation patterns

This section briefly reviews several primitives for manipulating futures that exist in languages with explicit futures and investigates whether these primitives would make sense and could be specified and implemented for explicit futures with data-flow synchronisation, i.e. added to the semantics of `DeF` and used in an implementation of `DeF`.

**Await.** Await is a primitive used in active object languages with cooperative multi-threading. It releases the current thread to allow another method to execute. Its semantics could easily be specified in `DeF` with the obvious difference that the *await* should succeed when the `get` operation on the future would succeed (when a basic value is available).

**Asynchronous future access.** Several languages like Akka and AmbientTalk feature asynchronous invocations on futures, which is triggered after the future value is available. In AmbientTalk for example a new request is added to the actor when the future is resolved. A similar constructs could be featured by data-flow explicit futures, like in AmbientTalk an asynchronous invocation on a future would be enqueued when the future is resolved with a proper object, i.e. in a data-flow oriented manner. *Future chaining* of Encore is a bit similar to asynchronous future access but is control-flow oriented because based on explicit futures. Encoding future-chaining in `DeF` would be similar to asynchronous invocation on futures, except that future-chaining operations can sometimes be performed in parallel with the single thread of the active object.

## 6.3 Conclusion

In this article we design a new future construct that is both a promising programming paradigm, and a convenient tool to compare the semantics of futures in existing programming languages. Our work show that the distinctive feature of the different future constructs is the way synchronisation is performed: either it is data-driven like with implicit futures and a synchronisation corresponds to a process expecting the availability of some data, or it is control-driven like with explicit futures and a synchronisation corresponds to a process waiting for the execution of a given instruction, like the return statement of a method.

Our future construct offers a valuable compromise between explicit and implicit futures. It forces the programmer to state where the synchronisation occurs, and thus makes him aware of synchronisation points and potential deadlocks. Indeed, in `DeF`, an active object can only be blocked if it is performing a `get` operation on an unresolved future. `DeF` also allows writing functions that return a future, and in particular recursive asynchronous function calls, like with implicit futures. We explained why writing such functions was impossible or very difficult with explicit futures. Finally, `DeF` allows better code reuse than explicit futures: a piece of code that is written to access a future can be executed with a non-future value. The code re-usability is however lower than with implicit futures where a piece of code written in a sequential setting can be invoked with a future. This is compensated by more safety because the programmer has to declare which code can accept a future, and is aware that (s)he must handle the potential synchronisations and deadlocks. From a typing perspective, in `DeF`, `Fut`$\ll T \gg$ types a term of type $T$ possibly needing the resolution of one or several futures (but always a single `get` statement) to be accessed.

This article did not discuss type inference, but intuitively, the translation from `IF` to `DeF` shows a naive and ineffective but safe inference of future types. A classical static analysis can then propagate the non-future types that are obtained after a get or after an assignment that is not an asynchronous call. Inferring future types in method parameters would also be possible provided the whole program is known.

Even if no implementation of the future construct presented in this paper exists yet, such an implementation could rely on several existing solutions. On one side, we have shown that the implementation of implicit futures features the same behaviour as `DeF` futures, thus one solution to implement `DeF` would be to provide a type system in a language for implicit futures and implement the future semantics based on the translation presented in Section 4.3. ASP is the language that would fit the best this approach but its implementation, ProActive, is a Java library and it is difficult to implement a dedicated type system in this context. The distributed Java backend for ABS [21] can be considered as a better first step for the implementation of `DeF`. Indeed, this backend implements a data-driven synchronisation, and thus an implementation of `DeF` could rely on an adaptation of the ABS type system and minor modifications of the backend. Even if this is not very efficient, a first simple implementation of `DeF` would consist in encoding `DeF` futures as a datatype in ABS, as shown in Appendix **??**.

# Bibliography


[1] Futures and promises. URL: `https://en.wikipedia.org/wiki/Futures_and_promises`. 2

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986. 2

[3] Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. Formal Properties of the Eiffel// Model. In *Object-Oriented Parallel and Distributed Computing*. Hermes Science Pub., 2000. 5

[4] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Programming, Composing, Deploying for the Grid*, pages 205–229. Springer London, London, 2006. 5

[5] Laurent Baduel, Françoise Baude, and Denis Caromel. Object-oriented spmd. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 824–831, Washington, DC, USA, 2005. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=1169223.1169589`. 7

[6] Henry. G. Baker Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages*, pages 55–59. New York, NY, USA, 1977. 2, 4

[7] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017. URL: `http://doi.acm.org/10.1145/3122848`, `doi:10.1145/3122848`. 2, 11

[8] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104, pages 1–56. 2015. 5, 25

[9] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2004. 12, 15

[10] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004. 5, 7

[11] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421, pages 316–330. Springer, 2007. 5

[12] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of 20th European Conference on Object-oriented Programming (ECOOP)*. Springer, 2006. 5

[13] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. ParT: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch-Lafuente and José Proença, editors, *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*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016. URL: `https://doi.org/10.1007/978-3-319-39519-7_7`, `doi:10.1007/978-3-319-39519-7_7`. 6

[14] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. pages 209–220, 1995. 4

[15] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999. 4

[16] Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *PPDP 2016 18th International Symposium on Principles and Practice of Declarative Programming* , Edinburgh, United Kingdom, September 2016. URL: `https://hal.inria.fr/hal-01345315`. 10, 24

[17] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006. 5

[18] Phillip Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009. 5

[19] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985. `doi:http://doi.acm.org/10.1145/4472.4478`. 4

[20] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. First class futures: Specification and implementation of update strategies. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannataro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Selected Papers Coregrid Workshop On Grids, Clouds and P2P Computing*, volume 6586, pages 295–303. August 2010. 5

[21] Ludovic Henrio and Justine Rochas. Multiactive objects and their applications. *Logical Methods in Computer Science*, Volume 13, Issue 4, November 2017. URL: `http://lmcs.episciences.org/4079`, `doi:10.23638/LMCS-13(4:12)2017`. 8, 10, 26, 27, 28

[22] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957, pages 142–164. 2011. 5, 12

[23] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. 6(1):35–58, March 2007. 5

[24] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In Dag Langmyhr, editor, *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, November 2003. 5

[25] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. 2005. 5

[26] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd Conference on Mathematical Foundations of Programming Semantics*, ENTCS, New Orleans, April 2007. 4

[27] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006. 4

[28] Jan Schafer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010. 5

[29] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. Modular verification of a component-based actor language. *Journal of Universal Computer Science*, 11(10):1695–1717, 2005. 11

[30] Kenjiro Taura, Satoshi Matsuoka, and Akinori 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. 4

[31] Derek Wyatt. *Akka Concurrency*. Artima, 2013. 2, 5

[32] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *The Journal of Logic and Algebraic Programming*, 78(7):491 – 518, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007). URL: `http://www.sciencedirect.com/science/article/pii/S1567832609000022`, `doi:https://doi.org/10.1016/j.jlap.2009.01.001`. 4

# Appendices

These appendices will be published in a research report if the paper is accepted, and the report will be cited in the published paper.

## 1 Creating a Cycle of Futures in DeF

The following DeF program creates a cycle of futures, i.e. a runtime configuration containing $f_1(f_1)$. This justifies the fact that information about the type of future entries must be recorded at runtime in the configuration in order to type it (and to prove subject reduction for example). Indeed, at runtime, without static information on the future type it is not possible to infer the type of a future in a cycle. Fut«» is the future type constructor.

```
1  Act{ Fut«Int» field;
2    Int start(Act y) {
3          field = y.f1(this);
4          return 0
5    }
6    Fut«Int» f2() {
7      return field
8    }
9    Int f1(Act x){
10         Fut«Int» i;
11         i = x.f2();
12         return i
13   }
14 //MAIN
15 { Act x,y; Fut«Int» i;
16   x = new Act();
17   y = new Act();
18   i = x.start(y)
19 }
```

## 2 Proof of Theorem 1

**Lemma 1** ($\mathcal{R}_1$ Symmetry)**.** *Consider $v$ a term appearing in activities $\alpha(a, q : \{\ell'|s\}, \overline{q})$ and $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell'_{\text{D}}|s_{\text{D}}\}, \overline{q_{\text{D}}})$, suppose $\ell = a + \ell'$ and $\ell_{\text{D}} = a_{\text{D}} + \ell'_{\text{D}}$. We have: $cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash v \,\mathcal{R}_1\, v$.*

The proof is a simple case study on the potential values of $v$ and the applicable rules.

**Lemma 2** (Equivalent runtime values)**.** *Consider $w$ and $w_{\text{D}}$ two runtime values of activities $\alpha(a, q : \{\ell'|s\}, \overline{q'})$ and $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell'_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}})$, suppose $\ell = a + \ell'$ and $\ell_{\text{D}} = a_{\text{D}} + \ell'_{\text{D}}$. We have*

$$cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w_{\text{D}} \iff \begin{array}{l} w = w_{\text{D}} \\ \vee \exists w'.\, w(w') \in cn \wedge cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w' \,\mathcal{R}_1\, w_{\text{D}} \quad (w \text{ is a future}) \\ \vee \exists w'_{\text{D}}.\, w_{\text{D}}(w'_{\text{D}}) \in cn_{\text{D}} \wedge cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w'_{\text{D}} \quad (w_{\text{D}} \text{ is a future}) \end{array}$$

*The lemma also holds if $w$ and $w_{\text{D}}$ are the values of future $f$ in both configurations and $\ell = \ell_{\text{D}} = \varnothing$.*

In practice, we have that $(f(w) \in cn_{\text{D}} \wedge \nexists f'.\, w = f') \implies f(w) \in cn$ because future updates happen at any time in `IF` while they are delayed to the synchronisation point in `DeF`. Consequently at the point of `return` a future reference might be updated in `IF` but not in `DeF`, but if a future is updated in `DeF` it means that a synchronisation occurred and the update must also have been done in `IF`.

*Proof.* The proof is a simple case analysis on the applicable rule in the definition of $\mathcal{R}_1$ : except for structural equivalence, the only applicable rule is the unfolding of future values. □

**Corollary 1** (Equivalent runtime values can be identified by following chains of futures)**.** *Consider $w$ and $w_{\text{D}}$ two runtime values of activities $\alpha(a, q : \{\ell'|s\}, \overline{q'})$ and $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell'_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}})$, let $\ell = a + \ell'$ and $\ell_{\text{D}} = a_{\text{D}} + \ell'_{\text{D}}$. We have (for some $n \geqslant 0$, $k \geqslant 0$)*

$$cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w_{\text{D}} \iff \exists w'.\, \begin{cases} w = w' \vee \exists f_0..f_n.\, f_0 = w \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w') \in cn \\ \wedge \\ w_{\text{D}} = w' \vee \exists f'_0..f'_k.\, f'_0 = w_{\text{D}} \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w') \in cn_{\text{D}} \end{cases}$$

*The equivalence also holds if $w$ and $w_{\text{D}}$ are the values of future $f$ in both configurations and $\ell = \ell_{\text{D}} = \varnothing$.*

*Proof.* The proof is done by structural induction on the derivation stating that $cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w_{\text{D}}$. If the length is 1, then $w = w_{\text{D}}$, else one of the cases of Lemma 2 is applicable, additionally to the recurrence hypothesis, thus if $cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w_{\text{D}}$, then (the case $w = w_{\text{D}}$ has been eliminated):

$$\exists w'.\, w(w') \in cn \wedge cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w' \,\mathcal{R}_1\, w_{\text{D}} \vee \exists w'_{\text{D}}.\, w_{\text{D}}(w'_{\text{D}}) \in cn_{\text{D}} \wedge cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w'_{\text{D}}$$

Consider the first case, by recurrence hypothesis we have:

$$\begin{array}{l} cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w' \,\mathcal{R}_1\, w_{\text{D}} \\ \implies \exists w''.\, \left\{ \begin{array}{l} w' = w'' \vee \exists f_0..f_n.\, f_0 = w' \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w'') \in cn \\ \wedge \\ w_{\text{D}} = w'' \vee \exists f'_0..f'_k.\, f'_0 = w_{\text{D}} \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w'') \in cn_{\text{D}} \end{array} \right. \end{array}$$

Which ensures the desired property with a longer chain on the first line (the equality case is not possible here):

$$\exists w''.\, \left\{ \begin{array}{l} \exists f''_0 = w, f''_1 = w' = f_0, .., f''_{n+1} = f_n.\, f''_0 = w \wedge \forall i < n+1.\, f''_i(f''_{i+1}) \in cn \wedge f''_{n+1}(w'') \in cn \\ \wedge \\ w_{\text{D}} = w'' \vee \exists f'_0..f'_k.\, f'_0 = w_{\text{D}} \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w'') \in cn_{\text{D}} \end{array} \right.$$

The other cases and the converse implication are similar. □

**Corollary 2** (Equivalent runtime values can be identified by chains of futures pointing to equivalent values)**.** *Consider $w$ and $w_{\text{D}}$ two runtime values of activities $\alpha(a, q : \{\ell'|s\}, \overline{q'})$ and $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell'_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}})$, let $\ell = a + \ell'$ and $\ell_{\text{D}} = a_{\text{D}} + \ell'_{\text{D}}$. We have*

$$cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w \,\mathcal{R}_1\, w_{\text{D}} \iff \exists w', w'_{\text{D}}.\, \left\{ \begin{array}{l} cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash w' \,\mathcal{R}_1\, w'_{\text{D}} \\ \wedge \\ w = w' \vee \exists f_0..f_n.\, f_0 = w \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w') \in cn_{\text{D}} \\ \wedge \\ w'_{\text{D}} = w' \vee \exists f'_0..f'_k.\, f'_0 = w_{\text{D}} \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w'_{\text{D}}) \in cn_{\text{D}} \end{array} \right.$$

*The equivalence also holds if $w$ and $w_{\text{D}}$ are the values of future $f$ in both configurations and $\ell = \ell = \varnothing$.*

This is proven by two applications of the previous corollary.

The following lemma relates expression evaluation with the equivalence.

**Lemma 3** ( $\mathcal{R}_1$ and evaluation)**.** *Suppose* $\alpha(a, q : \{\ell|s\}, \overline{q})\ cn\ \mathcal{R}_1\ \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q_{\text{D}}})\ cn_{\text{D}}$. *Then for any expression* $e$ *of* `IF`*:*

$$\alpha(a, q : \{\ell|s\}, \overline{q})\ cn, \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q_{\text{D}}})\ cn_{\text{D}}, \ell, \ell_{\text{D}}\ \vdash\ [\![e]\!]_{a+\ell}\quad \mathcal{R}_1\quad [\![e]\!]_{a_{\text{D}}+\ell_{\text{D}}}$$

This lemma is proven by a case analysis on the rules for the evaluation of expressions. All cases are solved trivially, or by application of Lemma 1.

*Proof of Theorem 1.* The proof is a classical case analysis on the rule applied on each side. We focus ion the proof below on the most significant cases, i.e. all the non-observable rules, and the simulation of invocation in both directions. Proofs of other cases are similar.

UPDATE **(non-observable) rule of** `IF`**.** Suppose $cn \overset{\text{UPDATE}}{\rightarrow_I} cn'$ and $cn\,\mathcal{R}_1\,cn_{\text{D}}$ suppose the update occurs in $\alpha(a, q : \{\ell|s\}, \overline{q'}) \in cn$ and that $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$ (the activity $\alpha$ exists in $cn_{\text{D}}$ and has this form by definition of $\mathcal{R}_1$ ). Let $\ell' = a + \ell$ and $\ell'_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$. We have on the `IF` side:

$$\frac{\textsc{Update}\quad [\![x]\!]_{a+\ell} = f \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, q : \{\ell \mid s\}, \overline{q'}))\ f(w) \overset{\text{UPDATE}}{\rightarrow_I} \alpha(a', q : \{\ell' \mid s\}, \overline{q'})\ f(w)}$$

By Lemma 3 we have $[\![x]\!]_{a_{\text{D}}+\ell_{\text{D}}} = w_{\text{D}}$ and $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash f\,\mathcal{R}_1\,w_{\text{D}}$ for some $w_{\text{D}}$. By corollary 1, we obtain:

$$\exists w'.\ \begin{cases} f = w' \vee \exists f_0..f_n.\ f_0 = f \wedge \forall i < n.\ f_i(f_{i+1}) \in cn \wedge f_n(w')cn \\ \wedge \\ w_{\text{D}} = w' \vee \exists f'_0..f'_k.\ f'_0 = w_{\text{D}} \wedge \forall i < k.\ f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w') \in cn_{\text{D}} \end{cases} \tag{1}$$

Note that $f_1 = w$. By definition of $\mathcal{R}_1$ , $f(w) \in cn$ implies $f(w'_{\text{D}}) \in cn$ with $cn, cn_{\text{D}}, \varnothing, \varnothing \vdash w\,\mathcal{R}_1\,w'_{\text{D}}$.

To prove that $cn'\,\mathcal{R}_1\,cn_{\text{D}}$, it is sufficient to consider only the activity $\alpha$ and more precisely the value associated to $x$ in the field or the local store. Suppose the update occurs in the fields (the store case is similar): $a' = a[x \mapsto w]$. Overall we have to prove that: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash w\,\mathcal{R}_1\,w_{\text{D}}$. We consider the 4 different cases in equation (1). (1) if $w' = f$ and $w_{\text{D}} = f$, then the fifth rule of Figure 9 (following futures in `DeF`) allows us to conclude. (2) and (3) if $f(w)..f_n(w') \in cn$ and $w_{\text{D}} = f$ or $w_{\text{D}}(f'_1)..f'_k(w') \in cn_{\text{D}}$, then Corollary 1 allows us to conclude with the same $w'$ and a chain of length $n-1$ (or $w' = w$ if $n = 1$). (4) if $w' = f$ and $w_{\text{D}}(f'_1)..f'_k(w') \in cn_{\text{D}}$, we have $w_{\text{D}}(f'_1)..f'_k(w')\,w'(w'_{\text{D}}) \in cn_{\text{D}}$ with $cn, cn_{\text{D}}, \varnothing, \varnothing \vdash w\,\mathcal{R}_1\,w'_{\text{D}}$ and conclude by Corollary 2.

**Invk rule of** `IF`**.** Suppose $cn \overset{\text{Invk}}{\rightarrow_I} cn'$ and $cn\,\mathcal{R}_1\,cn_{\text{D}}$ suppose the invocation occurs from $\alpha(a, q : \{\ell|s\}, \overline{q'}) \in cn$, we have $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$. Let $\ell' = a + \ell$ and $\ell'_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$. We have the following reduction from $cn$:

$$\frac{\textsc{Invk}\quad [\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\alpha(a, q : \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}) \rightarrow \alpha(a, q : \{\ell \mid x = f\,;s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}\#(f, m, \overline{w}))\ f(\bot)}$$

By definition of $\mathcal{R}_1$ we have $s\,\mathcal{R}_1\,s_{\text{D}}$ and thus three cases are possible, corresponding to the three last cases of Figure 9 (the first rule relating statements is also applicable but it is a particular case of the first case below):

(1) $s_{\mathrm{D}} = (u_T = \texttt{get}\ w_{\mathrm{D}}\,;\, x = u_T.\mathtt{m}(\overline{v})\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}})$ with $cn, cn_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash [\![v]\!]_\ell \,\mathcal{R}_1\, [\![w_{\mathrm{D}}]\!]_{\ell'_{\mathrm{D}}}$.
Because $[\![v]\!]_{a+\ell} = \beta$, by Lemma 1 two cases are possible ($\beta$ is not a future): (a) $w_{\mathrm{D}} = \beta$, or (b) there exist $f'_0..f'_k$ s.t. $w = f'_0$, $\forall i < k.\, f'_i(f'_{i+1}) \in cn_{\mathrm{D}}$, and $f'_k(\beta) \in cn_{\mathrm{D}}$
We then have $cn_{\mathrm{D}} \xrightarrow{\tau}{}^{*} cn'_{\mathrm{D}}$ applying 0 times in case (a), or $k+1$ times in case (b), the Get-Update rule. We finally have $cn'_{\mathrm{D}}$ identical to $cn_{\mathrm{D}}$ except the current statement in $\alpha$ is $u_T = \texttt{get}\ \beta\,;\, x = u_T.\mathtt{m}(\overline{v})\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}}$. Then we have $cn'_{\mathrm{D}} \xrightarrow{\tau} cn''_{\mathrm{D}}$ applying once the Get-Resolved rule. The current statement in $\alpha$ is $u_T = \beta\,;\, x = u_T.\mathtt{m}(\overline{v})\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}}$. Then $cn''_{\mathrm{D}} \xrightarrow{\tau} cn^3_{\mathrm{D}}$ by applying once the Assign rule on the local intermediate variable $u_T$. The current statement in $\alpha$ is now $x = u_T.\mathtt{m}(\overline{v})\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}}$ and the local store has changed in $\alpha$, we call it $\ell^3_{\mathrm{D}}$ such that $[\![u_T]\!]_{a_{\mathrm{D}}+\ell^3_{\mathrm{D}}} = \beta$. Finally we can apply the invocation rule in DeF (note that $f$ is necessarily fresh in $cn^3_{\mathrm{D}}$):

Invk
$$\frac{[\![u_T]\!]_{a_{\mathrm{D}}+\ell^3_{\mathrm{D}}} = \beta \qquad [\![\overline{v}]\!]_{a_{\mathrm{D}}+\ell^3_{\mathrm{D}}} = \overline{w_{\mathrm{D}}} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\begin{array}{l}\alpha(a_{\mathrm{D}}, q_{\mathrm{D}} : \{\ell^3_{\mathrm{D}} \mid x = u_T.\mathtt{m}(\overline{v})\ ;\ [\![s]\!]_{\texttt{IF}\to\texttt{DeF}}\}, \overline{q'_{\mathrm{D}}})\ \beta(a'_{\mathrm{D}}, p_{\mathrm{D}}, \overline{q''_{\mathrm{D}}}) \\ \to \alpha(a_{\mathrm{D}}, q_{\mathrm{D}} : \{\ell^3_{\mathrm{D}} \mid x = f\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}}\}, \overline{q'_{\mathrm{D}}})\ \beta(a'_{\mathrm{D}}, p_{\mathrm{D}}, \overline{q''_{\mathrm{D}}\#(f, m, \overline{w_{\mathrm{D}}})})\ f(\bot) = cn^4_{\mathrm{D}}\end{array}}$$

The equivalence $\mathcal{R}_1$ of the obtained configuration is easy to assert: futures are equivalent because the new one is undefined in both cases, activity $\alpha$ is equivalent by structural equivalence (plus the hypothesis that original configurations are equivalent). Finally, we only have to prove $cn', cn^4_{\mathrm{D}}, \varnothing, \varnothing \vdash (f, m, \overline{w})\,\mathcal{R}_1\,(f, m, \overline{w_{\mathrm{D}}})$ inside $\beta$. This is done by lemma 3 that states that: $cn, cn_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash [\![\overline{v}]\!]_{\ell'}\,\mathcal{R}_1\,[\![\overline{v}]\!]_{\ell'_{\mathrm{D}}}$ and thus: $cn, cn^3_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash [\![\overline{v}]\!]_{\ell'}\,\mathcal{R}_1\,[\![\overline{v}]\!]_{a_{\mathrm{D}}+\ell^3_{\mathrm{D}_{\mathrm{D}}}}$ because $\overline{v}$ does not use $u_T$ and is not affected by the modification of the local environment. Finally, $cn', cn^4_{\mathrm{D}}, \varnothing, \varnothing \vdash \overline{w}\,\mathcal{R}_1\,\overline{w_{\mathrm{D}}}$: the stores can be discarded because the result of an evaluation function $[\![]\!]$ contains no variable, and the changes in the configuration implied by the invocation have no influence on this equivalence (variables, futures, and active object references inside $\overline{v}$ are by nature not affected by the invocation).
This allows us to conclude that $cn'\,\mathcal{R}_1\,cn^4_{\mathrm{D}}$.

(2) $s_{\mathrm{D}} = (u_T = w_{\mathrm{D}}\,;x = u_T.\mathtt{m}(\overline{v}))\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}})$ with $cn, cn_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash [\![v]\!]_\ell\,\mathcal{R}_1\,[\![w_{\mathrm{D}}]\!]_{\ell'_{\mathrm{D}}}$. Because of typing, as $\vdash u_T : T$ where $T$ is a base type (not future), we know that $\vdash w_{\mathrm{D}} : T$ (in the right context), and thus $w_{\mathrm{D}}$ is not a future. Additionally, we have $cn, cn_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash \beta\,\mathcal{R}_1\,[\![w_{\mathrm{D}}]\!]_{\ell'_{\mathrm{D}}}$, and thus $[\![w_{\mathrm{D}}]\!]_{\ell'_{\mathrm{D}}} = \beta$, and $w_{\mathrm{D}} = \beta$. We conclude by following the last steps (Assign and Invk) similarly to case (1).

(3) $s_{\mathrm{D}} = (x = u_T.\mathtt{m}(\overline{v}))\,;[\![s]\!]_{\texttt{IF}\to\texttt{DeF}})$ with $cn, cn_{\mathrm{D}}, \ell', \ell'_{\mathrm{D}} \vdash [\![v]\!]_\ell\,\mathcal{R}_1\,[\![u_T]\!]_{\ell'_{\mathrm{D}}}$. Again, by typing we have $[\![u_T]\!]_{\ell'_{\mathrm{D}}}$ is not a future, and by the properties of $\mathcal{R}_1$: $[\![u_T]\!]_{\ell'_{\mathrm{D}}} = \beta$. We conclude by applying the Invk rule similarly to (1).

Get-Update **(non-observable) rule of** DeF**.** Suppose $cn_{\mathrm{D}} \xrightarrow{\text{Get-Update}} cn'_{\mathrm{D}}$, $cn\,\mathcal{R}_1\,cn_{\mathrm{D}}$, the update occurs in $\alpha(a_{\mathrm{D}}, q_{\mathrm{D}} : \{\ell_{\mathrm{D}} | s_{\mathrm{D}}\}, \overline{q'_{\mathrm{D}}}) \in cn_{\mathrm{D}}$, and (by definition of $\mathcal{R}_1$) $\alpha(a, q : \{\ell | s\}, \overline{q'}) \in cn$. Let $\ell' = a + \ell$ and $\ell'_{\mathrm{D}} = a_{\mathrm{D}} + \ell_{\mathrm{D}}$. We have the following reduction inside $cn_{\mathrm{D}}$:

Get-Update
$$\frac{[\![w_{\mathrm{D}}]\!]_{a_{\mathrm{D}}+\ell_{\mathrm{D}}} = f}{\alpha(a_{\mathrm{D}}, q_{\mathrm{D}} : \{\ell_{\mathrm{D}} \mid u_T = \texttt{get}\ w_{\mathrm{D}}\ ;\ s'_{\mathrm{D}}\}, \overline{q'_{\mathrm{D}}})\ f(w'_{\mathrm{D}}) \to \alpha(a_{\mathrm{D}}, q_{\mathrm{D}} : \{\ell_{\mathrm{D}} \mid u_T = \texttt{get}\ w'_{\mathrm{D}}\ ;\ s'_{\mathrm{D}}\}, \overline{q'_{\mathrm{D}}})\ f(w'_{\mathrm{D}})}$$

By definition of $\mathcal{R}_1$ (cases on statements) we have two equivalence rules applicable (the 6th and the 7th), both entail that: $\big([\![s]\!]_{\texttt{IF}\to\texttt{DeF}} = \begin{array}{l} u_T = \texttt{get}\ v\,; \\ s'_{\mathrm{D}}\end{array} \vee [\![s]\!]_{\texttt{IF}\to\texttt{DeF}} = \begin{array}{l} y_{T'} = \texttt{get}\ v'\,; \\ u_T = \texttt{get}\ v\,; \\ s'_{\mathrm{D}}\end{array}\big)$

where $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'} \mathcal{R}_1 [\![w_{\text{D}}]\!]_{\ell'_{\text{D}}}$.

Consequently we have: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'} \mathcal{R}_1 f$ and by Corollary 1

$$\exists w'. \begin{cases} [\![v]\!]_{\ell'} = w' \vee \exists f_0..f_n.\, f_0 = [\![v]\!]_{\ell'} \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w') \in cn \\ \wedge \\ f = w' \vee \exists f'_0..f'_k.\, f'_0 = f \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w') \in cn_{\text{D}} \end{cases} \tag{2}$$

We need to prove $cn \mathcal{R}_1 cn'_{\text{D}}$. By definition of the equivalence on statements, it is sufficient to prove: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'} \mathcal{R}_1 [\![w'_{\text{D}}]\!]_{\ell'_{\text{D}}}$.

We reason on the two cases for the second line of (2):

- Either $f = w'$. As $f(w_{\text{D}}) \in cn_{\text{D}}$ and $cn \mathcal{R}_1 cn_{\text{D}}$, there is a $w$ s.t. $f(w) \in cn$ and $cn, cn_{\text{D}}, \varnothing, \varnothing \vdash w \mathcal{R}_1 w'_{\text{D}}$. For each of the two cases of the first line of (2) we find a longer chain of futures that allows us to apply Corollary 2: if $[\![v]\!]_{\ell'} = w' = f$ then there exists $f$ such that $[\![v]\!]_{\ell'} = f$ and $f(w) \in cn$. Else $\exists f_0..f_n.\, f_0 = [\![v]\!]_{\ell'} \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w') \in cn$, and then $\exists f_0..f_n, f_{n+1}.\, f_0 = [\![v]\!]_{\ell'} \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_{n+1}(w) \in cn$ (the futures are the same except we add $w = f$ to the chain). Finally we have:

$$\exists w, w_{\text{D}}. \begin{cases} cn, cn_{\text{D}}, \varnothing, \varnothing \vdash w \mathcal{R}_1 w_{\text{D}} \\ \wedge \\ \exists f_0..f_n.\, f_0 = [\![v]\!]_{\ell'} \wedge \forall i < n.\, f_i(f_{i+1}) \in cn \wedge f_n(w) \in cn \\ \wedge \\ w_{\text{D}} = w_{\text{D}} \end{cases}$$

  By Corollary 2 $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'} \mathcal{R}_1 [\![w'_{\text{D}}]\!]_{\ell'_{\text{D}}}$.
- Or $\exists f'_0..f'_k.\, f'_0 = f \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w') \in cn_{\text{D}}$. In this case, by removing the first future we have: $\exists f'_1..f'_k.\, f'_1 = w \wedge \forall i < k.\, f'_i(f'_{i+1}) \in cn_{\text{D}} \wedge f'_k(w') \in cn_{\text{D}}$, and by Corollary 1 with the same $w'$ as in (2) we obtain: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'} \mathcal{R}_1 [\![w'_{\text{D}}]\!]_{\ell'_{\text{D}}}$.

This concludes about the equivalence $cn \mathcal{R}_1 cn'_{\text{D}}$.

Get-Resolved **(non-observable) rule of** `DeF`**.** Suppose $cn_{\text{D}} \overset{\text{Get-Resolved}}{\rightarrow} cn'_{\text{D}}$ and $cn \mathcal{R}_1 cn_{\text{D}}$. We have the reduction:

$$\text{Get-Resolved}\ \frac{[\![w_{\text{D}}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = w'_{\text{D}} \qquad \nexists f. w'_{\text{D}} = f}{\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid y = \texttt{get}\ w_{\text{D}}\ ;\ s\}, \overline{q'_{\text{D}}}) \rightarrow \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid y = w'_{\text{D}}\ ;\ s\}, \overline{q'_{\text{D}}})}$$

The statement is of the form $s_{\text{D}} = (u_T = \texttt{get}\ w_{\text{D}}\,; s')$, thus only the $6^{\text{th}}$ or $7^{\text{th}}$ rule of the definition of $\mathcal{R}_1$ (Figure 9) can be used to assert equivalence on the original configuration. Thus the $8^{\text{th}}$ rule, for statements of the form $s_{\text{D}} = (u_T = w_{\text{D}}; s')$, can be used to conclude that the destination configuration verifies the same equivalence as the original one: $cn \mathcal{R}_1 cn'_{\text{D}}$.

Assign **of local intermediate variable (non-observable) rule of** `DeF`**.** $cn_{\text{D}} \overset{\text{Assign}}{\rightarrow} cn'_{\text{D}}$. The argument on statements is similar to the case above. Local intermediate variables are only introduced when the translation inserts a `get`, and the value of these variables is assigned only if the $8^{\text{th}}$ rule of Figure 9 is applicable. After reduction we can apply the last rule of Figure 9 where the value of intermediate variable is checked. The premise $cn, cn', \ell, \ell' \vdash [\![v]\!]_{\ell} \mathcal{R}_1 [\![u_T]\!]_{\ell'}$ (last rule of $\mathcal{R}_1$) to be ensured inside $cn'_{\text{D}}$ is obtained from the premise: $cn, cn', \ell, \ell' \vdash [\![v]\!]_{\ell} \mathcal{R}_1 [\![w_{\text{D}}]\!]_{\ell'}$ of the $8^{\text{th}}$ rule, that is verified on $cn_{\text{D}}$.

Invk **rule of** `DeF`**.** Suppose $cn_{\text{D}} \overset{\text{Invk}}{\rightarrow} cn'_{\text{D}}$ and $cn \mathcal{R}_1 cn_{\text{D}}$ suppose the invocation originates from $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$. Note that the invocation is necessarily performed on an

intermediate variable due to the translation rules. Let $\ell' = a + \ell$ and $\ell'_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$.

$$\textsc{Invk}\ \frac{[\![u_T]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \beta \qquad [\![\overline{v}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \overline{w_{\text{D}}} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\begin{array}{l}\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = u_T.\mathtt{m}(\overline{v})\ ;\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}}) \\ \rightarrow \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = f\,;s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}} \# (f, m, \overline{w_{\text{D}}}))\ f(\bot)\end{array}}$$

The key argument is to notice that the only way to have the equivalence of statements is to apply the last rule of Figure 9. Indeed, `get` statements are introduced for all method invocations of the original program and thus other equivalence cases are non-applicable. This way, we have in $cn$ two equivalent activities $\alpha$ and $\beta$: $\alpha(a, q : \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q'})\ \beta(a', p, \overline{q''})$ with $s'_{\text{D}} = [\![s]\!]_{\texttt{IF}\rightarrow\texttt{DeF}}$ and $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'}\, \mathcal{R}_1\, [\![u_T]\!]_{\ell'}$ and $[\![u_T]\!]_{\ell'} = \beta$. According to Corollary 1, either $[\![v]\!]_{\ell} = \beta$ or there is a chain of futures such that $[\![v]\!]_{\ell} = f_0$, and $f_0(f_1)\ ..\ f_n(\beta) \in cn$. Consequently, $cn$ can be reduced, after $n$ application of the UPDATE non-observable rule into $cn'$, identical to $cn$ except that $[\![v]\!]_{\ell''} = \beta$ with a new $\ell'' = a_2 + \ell_2$ where some local variables and fields have been updated. Finally $cn' \overset{\textsc{Invk}}{\rightarrow_I} cn''$:

$$\textsc{Invk}\ \frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a_2+\ell_2} = \overline{w} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\alpha(a_2, q : \{\ell_2 \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q'})\ \beta(a', p, \overline{q''}) \rightarrow_I \alpha(a, q : \{\ell \mid x = f\,;s\}, \overline{q'})\ \beta(a', p, \overline{q''} \# (f, m, \overline{w}))\ f(\bot)}$$

Only the equivalence of the enqueued requests still needs to be asserted; this is proven similarly to the reverse case: the proof that the invocation in `IF` can be simulated by the invocation in `DeF`. Indeed Lemma 3 states that: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![\overline{v}]\!]_{\ell'}\, \mathcal{R}_1\, [\![\overline{v}]\!]_{\ell'_{\text{D}}}$ and thus: $cn', cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![\overline{v}]\!]_{\ell'}\, \mathcal{R}_1\, [\![\overline{v}]\!]_{a_{\text{D}}+\ell^3_{\text{D}_{\text{D}}}}$ because $cn'$ only differs from $cn$ by the update of futures and the equivalence relation is not sensitive to the update of futures (e.g. because of Lemma 2)). Finally, $cn'', cn'_{\text{D}}, \varnothing, \varnothing \vdash \overline{w}\, \mathcal{R}_1\, \overline{w_{\text{D}}}$: the stores can be discarded because the result of an evaluation function $[\![]\!]$ contains no variable, and the changes in the configuration implied by the invocation have no influence on this equivalence.

The other reduction cases involve similar or simpler arguments. We thus show that there is a branching bisimulation between an `IF` program and its translation in `DeF`.

This proof also shows that futures are updated earlier in `IF` than in the `DeF` translation because future updates occur automatically in `IF`. The only exception is between the `get` instruction introduced by the translation and the real use of the value (one or two statements later). If we were interested in weak bisimulation, there would be no need to follow futures in the definition of equivalence in `IF` because the equivalent `IF` configuration could always catch up by updating futures to assert equivalence (future update can occur at any time in `IF`). However to obtain branching bisimulation we need to follow futures on both sides. On the other side, following futures in `DeF` is not avoidable since future values are only retrieved upon a `get` statement in `DeF` and thus it is not possible for a `DeF` execution to catch up on future updates at any time. □

# 3 Proof of Theorem 2

First note that all the Lemmas and corollary that were introduced for $\mathcal{R}_1$ are also valid when replacing $\mathcal{R}_1$ by $\mathcal{R}_2$, this is due to the fact they only rely on the rules defining equivalence on activities and futures, and the unfolding of future references, not ; these rules are similar for $\mathcal{R}_2$ and for $\mathcal{R}_1$ contrarily to the equivalence on statements.

*Theorem.* The proof is a case analysis on the rule applied to reduce either the `IF` or the equivalent `DeF` configuration. We focus below on all the non-observable transitions, the method invocation, and one assignment rule. The other cases are similar and less informative.

UPDATE **rule of `IF`.** This case is proven identically as for Theorem 1.

INVK **rule of `IF`.** Suppose $cn \overset{\text{INVK}}{\rightarrow_I} cn'$ and $cn\,\mathcal{R}_2\,cn_{\text{D}}$ suppose the invocation occurs from $\alpha(a, q : \{\ell|s\}, \overline{q'}) \in cn$, we have $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$. Let $\ell' = a + \ell$ and ${\ell'}_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$. We have the following reduction from *cn*:

$$\textsc{Invk}\ \frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\alpha(a, q : \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}) \rightarrow \alpha(a, q : \{\ell \mid x = f\,;s\}, \overline{q'})\ \beta(a', p, \overline{q_\beta}\#(f, m, \overline{w}))\ f(\bot)}$$

By definition of $\mathcal{R}_2$ we have $s\,\mathcal{R}_2\,s_{\text{D}}$ and thus necessarily $s_{\text{D}} = (x = v.m(\overline{v}))\,; s'_{\text{D}}$ where $[\![s'_{\text{D}}]\!]_{\texttt{DeF}\rightarrow\texttt{IF}} = s$. We can thus apply the rule INVK of `DeF`:

$$\textsc{Invk}\ \frac{[\![v]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \beta \qquad [\![\overline{v}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \overline{w_{\text{D}}} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\begin{array}{c}\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = v.\mathtt{m}(\overline{v})\ ;\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}}) \\ \rightarrow \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = f\,;s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}}\#(f, m, \overline{w_{\text{D}}}))\ f(\bot)\end{array}}$$

Note that typing ensures that $[\![v]\!]_{a_{\text{D}}+\ell_{\text{D}}}$ is not a future and thus $[\![v]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \beta$. Finally, the equivalence of the enqueued request is proven similarly to the same case in the proof of Theorem 1.

ASSIGN **rule of `IF`.**[6] Suppose $cn \overset{\text{ASSIGN}}{\rightarrow_I} cn'$ and $cn\,\mathcal{R}_2\,cn_{\text{D}}$. Additionally, suppose the assignment occurs in $\alpha(a, q : \{\ell|s\}, \overline{q'}) \in cn$, we have $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}}|s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$. Let $\ell' = a + \ell$ and ${\ell'}_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$. We have the following reduction from *cn*:

$$\textsc{Assign}\ \frac{[\![e]\!]_{a+\ell} = w \qquad (a + \ell)[x \mapsto w] = a' + \ell''}{\alpha(a, q : \{\ell \mid x = e\ ;\ s'\}, \overline{q'}) \rightarrow \alpha(a', q : \{\ell'' \mid s'\}, \overline{q'})}$$

Two cases are possible: the assign is the translation of the identical statement in `DeF`, in this two first cases the proof is trivial; or the assignment contains a `get` statement in `DeF`; we focus on this case. One of the two last rules of the definition of $\mathcal{R}_2$ (Figure 11) ensures the equivalence of statements.

(1) The `get` has already been resolved (last rule), we have:

$$\begin{array}{l}((e = w' + 0) \vee (e = w' \wedge \mathit{True}) \vee ((s = (x = w)) \wedge \nexists f.[\![w]\!]_\ell = f)) \\ \wedge s_{\text{D}} = (x = w_{\text{D}})\ ;\ s'_{\text{D}} \wedge cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![w']\!]_{\ell'}\,\mathcal{R}_2\,[\![w_{\text{D}}]\!]_{\ell'_{\text{D}}} \wedge \nexists f.w_{\text{D}} = f\end{array}$$

with $[\![s'_{\text{D}}]\!]_{\texttt{DeF}\rightarrow\texttt{IF}} = s'$. We detail the case where $w$ is an integer: $e = w' + 0$, the boolean case and the object case (last case of first line) are similar. Because $w_{\text{D}}$ is not a future, we know that $[\![w_{\text{D}}]\!]_{\ell'_{\text{D}}}$ is an integer $n$. Also $[\![w']\!]_\ell = n$ because ASSIGN is applicable and thus $[\![w' + 0]\!]_\ell$ can be computed and $[\![w]\!]'_\ell$ is not a future. Consequently $[\![w' + 0]\!]'_\ell = n = w$ and the rule ASSIGN can be applied in `DeF`:

$$\textsc{Assign}\ \frac{[\![w_{\text{D}}]\!]_{a+\ell} = n \qquad (a_{\text{D}} + \ell_{\text{D}})[x \mapsto n] = a'_{\text{D}} + \ell''_{\text{D}}}{\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = w_{\text{D}}\ ;\ s'_{\text{D}}\}, \overline{q'_{\text{D}}}) \rightarrow \alpha(a'_{\text{D}}, q_{\text{D}} : \{\ell''_{\text{D}} \mid s'_{\text{D}}\}, \overline{q'_{\text{D}}})}$$

[6] This is probably the most important case of the proof as it ensures that the same futures are resolved in any two equivalent configurations, and that the synchronisation points are the same in a program and in its translation.

We obtain a configuration $cn'_{\text{D}}$ and the term-by-term equivalence can be checked in a straightforward manner to prove that $cn' \,\mathcal{R}_2\, cn'_{\text{D}}$.

(2) The get has not been resolved yet (penultimate rule):

$$\begin{array}{l}(e = w'+0 \vee e = w' \wedge \mathit{True} \vee ((s = (x = w)) \wedge \nexists f.[\![w]\!]_\ell = f)) \\ \wedge s_{\text{D}} = (x = \texttt{get}\ w_{\text{D}}\ \texttt{;}\ s'_{\text{D}}) \wedge cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash [\![w']\!]_{\ell'}\, \mathcal{R}_2\, [\![w_{\text{D}}]\!]_{\ell'_{\text{D}}}\end{array}$$

with $[\![s'_{\text{D}}]\!]_{\texttt{DeF}\to\texttt{IF}} = s'$. We also only detail the case where $w$ is an integer: $e = w' + 0$ (other two cases are similar). Because $[\![w' + 0]\!]_\ell$ can be computed we know that $w'$ is not a future and $[\![w' + 0]\!]_\ell = [\![w']\!]_\ell = n = w$. By Corollary 1, $[\![w']\!]_{\ell'}\, \mathcal{R}_2\, [\![w_{\text{D}}]\!]_{\ell'_{\text{D}}}$ implies that we have either (a) $[\![w_{\text{D}}]\!]_{\ell'_{\text{D}}} = n$ or (b) $[\![w_{\text{D}}]\!]_{\ell'_{\text{D}}} = f_0$ and $\exists f_1..f_k.\, f_0(f_1)..f_k(n) \in cn_{\text{D}}$.

- In the case (b), we first apply $k$ times the rule Get-Update and obtain a configuration $cn'_{\text{D}}$ such that $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | x = \texttt{get}\ w'_{\text{D}}\ \texttt{;}\ s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn'_{\text{D}}$ where $[\![w'_{\text{D}}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = n$. From the configuration $cn'_{\text{D}}$, we apply the DeF reduction rule Get-Resolved and obtain the configuration $cn''_{\text{D}}$:

$$\textsc{Get-Resolved}\quad \frac{[\![w'_{\text{D}}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = n \qquad \nexists f.n = f}{\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | x = \texttt{get}\ w'_{\text{D}}\ \texttt{;}\ s'_{\text{D}}\}, \overline{q'_{\text{D}}}) \to \alpha(a'_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | x = n\ \texttt{;}\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})}$$

  We can the prove that $cn\, \mathcal{R}_2\, cn''_{\text{D}}$. Indeed, the only difference is in the current statement, and the statement equivalence can be established with the last rule of Figure 11. Note that the premise of Get-Resolved ensuing that $n$ is not a future is directly used to show the equivalence. We can then apply the same proof as in case (1).
- In the case (a) the configuration $cn_{\text{D}}$ already verifies the same property as $cn'_{\text{D}}$ above, we can directly apply the reduction rule Get-Resolved of DeF. The last steps are identical.

**If-True/If-False rule of IF where the condition contain one *nullAct* variable (non-observable)..** Suppose $cn \overset{\text{If-True}}{\to} cn'$ and $cn\, \mathcal{R}_2\, cn_{\text{D}}$. In this case one of the two last rules of Figure 11 is applicable for proving equivalence and the part $s = (\texttt{if}\ (w == \mathit{nullAct})\ \{\, x = w \,\}\ \texttt{else}\ \{\, x = w \,\})$ of the premise is true. In both cases, after the reduction, we know that $\nexists f.[\![w]\!]_\ell = f$ which verifies another part of the same disjunction, where $(s' = (x = w)) \wedge (\nexists f.[\![w]\!]_\ell = f)$. Consequently the configuration $cn'$ reached after the reduction is also equivalent to $cn_{\text{D}}$: $cn' \,\mathcal{R}_2\, cn_{\text{D}}$.

**Get-Update (non-observable) rule of DeF.** Suppose $cn_{\text{D}} \overset{\text{Get-Update}}{\to} cn'_{\text{D}}$ and $cn\, \mathcal{R}_2\, cn_{\text{D}}$ suppose the update occurs in $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$ and that (by definition of $\mathcal{R}_2$) $\alpha(a, q : \{\ell | s\}, \overline{q'}) \in cn$. Let $\ell' = a + \ell$ and $\ell'_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$. We have the following reduction inside $cn_{\text{D}}$:

$$\textsc{Get-Update}\quad \frac{[\![w_{\text{D}}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = f}{\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid u_T = \texttt{get}\ w_{\text{D}}\ \texttt{;}\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ f(w'_{\text{D}}) \to \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid u_T = \texttt{get}\ w'_{\text{D}}\ \texttt{;}\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ f(w'_{\text{D}})}$$

By definition of $\mathcal{R}_2$ (cases on statements) we have two equivalence rules applicable, but both entail that $s = s_s\ \texttt{;}\ [\![s'_{\text{D}}]\!]_{\texttt{DeF}\to\texttt{IF}}$ with:

$$\begin{array}{l}\left(\begin{array}{l} s = (x = w+0) \vee s = (x = w \wedge \mathit{True}) \vee (s = (x = w)) \wedge (\nexists f.[\![w]\!]_\ell = f) \\ \vee s = (\texttt{if}\ (v == \mathit{nullAct})\ \{\, x = w \,\}\ \texttt{else}\ \{\, x = w \,\}) \end{array}\right) \\ \wedge \quad cn, cn_{\text{D}}, \ell, \ell_{\text{D}} \vdash [\![w]\!]_{\ell'}\, \mathcal{R}_2\, [\![w_{\text{D}}]\!]_{\ell'_{\text{D}}}\end{array}$$

Consequently we have: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![w]\!]_{\ell'}\ \mathcal{R}_2\ f$. We need to prove $cn\,\mathcal{R}_2\,cn'_{\text{D}}$. By definition of the equivalence on statements, it is sufficient to prove: $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![w]\!]_{\ell'}\ \mathcal{R}_2\ [\![w'_{\text{D}}]\!]_{\ell'_{\text{D}}}$ which is done similarly to the same case in the proof of Theorem 1.

GET-RESOLVED **rule of** `DeF`**.** Like in Theorem 1, it is sufficient to check that, the statement necessarily verifies the penultimate rule defining $\mathcal{R}_2$, and thus the resulting statement verifies the last rule of the definition of $\mathcal{R}_2$.

INVK **rule of** `DeF`**.** Suppose $cn_{\text{D}} \overset{\text{INVK}}{\rightarrow} cn'_{\text{D}}$ and $cn\,\mathcal{R}_2\,cn_{\text{D}}$ suppose the invocation originates from $\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} | s_{\text{D}}\}, \overline{q'_{\text{D}}}) \in cn_{\text{D}}$. Let $\ell' = a + \ell$ and $\ell'_{\text{D}} = a_{\text{D}} + \ell_{\text{D}}$.

$$\text{INVK}\ \frac{[\![v]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \beta \qquad [\![\overline{v}]\!]_{a_{\text{D}}+\ell_{\text{D}}} = \overline{w_{\text{D}}} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\begin{array}{c}\alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = v.\mathtt{m}(\overline{v})\ ;\ s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}}) \\ \rightarrow \alpha(a_{\text{D}}, q_{\text{D}} : \{\ell_{\text{D}} \mid x = f\,;s'_{\text{D}}\}, \overline{q'_{\text{D}}})\ \beta(a'_{\text{D}}, p_{\text{D}}, \overline{q''_{\text{D}}\#(f, m, \overline{w_{\text{D}}})})\ f(\bot)\end{array}}$$

By definition of $\mathcal{R}_2$, we have in $cn$ two equivalent activities $\alpha(a, q : \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ [\![s'_{\text{D}}]\!]_{\mathtt{DeF}\rightarrow\mathtt{IF}}\}, \overline{q'})$ and $\beta(a', p, \overline{q''})$. Additionally, Lemma 3 ensures that $cn, cn_{\text{D}}, \ell', \ell'_{\text{D}} \vdash [\![v]\!]_{\ell'}\ \mathcal{R}_1\ [\![v]\!]_{\ell'_{\text{D}}}$. Consequently by (Corollary 1) we have either (a) $[\![v]\!]_{\ell'} = \beta$, or (b) $[\![v]\!]_{\ell'} = f_0$ and $\exists f_1..f_k.\ f_0(f_1)..f_k(\beta) \in cn$

In case (a), we can apply the rule INVK of `IF`, with $cn \overset{\text{INVK}}{\rightarrow_I} cn'$:

$$\text{INVK}\ \frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a_2+\ell_2} = \overline{w} \qquad \beta \neq \alpha \qquad f \ \text{ fresh}}{\begin{array}{c}\alpha(a, q : \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ [\![s'_{\text{D}}]\!]_{\mathtt{DeF}\rightarrow\mathtt{IF}}\}, \overline{q'})\ \beta(a', p, \overline{q''}) \\ \rightarrow_I \alpha(a, q : \{\ell \mid x = f\,;[\![s'_{\text{D}}]\!]_{\mathtt{DeF}\rightarrow\mathtt{IF}}\}, \overline{q'})\ \beta(a', p, \overline{q''\#(f, m, \overline{w})})\ f(\bot)\end{array}}$$

The equivalence $cn'\,\mathcal{R}_2\,cn'_{\text{D}}$ can be asserted in the same way as in the preceding proof, the only non-trivial part being the equivalence of the enqueued request.

Consider now case (b)[7]. According to the syntax and by typing, $v$ is necessarily a variable or `null`, but $[\![v]\!]_{\ell'} = f_0$ and thus $v = x$ with $x$ the name of a field or a local variable. We can thus apply the rule UPDATE of `IF`, $cn \overset{\text{UPDATE}}{\rightarrow_I} cn'$ with:

$$\text{UPDATE}\ \frac{[\![x]\!]_{a+\ell} = f_0 \qquad (a + \ell)[x \mapsto f_1] = a' + \ell'}{\alpha(a, q : \{\ell \mid s\}, \overline{q'}))\ f_0(f_1) \rightarrow_I \alpha(a', q : \{\ell' \mid s\}, \overline{q'})\ f_0(f_1)}$$

We have $cn'\,\mathcal{R}_2\,cn_{\text{D}}$, this is indeed an application of the UPDATE rule and it was proven above that it preserves equivalence. The UPDATE rule can be applied recursively until $[\![x]\!]_{a^{(k)}+\ell^{(k)}} = \beta$, with $cn^{(k)}\,\mathcal{R}_2\,cn_{\text{D}}$. We are then in the same situation as in case (a) which is sufficient to conclude.

The cases for other reduction rules are simpler or similar to the ones presented above. This allows us to conclude about branching bisimulation as stated in Theorem 2. □

# 4 `DeF` futures as a datatype inside `EF`

This appendix shows briefly how to provide a datatype (tagged union) encoding of `Fut`≪≫. Note that the initial purpose of the paper is not to use datatypes to encode futures because

[7] The careful reader would notice that in the `IF` execution case (b) never occurs. Indeed, the translation ensures that the `IF` program performs the synchronisations at the same place as the `DeF` program and thus the futures contained in $v$ have already been synchronised. However, according to the equivalence relation, case (b) is possible, we thus detail it for the completeness of the proof.

they add additional structure and control. Consider a language with datatypes, and only consider integers for simplicity (there is a similar but more verbose solution with objects):

```
datatype FutStarInt = Value Int | Future FutStarInt
```

It is easy to define get:

```
getStar(f) = { match f with Value x => x |  Future f' => x=get f'; getStar(x) }
```

Then one needs to systematically create a `FutStar` instead of `Future` when one does an asynchronous invocation and also to create a `FutStar` when before doing a `return`. An encoding of `EF` into `DeF` is possible this way. This solution was omitted in the paper because it uses additional objects (both upon invocation and upon return), and pattern-matching to replace futures (and not simply futures and simple future operations).