An Evaluation of Interaction Paradigms for Active Objects

Farzane Karami, Olaf Owe, Toktam Ramezanifarkhani

Department of Informatics, University of Oslo, Norway

Abstract

Distributed systems are challenging to design properly and prove correctly due to their heterogeneous and distributed nature. These challenges depend on the programming paradigms used and their semantics. The *actor paradigm* has the advantage of offering a modular semantics, which is useful for compositional design and analysis. Shared variable concurrency and race conditions are avoided by means of asynchronous message passing. The object-oriented paradigm is popular due to its facilities for program structuring and reuse of code. These paradigms have been combined by means of concurrent objects where remote method calls are transmitted by message passing and where low-level synchronization primitives are avoided. Such kinds of objects may exhibit active behavior and are often called *active objects*. In this setting the concept of *futures* is central and is used by a number of languages. Futures offer a flexible way of communicating and sharing computation results. However, futures come with a cost, for instance with respect to the underlying implementation support, including garbage collection. In particular this raises a problem for IoT systems.

The purpose of this paper is to reconsider and discuss the future mechanism and compare this mechanism to other alternatives, evaluating factors such as expressiveness, efficiency, as well as syntactic and semantic complexity including ease of reasoning. We limit the discussion to the setting of imperative, active objects and explore the various mechanisms and their weaknesses and advantages. A surprising result (at least to the authors) is that the need of futures in this setting seems to be overrated.

Keywords: Active objects; asynchronous methods; interaction mechanisms; concurrency; distributed systems; futures; cooperative scheduling.

1. Introduction

Programming paradigms are essential in software development, especially for distributed systems since these affect large programming communities and

Email addresses: farzanka@ifi.uio.no (Farzane Karami), olaf@ifi.uio.no (Olaf Owe), toktamr@ifi.uio.no (Toktam Ramezanifarkhani) a large number of applications users. The *actor model* [18] has been adopted by a number of languages as a natural way of describing distributed systems. The advantages are that it offers high-level and yet efficient system designs, and that the operational semantics may be defined in a modular manner, something which is useful with respect to scalability. The actor model is based on concurrent autonomous units (actors) communicating by means of asynchronous message passing, and with a “sharing nothing” philosophy, meaning that no data structure is shared between actors. The actor model offers high level, yet efficient, constructs for synchronization and communication.

A criticism of the interaction mechanism of the actor model has been that its one-way communication paradigm may lead to complex programming when there are dependencies among the incoming messages. It is easy to make programming errors such that certain messages are never handled. And it is not straight forward to augment an actor model with support of additional messages and functionality. In the actor model one may not classify and organize the communication messages in request messages and reply messages, and it does not support object-oriented (OO) principles such as inheritance, late binding, and reuse (even though the original actor concept was inspired by the ideas behind object-orientation).

To overcome these limitations, one may combine the actor model and object-orientation, using the paradigm of concurrent, *active objects* and using methods rather than messages as the basic communication mechanism, thereby supporting imperative programming in a natural manner. The active object model has gained popularity and is an active research area [3]. A call of method m on a remote object o could have the form $x := o.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. This opens up for two-way communication where both the method call and the corresponding return value are transmitted by message passing between then caller and callee objects. The naive execution model is that the caller waits while the callee performs the call, and then stores the result in the program variable x. However, this can result in undesired blocking and possibly deadlock. Therefore non-blocking call mechanisms are needed.

One way of avoiding unnecessary waiting is provided by the *future mechanism*, originally proposed in [2] and exploited in *MultiLisp* [15], *ABCL* [34], and several other languages. A future is a read-only placeholder for a result that is desirable to share by several actors, where the placeholder may be referred to using the *identity* of the future. In particular, one may refer to a result even before it is produced, and a future identity may refer to a *future* method result. In languages with first-class futures, future identities can be passed around as first-class objects like references. Futures can give rise to efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. The notion of *promises* gives even more flexibility than futures by allowing the programmer to refer to a computation result before it is known how to generate it, and by which process. For instance, a promise may be used to refer to the result of one of several futures.

A future object with its own identity can be generated when a remote method call is made. Then the caller may go on with other computations until it needs the return value, while the callee executes. The callee executes the called method and sends the return value back to the future object upon termination of the method invocation, at which time the future is said to be *resolved* (i.e., the future value is available). When the caller needs the future value it may request the future value, and is blocked until the future is resolved. A programming language may have implicit or explicit support of futures. Consider first explicit futures: Typically a call statement defines the future identity, say $f := o!m(\overline{e})$, where f is a *future variable* used to hold the future identity of the call (with m, o, and $\overline{e}$ as above). Here the symbol "!" indicates the difference from a blocking call (assuming both are allowed). When the result of the call is needed, the caller uses a construct like **get** f where f is an expression giving the future identity, for instance in an assignment $x :=$ **get** f. By letting futures be first-class entities, the objects may communicate future identities and thereby allow several objects to share the same method result, given as a future. Any object that has a reference f to a future may perform **get** f. Implementation of call requests and future operations can be done by means of message passing.

Implicit futures are similar, except that the future variable is not available for the programmer, and the **get** operations are made implicitly as defined by the semantics. The call may now look like $x := o.m(\overline{e})$ (or say $x := o!m(\overline{e})$ to distinguish it from that of a synchronous call, if both are desired) where x is of the return value type, and the implicit **get** operations may happen when the value of x is needed (first time after the call). This is attractive in functional languages, avoiding the distinction between a function returning a future and one returning the future value, or receiving a future input versus a future value. However, implicit futures make static program analysis difficult since the waiting points are implicit, possibly depending on dynamic factors. In particular, certain kinds of textual analysis become infeasible.

In languages with futures, the two-way communication mechanism is replaced by a more complex pattern, namely that a method call generates a future object where the result value can be read by a number of objects, as long as they know the future identifier. A normal two-way call can be done by letting the caller ask and wait for the future. This means that each call has a future identity, and that the programmer needs to keep track of which future corresponds to which call. This gives an additional layer of indirectness in programming. Our experience is that the full functionality of futures is only needed once in a while, and that basic two-way communication suffices in most cases. Thus the flexibility of futures (and promises) comes at a cost. Implementation-wise, garbage collection of futures is non-trivial, and static analysis of various aspects including deadlock detection in presence of futures is more difficult. Even if one introduces a short-hand notation for the simple two-way call interaction, there is still a future behind the scene, and thus all calls are typically handled uniformly by this more expensive implementation mechanism.

Another drawback of the basic future mechanism is that once a **get** operation is done, the current object is blocked as long as the future is not yet resolved. To overcome this, one may allow *polling*, i.e., testing if a future is resolved or not, without blocking, for instance used in an if-test where the branches deal with the two cases. But polling may result in complex program structures since it opens up for explicit program control of the possible message ordering.

Another way of avoiding blocking is the notion of *cooperative scheduling* suggested in the *Creol* language [22], and the *OUN* language [7, 27], generalizing the concept of guards from the guarded command language of Dijkstra [9] by adding a notion of process suspension. Cooperative scheduling can be achieved by a language construct, **`await`** c, where c is a condition, either a boolean condition or a waiting condition, such as the presence of the result of a remote method call. If c is not satisfied, the current executing method invocation ("process") is placed on the process queue of the object, which allows another enabled process on the queue, or an incoming request, to continue. A process on the queue is not enabled if it starts with an await with a condition that is not satisfied. Thus a method invocation may passively wait in the queue while the object is active and able to take care of other (enabled) processes. Thus cooperative scheduling provides local synchronization control and provides a constructive approach to the scheduling of processes internally in an object. Cooperative scheduling may be combined with the future mechanism, for instance with first-class futures as in the ABS language, or with non-first-class futures as in the Creol language, where the futures are local to a process. We refer to such non-first-class futures as *local futures*; and in general, we may talk about object-local and method-local futures. Object-local futures may not be communicated to other objects, but assignment of futures to fields and local variables is acceptable as well as passing of futures through parameters or return values of local methods. Method-local futures are local to a method instance (process) and may not be assigned to fields and may not be passed as parameters or return values of method calls.

In this paper, we will focus on the interaction mechanisms in imperative, active object languages, especially the paradigm of asynchronous call/return without use of futures, versus the different versons of the future mechanism, as well as cooperative scheduling and polling. The contibution of the paper is a comparison on the different interaction mechanisms, based on a survey of representative languages, and a unified syntactic and semantic formalization of the various language combinations. We give a critical discussion on the pros and cons of the various combinations of these mechanisms. As most recent imperative languages for active objects support explicit, first-class futures, we leave out implicit futures from our discussion. To complement the discussion, we suggest some language improvements in the setting of asynchronous call/return without use of futures. We compare the various interaction paradigms wrt. the following criteria:

- *expressiveness*
- *efficiency*
- *syntactic and semantic complexity*
- simplicity of program reasoning and static analysis
- information *security* aspects.

The paper is organized as follows: Section 2 provides the context of the work, giving an overview of the interaction mechanisms of a number of active object languages, including ABCL, Rebeca, Creol, ABS, Encore, and ASP/ProActive. A complementary communication model and language is proposed in Subsection 2.7. In order to make a comparison easier, Section 3 defines a unified syntax and semantics for the different interaction paradigms. Then Section 4 evaluates the different interaction mechanisms along the comparison dimensions. Finally, conclusions are given in Section 5.

2. Background

In this section we review some representative languages based on *active objects*, and give a summary of their interaction models. We limit the discussion to imperative languages, since a majority of modern active object languages (with some exceptions like Scala) are imperative. For each language we identify support of active and/or passive behavior and interaction mechanisms, including synchronization mechanisms involving waiting and blocking, as well as cooperative scheduling. In particular, we explain support of explicit or implicit futures and polling mechanisms. We focus on explicit futures since the semantical issues of these are more clearly connected to syntactic constructs. This allows us to make a syntax-oriented, language-based comparison.

Furthermore, we look at shared futures as well as local futures. Local futures include *object-local futures*, not permitted to be communicated and shared with other objects, and *method-local futures*, not permitted to be stored in fields or passed to other method invocations than the one creating the future. Local as well as shared futures may in principle be read multiple times, but for method-local futures the value of multiple reads is questionable. It can be statically checked that a method-local future is read at most once, leading to a notion of single-use, method-local futures, which gives the simplest form of futures.

To illustrate and compare the interaction mechanisms in the different languages, we use a running example. It is part of a subscriber service that was originally made to take advantage of the benefits of first-class futures. The ABS solution in Figure 5 is most close to the original version, and should be read first. In this example, the server, defined by class *Service*, searches for news and publishes them to subscribing clients, using proxies. The server communicates news to the proxies by means of first-class futures, so that the server itself does not wait for incoming news and is free to respond to any client request (apart from doing synchronized database operations). Instead the proxies wait for the incoming news. The proxies are organized in a list (growing upon need), letting each proxy handle a limited number of clients.

2.1. ABCL

The integration of the actor model with object-oriented concepts was first introduced in ABCL [34]. In this language, concurrent objects interact via asynchronous message passing and futures. An object definition as depicted in

```
[object object-name
    (state representation of local memory ...)
        (script
           (=> message pattern1)
           (=> message pattern2))]
```

Figure 1: Object definition in ABCL.

Figure 1 includes: the object's name, its state declaring the local object variables (fields) and initialization, and its script including patterns of messages received by the object, and a set of corresponding actions. Each object has its own queue for storing the messages according to their arrival time. When an object receives a message matching one of the declared patterns, it performs the corresponding actions. ABCL uses first-class futures, which are explicitly created by the syntax *make-future*. Moreover, a future is a queue, and all receiving objects can write to it, but only the object which creates it, can access and check the future values, in contrast to other languages supporting first-class futures. In fact, most other languages implement a future object as once-writable and multiple-readable (by many objects).

Assuming an object o sends a message m to an object o', ABCL supports three types of message passing [34]:

1. *Past-time* message passing (send and no wait):
   After sending the message, the sender o immediately continues its process without waiting for a reply or delivery. If the reply should be sent to other objects, the (optional) *reply-destination* is the destination of those objects. The syntax for this kind of message passing is:

   $$o' <= m \ @\textit{reply-destination}$$

2. *Now-time* message passing (send and wait):
   Object o blocks while waiting for the result from o', then assigns the result to a program variable x. The notation for this type of message passing is:

   $$x := \ o' <== m$$

3. *Future-type* message passing (reply to me later):
   In this case, o does not need the result immediately, and instead of blocking it can continue and later on check whether the future object contains the result or not. In this case of message passing, the reply destination is the specified future object. The notation for this kind of message passing is:

   $$o' <= m \ \$f$$

   where the future variable f is bound to the future object.

In ABCL, an object that creates a future can check its values by the operation:

$$\texttt{ready?}\ f$$

If at least one reply is stored in the future f, the value of this form is t (i.e., true); otherwise, nil. Thus polling is supported. Moreover, the operation

`next-value` f *options*

returns the first element stored in the future f, and if the future is empty the owner object waits until a reply arrives. And with a `:remove t` option the future value is removed from the *future-object*. Whereas, with the `:remove nil` option, it still remains in the future queue even after evaluation of this form. The default option is `:remove t`.

Example. Figure 2 shows a subscriber example in the ABCL language. In ABCL, bracket forms are often used to build message patterns; and in a message pattern, a symbol starting with a colon (:) represents a tag, and other symbols are pattern variables. Executing an expression [*object*...] creates an object with a specified behavior defined in the expression. In this language, the symbol `Me` stands for the object which executes the operation in which the "`Me`" exists. Moreover, by using a reply form !*form*, the evaluation result of the form is sent back as a reply to the currently processed message.

In the subscriber example, the *publish* call in the state definition of *service* creates a cycle between service and proxy objects, since each such call leads to a *produce* and indeed another *publish* call in the object proxy. In the object service, since *publish* and *detectNews* calls are past-time, interleaving of other calls (such as *subscribe* and *unsubscribe* calls) is possible between each execution of *publish* or *produce*. In line 2, object service sends a [:publish] message to proxy. As a response, in line 15 and 19, object proxy creates a future and appends it to a produce message toward object service, respectively. Therefore, waiting points for detecting news are delegated to the proxy by using futures. Object proxy owns the *future*, and only this object has the access to values, while other objects can only write to the *future*. In line 6, when object service receives a *produce* message, it sends a past-time *detectNews* message to the object producer, searching for news, with a reply destination *Me*. And according to an exclamation mark ! in line 35, the reply from evaluation of *detectNews* is sent back to *Me*. In line 6, according to the exclamation mark ! the result from the evaluation is replied to the *future* variable. In line 21, object proxy first checks if the future is available, then by the command *next-value* retrieves the value and multi-casts it to clients. In line 24, if nextProxy is empty, the object proxy continues to search for new news; otherwise, it publishes current news to clients subscribed to the nextProxy.

2.2. Rebeca

Rebeca [31, 32] is an active object language that is more close to the actor-based model than the other languages considered here. In this language, active objects are called *rebecs* (<u>re</u>active o<u>bjec</u>t). Each rebec is instantiated from a reactive class and has its own thread of control. A reactive class consists of an interface, variables, method definitions (*message server*) for dealing with

```
[object service
 (state dataBase db; int limit; [proxy <= [:publish]]; )
      // proxy does the main job and initiates a produce call
 (script
 (=> [:produce proxy]
      ![producer <= [:detectNews] @ Me]; //reply destination is Me
      db <= [:logging];)
      ...
  (=> [:subscribe]...)
  (=> [:unsubscribe]...))
]

[object proxy
  (state list myClients:=nil; News ns; proxy nextProxy:=nil;
 future := (make-future);)
  (script
   ...
  (=> [:publish]
    [service <= [:produce Me] $ future];
     // replies from object service saved in the future
    if (ready? future){ // polling on the future
      [ns := (next-value future)];
      [myClient <= [:signal ns]];} // multi-cast the result
    if (= nextProxy nil)
      [Me <= [:publish]]
    else
      [nextProxy <= [:publish]];))
]

[object producer
  (state News ns;)
  (script
  (=> [:detectNews]
     ...
    ! ns:=...))
]

[object myClient
 (script
 (=> [:signal ns] ....))
]
```

Figure 2: A version of the subscriber example in the ABCL language.

messages and initial methods. An initial method of a rebec triggers declared messages toward other rebecs. The receiving objects react to these messages according to their method definitions. Communication in this model is one-way asynchronous message passing, without shared variables, blocking receive, nor futures. Since the communication is by asynchronous message passing, each rebec has its own message queue, with FIFO order. Rebeca actors are isolated, therefore their analysis and verification become feasible.

Sometimes it is necessary to have synchronous communication, thus in extended versions of Rebeca the *component* concept is defined. A component encapsulates rebecs that may have internal synchronous communications [30]. External communication beyond a component is either an asynchronous broadcast or an asynchronous message toward another rebec. RebecaSys [28] is another extended model of Rebeca supporting global variables and the $wait(e)$ statement. This statement temporarily stops the execution of the process. The Boolean expression e may only contain global variables that all rebecs have access to. Hence, the *wait* statement depends on the rebecs that update these variables.

Example. Figure 3 represents the subscriber example with the extended version of Rebeca, considering futures as global variables. The initial *produce* message in reactiveclass *Service* creates a cycle since each such message leads to a *publish* message, which in turn leads to another *produce* message. The *future* variable is a Boolean global variable that the prod rebec sets to true when it completes a *detectNews* call, in line 19. The service rebec as a server asynchronously broadcasts a *detectNews* message to anonymous receivers, which only one of the rebecs providing these messages reacts to, and also sends an asynchronous *publish* message to object proxy. In line 30, proxy rebec is blocked waiting for the *future* to become true. Then in line 31, it sends a *send* message, provided in prod rebec, to signal news to subscribed clients. It is possible to make a simpler version in Rebeca without global variables, but we here want to illustrate how futures can be simulated.

2.3. Creol

Creol was developed from the OUN language [7] based on the notion of active concurrent objects. Interaction is by means of asynchronous methods, implemented by message passing, and remote field access is not allowed. The synchronization mechanisms include suspension, allowing passive (non-blocking) waiting on a Boolean condition or on the arrival of a return value from another object [23, 22, 24, 20]. This allows non-blocking as well as blocking method calls.

The visible behavior of objects is specified through interfaces. Thus methods not exported through an interface may only be used for self calls. The behavior of objects can change dynamically between active and passive (reactive) by means of asynchronous self calls. Multiple inheritance is supported as well as dynamic code modification. Basic Creol supports method-local futures (so-called "call labels"), i.e., futures may neither be passed as parameters nor

```
globalvariables {boolean future;}
reactiveclass Service() {
  init() {Producer prod; Proxy nextProxy; self.produce(DataBase db); }
  // initial action, starting a produce cycle
  produce(DataBase db){
    detectNews();
    proxy.publish(self, prod, nextProxy); // no waiting
    logging(){...} } // logging in a database for services
  ...
  subscribe(Client me){...}
  unsubscribe(Client me){...}
}

reactiveclass Producer() {
  News ns;
  init() { future= false;} // initialization
  detectNews(){
    ns=...; // wait for more news
    future= true;
  }
  send(Client myClients){
    if (future)
      myClients.signal(ns); } // assuming multi-casting is ok in Rebeca
}

reactiveclass Proxy() {
  List[Client] myClients:=null;
  ...
  publish(Service s, Producer prod, Proxy nextProxy){
  wait(future); // wait for the future
  send(myClients);
  if nextProxy == null
    s.produce();
  else
    nextProxy.publish();}
}

reactiveclass Client(){
  News latestNews;
  signal(News ns){ latestNews= ns;}
}

main {Service s(Database db); Proxy proxy;}
```

Figure 3: A subscriber example in the Rebeca language.

assigned to fields. However, first-class futures are allowed in extensions of Creol. Methods are given with the keyword **op** and Creol methods may have several inputs as well as several outputs (indicated by the keyword **out**). Variables are declared by the syntax **var** *name* : $T = e$ where e is the initial value of type T. Creol has a small-step operational semantics defined by a set of rewrite rules in the Maude format [12], used for proving the soundness of analysis and verification [14, 10, 11], and also providing an executable interpreter.

Communication between Creol objects is two-way, passing actual parameters from the caller to the callee object when a method is called, and passing method return values from the callee to the caller when the method execution terminates.

The asynchronous method call command $t!o.m(\overline{e})$ where t is a call label ("tag"), sends a call request message to the callee o and the caller object proceeds without waiting. This call generates a unique call identity for referencing the call, assigned to t. Passive waiting for return values is possible by means of *cooperative scheduling.* Each active object has an *internal process queue* containing the processes that are suspended, either waiting for a return value or a Boolean condition. In addition there is an external queue for receiving method call requests from other objects.

A process is suspended when the suspension statement **await** c is executed in a state where the condition c is false. The executing process is moved to the *process queue* of the object, and the object is then free to do something else, like serving an incoming call request or continuing an enabled process in the process queue of the object. Similarly the statement **await** $t?$ suspends when the return value for the call with the identity of t has not arrived. Otherwise, the await statement is enabled, and execution continues with the next statement. In contrast, the command $t?(\overline{x})$ blocks while waiting for the return values, and assigning these to the variable list $\overline{x}$. A label t is local to the current process and cannot be passed to other processes, nor assigned or read by other kinds of statements. The sequence $t!o.m(\overline{e}); t?(\overline{x})$ (abbreviated $o.m(\overline{e}; \overline{x})$) corresponds to a synchronous method call, blocking the processor of the current object until the return values are available, whereas the sequence $t!o.m(\overline{e});$ **await** $t?(\overline{x})$ (abbreviated **await** $o.m(\overline{e}; \overline{x})$) corresponds to a non-blocking call, where **await** $t?(\overline{x})$ abbreviates **await** $t?;\ t?(\overline{x})$. The label t may be omitted in a ! call statement if that t is not needed in a ? statement. Multi-casting can be allowed by the syntax $!o.m(\overline{e})$ where o is a list of objects, in which case the replies cannot be received (since there is no associated label). Note that labels (of type *Label*) are not typed by the return value. Static type checking is possible by certain language restriction (avoiding that the same label is used for several return value types, at a given program point).

Example. The subscriber example, which originally makes use of first-class futures, must be redesigned in Creol, for instance as done in Figure 4. Here the passing of a future is replaced by suspension, which means that the Service object may continue with other tasks as in the version with first-class futures. The suspended process can be compared to an added proxy-like object, while the *Proxy* objects in the Creol solution are not blocked in contrast to the ABS

```
type News = ...

interface ServiceI{
 with ClientI
  op subscribe(out result:Bool)
  op unsubscribe(out result:Bool)
  with Any
  op produce()
}
interface ProxyI{
 with ServiceI, ProxyI
  op publish(ns:News)
  ...
}
interface ProducerI{
 with ServiceI op detectNews(out result:News)
  ...
}
interface ClientI{with Any op signal(ns:News) ...}
interface DataBase{with Any op logging(...) ...}

class Service(limit:Nat, prod:ProducerI, db:DataBase) implements ServiceI {
  var proxy:ProxyI = new Proxy(limit,this); //proxy does the main job
  {!this.produce() } // initial action, starting a produce cycle

  op produce(){var ns:News;
    var t: Label;
    t!prod.detectNews();
    db.logging(...) // logging in a database for services
    await t?(ns)// waiting while suspending
    !proxy.publish(ns) } // sends the value

  with ClientI
   op subscribe(out result:Bool) {...}
   op unsubscribe(out result:Bool) {...}
}

class Proxy(limit:Nat, s:ServiceI) implements ProxyI{
  var myClients:List[ClientI]=Nil; var nextProxy:ProxyI;
  ...
   op publish(ns:News){
    !myClients.signal(ns);  // multi-cast the result
    if nextProxy=null
    then !s.produce() else !nextProxy.publish(ns) fi}
}
```

Figure 4: A version of the subscriber example in the Creol language.

version with first-class futures. Note that Creol insists on typing of object variables by interfaces, and we therefore sketch all interfaces. An interface consists of a number of operations (and semantic specifications, ignored here), and each operation has a *co-interface*, restricting what kind of objects may appear as callers. For instance, *subscribe* has *ClientI* as co-interface, meaning that method *subscribe* may only be called by objects supporting *ClientI*. This implies that the implicit *caller* parameter is of interface *ClientI*, which allows us to ensure statically that *myClients* is a list of *ClientI*, say by passing *caller* to a method

```
op add(c : ClientI){
    if length(myClients) < limit
    then myClients := append(myClients, c)
    else if nextProxy = null then nextProxy := new Proxy(limit, s) fi ;
    !nextProxy.add(c) fi}
```

of class *Proxy*, by the asynchronous call !proxy.add(caller). Thus the co-interface *ClientI* is needed in the implementation of *subscribe* and *unsubscribe* in order to obtain a type-correct program, but it is not needed for the implementation of *subscribe* since *caller* is not used. By using the implicit *caller* parameter, one does not need the explicit caller parameter (ClientI me) used in the other solutions for *subscribe* and *unsubscribe*. For simplicity, we use the syntax {. . .} rather than **begin** . . . **end**.

2.4. ABS

Abstract Behavioral Specification language (ABS) [21] is an object-oriented language, inspired by Creol and JCoBox [29]. It is a concurrent programming language based on the cooperative scheduling from Creol and the notion of object groups from JCoBox, named Concurrent Object Groups (COG) [3]. Software product lines with Deltas are supported, but not class inheritance. In ABS, the unit of concurrency and distribution is the COG. Each COG includes a group of objects, a queue, and a processor. Objects in a COG share a common heap and processor, and there is no data sharing between COGs. At most one process (method activation) is active in a COG, while other processes are suspended in a process pool. In other words, parallel processes are executed by multiple threads in different COGs, but only one thread is active in a particular COG.

Objects in different COGs call each other asynchronously. Inside a COG, objects can call each other asynchronously or synchronously. The communication syntax is like Creol as well, using conditional await or await on a result/future. The statement **release** gives unconditional suspension. The **await** releases the thread if the specified condition does not hold, or if the future is not resolved (in case of await on a future), whereas the **get** f statement blocks the thread until the future f is resolved. Thus, the whole COG gets blocked. ABS futures are explicit, first-class, and typed by a parametric type **Fut** $[T]$, where T is the type of the future value.

```
data News = ...

interface ServiceI{
  Bool subscribe(ClientI me)
  Bool unsubscribe(ClientI me)
  Void produce()
}
interface ProxyI{
  Void publish(Fut[News] fut)
  ...
}
interface ProducerI{
 News detectNews()
  ...
}
interface ClientI{Void signal(ns:News) ...}
interface DataBase{Void logging(...) ...}

class Service(Int limit, ProducerI prod, DataBase db) implements ServiceI {
  ProxyI proxy:= new Proxy(limit,this); //proxy does the main job
  {this!produce() } // initial action, starting a produce cycle

  Void produce(){
    Fut[News] fut := prod!detectNews();
    proxy!publish(fut); // sends future, no waiting
    db.logging(. . .) } // logging in a database
  ...
  Bool subscribe(ClientI me){...}
  Bool unsubscribe(ClientI me){...}
}

class Proxy(Int limit,ServiceI s) implements ProxyI{
  List[ClientI] myClients:=nil; ProxyI nextProxy;
  ...
  Void publish(Fut[News] fut){
    News ns := get fut; // wait for the future
    myClients!signal(ns);  // multi-cast the result
    if nextProxy==null
    then s!produce() else nextProxy!publish(fut) fi}
}
```

Figure 5: A subscriber example in the ABS language.

Example. Figure 5 illustrates the subscriber example in ABS, making use of first-class futures, passing a future in the *publish* calls rather than the news value. In line 24, the asynchronous call creates a future identity, assigned to *fut*. Then *fut* is passed to a proxy object in line 25. In line 36, the proxy object is blocked until *fut* is resolved, after which the proxy continues to execute the next statement. For simplicity, we omit **return** *void* at the end of the body of a void method.

2.5. Encore

Encore [4] is a parallel programming language based on active objects with explicit first-class futures, inspired by Creol and ABS. It is designed for multicore platforms and is optimized for efficient execution. Encore supports both active object parallelism for coarse-grained parallelism, as in Creol, and parallelism within an object, using *parallel combinators* for building high-level coordination of active objects and low-level data parallelism. Encore offers high-level language constructs for coordination of parallel computations such as building pipelines of these computations. It offers parallel types, an abstraction of parallel collections, and also parallel combinators for operating on them. The parallel type **Par T** is a handle to a collection of parallel computations, and it can be thought of as a list of futures, which will eventually produce zero to multiple values of type **T**. Then operations on parallel types are called parallel combinators; accordingly, high-level typed coordination patterns, parallel dataflow pipelines, speculative evaluation and pruning, and low-level data parallel computations are supported by Encore [4].

Encore provides both passive and active classes (with the latter as default). Active objects have their own thread of control, or possibly multiple threads of control, and a FIFO message queue, and interact via asynchronous method calls. Passive objects do not have their own thread of control, like standard objects of Java. An object class is active by default or is declared as passive by a keyword **passive**. An asynchronous method call to an active object is stored inside the active object's queue. And the result of this asynchronous method call is a future. If the type of the return value is T, the returned future would be of type $Fut\ T$. Explicit synchronization constructs for accessing a future are **get**, **await**, and future chaining. Like Creol/ABS, **get** blocks an active object until the future is resolved, and **await** waits for resolving the future and blocks the current process, but not the current active object. Thus other methods of the active object can be invoked. In the chaining construct ($\sim\sim>$), a closure is attached to a future, and when resolved, the thread executes the closure, which might result in another future, containing the result of the executed closure. A closure is a set of computations, possibly including method calls.

Example. To illustrate these operations on futures, Figure 6 represents the subscriber example with the Encore language. In line 7, a future with identity *fut* is created as a result of an asynchronous call to the active object prod, and then passed to the object proxy in line 8. In line 20, object proxy blocks until *fut* is resolved.

```
passive class DataBase{...}
class Service(limit: int, prod: Producer, db: DataBase) {
 proxy:= new Proxy(limit,this); //proxy does main job
 {this.produce();} // initial action, starting a produce cycle
 ...
 def produce(): void{
   let fut := prod.detectNews();
   proxy.publish(fut); // sends future, no waiting
   db.logging(. . .) } // logging in a database for services
 def subscribe(me:Client): bool;
 def unsubscribe(me:Client): bool;
}

class Proxy(limit: int, s: Service){
 myClients: [Client];
 nextProxy: Proxy;
 ...
 def publish(fut: Fut News): void{
   ns : News;
   ns := get fut; // wait for the future
   myClients.signal(ns);  // multi−cast the result
   if nextProxy==null;
     s.produce();
   else nextProxy.publish(fut);}
}
```

Figure 6: A subscriber example in the Encore language.

2.6. ASP/ProActive

The goal of ASP [6] and ProActive [5] is to design a transparent concurrent programming language. ProActive is a Java library programming language [5], which implements ASP semantics in Java and inherits many properties from ASP. In ASP, an active object with its thread of control, its request queue, and its passive objects is called an *activity*. In addition, active objects are defined by a **newActive** command, and passive objects are standard Java objects. Only active objects are accessible between activities. Method calls to active objects are transparently turned into asynchronous calls, and those to passive objects are turned into synchronous local calls. Moreover, futures are created implicitly as a result of asynchronous method calls to an active object. In other words, an asynchronous method call is stored in the request queue of the callee, and the caller creates a future object with a unique identity, referencing this request. When a future gets resolved, a reference to the corresponding request gets updated by the value.

ASP supports first-class futures, and its synchronization mechanism for accessing a future is *wait-by-necessity.* This synchronization mechanism blocks the thread whenever it needs to access a future value, until it is resolved. Although futures in ASP are implicit, the ASP runtime system needs constructs for implementation, update, garbage collection and synchronization of futures. It supports an explicit synchronization primitive *waitfor* which triggers an explicit *wait-by-necessity* on a future. It also provides primitives to test whether a future is updated or not. The primitive is denoted by *awaited(a)*, which returns true if a is a future and false otherwise. An extension to multiactive objects has been made recently in [17].

Example. Figure 7 shows the subscriber example with the ProActive language. In ProActive, active objects are instantiated using the ProActive API:

$$B\ b = (B)\ ProActive.newActive(''B'', params, node);$$

It creates a new active object of type B, in which *params* specifies constructor parameters, and *node* specifies the location to put the active object. Another method to create an active object is by using **`turnActive(obj, node)`**, which makes an existing object (*obj*) active on a specified location (*node*). In fact, a thread is created and an associated pending request queue. In line 5, an active object `proxy` is defined by the **`newActive`** command, and in line 6 a passive object `prod` is transformed to an active one by the keyword **`turnActive`**. In addition, we assume that all object instantiations of class service is active as well. Correspondingly, all method calls toward these objects are implicitly transformed to asynchronous ones. Line 7 starts a *produce* cycle. In line 10, variable `v` is the result of an asynchronous *detectNews* call toward object `prod`, which is an implicit future. In line 11, this future is passed to object `proxy` without blocking. Then, in line 20, when the future value is needed to continue execution, an explicit wait-by-necessity synchronization **`waitfor`** is applied on the future `v`.

Implementation strategies. To represent flow of futures and different update strategies for implicit futures, Figure 8 is adopted from [6]. The gray arrows with numbers show the flow of futures between activities, and the black ones, indexed with letters, show the future references. In this example, activity γ initiates a remote method call to activity δ and future f_1 is associated to the result of this call. Future flow number 1 corresponds to the creation of future f_1 involving γ and δ. Then γ sends f_1 to β, for instance as the result of a request, flow number 2, and β forwards the f_1 reference to α as the result of another request, flow number 3a. In parallel, γ sends a request to $\alpha\prime$ with f_1 as a request parameter, flow number 3b. Finally, δ consumes the result associated with f_1, flow number 4.

In the case of first-class futures, a future value list F_α inside an activity stores future values calculated by itself or other activities. There are three strategies for updating a future value: 1) *no partial object forwarding*, 2) *eager strategy*: either *forward-based* or *message-based*, and 3) *lazy strategy.* The simplest strategy is that no partial object (the objects containing future references or values) can be forwarded between activities. In other words, futures are not first-class. This

```
class Service(Int limit, ProducerI prod, DataBase db) extends ServiceI
implements Active {
 // assuming active instantiation of object service
 Object [] params= {limit,this}
 Proxy proxy:= (Proxy) ProActive.newActive ("Proxy", params, Node);
 prod = (Producer) ProActive.turnActive (prod, Node);
 {this.produce(); }

 void produce(){
   News v := prod.detectNews(); // implicit asynchronous method call
   proxy.publish(v); // v can be passed without blocking
   db.logging(...) }
   ...
}

class Proxy(Int limit,ServiceI s) extends ProxyI implements Active{
 Client[] myClients:=null; ProxyI nextProxy;
 ...
 void publish(News v){
   News ns := waitfor(v); // explicit wait-by-necessity on v
   myClients.signal(ns);
   if nextProxy==null;
   then s.produce(); else nextProxy.publish(v);}
}
```

Figure 7: A subscriber example in the ProActive language.

strategy leads to fewer number of future references, simpler update process, and avoids maintaining a future value list inside an activity. However, it is too synchronized and may lead to waste of time and deadlock. For example, according to Figure 8, while γ is waiting for a response from δ, other activities are stuck (waste of time).

The second strategy is called the *eager strategy*, a future gets updated as soon as it is resolved, thus future value lists are avoided. In the case of *forward-based strategy*, when a future reference is sent to an activity, the sender is responsible for updating its value, but not the source activity; hence the source activity does not need to keep the future value any longer. Consequently, when there are too many intermediate nodes, this strategy increases the delay between when a future is resolved and when it gets updated. In Figure 8, when based on this strategy, the future f_1 is first updated in γ, then it is sends this value to β, α', and then β can forward this value to α. Consequently, there is a delay before updating the future value of α.

In the case of *message-based strategy*, when a future is forwarded between two activities, a message, created from the receiver or the sender activity, is

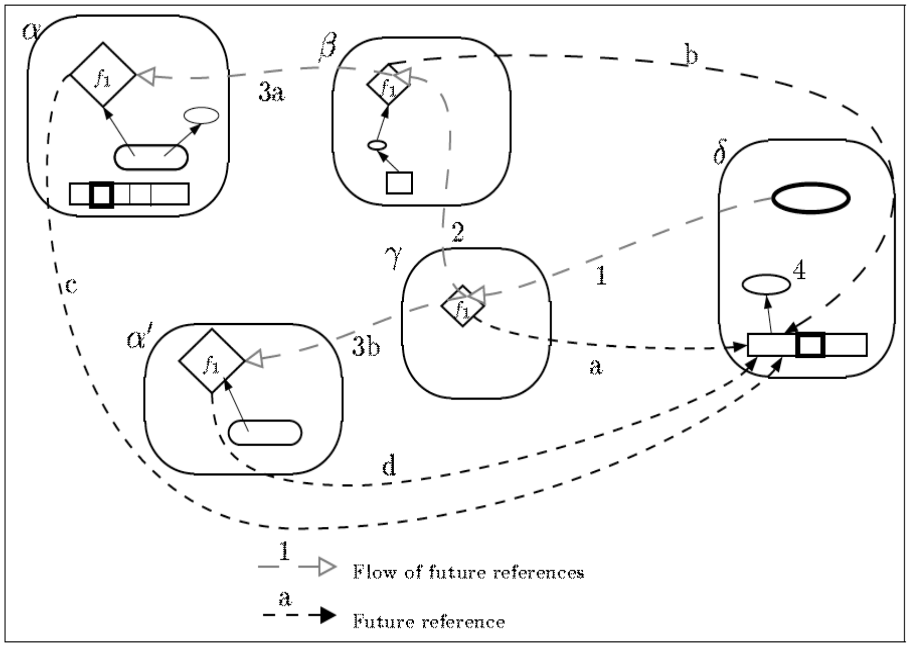

Figure 8: Future flow in ASP [6].

sent to the source activity. Hence the source activity gets informed about them, and when the future value is calculated, it can directly update it in all other activities. This message-based mechanism minimizes the delay of updating. For example, in Figure 8, when γ sends the future reference f_1 to α', either γ or α' sends a message to the source activity δ; correspondingly, when the future value is calculated in δ it can be immediately updated in α'.

The third strategy is the *lazy* future update. A future gets updated only when an activity requires it (wait-by-necessity). The activity directly asks for the future value by sending a message to the source activity. In the lazy strategy, a future value list is required to be kept in the source activity in order to store the future values and update them whenever there is a request for them.

The implementation of ProActive supports several update strategies, including no partial object forwarding and forward-based strategy [6], and Henrio et al. have implemented the four strategies in ProActive as a middleware to study the efficiency of different update strategies [6]. The lazy strategy is faster than the two eager ones, since less updates are required. This strategy is suitable for scenarios in which the number of processes requiring the future value are considerably less than the total number of processes. In this strategy, there is less load at the source process. However, it leads to additional delays and needs more resources to keep the future value list inside the source process for later updates. The eager forward-based strategy gives more delay since the intermediate nodes have to forward a future value to a target. As a result, this strategy is suitable for scenarios with small number of nodes. Eager message-based strategy necessitates more bandwidth and resources at the source process, since all other nodes communicate with it.

To complement the discussion we introduce some additional language features and a new language called *FutFree*, in the next section.

*2.7. A proposed future-free language with improved expressiveness (*FutFree*)*

As we have seen, implicit futures have the weakness that the implicit occurrences of the **get** operation cannot always be identified in a context-free manner. This makes modular reasoning and static analysis difficult. And a weakness of the future-free paradigm, as represented by the future-free subsets of the languages discussed above, is that there is no way of expressing that a **get** operation should be performed in a given state. For the purpose of this paper, we therefore propose a new language exploiting the future-free interaction paradigm while adding a new mechanism for non-blocking waiting. The language is an extension of the future-free subset of Creol/ABS and is called *FutFree*. We do this in order to complement the comparison of languages with and without futures.

FutFree is without call-labels, and without explicit/implicit futures, and consists of the asynchronous call statement $o!m(\overline{e})$, the high-level future-free call mechanisms of Creol, including the **await** construct, extended with a *"tail"* construct to better control return value points, and a *delegation* mechanism to enable simple sharing (to an object other than the caller). The tail construct may or may not be combined with **await**:

$$[\texttt{await}]\ x := o.m(\overline{e}) < s >$$

where the tail s is any statement list, being performed while waiting for the future of the call to m to be resolved, and therefore s may not use (the new value of) x. The syntax [**await**] denotes an optional **await**, allowing the statement to suspend after s if the future is not resolved at that point. Thus, the statement $x := o.m(\overline{e}) < s >$ will block after execution of s while waiting for the return value to appear (if it has not already appeared). And the statement **await** $x := o.m(\overline{e}) < s >$ will suspend after execution of s while waiting for the return value to appear. The former statement is equivalent to $f := o!m(\overline{e}); s; x := \texttt{get}\ f$, using ABS, and the latter is equivalent to $f := o!m(\overline{e}); s; \texttt{await}\ x := \texttt{get}\ f$.

Note that $< s >$ may be empty or include additional calls as in for instance

$$[\texttt{await}]\ x := o1.m1(\overline{e1}) < \textit{calculate}\ e2; [\texttt{await}]\ y := o2.m2(\overline{e2}) < s >; \textit{use}\ y >$$

Here the first suspension point is after s, passively waiting for the *last* call to complete (receiving the result in y), and the second suspension point passively waits for the $m1$ call to complete. With respect to the expressiveness of this construct, we observe that programs with nested call-get structures can be expressed without futures. In particular, one may wait for completions of several calls and continue when all calls have completed by letting the right brackets ("$>$") stand together, as in:

$$[\texttt{await}]\ \mathit{call}1 < ...; [\texttt{await}]\ \mathit{call}2 < ... >>$$

To await completions in one specific order, one would need to make the calls in the opposite order, as in

$$[\texttt{await}]\ \mathit{call}1 < ...; [\texttt{await}]\ \mathit{call}2 < ... >; ... >$$

where the return value of $\mathit{call}2$ is handled before that of $\mathit{call}1$. We use **await** when passive waiting is desired.

Delegation. Consider the case that a method body (say method m) ends by returning the result of a call to n, as in the method body $\{...; x := o.n(\overline{e}); \texttt{return}\ x\}$ where x is a local variable in the body. Here the result of the call is not used by the current process, and it may be desirable to avoid the waiting. With futures this could be done efficiently by returning a future, as in $\{...; f := o!n(\overline{e}); \texttt{return}\ f\}$ changing the type of the return value accordingly. The same efficiency can be achieved in the future-free setting by a form of *delegation* [25]. The body of m is now written as $\{...; \texttt{delegate}\ o.n(\overline{e})\}$. The statement

$$\texttt{delegate}\ o.n(\overline{e})$$

makes the current call (of m) terminate without producing a result, while delegating to the remote call $o.n(\overline{e})$ to send a result back to the caller of m. Type checking must ensure that the result type of n is appropriate. This gives the same efficiency for the current process as in the solution with futures, and without the need to change the return type.

FutFree is more high-level than languages with futures or call labels, since the syntactic complication of futures/labels is avoided, and it is more expressive than the future-free restriction of both Creol and ABS. However, *FutFree* does not support non-parenthetic nesting of calls and returns, nor delegation to more than one object.

A weakness of the tail construct is that it involves blocking at the end of the tail. The delegation mechanism $\texttt{delegate}\ o.n(\overline{e})$ avoids this waiting point and still makes use of the result. If o here is this, a local continuation (i.e., relative to the caller) will be triggered asynchronously. This is a bit similar to a continuation associated to a future, which could be added as a suspended process and enabled when the associated future is resolved. Such a continuation should then maintain the class invariant. When a method body contains a number of continuations, there may be a need for coordination of the different activities. Encore has solutions for this using futures [4]. A syntax for this concept of asynchronous continuation can also be made in the future-free setting (i.e., without explicitly mentioning the associated future). However, we will not add further syntax to *FutFree* since this can be simulated (even though less elegant).

```
class Service(Int limit, ProducerI prod) implements ServiceI {
  ProxyI proxy:= new Proxy(limit,this); //proxy does the main job
  {this!produce{}} // initial call
  ...
  Void produce(){ News ns;
    await ns := prod.detectNews()                // no blocking
      < db.logging(...) >;
    proxy!publish(ns)}                           // send the news
}
class Proxy(Int limit,ServiceI s) implements ProxyI{
  List[ClientI] myClients:=Nil; ProxyI nextProxy;
  ...
  Void publish(News ns){
    myClients!signal(ns); // multi-cast the result
    if nextProxy==null
    then s!produce() else nextProxy!publish(ns)}
}
```

Figure 9: The Publishing Example rewritten in the future-free language

Example. In Figure 9 we show the publishing example rewritten to the new format without futures (with changes in blue). This is essentially the same solution as that in Creol, Figure 4, but expressed without call labels/local futures. We also assume interfaces as in the Creol. The changes are straight forward. The Service object makes the same call to publish, but at a later time, when the news are available. By using a suspending publish call, the Service object is not blocked and has therefore similar efficiency as in the first version. And the blocking that used to be in the Proxy object is removed, and thus the Proxy objects will be more responsive. This indicates that passing futures as parameters can be avoided without loss of efficiency by using asynchronous call/return in combination with suspension. Instead of having a responsive Service object at the cost of blocking in Proxy objects, as in the original version, the future-free version has a responsive Service object as well as Proxy objects, but now the process queue of the Service object may be non-empty. Thus there is less need for first-class futures in a future-free language with cooperative scheduling than in one without. In general one may use the process queue and suspension rather than blocking separate object(s) in get statements. Forwarding a future can be replaced by a suspended process forwarding the future value. And this gives a deadlock-free solution.

The versions in Figures 9 and 5 are similar in that the Service objects are not blocked (apart from the logging part). And if there are no other *produce* and *publish* calls than the ones in Service and Proxy, there will be at most one uncompleted *produce* process and also at most one uncompleted *publish* process, for either version. For the version in Figure 9 there is an explicit

	ABCL	Rebeca	Creol	ABS	Encore	ASP	*FutFree*
futures	yes	no	yes	yes	yes	no	no
syntactic	yes	yes	yes	yes	yes	no	yes
first-class	yes	no	no	yes	yes	yes	no
cooperative	no	no	yes	yes	yes	no	yes
polling	yes	no	no	no	no	yes	no
dyn. creation	yes	no	yes	yes	yes	yes	yes
passive data	obj	obj	adt	adt	obj	obj	adt

Figure 10: Overview of future support in the selected languages.

interleaving point after the logging, whereas in the other version (in Figure 5) this interleaving is implicit since the *produce* process terminates. Thus the two versions may give rise to different executions.

2.8. A summary of active object languages

A summary of the main interaction-related features of some different languages for active objects is given in Figure 10. In the table, the entry "futures" is indicating whether a language supports the future implementation explicitly or not. The entry "syntactic" shows whether the waiting points are syntactically identified or not, in the context of a given class. For example, in ASP when a future is passed to an activity and its value is needed inside that, it is not textually clear whether it is a waiting point or not. For Rebeca the "yes" refers to the extension with *wait*. Moreover the table compares whether these languages support first-class, cooperative, polling, cooperative scheduling or not. The table entry "dyn. creation" indicates if dynamic object creation is supported, and the entry "passive data" indicates how temporary internal data structure is built, either by means of (passive) objects without their own execution thread, marked "obj", or user-defined data types, marked "adt".

An advantage of explicit futures is that it provides explicit (syntactic) identification of waiting points, typically by the **get** construct. For the category of languages with implicit future support, we may distinguish between those with explicit and implicit identification of waiting points.

The table summarizes the support of futures and related concepts for the different languages. The considered languages are chosen to give a certain variety, all in the setting of imperative programming with active objects.

For the category of implicit futures, there is a distinction between languages where the waiting points are syntactically given and those where they are not, as in the case of wait-by-necessity where the usage of a variable x bound to the result of a method call requires that the value is available. Thus sending x as a parameter does not require that the value is there, but updating or testing the value would normally require the value to be available. For instance, this means that a method $m(Int\ x)\{s_1; y := x{+}1; s_2\}$ will have an implicit potential waiting point at $y := x + 1$ when m is called with an actual parameter representing an implicit future (and when s_1 does not use x) whereas there is no such waiting point when m is called with an actual parameter representing an available value. This makes program reasoning complicated, for instance deadlock reasoning, and modular semantics is not possible since waiting depends on external objects. We will therefore limit the discussion to languages with explicit waiting points.

Operations on futures can be blocking, such as getting the result from a future, suspending, or it can be asynchronous and non-blocking, such as attaching a callback or a continuation to a future. For example, in AmbientTalk the future access is a non-blocking asynchronous operation, in which actors desiring a future value are registered as observers, then when the future is resolved, its value is sent to these registered observers. An observer actor can register a closure, a block of code, to a future that is applied when the future gets resolved [8]. An extended version of Encore [13] also offers high-level non-blocking coordination constructs operating on futures. For instance, it can apply a function on the first result of a bunch of futures and terminate the computations associated with the other futures. It offers complex coordination, including pipe-lining and speculative parallelism on futures, when they might be dependent on other futures. To have these complex coordinated workflows, they offer a new non-blocking asynchronous parallel abstraction $ParT$ (or $Par\ T$), which is a handle to parallel computations or a data structure for collecting future values, and also offer parallel combinators to operate on. Parallel combinators are non-blocking constructs that control and coordinate $ParT$ collections without blocking threads.

A recent survey [3] compares several active object languages such as ABS, Encore, Rebeca and ASP/Proactive according to their design aspects, the degree of synchronization, the degree of transparency and the degree of data sharing. It identifies the design purpose of these languages. For example Creol, ABS, and Rebeca are designed with program analysis in mind, while Encore and ASP/ProActive are optimized for efficient execution. The degree of synchronization is compared according to their synchronization primitives. This survey compares explicit and implicit futures as a degree of transparency. The degree of data sharing between active objects are compared as well. None of the active object languages support data sharing, apart from futures, since they are oriented toward distributed systems, where copying and sending data is more safe and efficient.

3. Unified Syntax and Semantics

Based on the overview above, we consider the main categories of language support for interaction mechanisms, given by

- support of first-class futures, object-local and method-local futures, and future-free (asynchronous call/return) interaction
- cooperative scheduling or not
- polling or not (for languages with futures)

This gives the following main categories for interaction mechanisms, where the six first categories (those with futures) may be with or without polling: i) cooperative scheduling and first-class futures with ABS/Encore as representatives, ii) cooperative scheduling and object-local futures, iii) cooperative scheduling and method-local futures with Creol as a representative, iv) non-cooperative scheduling and first-class futures with something in the direction of ABCL as a representative, v) non-cooperative scheduling and object-local futures, vi) non-cooperative scheduling and method-local futures, vii) cooperative scheduling and no support of futures, with *FutFree* as a representative, and viii) non-cooperative scheduling and no support of futures, with Rebeca (or the await-free restriction of *FutFree*) as representative. However, basic Rebeca does not use the call/reply paradigm.

As mentioned, we restrict ourselves to imperative languages, and assume syntactic identification of any waiting points. In particular, to keep the discussion uniform, we use the following ABS-inspired syntax for the different language mechanisms representing the basic ways of using futures or asynchronous call/return interaction without futures.

3.1. Syntax

The unified syntax is given in Figure 11. We assume static type checking. The different language combinations are obtained by including/excluding polling, await, first-class, or object/method-local futures. In the case of method-local futures, futures may not be assigned to fields nor passed as parameters or method results. In the case of object-local futures, futures may be assigned and passed as parameters/result but only for local methods (methods not exported through any interface). This guarantees static control of what is legal. For first-class future languages, futures may be assigned and passed to parameters/result to any method (modulo static typing restrictions).

Moreover, the different language combinations are achieved by taking the basic and future-free constructs, adding no futures, or adding futures, either with first-class future operations (assignment and parameter passing), no first-class future operations (other than assignment to local variables) for the case of method-local futures, or assignments to future variables and passing of futures in local methods for the case of object-local futures, and furthermore adding suspension (by the **await** keyword) or not, and adding polling or not for combinations with futures. Furthermore, languages with method-local futures may be divided further by allowing multiple-read or single-read futures. Single-read futures depend on static constraints ensuring that each path through a method has at most one read of a given future, whereas multiple-read languages are free from this restriction. Thus we cover in all *eighteen language combinations.*

The future-free constructs (including tail) are included in all language combinations, however, some of these constructs can be omitted in languages with futures since they can be expressed by means of futures. For instance, the following statements are inter-definable: The call [**await**] $x := o.m(\overline{e})$ is the same as [**await**] $x := o.m(\overline{e}) < skip >$, and the call [**await**] $x := o.m(\overline{e}) < s >$ may be simulated by means of futures by $f := o!m(e); s;$ [**await**] $x :=$ **get** f, for

Basic constructs	
$x := e$	assignment (x a variable, e an expression)
$x :=$ **new** $C(\overline{e})$ [**at** o]	object creation ($\overline{e}$ actual class parameters)
return e	creating a method result/future value
if c **then** s [**else** s'] **fi**	if-statement (c a Boolean condition)
while c **do** s **od**	while-statement
Future-free constructs	
$o!m(\overline{e})$	simple asynchronous call, non-blocking
[**await**] $x := o.m(\overline{e})$	blocking/non-blocking call
[**await**] $x := o.m(\overline{e}) < s >$	blocking/non-blocking call with tail
delegate $o.m(\overline{e})$	termination and delegation
Basic future constructs	
$f := o!m(\overline{e})$	asynchronous call, non-blocking
[**await**] $x :=$ **get** f	getting a future value
Cooperative scheduling	
await c	conditional suspension
await $f?$	await on a future
Polling	
$f?$	checking if a future is resolved

Figure 11: Unified Syntax. Here f is a declared future variable and x an ordinary program variable, s, s' denote statement lists, and $[\ldots]$ denotes optional parts.

some fresh future variable f, where [**await**] $x :=$ **get** f again can be seen as a shorthand for [**await**] $f?; x :=$ **get** f. Furthermore, we may extend the **await** notation to a conjunction of futures. We use dot-notation for suspending/blocking calls and ! for making an invocation request.

Polling of a future means checking if a future is resolved or not, for instance in an if-test, say **if** $f?$ **then** $x :=$ **get** f **else** ... **fi**. Polling may lead to complicated branching structures, and is often avoided in languages with support of explicit futures. Basic constructs are quite standard, but object creation has as an optional part (**at** o) for specifying the placement of the new object. The default is **at** this, i.e., the same location as the parent object.

3.2. Operational Semantics

In this part, we present (relevant parts of) the operational semantics of the different language combinations, including future-free, method-local, object-local, and first-class future languages, using the style of structural operational semantics. The purpose of the operational semantics is to show the underlying asynchronous communication between active objects at runtime. Each rule in the operational semantics corresponds to one execution step.

A system state is given by a configuration, which is a multi-set of objects and messages, either invocation messages or reply messages, in addition to future objects in the case of first-class future languages. An object is represented as

$$\texttt{ob}(o \mid s, a, l, q)$$

where o denotes the object identity, s is the list of active statements, a the state of object fields (attributes), l the state of local variables defined in a method (including the parameters), and q the internal process queue. The process queue is only needed in languages with cooperative scheduling. The pair $(s \,\&\, l)$ represents (the remaining part of) the active process, with statements s and local state l. At suspension, this active process is moved to the process queue, making the object *idle*. When idle, an object may continue with another (enabled) process from the process queue, or start a new method invocation. We use the syntax

$$\texttt{invoc}\,(o, u, m, \overline{d})\,\texttt{to}\;\; o'$$

for an invocation message from object o to object o' where m is the name of the called method, u the identity of the call/future, and $\overline{d}$ the list of actual parameters. An object will have an associated invocation queue, and also a reply queue in the case of non-first-class future languages. We let $\texttt{iq}(o \mid p)$ denote the invocation queue associated with o containing messages (p) to o.

It is worth mentioning that an object identity is unique, which can be achieved by using the parent object identity and a counter [22]. Similarly a unique call or future identity u can be achieved by using the caller object identity and a separate counter. The counters can be represented by implicit fields in a. In the case of first-class future languages, this identity acts as the future identity.

The operational semantics is given by a number of rewrite rules. A rule can be applied to a configuration if the left-hand-side matches a subset of the configuration (possibly reordered). If the left-hand-sides of two rules match disjoint parts, they can be applied at the same time, reflecting concurrent behavior. Each rule involves at most one object, reflecting that the objects are executing independently from each other. Thus the objects execute in parallel. Nondeterminism is achieved when (at least) two left-hand-sides match overlapping parts of a configuration.

States are given by mappings from variable names to values, and $l[x \mapsto d]$ denotes the local state l updated so that variable x binds to data value d (adding such a binding if x is not bound in l). We use $+$ for map composition with overwriting, so that $a + l$ is the union of map a and l, using l for variable names with a binding in both a and l, reflecting that the binding of a variable name in the inner scope l shadows any binding of that name in the outer scope a. Therefore, $a + l$ represents the total state of an object as used for evaluation. For an expression e, the notation $[e]_{a+l}$ abbreviates the evaluation of e in the context of $a + l$. For a variable x, the evaluation $[x]_{a+l}$ equals $[x]_l$ if l has a binding for x, otherwise $[x]_a$, i.e., the value of a variable x is found in l if l has a binding for x, otherwise in a. Therefore the semantics of an assignment statement is given by two rules; one for the case that x is a local variable and one for the case that x is a field:

$$\begin{array}{lrl}
\textsf{assign-local}: & & \texttt{ob}(o \mid x := e; s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid s, a, l[x \mapsto [e]_{a+l}], q) \\
& \texttt{if} & x \in l
\end{array}$$

$$\begin{array}{lrl}
\textsf{assign-field}: & & \texttt{ob}(o \mid x := e; s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid s, a[x \mapsto [e]_{a+l}], l, q) \\
& \texttt{if} & x \notin l
\end{array}$$

Type checking ensures that x is defined in l or a. Similarly there are two rules for if-statement, depending on the value of the if-condition.

$$\begin{array}{lrl}
\textsf{if-then}: & & \texttt{ob}(o \mid \texttt{if } c \texttt{ then } s1 \texttt{ else } s2 \texttt{ fi}; s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid s1; s, a, l, q) \\
& \texttt{if} & [c]_{a+l} = true
\end{array}$$

$$\begin{array}{lrl}
\textsf{if-else}: & & \texttt{ob}(o \mid \texttt{if } c \texttt{ then } s1 \texttt{ else } s2 \texttt{ fi}; s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid s2; s, a, l, q) \\
& \texttt{if} & [c]_{a+l} = false
\end{array}$$

The semantics of the call $[\texttt{await}]\ x := o.m(\overline{e})$ is given by that of $f := o!m(e); s; [\texttt{await}]\ x := \texttt{get}\ f$, for some fresh future variable f. And similarly the semantics of the call $o!m(\overline{e})$ is given by that of $f := o!m(e)$, for some fresh future variable f. Thus the semantics of the future-free languages is given by means of futures at the run-time level.

The operational semantics of languages with first-class futures differs from those with local futures or without futures, since they need a representation of shared future objects, and we therefore consider the two classes of operational semantics in two subsections.

3.2.1. Operational semantics of languages without first-class futures

In the case of local or no futures, we need to deal with reply messages, and let $\texttt{reply}\,(u, v)\ \texttt{to}\ \ o$ represent a reply message to the caller o, where u is the call identity and v the returned value. Each object o must keep track of the received reply messages, in a reply queue $\texttt{rq}(o \mid r)$, where r is a queue of (u, v) pairs. When a $\texttt{get}\ \ f$ operation is executed, one must check if there is a value for f in the reply queue. Thus the order of the reply queue is irrelevant, and the handling of the internal process queue can be seen as non-deterministic. Call identities are invisible to the programmer and cannot be passed to other objects (where these call identities would be unknown). For a statement containing $\texttt{get}\ \ f$, the future (expression) f must first be evaluated. Therefore $\texttt{ob}(o \mid [\texttt{await}]\ x := \texttt{get}\ \ f; s, a, l, q)$ reduces to $\texttt{ob}(o \mid [\texttt{await}]\ x := \texttt{get}\ \ [f]_{a+l}; s, a, l, q)$.

Figure 12 defines the operational semantics for local futures, with or without suspension (ignoring all rule instances with $\texttt{await}$ in the latter case), and with polling if including polling rules similar to those in Figure 13 (using the reply queue for checking the presence of a future). The difference between object-local and method-local futures is not visible in the rules, but in the underlying language, restricting assignment and passing of futures (in the ways mentioned). The operational semantics for the future-free languages is obtained by replacing future-free calls by calls with futures as explained above, and allowing/disallowing suspension as desired. The syntax [**await**] denotes an optional **await**, o and o' denote object identities, and an object is **idle** when there is no active process. In a left hand side or condition, the symbol "_" matches any term.

The async-call rule executes an asynchronous call where o is the caller invoking method m of object e with $\overline{e}$ as actual parameters. This rule generates a globally unique call identity (u), which is assigned to the future variable f. The call creates the invocation message **invoc** $(o, u, m, [\overline{e}]_{a+l})$ **to** $[e]_{a+l}$, where the actual parameters $\overline{e}$ and the callee e are evaluated before sending this message.

The start rule says that when an object is **idle** and there is an invocation message in its invocation queue, the object may start to execute the corresponding method. It then captures the method's body s as its active process with local variables given by the method declaration (bound to default values), binding formal parameters to the actual ones, and storing the caller object in the (implicit) caller parameter and the call identity in the (implicit) callid parameter. These two implicit variables are needed to execute the return statement.

The reply rule represents the case when an object o returns a value e to the caller o'. It creates a reply message with the callid as its first argument and the evaluation of $[e]_{a+l}$ as the value and forwards this message back to the caller, as given by the local variable caller. We assume each method body ends with a return statement.

The get-reply rule describes how a reply message to an object o with label u and return value v enters into the reply queue (**rq**$(o \,|\, r)$) of object o.

The get, suspend, await-pq rules represent query statements for retrieving the reply. For the first two rules the query is already in the active process, and for the third it is in the internal queue. The reply value is not removed from the reply queue, thus multiple reads are possible. (For single-read futures it should be removed.) When a reply is needed and the reply message is in the queue, the response value v is fetched. The suspend rule takes care of the case when a reply is needed in an await statement, but the corresponding reply message is not in the queue. Then the whole process with its local variables are suspended and placed at the end of the internal queue. The notation $q \cdot z$ denotes that a process z is appended to q. A suspended process with the state of its local variables are represented by a pair, written (**await** $x :=$ **get** $u; s \,\&\, l$)), where **await** $x :=$ **get** $u; s$ is the suspended process with local state l.

The tail rule says that we implicitly create a fresh local variable f (like a future) to talk about the value resulting from the method call. Then in the next step, the async-call rule binds the f variable to a unique identity u. We ignore here the rules for other statements such as if and while, since they are not central to our discussion.

The rules in Figure 12 can be used to define object-local futures as well as method-local futures and future-free languages, restricting the language accordingly (disallowing passing of futures in non-local methods, disallowing all

$$\begin{array}{lll}
\textsf{async-call :} & & \texttt{ob}(o \mid f := e!m(\overline{e}); s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid f := u; s, a, l, q) \\
& & \texttt{invoc}\,(o, u, m, [\overline{e}]_{a+l})\,\texttt{to}\;\; [e]_{a+l} \\
& & \texttt{where}\;\; u \text{ is a fresh locally unique identity}
\end{array}$$

$$\begin{array}{lll}
\textsf{invocation :} & & \texttt{invoc}\,(o, u, m, \overline{d})\,\texttt{to}\;\; o' \\
& & \texttt{iq}(o' \mid p) \\
& \longrightarrow & \texttt{iq}(o' \mid p \cdot \texttt{invoc}(o, u, m, \overline{d}))
\end{array}$$

$$\begin{array}{lll}
\textsf{start :} & & \texttt{iq}(o \mid \texttt{invoc}(o', u, m, \overline{d}) \cdot p) \\
& & \texttt{ob}(o \mid \textbf{idle}, a, \mathit{empty}, q) \\
& \longrightarrow & \texttt{iq}(o \mid p) \\
& & \texttt{ob}(o \mid s, a, [\textsf{caller} \mapsto o', \textsf{callid} \mapsto u, \overline{x} \mapsto \overline{d}, \overline{l} \mapsto \overline{l_0}\,], q) \\
& & \texttt{where}\;\; \text{method } m \text{ binds to } m(\overline{x})\{\overline{T\;l}; s\} \text{ (with initial values } \overline{l_0} \text{ of } \overline{l}\text{)}
\end{array}$$

$$\begin{array}{lll}
\textsf{reply :} & & \texttt{ob}(o \mid \texttt{return}\;\; e, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid \textbf{idle}, a, \mathit{empty}, q) \\
& & \texttt{reply}\,([\textsf{callid}]_l, [e]_{a+l})\,\texttt{to}\;\; [\textsf{caller}]_l
\end{array}$$

$$\begin{array}{lll}
\textsf{get-reply :} & & \texttt{reply}\,(u, v)\,\texttt{to}\;\; o \\
& & \texttt{rq}(o \mid r) \\
& \longrightarrow & \texttt{rq}(o \mid r \cdot (u, v))
\end{array}$$

$$\begin{array}{lll}
\textsf{get :} & & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid [\texttt{await}]\; x := \texttt{get}\;\; f; s, a, l, q) \\
& \longrightarrow & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid x := v; s, a, l, q) \\
& \texttt{if} & ([f]_{a+l}, v) \in r
\end{array}$$

$$\begin{array}{lll}
\textsf{suspend :} & & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid \texttt{await}\;\; x := \texttt{get}\;\; f; s, a, l, q) \\
& \longrightarrow & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid \textbf{idle}, a, \mathit{empty}, q \cdot (\texttt{await}\;\; x := \texttt{get}\;\; [f]_{a+l}; s \,\&\, l)) \\
& \texttt{if} & ([f]_{a+l}, \_) \notin r
\end{array}$$

$$\begin{array}{lll}
\textsf{await-pq :} & & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid \textbf{idle}, a, \mathit{empty}, q \cdot (\texttt{await}\; x := \texttt{get}\;\; u; s \,\&\, l) \cdot q') \\
& \longrightarrow & \texttt{rq}(o \mid r) \\
& & \texttt{ob}(o \mid x := v; s, a, l, q \cdot q') \\
& \texttt{if} & (u, v) \in r
\end{array}$$

$$\begin{array}{lll}
\textsf{tail :} & & \texttt{ob}(o \mid [\texttt{await}]\; x := e.m(\overline{e}) < s >; s', a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid f := e!m(\overline{e}); s; [\texttt{await}]\; x := \texttt{get}\;\; f; s', a, l[f \mapsto \mathit{nil}], q) \\
& & \texttt{where}\;\; f \text{ is a fresh local (future) variable}
\end{array}$$

Figure 12: Operational rules for local futures and future-free statements.

passing of futures and assignments of futures to fields, and disallowing the use of futures variables in a program, respectively). In the latter case, future variables and the **get** statement are not part of language syntax. However, call identities and **get** statements are implicitly generated by the rules. Asynchronous calls, which use futures, are therefore not part of the language as seen by the programmer, but simple asynchronous call is. The rule for a simple asynchronous call is similar to that of asynchronous call, i.e.,

$$\texttt{ob}(o \mid e!m(\overline{e}); s, a, l, q) \longrightarrow \texttt{ob}(o \mid s, a, l, q) \quad \texttt{invoc}\,(o, u, m, [\overline{e}]_{a+l})\, \texttt{to} \;\; [e]_{a+l}$$

where u is locally fresh as before. There is no state update since there is no future variable involved. (In fact, one could use *nil* instead of u in the invocation message to indicate that no return is needed.)

As mentioned, the rules in Figure 12 define object-local and method-local futures. For both these language classes we can define the versions without cooperative scheduling by removing the process queue (pq) and removing rules involving **await** statements. However, the addition of first-class futures requires sharing of futures, which is not possible with the rules of Figure 12. This is considered below.

The rules for polling (given for the case of first-class futures in Figure 13) describe that the test $f?$ gives true if and only if the future is resolved/the reply is in the reply queue.

3.2.2. Operational semantics of languages with first-class futures

In Figure 13, we define operational semantics of first-class futures, representing a future as a global object with a unique identity (the future/call identity). We let $(\texttt{fut}(u \mid))$ denote an unresolved future object with identity u, and $(\texttt{fut}(u \mid v))$ denote a resolved future object with value v. In this paradigm, a future is once writable, but readable many times by objects that have a reference to it. Objects can individually and synchronously access the future object value when resolved. Some of the rules from Figure 12 must then be modified, as shown in Figure 13 with primed versions of the relevant rules. The rules invocation, start, and tail are as before and are therefore not redefined.

The async-call' rule represents the creation of a unique future object corresponding to a method call. It creates an unresolved future object with a unique identity (details omitted). This rule creates an invocation message to the callee as before. However, the caller identity (o) is not needed in the case of first-class futures unless the language supports the implicit caller parameter, and one could remove o from the invocation message in this case (simplifying both the async-call' and the invocation rule).

The reply' rule says that when object o returns the value of e, it becomes **idle** and the reply goes to the future object with identity u. The get', await-pq', and suspend' rules capture the cases when there is a query statement to get the result as before, but now using the future object. In the get' rule, the query succeeds since the reply value is resolved in the future object $\texttt{fut}(u \mid v)$, hence if the evaluation of $[f]_{a+l}$ according to the local variables and object fields equals

$$\begin{array}{lll}
\textsf{async-call'}: & & \texttt{ob}(o \mid f := e!m(\overline{e}); s, a, l, q) \\
& \longrightarrow & \texttt{ob}(o \mid f := u; s, a, l, q) \\
& & \texttt{fut}(u \mid\,) \\
& & \texttt{invoc}\,(o, u, m, [\overline{e}]_{a+l})\ \texttt{to}\ [e]_{a+l} \\
& & \texttt{where}\ \ u \text{ is a fresh and globally unique identity}
\end{array}$$

$$\begin{array}{lll}
\textsf{reply'}: & & \texttt{fut}(u \mid\,) \\
& & \texttt{ob}(o \mid \texttt{return}\ e, a, l, q) \\
& \longrightarrow & \texttt{fut}(u \mid [e]_{a+l}) \\
& & \texttt{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q) \\
& \texttt{if} & [\textsf{callid}]_l = u
\end{array}$$

$$\begin{array}{lll}
\textsf{get'}: & & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid [\texttt{await}]\ x := \texttt{get}\ f; s, a, l, q) \\
& \longrightarrow & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid x := v; s, a, l, q) \\
& \texttt{if} & [f]_{a+l} = u
\end{array}$$

$$\begin{array}{lll}
\textsf{suspend'}: & & \texttt{fut}(u \mid\,) \\
& & \texttt{ob}(o \mid \texttt{await}\ x := \texttt{get}\ f; s, a, l, q) \\
& \longrightarrow & \texttt{fut}(u \mid\,) \\
& & \texttt{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\texttt{await}\ x := \texttt{get}\ u; s \,\&\, l)) \\
& \texttt{if} & [f]_{a+l} = u
\end{array}$$

$$\begin{array}{lll}
\textsf{await-pq'}: & & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\texttt{await}\ x := \texttt{get}\ u; s \,\&\, l) \cdot q') \\
& \longrightarrow & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid x := v; s, a, l, q \cdot q')
\end{array}$$

$$\begin{array}{lll}
\textsf{polling1}: & & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid \texttt{if}\ f?\ \texttt{then}\ s1\ \texttt{else}\ s2\ \texttt{fi}; s, a, l, q) \\
& \longrightarrow & \texttt{fut}(u \mid v) \\
& & \texttt{ob}(o \mid s1; s, a, l, q) \\
& \texttt{if} & [f]_{a+l} = u
\end{array}$$

$$\begin{array}{lll}
\textsf{polling2}: & & \texttt{fut}(u \mid\,) \\
& & \texttt{ob}(o \mid \texttt{if}\ f?\ \texttt{then}\ s1\ \texttt{else}\ s2\ \texttt{fi}; s, a, l, q) \\
& \longrightarrow & \texttt{fut}(u \mid\,) \\
& & \texttt{ob}(o \mid s2; s, a, l, q) \\
& \texttt{if} & [f]_{a+l} = u
\end{array}$$

Figure 13: Operational rules for first-class futures.

u, then the object can update x. For the await-pq' rule, this query is in the internal queue and the active process is **idle**, and then the object deals with this suspended query. If the reply value is resolved, the object can update x. For the suspend' rule, an **await** statement is in the internal queue and the reply is not resolved yet, therefore the whole process with its corresponding local variables is suspended in the internal queue.

The semantics of polling is similar to that of get/get' for the case that the future is resolved, in which case the expression $f?$ is replaced by *true*, and suspend/suspend' for the case that the future is not resolved, in which case the expression $f?$ is replaced by *false*.

For languages without cooperative scheduling we omit *pq* and omit the suspend' and await-pq' rules, and the optional **await** in get'. This means that we have defined the operational semantics of all the chosen classes of languages. For simplicity we have not considered aspects such as garbage collection of futures.

4. Evaluation

We evaluate the various mechanisms for interaction between active objects with respect to the dimensions stated in the introduction. As mentioned we consider imperative languages with or without futures but not implicit futures.

4.1. Syntactic complexity

We compare the different versions of the unified syntax in Figure 11, including related static checking aspects and also immediate pragmatic issues. It is obvious that explicit futures require programming awareness of call identities or futures, representing a new kind of entity, and new programming choices like how and when to use them. Thus the notion of future variables comes with a syntactic and pragmatic cost, especially since the addition of future variables does not reduce the need of ordinary variables.

When a new method is declared in a language with first-class futures, one must decide for each parameter and return value if it should be represented by a future or not. Moreover, the passing of futures in a typed language requires typing support of future types, as in $Fut[T]$. And if T' is a subtype of T, one may want $Fut[T']$ to be a subtype of $Fut[T]$. This means that first-class futures and object-local futures give a more complex type system, whereas method-local futures can be handled with a simple predefined type, say Fut. Pragmatically, the resolving of futures of futures may lead to some confusion.

For local futures, we have seen in subsection 3.2.1 that it is possible to ensure that futures really are object-local or method-local, as appropriate, by means of simple language restrictions. First-class future operations reuse basic language mechanisms and do not add further to the syntactic restrictions of a language. For all languages with futures, one may add static restrictions to ensure that a get operation is not done on a *nil* future (one that has not been associated with a call yet), a problem which is similar to avoiding nil references. For first-class or object-local futures, one may for instance insist that a future can only be passed when statically not *nil*. For method-local futures, the situation is much simpler since there is no passing of futures. However, static restrictions are needed to ensure that a get statement $x :=$ **get** f is type-correct when x is simply typed by the predefined type Fut [24].

Future variables give a level of indirectness in that the retrieval of the result of a call is no longer syntactically connected to the call, compared to future-free languages. The connection might range from trivial (as in $f := o.m(...); x :=$ **get** f) to non-trivial, for instance when the future is received as a parameter. In the latter case a get statement in a given method may not statically correspond to a unique call statement. This is a complicating factor in static analysis, especially modular static analysis. One may overestimate the set of call statements that correspond to a given get statement, but this requires access to the whole program, which is not possible in open-ended software systems.

From a pragmatic point of view, we notice that in languages with first-class futures, the passing of futures is decided when a method is declared rather than when it is called, i.e., for each method parameter or return value one must decide at declaration time if it should be represented by a future or not, and all calls made later must obey this regime. Then, too few parameters given as futures may imply that desired passing of futures is not supported, and too many future parameters lead to dummy future variables on the call-site and syntactic overhead. This can be a problem in languages with first-class futures since the need of the future mechanism depends on the calling objects. Thus first-class future languages may not provide the desired flexibility. This is not a problem for method-local futures since there is no passing of futures. And the problem is rather limited for object-local futures since future passing is limited to local methods, for which all calls appear in the class definition (but class inheritance may reintroduce the flexibility problem).

Consider next cooperative scheduling. The **await** mechanism can be decided for each call upon need, and is not pre-decided in method declarations. This means that the same method may be called by means of blocking calls as well as by suspending calls. This gives a high degree of flexibility. The addition of cooperative scheduling is syntactically cheap since essentially only one keyword (**await**) is needed, and type analysis is not affected. Pragmatically, the use of suspension should reflect the need to wait for something without blocking, and the choice between blocking and non-blocking waiting should be obvious for a conscious programmer. However, the combination of first-class futures and cooperative scheduling gives two different mechanisms that may be used to deal with waiting without blocking, and the choice between them is less obvious.

Finally we consider the concept of polling. Adding polling is syntactically cheap, essentially a new predefined function ("f?") is added. Type checking issues are trivial, but some language restrictions may be added to control where polling is allowed, for instance restricting it to if-conditions (possibly allowing Boolean expressions involving several polling checks). Pragmatically, the use of polling gives increased programmer control. For instance, one may wait for two results and react to them in the order they appear. This may look like

if f1? **then** react1; **get** f2; react2 **else if** f2? **then** react2; **get** f1; react1 **fi fi**.

However, when there are more than two futures that should be handled individually, the branching becomes rather involved and with repetition of parts of the code. Use of futures or cooperative scheduling may provide more elegant solutions: With cooperative scheduling one may start one process for each future, and let each process await the corresponding future. With first-class futures one may pass each future to a new object taking care of the proper reaction.

All in all, we have considered syntactic complexity and immediate pragmatic issues. We have seen that futures add more syntactic complexity and more pragmatic complications than cooperative scheduling and polling. And first-class futures add more pragmatic complications than object-local futures, which again add more pragmatic complications than method-local futures. Polling without cooperative scheduling may lead to a complex programming style, whereas languages with first-class futures or cooperative scheduling are less depending on polling.

4.2. Semantic complexity

The operational semantics (Figures 12 and 13), although abstractly defined, shows that local futures and future-free languages can be handled by reply queues (or sets) on the caller side, whereas first-class futures is handled by shared future objects. For method-local futures, the replies can be removed when the method instance (process) that generated the future has terminated. This can be formalized in the semantics by including the identity of the calling process (given by *callid*) in the future identity, for instance as a pair (i, j) where i is the process identity and j the future identity relative to i. We may then add a rule

$$\mathtt{rq}(o \mid r \cdot ((i,j),v) \cdot r')\ \mathtt{ob}(o \mid s,a,l,q) \longrightarrow \mathtt{rq}(o \mid r \cdot r')\ \mathtt{ob}(o \mid s,a,l,q)\ \ \mathtt{if}\ \ i \not\in q$$

where $i \not\in q$ checks whether a process with callid equal to i is in the process queue q. In the case of single-read futures as well as future-free languages, a reply in rq may even be removed as soon as it is read, in the rules get and await-pq (by removing the reply message from r in the right-hand-side). This still requires that removal is handled upon method termination, as for multiple-read local futures, in case the future is not read. (Alternatively this could be controlled by commands inserted in the code during static checking.) Thus for method-local futures (both single-read and multiple-read), the reply messages can be removed efficiently without general garbage collection. The same scheme can be used for future-free languages.

First-class futures involve the creation of future objects. The placement of the futures in a distributed system is not specified by the abstract operational semantics (other than being placed somewhere in the system configuration), nor is the removal of these objects, but could be added by rules that consider the total system (letting futures that are no longer referred to in the system be deleted). A more efficient implementation is possible, as discussed in subsection 4.4.

The notion of cooperative scheduling adds an internal process queue to each active object, together with en-queuing and de-queuing operations. The semantics of basic statements (other than call, return, get, await, polling) is not affected by futures nor cooperative scheduling. For the considered mechanisms, first-class futures make the most complex change of the basic operational semantics, since this mechanism changes the notion of system configuration and necessitates garbage collection of futures, while that of cooperative scheduling is less involved, and that of polling is fairly trivial.

4.3. Expressiveness

We first show that the future mechanism can be encoded in an active object language with future-free constructs, using the asynchronous call/return paradigm together with dynamic object creation. Given a method $Vm(\overline{\mathsf{T\ x}})$ declared in an interface I (where V is the type of the return value), we define an interface for futures for this call by

```
interface Future_m {
   Bool resolve()        // waiting until resolved
   V get()                // gets the value when resolved
}
```

and an extension of this interface for the case that we allow *polling*:

```
interface PFuture_m extends Future_m{
   Bool resolved()       // polling
}
```

These interfaces are implemented by two classes, FUTURE_m and PFUTURE_m respectively, given in Figure 14. A new future then corresponds to a new future object, using the appropriate class. Thus, if f is an object of interface Future_m, the creation of a future by

$$f := \texttt{new}\ \ FUTURE\_m(o, e)$$

corresponds to the call $f := o!m(e)$ in a language with futures. The call $f.resolve()$ corresponds to blocking while waiting for a result, and the call $x := f.get()$ corresponds to $x :=$ **get** f in a language with futures. For f of interface PFuture_m, the call $f.resolved()$ corresponds to the test $f?$. Furthermore, first-class operations on f correspond to first-class future operations. In case of cooperative scheduling, the call **await** $x := f.get()$ corresponds to **await** $x :=$ **get** f in a language with futures, and the call **await** $f.resolve()$ corresponds to **await** $f?$ in a language with futures.

Consider the implementation of futures with polling given in Figure 14 by class PFUTURE_m (for a given method m as above). The call to m in the class implementation uses **await** so that the future object will be able to perform incoming calls before the future is resolved, and in this way be able to return the appropriate result of polling requests. Thus, implementation of polling requires suspension (or could be predefined outside the language).

```
class PFUTURE_m(I o, T̅ x̅) implements PFuture_m {
  Bool res:= false; // is the future resolved?
  V val; // the value of the future when resolved
 // initial code
 {await val:=o.m(x̅); res:=true}// non-blocking
  Bool resolved(){return res}
  Bool resolve(){await res; return true}
  V get(){await res; return val}
}
```

```
class FUTURE_m(I o, T̅ x̅) implements Future_m {
  Bool res := false; // is the future resolved?
  V val; // the value of the future when resolved
  // initial code
  {val:=o.m(x̅); res := true} // blocking
  Bool resolve(){return true} // await res is implicit
  V get(){return val} // await res is implicit
}
```

Figure 14: Implementation of simulated futures with and without polling.

For the case without polling we may implement the future mechanism without use of suspension, by class FUTURE_m in Figure 14. Here we can use a blocking call to *m* since nothing can be done by the future object when not resolved. Therefore we do not need **await** *res* in the implementation since the object cannot respond to any request when the future is not resolved. This could lead to more efficient scheduling. Since this implementation requires only the blocking call construct, it can be expressed in all considered languages. Thus built-in futures are not strictly needed in languages without futures since they can be expressed (simulated) within these languages, but polling requires suspension (or similar mechanisms). In either case, the implementation of a future uses one object with its own thread. This gives a simple high-level model of the future mechanism. It may seem like an inefficient solution, but due to the passiveness of the future objects, it suffices that they are given CPU time only when there is an incoming call/reply from the environment.

Each call involving a future is implemented as an object of class FUTURE (or PFUTURE). The future object makes the call and (eventually) receives the return value, as well as dealing with requests about the future value. We support polling of future values as well as blocking and non-blocking waiting primitives for requesting the future value.

On one hand it is obvious that the addition of language features may increase the expressive power. With our unified syntax the future-free language is a subset of that for method-local futures, which is a subset of that for object-local futures, which again is a subset of that for first-class futures. Their expressiveness is accordingly. And it is clear that the addition of cooperative scheduling adds expressive power. On the other hand we have seen that first-class futures can be expressed in languages without first-class futures or without futures, by means of object generation using predefined classes. Thus if shared futures are occasionally desirable, they can be defined by means of explicit future objects definable within the language. We have also seen that simulation of polling requires the await mechanism (or similar).

4.4. Efficiency

In this part we compare the efficiency of active object languages, considering the amount of message exchange and complexity of the garbage collection process. For example, in a distributed system with wireless IoT devices, the amount of network traffic is vital, and should be taken into account. Since IoT devices suffer from resource and power limitations, communication efficiency, as well as space and time limitations, matter. Correspondingly, a high amount of message passing and cumbersome garbage collection cause high resource and power consumption, which in general should be avoided.

In active object languages, the future paradigm and the interaction mechanisms influence the number of communication messages and the garbage collection. Consider a method result that is needed by several objects in addition to the caller. For languages supporting the asynchronous call/return paradigm, the network traffic consists of two messages (i.e., from caller to callee and back). Assuming n (other) objects need the value and that the caller can reach them indirectly, it then takes at least n messages to forward this value to them, say n' ($n' \geq n$). Thus the number of exchanged messages is $2 + n'$. We assume that the forwarding is done by parameter passing, say through void methods with no return message, and we count the number of messages needed for all n objects to receive the value (including the forwarding), not counting other related messages. Moreover, by using suspension one can avoid blocking the caller.

In the case of first-class futures, the number of exchanged messages depends on the update strategy, as mentioned in the ASP part (in subsection 2.6). We here assume that the future object is kept on the caller side. In the *eager-forward-based strategy*, the caller forwards the future reference to n objects and forwards the future value when resolved. With the assumptions above, the number of exchanged messages would be $2{+}2\,n'$. Considering the *eager-message-based strategy* or subscription scheme, the caller sends the future reference to n objects, each object receiving the future subscribes itself to the future, and then the future value is sent back to all of them when resolved. Hence the number of exchanged messages would be $2 + 3\,n'$, assuming the n' messages go to distinct objects. In the case of *lazy strategy* with blocking when needed, the caller sends the future reference to n objects, and when the value is required, they ask for it from the future object, which sends back the value. Hence the number of messages would be $2 + n' + 2\ n$, assuming all n objects really need the value.

We see that the number of messages gets higher with the future mechanism. In a distributed system this could cause problems. And in particular for IoT systems, this could lead to exhaustion of network capacity or battery time if n is greater than 1 for a large portion of the calls.

Consider next *garbage collection*. The purpose of garbage collection is to deallocate memory resources that are no longer in use at runtime. There are different kinds of strategies for garbage collection. One is *reference counting*, in which the number of references to an object (say a future object) is counted. An object can be deallocated when the number of references to it reaches zero. A disadvantage of reference counting is that for every object a resource must be reserved for storing the number of references to it, and this number must continuously be updated.

Another common way of garbage collection is *reference tracing*, i.e., following all references. It distinguishes between reachable and unreachable objects, and deallocates the latter ones. An object is reachable if it is referenced by a variable in an (potentially) active object directly or through a chain of references. A traditional reference tracing collector temporarily stops the program execution when it wants to deallocate unreachable objects. This guarantees that reachablility does not change while doing garbage collection. Halting the program execution can be undesirable in distributed systems with active entities. In these systems, an object might be used by several distributed units, necessitating distributed garbage collection which is complex, slow, and costly.

The given operational semantics does not include garbage collection. Futures are typically many and short-lived, which may cause much garbage. In contrast, the active objects are long-lived. Thus in languages without passive objects, there is then little or no need for garbage collection other than the future objects. Therefore the garbage collection of futures can be a problematic issue. For the given operational semantics, a straight forward implementation of garbage collection for first-class futures requires distributed garbage collection of future objects, since the futures are global runtime objects, whereas object-local futures require local garbage collection within each object, since the futures are stored and referenced locally in each object.

As already explained in subsection 4.2, method-local futures can be disposed without use of garbage collection, and the reply messages used in future-free languages can be removed efficiently. And the same implementation can be used for single-read futures.

For languages with first-class futures, the garbage collection mechanism for futures highly depends on the future update strategy, as already discussed in subsection 2.6. If a strategy is eager, a future is updated as soon as its value is computed, then a local garbage collection is enough since it does not need to be stored globally. Local garbage collection can be performed by classical garbage collection within an object or by combining different garbage collection techniques with static analysis approaches [6]. If an update strategy is lazy, a future value must be kept in a future value list in the callee for potential later requests; moreover, a future can be disseminated to many active objects, thus it is non-trivial to decide when a future value can be removed from a future value list. In this case, distributed garbage collection is required, which can be done by reference counting or a combination of garbage collection mechanisms [6].

4.5. Security/Privacy challenges

The future concept comes with a notion of future identity, but not a notion of associated caller, callee, or creator. However, if the identity of the caller of the associated method call is incorporated in the future identity (as indicated in the operational semantics), it could in principle be possible to extract this at runtime. But the object creating the value would still in general be unknown. For instance, in the case of first-class futures, a caller may create a future corresponding to a method call on *o* and pass the future reference as a parameter to other objects. When the future is resolved, they obtain its value from the future object without knowing who has created this value. In fact, for a future received as a parameter there is in general no available static or dynamic information about the creator. It is not known where the future value comes from, who has generated it, or how fresh it is. This opens up for third party information with indirect/implicit handling of private information. An object may implicitly reveal futures with private information. However, dynamic information could be provided by the addition of language attributes. Thus with some overhead dynamic checking would be possible.

Static information flow analysis of the secrecy level of first-class futures is therefore imprecise. In order to have secure information flow for first-class futures, dynamic checking is required, which is expensive compared to static analysis. Class-wise information flow analysis has been suggested for a future-free version of Creol [26]. The approach is based on static declaration of secrecy levels for each input parameter and result value of a method. This would be difficult when allowing futures as parameters, because the set of external calls that may result in an actual parameter is not statically given, and because the calls in this set are not uniform with respect to secrecy levels. Static declaration of secrecy levels must consider the worst case possibility (i.e., the highest secrecy level), and this will easily lead to inflation of secrecy levels, something which is not desirable since then it would severely limit statically acceptable information passing and call-based interaction, or require dynamic checking.

In a paper by Attali et al. [1], secure information flow for the ASP language is provided by dynamically checking for unauthorized information flows. In their approach security levels are assigned to activities and transmitted data between these activities. For verification, dynamic checks are implemented at activity creation, requests, and replies. However, future references can be freely transmitted between activities because they do not hold any valuable information, they just hold addresses. But for updating a future and getting its value, the secrecy level of this transmission will be performed dynamically by security rules of a secure reply transmission.

Therefore static information flow checking of first-class futures is problematic, which means that it must be compensated by dynamic checking. In this sense, first-class futures do not promote security at the programming level.

4.6. Program reasoning

We compare the simplicity of rules for reasoning about method calls for future-free and future-based languages, and the usefulness of these rules in practice, considering two typical kinds of applications. The rules given below cover future-free and future-based languages, with or without suspension and the tail construct.

In the setting of active objects, compositional reasoning is typically based on the notion of communication traces, often called *histories* [33]. The specification of an object can then be given by specifying invariants by means of predicates over the local history, i.e., the history of events generated or observed by that object. Modular reasoning about classes can be based on *class invariants* referring to the fields of the class and the local history h. The class invariant considers a given object (this) and should hold whenever the object is idle, but may be violated when not idle [11]. In addition, one may use pre- and post-conditions to specify any additional properties of the methods in the class. The Hoare triple $\{P\}\ s\ \{Q\}$ specifies that if the program s is started in a state satisfying the precondition P then the final state will satisfy the postcondition Q, provided the program terminates, thereby expressing partial correctness [19]. For instance, an assignment statement $x := e$ satisfies the triple $\{Q^x_e\}x := e\{Q\}$, where the notation Q^x_e denotes textual substitution, replacing the expression e for all (free) occurrences of the variable x in Q. This assignment axiom holds in our language setting since there is no remote field access. This rule is left-constructive, expressing the weakest precondition. A class invariant I must hold after class initialization and be maintained by each method of the class, as well as between any suspension points. This allows us to rely on the invariant when a new method is started or re-started after suspension. The triple $\{I\}\ s\ \{I\}$ expresses that the statement list s maintains the invariant I.

Notation. For histories, we let h/S denote projection by means of a set S, i.e., the sub-sequence of all the events in the set S. And we let $h \cdot z$ denote the history h appended with the event z. Furthermore, $\leq$ denotes the sequence prefix relation, i.e., $h' \leq h$ expresses that h' is an initial part (prefix) of h, and $\subseteq$ denotes the subsequence relation, i.e., $h' \subseteq h$ expresses that h' is a subsequence of h.

The local history h of an object o is related to the global history, H, by the equation $h = H/\alpha(o)$ where $\alpha(o)$ is the alphabet of o (the set of events generated/observed by o), i.e., h is the projection of H by the events visible to o. Compositional reasoning is then possible by conjunction of the local invariants, replacing the local history h by $H/\alpha(o)$, together with a "compatibility" assertion stating the partial order between the events corresponding to meaningful execution order (i.e., an event reflecting a method invocation must come before the event reflecting the corresponding method completion), following the compositional principle of [33].

4.6.1. Rules for future-free calls

Reasoning rules for the future-free call constructs, including call with and without tail, are given in Figure 15, based on [25]. In this setting we consider two kinds of events: *call events* and *return events*. (Events reflecting the start and end of a method execution may be considered as well, but we here focus on simple call $$\{Q^{h}_{h\cdot(\mathsf{this}\rightarrow o.m(\overline{e}))}\}\ o!m(\overline{e})\ \{Q\}$$

sync-call $$\{o \neq \mathsf{this} \wedge \forall x' \,.\, Q^{x,h}_{x',h\cdot(\mathsf{this}\rightarrow o.m(\overline{e}))\cdot(\mathsf{this}\leftarrow o.m(\overline{e};x'))}\}\ x := o.m(\overline{e})\ \{Q\}$$

tail $$\frac{\{o' = o \wedge \overline{e}' = \overline{e} \wedge P\}\ s\ \{\forall x \,.\, Q^{h}_{h\cdot(\mathsf{this}\leftarrow o'.m(\overline{e}';x))}\}}{\{o \neq \mathsf{this} \wedge P^{h}_{h\cdot(\mathsf{this}\rightarrow o.m(\overline{e}))}\}\ x := o.m(\overline{e}) < s >\ \{Q\}}$$

await $$\{L \wedge I \wedge h = h'\}\ \mathtt{await}\ e\ \{L \wedge I \wedge h' \leq h \wedge e\}$$

Figure 15: Reasoning rules for call-related statements for future-free languages. Here L is a local condition (not referring to fields).

the treatment of calls.) The execution of the asynchronous call $o!m(\overline{e})$ by *this* object is represented by the call event $\mathit{this} \rightarrow o.m(\overline{e})$, and the observation by *this* object of the return value v generated by object o resulting from this call is represented by the return event $\mathit{this} \leftarrow o.m(\overline{e}; v)$. Thus, call statements have side-effects on the local history, a simple call appends the local history by a call event, and a synchronous call (with or without a tail) appends the local history by a call event as well as a return event. In the case of a synchronous call with a tail, the call and return events are added to the history before and after the tail, respectively.

Rule simple call expresses that the execution of the simple asynchronous call $o!m(\overline{e})$ has the effect of appending the corresponding call event to the local history. Rule sync-call expresses that a synchronous call $x := o.m(\overline{e})$ has the effect of appending the history with both the call event and the corresponding return event. The rules involving return events use universal quantification to reflect that the return values are under-specified in local reasoning, and primed variables are used to freeze pre-values of program variables that may change (when needed in a postcondition). Since self calls have a different semantics than remote synchronous calls, we add the restriction $o \neq \mathsf{this}$ in the precondition of a synchronous call to o with or without tail. The call rules are all left-constructive. In Rule tail, the return event is added when transforming the overall postcondition Q to a postcondition of the tail, and the call event is added when transforming the precondition of the tail to an overall precondition. The reasoning rules can be understood by letting the call to o have the side-effect

$$h := h \cdot (\mathsf{this} \rightarrow o.m(\overline{e}))$$

on the history, where $\overline{e}$ are the actual parameters, and letting the observation of a return have the side-effect

$$h := h \cdot (\mathsf{this} \leftarrow o.m(\overline{e}; v))$$

where v is the return value. More precisely, reasoning about the statement

$$\textsf{async-call} \qquad \{\forall f' \,.\, Q^{f,h}_{f',h\cdot(\textsf{this}\to o.m(f',\overline{e}))}\}\ f := o!m(\overline{e})\ \{Q\}$$

$$\textsf{get} \qquad \{\forall x' \,.\, Q^{x,h}_{x',h\cdot(\textsf{this}\leftarrow(e,x'))}\}\ x := \texttt{get}\ e\ \{Q\}$$

Figure 16: Hoare style rules for futures.

[**await**] $x := o.m(\overline{e}) < s >$ is the same as reasoning about the sequence

$$h := h \cdot (\textsf{this} \to o.m(\overline{e})); s; [\texttt{await}\ \ true;]\ x := \texttt{some}; h := h \cdot (\textsf{this} \leftarrow o.m(\overline{e}; x))$$

where **some** is a locally unknown value such that $\{\forall x \,.\, Q\}\ x := \texttt{some}\ \{Q\}$. The tail rule (without **await**) can be derived from this understanding. In fact, all call rules can be derived from this understanding when ignoring any optional parts (await/tail s) not used. And for languages supporting suspension, rules for await-call with or without tail can be derived as well, using the await rule.

The await rule says that the class invariant I will be maintained over suspension, even though fields may change. The history may only grow, as formalized by the postcondition $h' \leq h$ where h' is the history in the prestate. And since the local state is unchanged during suspension, any local condition L, not referring to fields, is also preserved. In addition the **await** condition e is guaranteed immediately after suspension. In the same way as a conditional await, an await-call statement must ensure the invariant at the beginning of suspension, and may assume the invariant immediately after suspension.

4.6.2. Rules for calls with futures

For languages with futures, the asynchronous call invocation is textually separated from the result query (the get statement). Thus in the analysis of a get statement, neither the method name (m) nor the callee are in general known at verification time, not even in the case of local futures. Therefore neither of these can be part of the event reflecting the completion of a get statement, in contrast to the case of future-free languages. Instead, the future identity must be included in the call and get events: We now let a call generate the call event this $\to o.m(u, \overline{e})$ where u is the future identity. And we let the *get event* $o \leftarrow (u, v)$ correspond to the observation by o of the value v of future u.

The reasoning rules for calls with futures are shown in Figure 16, based on [11]. Rule async-call is similar to Rule simple call, but the generated future identity is under-specified, which explains the quantifier. Rule get expresses that the history is appended with the corresponding get event, with an under-specified future value. Thus this setting gives rise to events where the connection between call and get events is made by means of the future identity, rather than the method name. The rules for futures are therefore more indirect and complex to apply in practice, as discussed next in Subsection 4.6.3.

4.6.3. Application of the rules

We will consider two typical cases of program reasoning, namely reasoning about method calls by means of input/output relations, and reasoning about the ordering of method calls, exemplified by verification of sequential ordering.

Reasoning about input/output relations. Reasoning about a method call often deals with the relationship between the input and output values of a method call. For a method body $s; \texttt{return}\, e$ with parameters $\overline{x}$ and return value e, one may specify and verify such a relationship, say $R(\overline{x}, e)$, by verifying the Hoare triple $\{true\}\, s\, \{R(\overline{x}, e)\}$ by ordinary Hoare analysis. However, for external calls with futures, it is not straight forward to use such facts, since the input and output to a call may not in general be known in a single state of the caller, due to the decoupling of the get statement form the invocation statement. But one may use the history. Consider the code $f := o!m(\overline{e}); ...; y := \texttt{get}\ f$ where o is an external object (different from this). This gives rise to the call event $(\mathsf{this} \rightarrow o.m(u, \overline{e}))$ where u is the future identity assigned to the future variable f, and the query gives rise to the get event $(\mathsf{this} \leftarrow (u, y))$ for some return value y. We may then state

$$(\exists u\, .\, (\mathsf{this} \rightarrow o.m(u, \overline{x})); (\mathsf{this} \leftarrow (u, y)) \subseteq h) \Rightarrow R(\overline{x}, y)$$

(for any values of $\overline{x}, y$) expressing that R holds for the input and output values (found in separate call and get events in h) for the same u. Therefore reasoning from this fact involves quantifiers.

In contrast, for a synchronous call $[\texttt{await}]\ y := o.m(\overline{e})$, where o is different from this, we may use the fact

$$(\mathsf{this} \leftarrow o.m(\overline{x}; y)) \in h \Rightarrow R(\overline{x}, y)$$

(for any values of $\overline{x}, y$), and we obtain $R(\overline{e}, y)$ in the post-state of the call since $(\mathsf{this} \leftarrow o.m(\overline{e}; y)) \in h$ obviously holds in the post-state of the call (for y not occurring in $\overline{e}$). Such return events appear for synchronous calls with or without a tail as well as for suspending calls with or without a tail. Thus reasoning about such calls from the history can be done in the same manner, without quantifiers.

Finally, we may look at simulated futures. In this case, events are generated from the creation of the future object o' and from the interaction with that object, typically through the *get* method. This gives one creation event, one call event, and one return event, of which the first (the creation event) and last are the ones needed to express an input/output relation R. Similarly to the case of first-class futures, the identity of the future object is arbitrary. This means that for a method m (with body as above, satisfying R), called by simulated futures, we may state

$$(\exists o'\, .\, (\mathsf{this} \rightarrow o'.new\ FUTURE_m(o, \overline{x})); (\mathsf{this} \leftarrow o'.get(; y)) \subseteq h) \Rightarrow R(\overline{x}, y)$$

where $\mathsf{this} \rightarrow o'.new\ C(\overline{x})$ denotes the creation event of an object o' of class C with class parameters $\overline{x}$. Thus the situation is quite similar to the case of futures and involves quantifiers. Hoare-style reasoning about method calls by means of input/output relations is therefore substantially more complicated for languages with futures than for future-free languages. And reasoning about simulated futures is comparable to reasoning with futures.

Reasoning about ordering of calls. A history-based invariant may for instance state that the remote m calls to o made by the current object are sequential, in the sense that a new m call can only be made when the results of the previous m calls to the same object o have been observed by the current object. To express this in the setting of future-free languages, we may write the class invariant

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow o.m\}) \ \texttt{follows}\ \#(h/\{\mathsf{this} \leftarrow o.m\})$$

where **`follows`** denotes sequential connection as defined by x **`follows`** $y \triangleq (x = y \vee x = y+1)$, and $\#$ denotes sequence length, and $\{\mathsf{this} \rightarrow o.m\}$ denotes the set of call events to o from this object for method m. (Alternatively, one could use equality since the invariant is only required to hold when the object is *idle*.)

In the case of languages for futures, the above invariant will be more complicated since the method name m is not visible in the event observed at a **`get`** statement. In this case the class invariant can be expressed by

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow o.m\}) \ \texttt{follows}\ \#(h/\{\mathsf{this} \leftarrow (u, \_) \mid u \in \mathit{futures}(h, o, m)\})$$

letting $\mathit{futures}(h, o, m)$ be defined as the set of futures generated by a call event of form $\mathsf{this} \rightarrow o.m$, and letting $\{\mathsf{this} \leftarrow (u, \_) \mid c\}$ be the set of all get events to this object with future identity u such that condition c is satisfied. Clearly, it is significantly harder to formulate the invariant in this case, and also reasoning becomes more complicated. Proving maintenance of the invariant by a call $y := o.m(\overline{e})$ is straight forward in the case of future-free languages. For instance

$$\{I\}\ y := o.m(\overline{e})\ \{I\}$$

reduces to $I \rightarrow \forall y' \,.\, I^{h}_{h\cdot(\mathsf{this} \rightarrow o.m(\overline{e}))\cdot(\mathsf{this} \leftarrow o.m(\overline{e};y'))}$, which is trivial for our invariant since m is explicit in the events. In the case of languages with futures, the reasoning becomes non-trivial, since the m is no longer explicit in the get event. One must reason indirectly through future identities across events, possibly also generated by different processes (on this object). This involves the u implicitly quantified through the conditional set expression.

For simulated futures we may use the invariant

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow \_.\mathit{new}\ \mathit{FUTURE}_m(o, \_)\}) \ \texttt{follows}$$
$$\#(h/\{\mathsf{this} \leftarrow o'.\mathit{get} \mid o' \in \mathit{futures}(h, o, m)\})$$

with *futures* defined almost as above. Reasoning in this case is similar to the case of languages with futures.

We may conclude that specifications and invariants are more indirect in languages with futures than in future-free ones. In particular it is harder to express relationships between input parameters and corresponding result values. Verification conditions get more complex to prove when they depend on get events, as demonstrated in the examples. We have also seen that the reasoning rules for local futures and first-class futures mainly have the same complexity. Furthermore reasoning about simulated futures is comparable to reasoning about futures. Thus reasoning with a future-free language is significantly simpler than reasoning about future-based calls and simulated futures.

5. Conclusion

Programming paradigms are essential in software development, especially for distributed systems since these affect large programming communities and a varied range of applications. Thus, investigation and comparison of different programming paradigms are valuable. We have focused on interaction paradigms for imperative programming languages based on the active object model, ignoring implicit futures. This model has gained popularity due to its modular semantics and natural support of concurrency and autonomy. Most of the recent active object languages adopt the future mechanism, rather than a two-way interaction paradigm. First-class futures give a possibility of sharing information and of partially avoiding blocking. An active object that has generated a call with a future may pass the future to a number of clients, and as long as it does not need the future value itself, it can continue to serve clients without being blocked. Waiting is then delegated to those clients that need the future value. This gives a programming style that avoids deadlocking and blocking in server objects, allowing the servers to be continuously responsive to client requests.

Cooperative scheduling for active objects is another recent mechanism that avoids blocking and allows passive waiting. Without use of futures, a server object can wait for the future value in a suspended (sleeping) process, avoiding blocking, and thereby be continuously responsive to client requests, even in the case that the server itself needs the value. In this case the clients will get the future value when it is available (when the suspended process is enabled), and thus waiting is also avoided in the clients as well. This means that there is less need for futures in languages with cooperative scheduling than in those without, even though cooperative scheduling does not provide sharing. This means that languages with first-class futures or cooperative scheduling more directly support propagation of method results without waiting, and are in this respect more expressive than those without neither of these two mechanisms.

Polling has the advantage of more fine-grained synchronization control. For instance one may await the results of a number of outstanding calls in a given order. However, polling may lead to more complex program structure. For instance to cover any ordering of the completions of n calls, one could end up with n! branches. In contrast, with cooperative scheduling there could be one suspended process for each of the outstanding calls. With first-class futures one could delegate to n other objects (by passing futures), so that each waits for one completion. Thus there is less need for polling in languages with cooperative scheduling or first-class futures.

Object-local futures offer more flexibility than method-local futures, but in the presence of cooperative scheduling one may use suspension to compensate. In particular single-read method-local futures give several advantages in the case of cooperative scheduling.

This leaves seven language paradigms for interaction as the most interesting: i) first-class futures and cooperative scheduling (FF+CS), ii) first-class futures without cooperative scheduling (FF), iii) method-local futures and cooperative scheduling (LF+CS), iv and v) local futures with polling (LF+P), with sub-

criteria	FF+CS	FF	LF+P	LF+CS	NF+CS	NF
expressiveness	+	0	0/0	0	0	−
efficiency	−	−	−/0	+	0	−
synt./sem. complexity	−	−	0/0	0	+	+
security aspects	−	−	0/+	+	+	+
static analysis	−	−	−/0	0	+	+
program reasoning	−	−	−/−	−	+	+

Figure 17: A simplified summary of the evaluation of the different paradigms. The case of local futures with polling (LF+P) is split in two subcases, object-local and method-local futures.

cases for object-local and method-local futures, vi) no futures and cooperative scheduling (NF+CS), and vii) no futures (NF). We have focused on these interaction paradigms and evaluated them along the chosen criteria. For a rough overview, some main points of the evaluation results are illustrated in Figure 17. Here + is better than 0, which is better than −.

With respect to expressiveness, we have seen that first-class futures can be expressed by means of explicit object generation (using predefined classes). Thus the need for built-in first-class futures is not so critical. Even though simulated futures are less flexible (at least syntactically) than built-in futures, they have the advantage that the cheaper (implementation-wise) alternatives are available by default when first-class futures are not strictly needed. Cooperative scheduling gives an expressiveness that cannot be simulated with (first-class) futures without use of dynamic object generation and blocking. Languages with built-in first-class and cooperative scheduling futures have the highest expressiveness (marked as "+" in Figure 17) whereas the ones without (only simulated ones) have somewhat less expressiveness. Future-free languages are less expressive than those with futures, but the addition of the tail construct results in similar expressiveness as languages with local futures. Polling allows non-blocking programming (while compromising program structure), and increases expressiveness and efficiency in the setting of local futures without cooperative scheduling.

We have seen that first-class futures require garbage collection, which is non-trivial in the case of distributed systems. And we have seen that first-class futures give raise to more messages than languages without. There is no uniformly best choice of update strategy for first-class futures [16]. Different implementation strategies may give more efficient garbage collection, but at the cost of more internal messaging. For IoT systems this could be critical. In contrast, cooperative scheduling adds to efficiency, and gives scheduling control. Local future languages may sometimes lead to less waiting than in future-free languages due to more expressive synchronization control.

We have also seen that first-class futures may cause difficulties with respect to information security. In particular information flow analysis is problematic. Furthermore, the notion of futures, even local futures, make program reasoning more complex than reasoning for future-free languages, by adding a level of indirectness in the reasoning. The considered examples show properties where simple reasoning in the future-free setting becomes non-trivial in the case of futures, due to additional quantifiers. Static analysis has similar problems, and for first-class futures this is in general more complex than for local futures. For several kinds of static reasoning it is necessary to detect the set of calls that corresponds to a given **get** statement. In the case of first-class futures this set cannot be detected in class-wise analysis.

In general the more constructs a language has, the more expressive it is, but on the negative side, the more complex it is wrt. syntax, semantics, security, and analysis. This is illustrated clearly in the (somewhat oversimplified) table in Figure 17. There is a trade-off between these different choices depending on the requirements in a given context, considering expressiveness, efficiency, and simplicity. The main benefits of first-class futures are the added flexibility and information sharing, some of which can be compensated by cooperative scheduling. For distributed IoT programs we have argued that first-class futures are less suited. This is also the case for information flow analysis. And if simplicity of program reasoning is a major concern, first class, and even local futures, raise non-trivial complications. In our treatment, the limitations of future-free programming have been reduced by the addition of delegation and a syntactic tail construct for calls. Consequently, future-free programming can be attractive in a number of settings.

We have focused on imperative languages for single-threaded active objects. The setting of multi-threaded active objects is attractive in that it may provide higher efficiency, but its semantics is more complex. To keep our framework simple, we have avoided the multi-threaded setting, as well as advanced coordination constructs for futures, such as asynchronous continuations associated with futures. With respect to our evaluation criteria, these constructs increase the expressiveness and efficiency of first-class futures.

Acknowledgment

We thank the reviewers for their insight and thorough feedback, which definitely has increased the quality of the paper.

This work was partially supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme of Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, and Norway.

References

[1] Isabelle Attali, Denis Caromel, Ludovic Henrio, and Felipe Luna Del Aguila. Secured information flow for asynchronous sequential processes. *Electronic Notes in Theoretical Computer Science*, 180(1):17–34, 2007.

[2] H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.

[3] 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 Computing Surveys*, 50(5):76, 2017.

[4] 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 *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 9104 of Lecture Notes in Computer Science, pages 1–56. Springer, 2015.

[5] Denis Caromel, Christian Delbé, Alexandre Di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology*, 12, issue 1:16, 2006.

[6] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components*. Springer, 2005.

[7] Ole-Johan Dahl and Olaf Owe. Formal methods and the RM-ODP. Research Report 261, Dept. of informatics, University of Oslo, Norway, 1998.

[8] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of Lecture Notes in Computer Science, pages 230–254. Springer, 2006.

[9] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[10] Crystal Chang Din, Johan Dovland, and Olaf Owe. Compositional reasoning about shared futures. In *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2012.

[11] Crystal Chang Din and Olaf Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(5-6):360–383, 2014.

[12] F Durán, S Eker, P Lincoln, N Martí-Oliet, J Meseguer, and C Talcott. All about maude: A high-performance logical framework. *Lecture Notes in Computer Science*, 4350, 2007.

[13] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In *International Conference on Coordination Languages and Models*, pages 101–120. Springer, 2016.

[14] Reiner Hähnle. The abstract behavioral specification language: a tutorial introduction. In *International Symposium on Formal Methods for Components and Objects (FMCO 2012)*, volume 7866 of Lecture Notes in Computer Science, pages 1–37. Springer, 2012.

[15] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[16] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. First class futures: Specification and implementation of update strategies. In *Euro-Par Workshops*, pages 295–303. Springer, 2010.

[17] Ludovic Henrio and Justine Rochas. Multiactive objects and their applications. *Logical Methods in Computer Science*, Volume 13, Issue 4:1–41, November 2017.

[18] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[19] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[20] Einar Broch Johnsen, Jasmin Christian Blanchette, Marcel Kyas, and Olaf Owe. Intra-object versus inter-object: Concurrency and reasoning in creol. *Electronic Notes in Theoretical Computer Science*, 243:89–103, 2009. Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08).

[21] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of Lecture Notes in Computer Science, pages 142–164. Springer, 2011.

[22] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software & Systems Modeling*, 6(1):39–58, 2007.

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

[24] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[25] Olaf Owe. Verifiable programming of object-oriented and distributed systems. In Luigia Petre and Emil Sekerinski, editors, *From Action Systems to Distributed Systems - The Refinement Approach.*, pages 61–79. Chapman and Hall/CRC, 2016.

[26] Olaf Owe and Toktam Ramezanifarkhani. Confidentiality of interactions in concurrent object-oriented systems. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 10436 of Lecture Notes in Computer Science, pages 19–34. Springer, 2017.

[27] Olaf Owe and Isabelle Ryl. OUN: A formalism for open, object oriented, distributed systems. Research Report 270, Dept. of informatics, University of Oslo, Norway, August 1999.

[28] Niloofar Razavi, Razieh Behjati, Hamideh Sabouri, Ehsan Khamespanah, Amin Shali, and Marjan Sirjani. Sysfier: Actor-based formal verification of systemc. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):19, 2010.

[29] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, 6183:275–299, 2010.

[30] Marjan Sirjani, Frank de Boer, Ali Movaghar, and Amin Shali. Extended Rebeca: A component-based actor language with synchronous message passing. In *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 212–221. IEEE, 2005.

[31] Marjan Sirjani, Ali Movaghar, and Mohammad Reza Mousavi. Compositional verification of an object-based model for reactive systems. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford, UK*, pages 114–118. Citeseer, 2001.

[32] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.

[33] Neelam Soundarajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6:646–662, 1984.

[34] Akinori Yonezawa, editor. *ABCL: An Object-oriented Concurrent System.* MIT Press, Cambridge, MA, USA, 1990.