Transparent First-class Futures and Distributed Components

Antonio Cansado, Ludovic Henrio, Eric Madelaine

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
Email :First.Last@sophia.inria.fr

Abstract

Futures are special kind of values that allow the synchronisation of different processes. Futures are in fact identifiers for promised results of function calls that are still awaited. When the result is necessary for the computation, the process is blocked until the result is returned. We are interested in this paper in *transparent first-class* futures, and their use within distributed components. We say that futures are *transparent* if the result is automatically and implicitly awaited upon the first access to the value; and that futures are *first-class* if they can be transmitted between components as usual objects. Thus, because of the difficulty to identify future objects, analysing the behaviour of components using first-class transparent futures is challenging. This paper contributes with first a static representation for futures, second a means to detect local deadlocks in a component system with first class futures, and finally extensions to interface definitions in order to avoid such deadlocks.

Keywords: Hierarchical components, distributed asynchronous components, formal verification, behavioural specification, model-checking, specification language.

1 Introduction

This paper provides a model for reasoning about futures in the context of distributed components. We define here a framework allowing us to find which components can be blocked on an access to a future, and extend the specification of component interfaces in order to avoid some of these blocked states.

Our approach consists of specifying statically a behavioural model for distributed systems with futures, in order to apply model-checking techniques on it. This approach can relate to [14,10], but in this paper we focus on the modeling of futures that were not taken into account in previous work in this domain. In our previous works [3,2] we gave behavioural models for active objects and asynchronous distributed components such as the GCM (Grid Component Model [4]), however, futures were local, i.e. they were not sent between activities (we call *activity* a unit of distribution). The behavioural models, as in this paper, are based on an intermediate language that we call *Parameterized Networks of Synchronised Automata (pNets)* [2].

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The remainder of this section reviews related works on futures, defines the context of our work, and our contribution. In Section 2 we provide an abstraction suitable for static analysis of programs with futures, as well as behavioural models for futures. In Section 3 we apply the behavioural model to an example and show some properties that can be checked. Section 4 builds on definitions useful for detecting blocked components. Finally, in Section 5 we suggest an extension for the definition of component interfaces that can prevent some deadlocks.

1.1 Futures

Futures [13,18,16] provide synchronisation for concurrent or parallel languages. A future is an "object" that can be filled or not. Accessing a future blocks if the future is not filled, and returns the encapsulated object otherwise. Explicit futures are typed: a (static) type "Future" exists and futures are statically identified as the objects with this type, they can only be accessed by a `getValue` method. In this paper we are rather interested in *transparent* futures. Transparent futures are accessed like the object they encapsulate, so futures and non-future objects are manipulated in the same way. Futures have some kind of proxy that automatically blocks the program when accessing the object if the future is not filled. This renders a data-driven synchronisation in a mechanism called *wait-by-necessity*.

The first question is: "when are futures created?". The answer depends on the programming language. Generally, futures are created by an explicit construct which delegates a thread for computing the future value, this construct is named `future` in Multilisp [13,11], and `thread` in [16]. In AmbientTalk [9], ProActive [5], and ASP [6], futures are created automatically upon a remote method invocation. In ProActive and ASP, there is no syntactic difference between a local method call (which may or may not create a future), and a remote method call (which creates a future). This means futures are implicit and their creation depend on the data-flow; synchronisation is also implicit and occurs on strict access to a future.

Another question is: "Which are the blocking operations on a future?". Whereas in a local setting most of the languages agree on the definition of a strict access to an object, in a distributed setting things get more complex. The question above boils down to: "Is transmitting a future to a remote object a strict operation?". In ASP and ProActive, we consider that futures can be transmitted between remote activities because we proved that this property had no influence on the possible execution paths, except that it can avoid some dead-locks [6]. We call such futures *first class futures* because they can be manipulated as first class objects.

As an interesting related work, [8] provides a language with futures that features "uniform multi-active objects": roughly, each method invocation is asynchronous because each object is active; each object has several current threads, but only one is active at each moment. However, their futures are explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties on such multi-active objects with futures.

Our point of view in this paper is to build a behavioural model for futures *à la* ASP calculus, but more generally the proposed model applies to any kind of transparent first-class futures featuring wait-by-necessity mechanism.

1.2 Components and Futures

Components are software entities with interfaces (or ports); those interfaces are connected together by bindings. In GCM, there are two kinds of components: *primitive components* that implement some behaviour in Java, and *composite components* that compose other components. The composites are in fact dispatchers of services: the composite dispatches the received requests to the bound interfaces.

If communications occurring over the bindings are synchronous, then the interfaces can be accessed as usual objects, having methods with parameters and a return type. When components are connected asynchronously, one must find a way to create a channel for the objects returned by the components. Futures can be used as identifier of the asynchronous invocations over components. Indeed, futures provide some kind of transparent channels that correspond to the original bindings, but taken in the opposite direction: from the server to the client.

But components and futures get more related when considering static analysis. Indeed, in an asynchronous component model like the GCM, only invocations on a component create a future. Thus, the components allow the static identification of future creation points, and thus a finer static analysis.

1.3 Components as an Abstraction for Distribution

Components relieve us from a difficult analysis task: in a distributed object-oriented language with implicit futures, it is difficult to identify the communication and the creation points of futures. Indeed, asynchronous method calls are syntactically similar to synchronous ones, and distinguishing one from the other can only be the result of a static analysis step which is by nature imprecise, consequently identifying the points where futures are created is also difficult. In a distributed component model like the GCM, however, the only method invocations that are asynchronous are the ones performed on interfaces. The topology of distribution and communication is directly given by the component structure.

Unfortunately, although the component model provides a good abstraction for distribution and specifies which calls are asynchronous, the flow of futures is still hard to approximate. In other words, the component abstraction tells us where futures are created but not where they can go. The dynamic and transparent nature of futures implies that each result and each parameter of an invocation may contain a future; thus the only safe assumption for parameters and results is that any object received can be a future, and every field of this object can itself be a future. This leads to a very imprecise approximation of the synchronisation in the system; this over-approximation can always be improved by static analysis (when the system is closed), or by specification, as illustrated in Section 5.

1.4 Contribution

Transparent first-class futures provide a natural and efficient data-flow synchronisation where a result is awaited only when it is necessary. However, providing a model of programs using transparent first order futures is challenging. The contribution of this paper is first to give a static representation of transparent first-class

futures, second to characterise how access to futures can block components indefinitely, third to use the previous results to identify local deadlocks, and finally extend the definition of interfaces to avoid some blocked states.

2 A Static Representation for Futures

The objective of this section is to give a behavioural model for transparent first-class futures, this model is intended at the static verification of the behaviour of components. We assume that the accesses to the component interfaces and the creation point of futures are given in the functional behaviour of the component (Body). We start this section by a brief definition of the pNets model, and of its (static) graphical representation on which we build our models.

2.1 Informal Description of pNets

The **pNets** model, formally defined in [2], is a generalisation of **Nets** [1]. It synchronises a (potentially infinite) number of processes by means of N-ary synchronisation vectors. The parameters set a symbolic representation of the system.

A pNets takes the form of a hierarchy of processes; each process encodes a particular Sort of the pNet. The Sort of a pNet can be filled with a parameterized LTS (pLTS) satisfying a Sort inclusion condition, or with another pNet.

In this paper we use a static subset of pNets in which synchronisation vectors don't change. A pNet is depicted by a set of boxes, and their synchronisations are given by arrows that express the synchronisation vectors; the direction of the arrow is an abstraction of the data-flow. For N-ary synchronisation, we use a synchronisation vector with an eclipse in the middle.

The pLTSs are represented by states (circles), and transitions (arrows). The transitions encode the actions that a process can perform in a given state.

A label with !*act* and ?*act* denote actions of emition and reception of *act* resp. An action *act* is a visible action without synchronisation; however, this action may be the result of synchronising other actions within inner pNets.

Moreover, within the behavioural model we adopt the following notation:

- `call(`f_{id}`,M(args))` is a method call `M`, with parameters `args`, which result should update the future f_{id}.
- `response(`f_{id}`,val)` is the result of a method call; it updates the future f_{id} with the value `val`.
- `forward(`f_{id}`,val)` is the message forwarding the value `val` of the future f_{id}.
- `getValue(`f_{id}`,val)` is the access to the future f_{id}; the body accesses the future f_{id} and receives from the proxy the value `val`.
- `serve(M)` is the request to serve a method `M` from the queue.
- `pNets(C)` is the behavioural model of component `C`.
- `Proxy(`f_{id}`)` is the proxy dealing with the future f_{id}.
- `Body` is the (user) functional behaviour of the component.
- `Queue` is the request queue of the component.

2.2 Representing Futures

In the examples below, we will use a Java-like language *à la* ProActive, where creation of futures are method calls on some required interfaces (named `itf` here), i.e. all future creations are of the form `f=itf.foo()`, resulting in a future stored in the variable `f`. We call *future update* the operation consisting in replacing a reference to a future by the value that has been calculated for it.

The representation of a future must allow the contained object to be accessed, i.e. to synchronise futures. We call `waitFor` the primitive allowing the update of a future to be awaited (this primitive has also been named `touch` or `get` [11]). When futures are transparent, this waiting operation is automatically performed upon an access to the content of the future. We describe in this section what behavioural model can be created for this kind of futures. For the moment, we consider that futures cannot be passed between remote entities, and thus the future is necessarily accessed by the same entity that created it (at another point of the execution).

The objective of this part is already to be able to provide a model for the following piece of code:

```
f=itf.foo();          // creation of a future
if (bool) f.bar1();   // wait-by-necessity if bool is true
f.bar2();             // wait-by-necessity only if bool is false
```

In here, if `f.bar1()` is executed, then `f` must be filled; in this case `f.bar2()` will be necessarily non-blocking. Otherwise, `f.bar2()` may or may not be blocking depending if the future `f` is already filled by the time the call is performed. Note that it is much simpler when futures are explicit, i.e. if futures are typed.

In this work we formalise the abstract domain of a future. The previous example shows that futures are filled transparently at any time. Thus, whenever it is not statically decidable whether an object is a future or a value, it must be assumed as a future. This is an over-approximation that will, at least, include all possible synchronisations a variable may trigger. Therefore, static analysis of a program with futures requires the set of abstract values to be multiplied by two.

Indeed, statically each variable is either known to contain a value which *is not a future*, or, equivalently, a filled future, ranging in the domain of the usual static domain for values; or the variable *may be a future*, and when the future will be filled its value will range in the domain of the usual static domain for values. Note that an object that is not a future is semantically equivalent to a filled future. In abstract interpretation [7] it would be easy to construct a lattice for this new abstract domain: suppose without futures, the abstract domain is a lattice $(D, \prec)$, then the new abstract domain taking futures into account is the *lattice* $D' = D \cup \{fut(a) | a \in D\}$ equipped with the order $\prec'$ built such that if $a \prec b$, then $a \prec' b$, $a \prec' fut(b)$, and $fut(a) \prec' fut(b)$. Indeed the abstract value a gives more information than $fut(a)$. To summarise, statically, the value for our objects are either "filled" or "potentially non-filled"; these abstract values are composed with the usual abstract values required for the analysis.

In the ProActive middleware, the example above creates a *proxy* in the first line, and all calls to the future stored in `f` would go through the proxy object, leading, if

necessary, to a wait-by-necessity. For our model, the idea is the same: the variables have the "classic" static abstract domain, and the augmented lattice is taken into account by an additional automaton with the behaviour of a proxy. Initially, the proxy is in an empty state where the object can only be filled with a value, so any access to the variable will be blocking. In general, two instances of the same method call have two different futures, so the proxies are parameterized by the instance of method call. In Fig. 1 we show a first model on how two components communicate. The action `call(`f`,M(args))` puts the request in the server's queue, and initialises the local proxy. The call contains the *identifier* of the future to be updated, f. Once computed, the value of f is updated (`response(`f`,val)`).

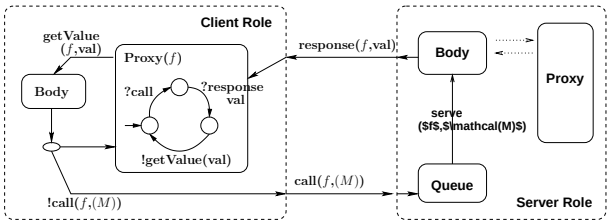


Fig. 1. Communication between two Active Objects

2.3 A Model for Transparent First Class Futures

We now extend the previous model to allow futures to be transmitted in a non-blocking manner. Futures can be transmitted in the parameters of a method call, or in the return value of a method call. Because of that, a future in an activity may have been created locally or by a third-party. In both cases, the activity is aware of the future *identifier*. The references of futures known by a component are local. Only when synchronising the components the references must match the data-flow. In pNets, synchronisation vectors allow us to synchronise different labels which we use to link the future references. This technique allows us to create the behavioural model for each component independently.

On the practical side, different future update strategies can be designed for propagating the values that should replace future objects. Despite having differences in performance, the update policies have equivalent behaviour, proved using ASP in [6]. This leaves freedom to choose any update policy. In this paper we use this result applied in the behavioural model.

Let $\sigma_{\mathtt{C}}$ be a valid execution on component `C`, `pNets(C)` the behavioural model of `C`, and f_{id} a future, then the model is built such that:

Property 1 *if* `getValue(`f_{id}`,val)` *in* σ_C*, then* `Proxy(`f_{id}`)` *is in* `pNets(C)`

As a consequence, the model has a proxy dealing with every future a component may receive. Due to imprecision of the abstraction, the component may even have proxies for futures that would never exist at run-time. However, any synchronisation is considered within the model.

Property 2 *if the value of* f_{id} *is computed, then all proxies of* f_{id} *are updated eventually*

Property 2 is true even for proxies that don't exist at run-time. The property is

guaranteed by construction: (i) *the proxy that creates* f_{id} initially synchronises with the remote method call. The proxy then waits for the result (value of f_{id}). When the value of f_{id} is updated, the proxy forwards the value of f_{id} to all components to which the local component sent the reference f_{id}. (ii) *all other proxies of* f_{id} are initially in a state in which they are ready to receive the value of f_{id}; this guarantees they will also be able to be updated. When the proxy is updated, it forwards the value of f_{id} to all components to which the local component sent the reference f_{id}. In fact, a proxy forwards the value of a future to all components it has sent the reference to synchronously. Therefore, each proxy only needs one port "`forward`" for each future, independently of the number of components to which it sent the reference.

Property 3 *the update of proxies that do not exist at run-time has no influence in the behavioural model*

Depending on the data-flow, some components will receive the value of f_{id}, though the reference f_{id} was not transmitted. In this case, the reference f_{id} is also unknown to the given component, and thus the content of the future is innaccessible, i.e. the business part of the component is not affected.

Sending a future as a method call parameter

In Fig. 2, the `Client` performs a method call M_1 on `Server-A`, and creates a `Proxy(`f`)` for dealing with the result. Then the `Client` sends the future to a third activity (`Server-B`) in the parameter of the method $M_2(f)$. From `Server-B`'s point of view, there is no way of knowing if a parameter is (or contains) a future, so every parameter in a method call must be considered as a potential future. `Server-B` includes, therefore, a proxy for dealing with the parameter f of the method call M_2. For the sake of comprehension, however, in the figure the identifiers for futures already match the data-flow.

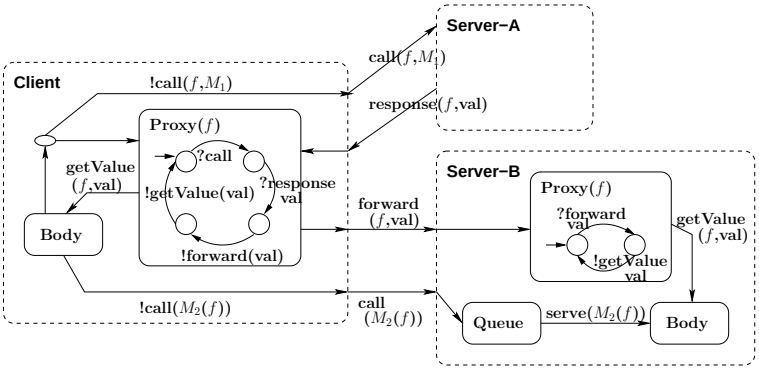


Fig. 2. Transmitting a future as method call parameter

Retransmit a future received as a method call parameter

The previous example is extended such that `Server-B` transmits the future f to `Server-C`. This is partially depicted in Fig. 3. The proxy in `Server-B`, after receiving the value of the future (`?forward(val)`), forwards the value to the components it has sent the future reference.

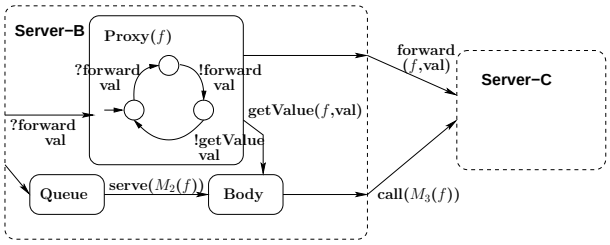


Fig. 3. Retransmitting a future as method call parameter

Returning a future

In Fig. 4 an activity (`Server-B`) creates a future f_2 and then transmits f_2 to the `Client` within the result of the method call $M_1(args)$. The behavioural model is slightly different to the one in ASP: instead of returning a future, there is a proxy on the server put in charge of forwarding the concrete value once it is known; no value or future is sent to the `Client` in the meanwhile. Using this mechanism, the behavioural model of the `Client` is the same no matter whether `Server-B` returns a value or a future. Moreover, `Client` remains as usual; the result of the method call $M_1(args)$ has a proxy `Proxy(`f_1`)` dealing with the result. It is up to the proxies of the `Client` and the `Server-B` to synchronise in order to match the expected behaviour. Concretely, the action with the response to the `Client` (`response(`f_1`,val)`) is synchronised with the forward action (`forward(`f_2`,val)`) of the `Proxy(`f_2`)`; it will then update the `Proxy(`f_1`)`. If the `Client` accesses the future, then it synchronises with its local proxy, `Proxy(`f_1`)`.

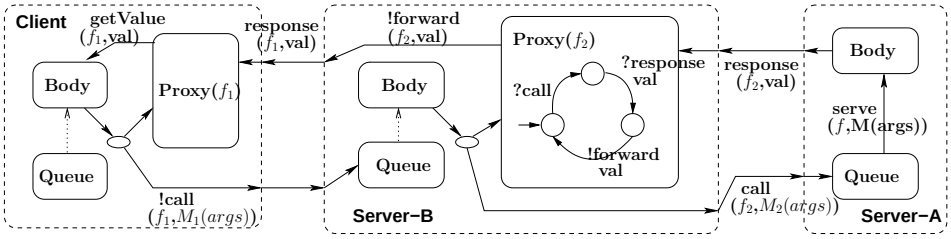


Fig. 4. Transmitting a future as a result for a method call

3 Illustrative Example

Consider a component system like in Fig. 5. It contains a component `A` that requests some services of `B`, and stores the return value in a variable f. Component `A` does not access the return value f immediately; instead, it forwards f to the component `E`, and possibly forwards f to the component `F`. Finally, `A` accesses f. Component `B` is a composite component that wraps a primitive component `C`. `C`, when serving the method `foo()`, requests a service to `D` by means of its wrapper, `B`, and returns.

In GCM/ProActive, this would instantiate 6 active objects; one per primitive component (`A`, `C`, `D`, `E`, `F`), plus one per composite component (`B`). The active object for `B` mediates services: requests coming from the composite's server interfaces are dispatched to a subcomponent, requests coming from its subcomponents client interfaces are dispatched towards an external component. For that it makes extensive use of first-class futures; it serves a request, performs a remote method call, creates

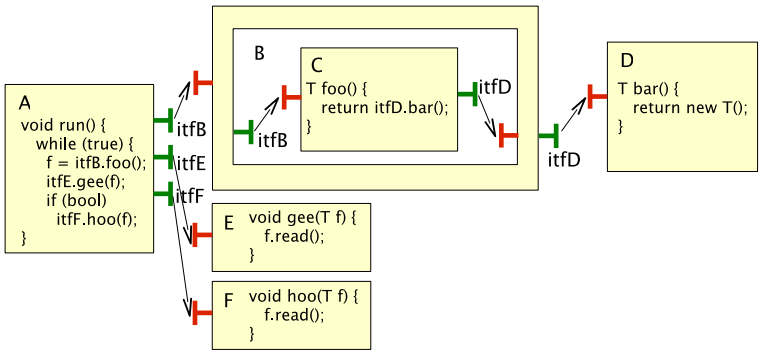


Fig. 5. Going through a composite thanks to first-class futures in ProActive

a future for holding the result, and then sends back the future to the caller. In other words, `B` *delegates* the requests it receives to components `C` and `D`, returning the future corresponding to the delegated method call.

3.1 Behavioural Model

Fig. 6 shows the model created for the system above. Components `E` and `F` have similar behaviours. Components `B` and `C` are synthesised by the pNets model `BC` depicted in Fig. 7 (which is as well a model for a composite component). In this example, we index each future by the name of the component that created it.

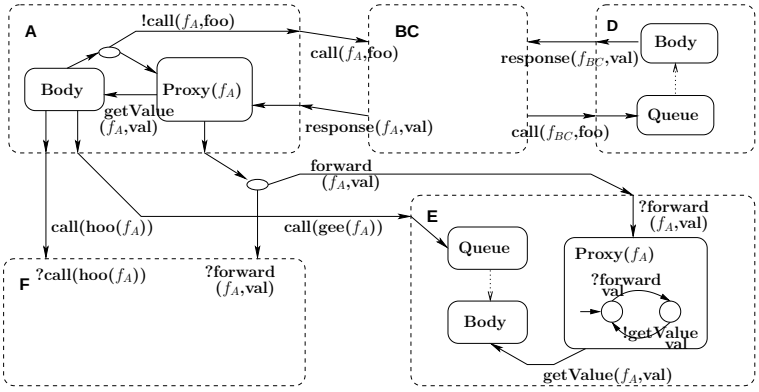
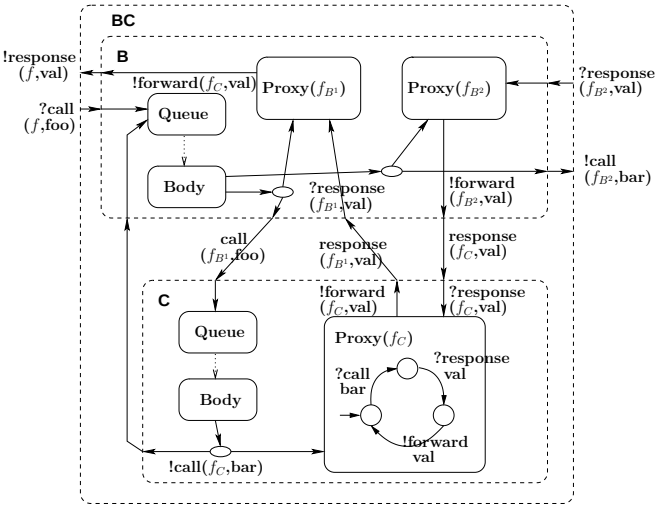


Fig. 6. pNets model of Fig. 5

In the pNets model of `A`, futures are forwarded to several activities; a future is sent as parameter of the method calls to `E` and `F` in `call(gee(`f_A`))` and `call(hoo(`f_A`))` resp. A proxy is created in each callee with the identifier (f_A) matching the proxy of the caller, i.e. `Proxy(`f_A`)`. `Proxy(`f_A`)` in `pNets(A)`, after receiving the concrete value, will forward the value to both activities `E` and `F`. This is seen as an action `forward(`f_A`,val)`. As a remark, the update of `Proxy(`f_A`)` in `F` is done no matter whether the component is called or not, however if the call is never performed the proxy is unreachable (its identifier is unknown).

Fig. 7 shows the behaviour of components `B` and `C`. Component `B` creates the proxies `Proxy(`f_{B^1}`)` and `Proxy(`f_{B^2}`)` for the calls `foo` and `bar` resp. `B` does not access the proxies, so the responses of the calls are forwarded directly by the proxies. The same models applies for component `C`. It creates a proxy `Proxy(`f_C`)` when



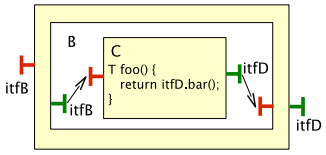


Fig. 8. Components `B` and `C`

Fig. 7. pNets model of components `B` and `C`

calling `bar`. `C` returns the future f_C, so `Proxy(`f_C`)` is the one forwarding the value it receives as a response to `B`.

3.2 Properties

In terms of behaviour, the value of f has no impact on the control flow, thus it is abstracted to a single abstract representative *dot*. It is the proxy that takes care of the abstract values *filled* and *non-filled*, meaning that we only care if the future has been filled or not and when it is accessed. We used the CADP [12] toolbox for generating the state-space and for the verification; the complete LTS for the system has: 12 labels, 575 states and 1451 transitions; when minimised using branching bisimulation 52 states and 83 transitions remain. Some properties can be found using alternation-free μ-calculus formulas [15]:

System is deadlock-free. As the program never terminates, we proved in CADP that, on the global-state space, every state has at least one successor.

All futures are necessarily updated. This is proved by stating that the call on `itfB.foo()` in component `A` will update all futures within a finite number of actions. In pNets, this is (see Fig. 9): starting in a state where `call(`f_A`,foo)` is performed, all leading traces will perform the future updates along the transmitting chain. More precisely, as no future is returned until a real value is known, when `D` computes the value, the components of the chain (`D`, `B`, and `C`) reply. Those `response` messages follow all the chain leading to `A`. Finally, `A` forwards the value to `E` and `F` (`forward(`f_A`,val)`).

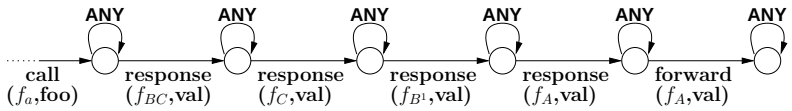


Fig. 9. Automaton representing the traces where futures are updated

System deadlocks if the composite does not support first-class futures. Suppose that the programming language does not support the transmission of futures, which implies that a method call must return a value (if any). Fig. 10

shows a modified version of the composite `B` with this behaviour. When the component `B` receives a request `?call(`f`,foo)`, the `Body` of `B` should: call the component `C` (action `!call(`f_{B^1}`,foo)`), access the return value (action `getValue(`f_{B^1}`,val)`), and then return the value of f_{B^1} (action `!response(`f`,val)`). The value of f_{B^1} is computed by component `C` on a service that must go through component `B`. Therefore, this value will never get computed as component `B` is blocked synchronising on `getValue(`f_{B^1}`,val)`. Such a scenario systematically results in deadlocks.

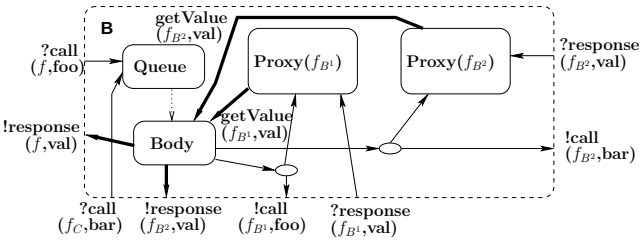


Fig. 10. pNets model of a composite without first-class futures

System deadlocks if `itfB.foo()` is synchronous The deadlock is similar to the previous one; if `foo()` is synchronous, then this call blocks component `B` until the result is known. What it means is that a synchronous call cannot trigger a flow that goes through a composite twice. This is a common pitfall for inexperienced programmers with GCM/ProActive that we can fortunately detect in our models.

4 Identifying Blocked Components

This section shows how to detect whether there are components blocked infinitely on a future access. We investigate definitions and properties adequate for this purpose based on ASP.

It is easier to start with the example of Fig. 11. A *Client* queries for some data. This data is properly formatted by the *QueryManager* and then forwarded to the *Database*. Once the *Client* creates the future d, it inserts a new entry into the table t with data from d; this is a method call performed directly towards the *Database*. The system may deadlock, though, due to a race condition on access to the *Database*. If the *Client* accesses the *Database* before the *QueryManager* does, the *Database* will access the future d – thus block –, but d will never be updated because the *Database* itself must update this future. The behavioural model of the previous section is enough to detect this problem.

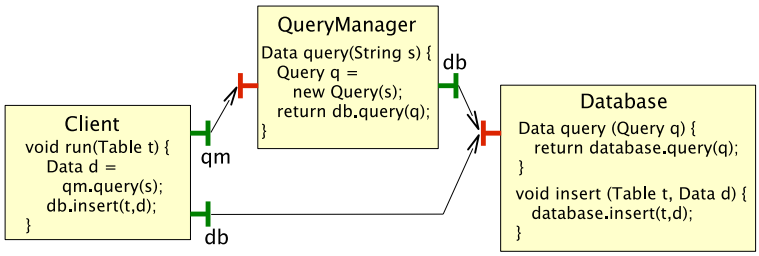
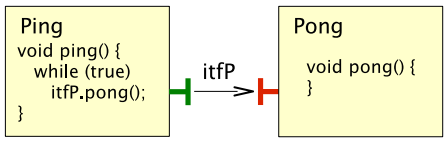


Fig. 11. Race condition in GCM / ProActive



Now, suppose the database example runs in parallel with two components that run continuously as the figure on the right. Using the same analysis over the complete system, no deadlock is found: indeed, some part of the system is constantly doing some work, i.e., in the global state-space every state has at least one transition. What we need is a finer grain definition of blocked component.

In ASP, synchronisations happen upon access to a future and when serving a request from the request queue. In the following we consider components that serve requests in a FIFO order, and thus no synchronisation on a request is made. Therefore, all deadlocks in a system must be related to access to a future. More precisely, there must be at least a future that is accessed and that is never updated. This gives us a starting point for defining what is a (non)-blocking future.

We refine the behavioural model in order to observe the accesses to futures in detail: first, there is a visible, non-synchronised action `waitFor(`f`)` signalling that a component wants to synchronise on the content of a future; and then a synchronised action `getValue(`f`,val)` where the component effectively retrieves the content of the future.

Unfortunately, due to an unfair scheduler, a subsystem (e.g. the Ping-Pong) could interact indefinitely while some components never progress. In this case, once the action `waitFor(`f`)` is performed, the action `getValue(`f`,val)` is reachable but not inevitable. Therefore, we impose some kind of fairness in traces. We use the definition of *fair reachability of predicates* as given by Queille and Sifakis [17].

Definition 4.1 A sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it.

Finally, we are able to define a *non-blocking future* and a *non-blocking component*.

Definition 4.2 A future f is *non-blocking* iff, under the fairness hypothesis, if each time the action `waitFor(`f`)` is performed, then the action `getValue(`f`,val)` is eventually reached.

Definition 4.3 A distributed component is *non-blocking* iff every future it accesses is *non-blocking*.

Moreover, if a distributed component system is *non-blocking*, and synchronisations are only due to future access, then the system is deadlock-free. In other words, if the system deadlocks and synchronisations are only due to access to futures, then there is at least one component blocked waiting for a future.

The main advantage of our approach is that it can be encoded in a model-checker, and thus we can ensure that every needed future reference is updated; in other words the program will have the expected behaviour: *all the object accesses of the program will occur*.

5 Extending the Interface Definition

By switching from an object-oriented to a component-oriented design, we make the application topology and dependencies explicit because: (i) every component contains a single thread; (ii) all method invocations are restricted to calls on client interfaces; and (iii) all future creation points are restricted to results of these method calls on client interfaces.

This removes some of the imprecision of the static analysis. Nevertheless, a source of ambiguity remains in open environments: a parameter (or any subfield) received in a method call may be either a future or a value due to transparency of first-class futures. This section suggests an extension to the Interface Description Language (IDL) to improve the precision of analysis and specification, we also explain how this extension prevents the occurrence of some deadlocks.

5.1 Principles

In order to be safe, the behavioural model must be an over-approximation of the implementation, including a proxy not only for futures, but also for variables or parameters which *may* be a future. Such imprecision is due to the undecidable nature of static analysis, and to the transparent nature of futures.

Moreover, for the database example of Fig. 11, one would like to offer means to correct the deadlock. For this, one can enforce further synchronisation on the *Client* side in order to guarantee that the *Database* always receives a value instead of a future. Up to now the only way to ensure such a synchronisation is to insert a call to the `waitFor()` primitive within the code of the *Client*. Nevertheless, from the server side, i.e., the *Database*, one does not know this information. Thus, the behavioural model for the *Database* still expects a future; the unneeded traces will only be pruned when synchronised with the environment.

The IDL used in the GCM specifies the interface signatures, but is insufficient to deal with transparent first-class futures. Based on the interface signature, one does not know whether method parameters are futures or not. Moreover, there is no way of controlling which parameters cannot be futures. Typing futures would solve the issue, however, we would lose all the good properties shown in Section 2.2. One way is to specify within the IDL which parameters (or fields) cannot be futures (i.e. marking them as *strict value*); the other parameters are allowed to be futures or not. Note that this is less restrictive than typing because some parameters can still be either a value or a future.

In an open system this information cannot be inferred by static analysis. It is a contract on futures that affects both client and server: client interfaces must ensure that method parameters match the interface specification; server interfaces assume – and may test – that method parameters agree with the interface specification. The contract also decreases the non-determinism in the server behaviour.

It is true that by the use of strict parameters there is less concurrency; components may enforce further synchronisations before performing remote invocations. On the other hand, behavioural models are more precise and closer to real executions; the programmer can specify parameters that are known to be non-futures.

5.2 Interface Specification

The difficulty is finding, statically, a proper abstraction for the parameter structure. In theory, every subfield of every parameter may be a future. Therefore a static representation of arbitrary types is impractical. Here we suggest a relatively precise approximation; marking a field as *strict value*, recursively, means that all its subfields (known at runtime during serialisation) are *strict values* as well. Similarly, not marking a field implicitly means that, recursively, all its subfields (except the marked ones) *may be futures*.

In the example of Fig. 11, an easy solution to the deadlock mentioned before is to force value-passing of d. Based on Java 1.5 Annotations the specification of the interface `DB` would look like:

```
interface DB {
  Data query(Query q);
  void insert(Table t,
      @StrictValue Data d);
}
```

On the practical side, if d is still a *non-filled* future by the time the method `insert(`t`,` d`)` is invoked, the invocation is halted until the future is updated. This way, the system is guaranteed to be *non-blocking*.

An implementation in ProActive would modify the *Meta-Object-Protocol* (MOP). Concretely, the MOP must: (i) *on the client side*: during serialisation any parameter marked as *strict value* will enforce an explicit synchronisation on the related object; the overhead is payed only for methods with annotated futures. (ii) *on the server side*: during deserialisation any parameter marked as *strict value* can be checked not to be a future; to avoid overhead, one may assume that the sender respects the contract because it was previously checked during serialisation. Moreover, the affected parameters will never block because they are guaranteed to be concrete values.

6 Conclusion

Throughout this paper we studied how to model transparent first-class futures in distributed components, as well as some necessary properties in order to avoid deadlocks related to futures. To our knowledge, the only previous work providing static reasoning on futures is [8], and focused on invariant proofs for explicit futures. We provides here behavioural static models for transparent futures that can be detailed as follows:

A Model for Transparent First-Class Futures. We defined an abstraction and a model for futures and their behaviour (synchronisation, update). This model expresses the flow of future references and future values. It extends our previous works by giving behavioural models for transparent first-class futures, relying heavily on the properties proved in the ASP-calculus.

A Framework for Detecting Blocked Components. Thanks to our model we are able to detect components indefinitely blocked on future access using model-checking techniques. This way, futures for which a value will never be computed can be identified. Our model greatly helps the programmer to find synchronisation issues in concurrent programs with futures.

Rich-Interfaces. Finally, we showed that the Interface Description Language of

GCM can be improved in order to specify synchronisation on futures at the interface level. This lifts some synchronisation from the behaviour up to the interface level, which yields more precise behavioural models and avoids some deadlocks.

An alternative model for futures would consider global references to further optimise the state-space. The properties on confluence inherited from ASP allows us to update all references of a future synchronously without other impact than generating traces with less interleaving. This effectively avoids the propagation of values found in our model, however it requires inter-procedural static analysis, so it does not allow the model to be built independently.

References

[1] A. Arnold. *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall, 1994.

[2] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, accepted for publication, 2008. also Research Report INRIA RR-6491.

[3] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, Oct. 2005. ENTCS.

[4] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: A grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, accepted for publication, 2008.

[5] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[6] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the Fourth Annual ACM SIGACT-SIGPLAN Symp. on Principles of Progr. Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[8] F. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[9] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming in ambienttalk. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[10] F. Fernandes and J. Royer. The STSLIB project: Towards a formal component model based on STS. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07)*, Sophia Antipolis, France, September 2007. To appear in ENTCS.

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

[12] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, Aug. 2002.

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

[14] ISO: Information Processing Systems - Open Systems Interconection. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, Aug. 1989.

[15] R. Mateescu. Efficient diagnostic generation for boolean equation systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 251–265, 2000.

[16] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.

[17] J. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, July 1983.

[18] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, Cambridge, Massachusets, 1987.