Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (2010) 35–53
www.elsevier.com/locate/entcs

Asynchronous Components with Futures: Semantics and Proofs in Isabelle/HOL

Ludovic Henrio and Muhammad Uzair Khan

INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
{ludovic.henrio, muhammad.khan}@inria.fr

Abstract

Components provide an easy to use programming paradigm allowing for better re-usability of application code. In the context of distributed programming, autonomous hierarchical components provide a simple model for creating efficient applications. This paper presents a model for distributed components communicating asynchronously using futures – placeholders for results. Our components communicate via asynchronous requests and replies where the requests are enqueued at the target component, and the invoker receives a future. Then, future references can be dispersed among components. When the result is available for a future, it needs to be transmitted to all interested components, as determined by a future update strategy. We present formal semantics of our component model incorporating formalisation of one such future update strategy. Our model has been mechanically formalised in Isabelle/HOL, together with the proof of properties. This approach validates the actual implementation of the future update strategy itself.

Keywords: Mechanised formalisation, Components, Futures, Distributed systems

1 Introduction

This paper is placed in the context of the GCM [1] component model, and aims at proving the correctness of its reference implementation (ProActive/GCM). Components are designed to increase the re-usability of programs. For this a component is defined as a piece of software with well-defined server and client interfaces (also called input and output ports). To increase scalability of the model, components can be designed in a hierarchical way: each component can be composed of other components. To better benefit from the component structure, GCM is one of the component models where components are represented and can be manipulated at run-time; this allows dynamic reconfiguration and adaptability of component-based applications.

Our component model goes one step further in the autonomicity of the components: each component is a unit of deployment and of concurrency, i.e. components only interact by asynchronous requests, each component has its own threads, and components do not share memory. In this context, structured communication

doi:10.1016/j.entcs.2010.07.004

impose the use of futures, empty objects representing an awaited result for such asynchronous requests. To increase asynchronism, our futures are first class; meaning a future may be passed as parameter of requests or as part of return values. As a consequence, futures spread everywhere. Under reasonable hypotheses, it has been shown that the order in which results are returned has no influence on the computation [4].

Even when the execution is insensitive to the order in which futures are returned, in a real implementation of the component platform, a strategy has to be chosen to optimally perform the communication of results. We call *future update* the operation that sends a result to replace a future reference; and *future update strategies*, the different ways of performing those operations. Formalising future updates is of little interest concerning the language properties, but it is crucial to study the implementation of this language. In order to prove the correctness of the implementation of GCM, our work aims at specifying formally future update strategies and proving correctness or efficiency properties on futures. This paper focuses on one particular strategy called *eager home*[10]. Our contribution can be summarised as follows:

- Formal specification of the component model and the eager home strategy,
- Formalisation in a theorem prover, Isabelle/HOL [13],
- Tools (lemmas, constructs) for expressing future update strategies and proving properties on components and futures,
- Mechanised proofs of correctness for future updates and registration.

To reflect the component model, we choose to specify the entire component structure (hierarchy, interfaces, bindings). One advantage is that we can navigate inside components, and directly reason on the application structure in the theorem prover. The second and even more important advantage is that it allows us to reason about component configuration and component reconfiguration, leading to the specification of an adaptive component model.

Our intent is to provide a reliable and strong basis for reasoning on futures and components. For this we prove a correctness property on the registration of futures along the reduction. The Isabelle/HOL development corresponding to this paper is already consequent and shows that: our model is adequate and precise, it can be used to reason about futures and components, and the specified future update strategy guarantees basic correctness properties.

This work is not restricted to the GCM component model, for example our formalisation should also provide a model for frameworks like Creol [9].

Next section presents the related works, Section 3 presents the principles of our component model and of future update strategies. Section 4 defines the semantics of our model. Formalisation is detailed in Section 5.

2 Related Works

Futures, first introduced in Multilisp [6] and ABCL/1 [17] are used as constructs for concurrency and data flow synchronisation. Futures are language constructs that improve concurrency in a natural and transparent way. Frameworks that make use of explicit constructs for creating futures include Multilisp [6], λ-calculus [12], Creol [9], SafeFuture API [16], and ABCL/f [15]. In contrast, futures are created implicitly in frameworks like ASP [4], AmbientTalk [5], ProActive [3] and ASP_{fun} [7]. In those object-oriented languages, implicit creation corresponds to asynchronous method invocation. A key benefit of the implicit creation is that no distinction is made between synchronous and asynchronous operations in the program. Additionally, the futures can be accessed explicitly or implicitly. In case of explicit access, operations like *claim* and *get*, *touch* are used to access the future [9,15]. For implicit access, operations that need the real value of an object (*blocking* operations) automatically trigger synchronisation with the future update operation.

Objects and futures

Creol [9] allows explicit control over data-flow synchronisations. In [2], Creol has been extended to support first class futures, although the future access is explicit (using *get* and *await*). ASP [4] and ProActive [3], have transparent first-class futures. Thus, the synchronisation is transparent and data-flow oriented. In AmbientTalk [5], futures are also first-class and are transparently manipulated; but the future access is a non-blocking operation: it is an asynchronous call that returns another future. This avoids the possibility of a dead lock as there is no synchronisation.

This differs from the approach adopted in other frameworks where access to a future is blocking. In [9], all processes interested in the future are registered as observers. When the result for the future is computed, all the registered observers are notified; this is very similar to eager-message based strategy as specified in [10]. In [16] a *safe* extension to Java futures is proposed, but with explicit creation and access.

In [2], the authors provide the semantics of an object-oriented language based on Creol [9]; it features active objects, asynchronous method calls, and futures. They provide a proof system for proving properties relating to concurrency. The model is multi-threaded, with only one thread active at a given time. Our approach is quite close to this work except that we study a component model featuring high level of abstraction, and hierarchical composition.

Also in the context of object-oriented languages, ASP_{fun} [7] is closely related to this paper. It formalises a functional language featuring active objects, asynchronous communication, first class futures, and a type system.

While the language provides for first class futures, it does not study future update strategies. Additionally, it does not deal with components.

Modeling components

In [11], a formalisation of the Fractal component model using Alloy, a specification language, is presented. Fractal allows for hierarchical composition of components, and separation of functional and non-functional concerns. The authors provide a analyser to check the consistency of model, they define key invariants and other properties of interest. Compared to [11], we consider asynchronous components and focus on the component dynamic behaviour. This is crucial when specifying future management procedures.

Our work extends [8] which presents a component model giving a semantics to GCM, including hierarchical components, asynchronous communication, and first class futures. Building on the structural description provided in GCM, [8] formalises the component composition and communication semantics in the presence of futures. In order to prove properties related to the implementation of futures, we have extended [8] with the precise definition of future update strategies, including semantics and constructs for the management of futures. With mechanised proofs, we show that our formalisation is complete and enables proofs on properties on futures and their update strategies, thus ensuring correctness of the ProActive/GCM implementation.

3 An Asynchronous Component Model With Futures

This section defines a subset of the GCM model, but with a precisely defined semantics. This model incorporates hierarchical components, and asynchronous communication with futures, it uses a request-reply model.

3.1 Component structure

Our model inherit most of its structure from GCM. GCM allows hierarchical composition of components. A coarse-grained component may be formed by composition of several fine-grained components. A component containing one or more *subcomponents* is referred to as a *composite component*. *Primitive components* do not contain other components, they are leaf-level components that implement the business functionality.

The only way to access a component is via its interfaces. *Client* interfaces allow the component to invoke operations on other components. On the other hand, *Server* interfaces receive invocations. Each client interface is plugged to a server interface. For this, a *binding* connects a client interface to the server interface that will receive messages sent by the client: requests transit on bindings. GCM model allows for a client interface to be bound to multiple-serve interfaces, but here, to simplify, bindings can only be one to one.

Figure 1 shows a composite component containing two primitive components, along with bindings, and the various interfaces of the components. The interfaces exposed to subcomponents are referred as *internal* interfaces, while the *external* interfaces are the ones exposed to other components. In our model, all interfaces of a given component, external or internal must have distinct names. Additionally, each

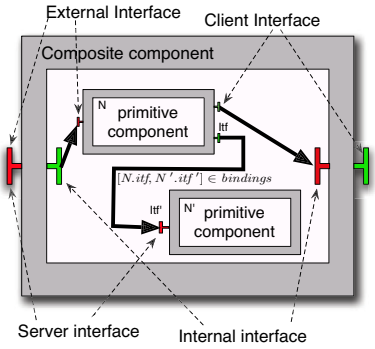

Fig. 1. A composite GCM component

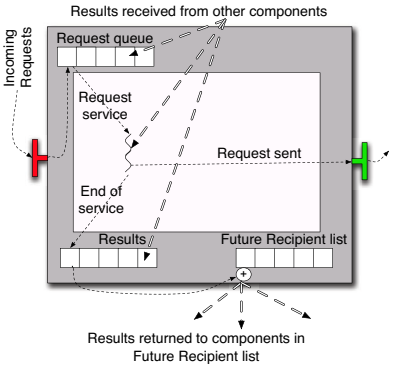

Fig. 2. Component Behaviour

internal interface have a corresponding external interface of the same name; with the implication that a call received on a external (resp. internal) server interface will be passed unchanged to the corresponding internal (resp. external) client interface. A GCM component may have functional or non-functional interfaces. Our model only deals with the functional interfaces and as such, we have excluded the non-functional interfaces from our representation. Figure 2 gives the internal structure of a component. It shows the request queue, the results list, and the future recipient list. All incoming requests are initially enqueued in the *request queue*. The requests are dequeued by the execution threads, and on the termination of execution (called "end of service"), the results are placed inside the *results list*. The *future recipient list* contains the components which need the results for a given request. When a result is produced, it is sent to all the components registered for it in the recipient list.

3.2 Informal semantics

As discussed in the previous sections, our communication model is strictly a request-reply model with no shared memory. Components are the unit of concurrency though each component may serve multiple requests concurrently.

Communication model

We use a simple communication model relying on asynchronous request and replies. The only interaction between components is the communication by means of requests. All request parameters are passed by copy semantics. There are no shared object/component references (except for futures). On the receiver side, the requests are enqueued in a *message queue*, which holds the messages until they can be treated by the receiver component. Our communication model is asynchronous. This means that the requests are not necessarily served/treated immediately upon arrival. Requests are only enqueued at the target component, then the component invoking the request can continue its execution without waiting for the result. Enqueuing a request is done synchronously but the receiver is always ready to receive a request.

To ensure transparent handling of asynchronous requests with results, we utilise *futures*. Futures are created automatically upon request invocation and represent the request result, while the treatment of the request is not finished. Once the result

of the computation is available, the future is replaced by the result value. We refer to this as *updating* the future value. Our futures are both transparent and implicit: they are created and are updated automatically.

Access to a future for which the result value has not been received yet is a blocking operation. The thread accessing such a future is blocked until the result value becomes available. Futures are first class objects: no thread is blocked when a future is transferred as part of requests or results.

Component behaviour

Primitive components are the basic components that implement business logic and therefore can have any internal behaviour. They treat/serve requests in the order they choose, providing replies for all the requests they receive. They can call other components by emitting a request on one of their client interface. Each primitive component must always be able to accept a request (enqueued in its request queue), and to receive a result (that will replace a future reference). Once the service of a request is finished, the produced result is stored in the computed results, which is a mapping between futures and computed values. It can then be transmitted to other components, as determined by the future update strategy.

As opposed to the primitive components, the behaviour of the composite components is strictly defined. Composite components serve the requests in a FIFO order, delegating the requests to other internal or external components. As mentioned in Section 3.1, a request received at a composite component is delegated unchanged to a bound component. Overall, a request is emitted by a client interface of a primitive component, and received unchanged by the server interface of the primitive component that is (indirectly) bound to it; this request might transit through several composite components and bindings.

3.3 Future update strategies

In a real implementation, updating a future value is not a simple task. Futures may be spread over a number of components, all requiring the future value. Additionally futures can appear in computed results, message queue, and current state of each component. To update all these futures efficiently, future update schemes have to be devised. The chosen scheme must ensure that any component needing a result that has been computed, receives it.

First class futures can be updated using different strategies [4,10,14]. We classify those strategies as either *eager* or *lazy*. Strategies are called eager when all the references to a future are updated as soon as the future value is calculated. They are called lazy if futures are only updated upon need, which minimises communications but increase the time spent waiting for the future value. Two eager strategies can be envisioned. *Eager forward* strategy, where each component remembers only the components to which it has sent the futures, and forward them the values when they become available; flow of future updates is along the same path as the futures themselves. On the other hand, in *eager home* strategy, each component is responsible for sending the future value to all components which have a reference to

the future. For this, all components receiving the future must register themselves as a future recipient. Finally, the *lazy home* strategy is the lazy version of eager home strategy where the future values are transferred on-demand: accessing a future reference triggers the future update.

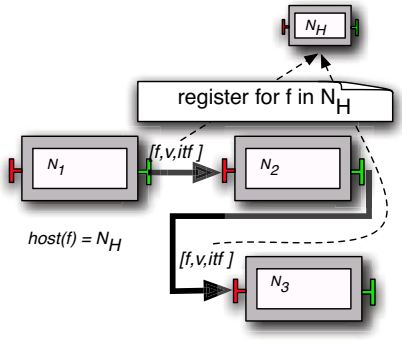

Fig. 3. Future registration

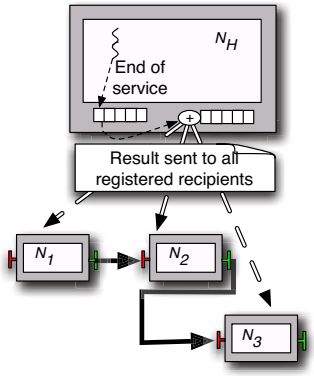

Fig. 4. Future update

Based on our component structure, we can derive semantics using any of the above mentioned strategies. All three strategies are semantically equivalent, as demonstrated in ASP [4]. Eager-forward strategy is simpler to implement as the flow of future updates follow the same path as the futures themselves. Therefore each component needs to remember only the component to which it has transferred the future. On the other hand, for the home-based strategies, the component serving the request needs to know about all components to which results should be sent. This is achieved by registering all components that require the future value with the component serving the request. Such registrations are more complex as compared to simple mechanism used in eager forward strategy. Eager-home and lazy-home strategies are similar in nature, the only deference being the on-demand nature of component registration in lazy strategy. For eager-home strategy, every forwarded future has to be registered with the component computing its value; including futures inside request parameters and result values. Registration mechanism for the lazy strategy is simpler because it is only triggered on future access. To conclude, eager-home strategy is the more complex strategy and we selected it for this paper in order to show that strategy can be formalised, and its properties shown. Finally, our formalisation can also be used in the context of Creol [9], which uses a update mechanism similar to eager-home.

Figure 3 shows the registration process for a future f which will be computed by the component N_H. N_1, N_2 and N_3 all have references to the future f, and consequently register with N_H. Once the result is computed, N_H sends this result to all registered components (N_1, N_2, and N_3) as shown in Figure 4.

4 Formal Model

In this paper, we build on the formal model presented in [8], adding precise semantics for the future update mechanism. We start this section by the general notations,

and gradually move to more component and GCM specific notations. The resultant model has been formalised [1] in Isabelle/HOL [13].

4.1 Structure and notations

We denote lists as $[a_i]^{i\in 1..n}$, while $\{a_i\}^{i\in 1..n}$ is used for a finite set. Pairs are represented with the notation (a, b). A number of operators are used in our formalism. The operator # is the list append operation whereas $[a_i]^{i\in 1..n} \setminus b$ removes b from the list $[a_i]^{i\in 1..n}$ whatever its position is. We use the notation $[a_i \mapsto b_i]^{i\in 1..n}$ to indicate a mapping from a_i to b_i. A new entry is added to an existing mapping simply by $([a_i \mapsto b_i]^{i\in 1..n})[c \mapsto d]$. $([a_i \mapsto b_i]^{i\in 1..n})[c \mapsto \emptyset\ ]$ removes the entry corresponding to c in the mapping, if it exists.

Let f range over futures, v range over values, *itf* range over interfaces and C range over components. Additionally, S denotes a composite component representing the component system (all components currently instantiated). A future f is a pair (identifier, component name): $f ::= (id, N)$. id is a unique identifier for the future, while N is the name of the component computing the value of the future. Similarly, we define a value v as a pair ("object value", set of referenced futures): $v ::= (V, f_i^{i\in 1..n})$, where "object value" (V) is a structure representing the values of the underlying language but that we abstracted away by integers. This prevents values from being defined recursively. We denote (V_f, f_0) the value containing only a future reference f_0. The second element of v, is the set of futures contained in the value. A Request R is a triple (future, value ,interface): $R ::= (f, v, itf)$.

Component structure

For presenting our component model, we choose a representation that include static information like component interfaces and bindings. This allows our model to be expressive enough to support properties and proofs interleaving the component structure and more dynamic features like future update strategies. On a longer term basis it will also allow us to prove properties on component reconfiguration.

Components in our model can be either composites or primitives:

$$C ::= \text{Comp}[\,N,\ itfs,\ subCp,\ bindings,\ CompState] \mid \ \text{Prim}[\,N,\ itfs,\ PrimState]$$

All components have a unique name N (there is only one component with a given name), a list of interfaces $itfs ::= [itf_i]^{i\in 1..n}$, and a component state s. Additionally, a composite has a list of subcomponents $subCp ::= [C_i]^{i\in 1..n}$, and a set of bindings $bindings ::= \{(N_i.itf_i,\ N'_i.itf'_i)^{i\in 1..n}\}$. $(N_i.itf_i,\ N'_i.itf'_i)$ belongs to *bindings* if interface itf_i of component named N_i is plugged to the interface itf'_i of N'_i (where N_i and N'_i can either be a component name or *This* if the plugged interface is the composite component that defines the bindings).

Each component state s contains a request queue: $queue ::= [R_i]^{i\in 1..n}$, a list of results mapping futures to computed values: $results ::= [f_i \mapsto v_i]^{i\in 1..n}$, and a list of futures recipients: $FRL ::= [f_i \mapsto \{N_j\}^{j\in 1..n_i}]^{i\in 1..n}$. A primitive component state additionally contains an internal state (*intState*), and an associated behaviour

[1] Prototype specification available at www.inria.fr/oasis/Ludovic.Henrio/misc

behaviour. A behaviour is a labelled transition system where the actions of the primitive components are the labels of transactions and the states are the values of *intState*. An internal state contains a list of current requests: $currReq ::= [f_i]^{i\in 1..n}$ and a list of futures referenced by the internal state: $refF ::= [f_i]^{i\in 1..n}$. Fields of a state are accessed through functions. For example, $queue(s)$ returns the current queue of the state s. Fields are modified by the operator $:=$ as shown below. $Enqueue(C, R)$ returns the component C where its state s is replaced by $s(\!|queue := queue(s) \# R |\!)$.

Additional constructs

We introduce a registration list RL to support the eager home strategy; it maps a future to the set of components that require the value for this future: $RL ::= [f_j \mapsto \{N_i\}^{i\in 1..n_j}]^{j\in 1..n}$. The structure of registration list RL is the same as for future recipients list FRL. The registration list of future f is accessed by $RL(f)$, to simplify, if $f \notin \mathrm{dom}(RL)$ then $RL(f) = \emptyset$. We define two new operators for manipulating lists of components. Operator $\uparrow$ is a find operation: $(subCp \uparrow N)$ is the element of $subCp$ which has the name N. Operator $\leftarrow$ is the list replacement operator: $(subCp \leftarrow C_1)$ replaces by C_1 the component in $subCp$ that has the same name as C_1. List append operator $\#$ is overloaded for recipient list RL as:

$$RL\#RL' \triangleq [f_j \mapsto M_j \mid f_j \in \mathrm{dom}(RL) \cup \mathrm{dom}(RL') \wedge M_j = RL(f_j) \cup RL'(f_j)]$$

To simplify our semantics we introduce a number of support functions.

$RqIdsSet(S)$ is the set of ids of all requests computed by S. It is the union of the domains of the request queue of S ($queue$), its currently executing requests ($currReq$), and its computed results ($results$), but also, recursively, requests computed by all its subcomponents.

$RefFutSet(S)$ is the set of all futures referenced by S and all its subcomponents recursively. It contains futures referenced in the current state ($refF$), futures in the parameters of the requests in the request queue ($queue$), futures in the the value of computed results ($results$), and futures referenced by subcomponents. By extension, we define similar function $RefFutSet(v)$, giving set of futures in a value v.

$host(f)$ is the name of the component computing future f ($snd(f)$).

$cpSet(C)$ is the set formed of C and all the components recursively contained in C.

$removeResult(f, C, N)$ looks recursively inside the component C until a component C' with name N is found. It returns C where the state s of C' is replaced by $s(\!|\, results = results(s)[f \mapsto \emptyset], FRL := FRL(s)[f \mapsto \emptyset] \,|\!)$

$updateFV(v, f, v')$ abstracts away the operation that updates a value v by replacing the occurrences of future f by v'. We simplified it so that it removes f from $RefFutSet(v)$, and replaces it by $RefFutSet(v')$. It returns the new value, and verifies the property:

$RefFutSet(updateFV(v, f, v')) = RefFutSet(v) \setminus \{f\} \cup RefFutSet(v')$

$getName(C)$ is the name of the component C.

$registerListFutures(S, RL)$ takes a component system S and a registration list

RL and returns a new component system S' such that all the entries in the RL have been added to the recipient lists (FRL) of relevant components: The state s of each component of S is replaced by $s\,(\!|FRL = FRL(s) \# RL|\!)$. This only affects components with name $host(f)$ where $RL(f) \neq \emptyset$.

A summary of the notations appears in Appendix B.

4.2 Semantics of component model

The formal semantics of our component model are given by a number of reduction relations defined by a set of inductive rules. The global reduction of the component system is $\rightsquigarrow$, it triggers either $\dashv f, v, N \mapsto_F$, or $\rightarrow_R$ reductions. $S \vdash C \rightarrow_R C', RL$, if in component system S, component C can be reduced to the component C'. RL contains the list of future registrations to be performed.

The parametrised relation $\dashv itf, f, v \mapsto_O$ emits messages. In order to be matched with a receive action, the statements $\dashv itf, f, v \mapsto_O$ are used as hypotheses to the rules for $\rightarrow_R$ for composite components. If $S \vdash C \dashv itf, f, v \mapsto_O C'$, then in the component system$S$, C emits a request on the interface itf, with parameter v, and is associated to a future f; after the emission, C becomes C'. A final parametrised relation $\dashv f, v, N \mapsto_F$ expresses that a component receives the new value for a future (future update message). if $C \dashv f, v, N \mapsto_F C', RL$, then the component C with name N receives the value v for the future f. N should register for all the futures in v via RL.

Figure 7 presents reduction rules dealing with composite components. The first rule embeds subcomponent reduction in composite contexts; the second rule allows composite components to emit requests on their external client interfaces. The three Comm-rules define the request transmission over the different kinds of bindings. Trigger future update defines the mechanisms for initiating future updates. Finally, the two Rcv rules performs future updates inside composite components based on whether it is the right component for doing the update or not. The last rule triggers $\rightarrow_R$ reduction. Figure 5 illustrates the different kinds of communications expressed by the Comm-rules and the composite call rule. The existence of the different form of rules is due to the component structure. Classically there are two rules for stating that a component is willing to send a communication (one for primitives and one for composites). Additionally, as there are three kind of bindings (from a parent component to a subcomponent, between two subcomponents, or from a subcomponent to its parent), there are three kind of communication rules (resp. CommChild, CommBrother, or CommParent).

Hierarchy: Hierarchy defines the compositionality of components. If a component C reduces to a component C' in isolation, then it also does so inside a composite. The registration list is the one for the sub-component.

CompositeCall: This rule describes how a composite component emits a call on the external client interface. This request will be handled by the enclosing composite. The request [f, v, itf], received on internal server interface itf, is sent on the matching external client interface (with same name). This call will be matched against a Comm rule that enqueues this request. A fresh future f' is found for this

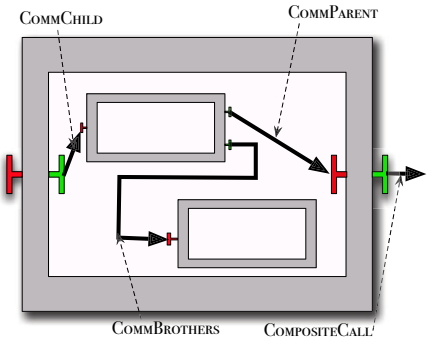

Fig. 5. Component Communications

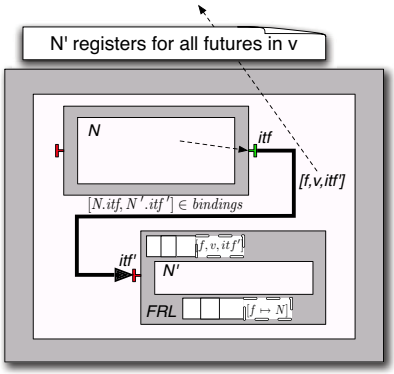

Fig. 6. CommBrother

new request. The composite component records that the value of f is now the new future f', and dequeues the request.

CommBrothers: This rule expresses communication between two sibling subcomponents of a composite component, as illustrated in Figure 6. If N and N' are the names of two subcomponents of component N_0, then component N can pass a call to component N' if the client interface itf of N is bound to the server interface itf' of N' ($[N.itf, N'.itf'] \in bindings$). The call parameters f, v are passed unchanged to interface itf' of subcomponent N'. The operation *Enqueue* is used to place the request $[f, v, itf']$ onto the request queue of the destination. N is reduced simultaneously, sending the request. Component N then register (in the RL list) for receiving the result for future f when it is available. Similarly, N' also registers for all futures inside the parameter v.

CommChild: This rule expresses request delegation between a composite component and its subcomponent as shown in Figure 8. The request $[f, v, itf]$ is dequeued from the request queue of the parent. A new future f' is created and added to the result list of the parent as the result for this request. The new request $[f', v, itf']$ is enqueued at the subcomponent. The exact subcomponent is determined using the bindings: a request delegated to a subcomponent necessarily arrived an external server interface, call it itf, if $This.itf$ is bound to $N'.itf'$ then the request is sent to the interface itf' of the subcomponent N'. The component N_0 registers in the RL list to receive the result for f', also the destination N' registers for any future inside the request parameter v.

CommParent: This rule expresses communication between a subcomponent and the composite component containing it, see Figure 9. When a subcomponent N of a composite component N_0 emits a request $[f, v, itf']$ to its parent, the request is added to the composite component's request queue. For this, the subcomponent interface $N.itf'$ must be bound to the parent component interface $This.itf$. The component N registers to receive the value for f when it is available; also, the values for any future inside v must be sent to N_0.

TriggerFutureUpdate: This rule selects a computed result of a component C in the component system S_o for initiating the future update process. The value v for the future f, has to be sent to all components $(N_i{}^{i\in 1..n})$ in future recipient list

HIERARCHY

$$\frac{(subCp \uparrow N) \;=\; C \qquad S \vdash\; C \rightarrow_R\; C',\; RL}{S \vdash \mathrm{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \mathrm{Comp}[N_0, itfs, (subCp \leftarrow C'), bindings, s],\; RL}$$

COMPOSITECALL

$$\frac{\begin{array}{c} queue(s) \;=\; [f, v, itf] \;\#\; Q \\ f' \notin RqIdSet(S) \qquad s' \;=\; s(\!|queue := Q,\; results := results(s)[f \mapsto (V_f, \{f'\})]|\!) \end{array}}{S \vdash \mathrm{Comp}[N, itfs, subCp, bindings, s] \dashv itf, f', v \mapsto_O \mathrm{Comp}[N, itfs, subCp, bindings, s']}$$

COMMBROTHERS

$$\frac{\begin{array}{c} C = (subCp \uparrow N) \qquad [N.itf,\; N'.itf'] \in bindings \qquad S \vdash\; C \dashv itf, f, v \mapsto_O C' \\ host(f) = N' \qquad subCp' = subCp \leftarrow C' \qquad subCp'' = subCp' \leftarrow (Enqueue(subCp' \uparrow N', [f, v, itf'])) \end{array}}{\begin{array}{c} S \vdash \mathrm{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \\ \mathrm{Comp}[N_0, itfs, SubCp'', bindings, s],\; [(f'',\; N') \mid f'' \in RefFutSet(v)] \;\#\; [f, N] \end{array}}$$

COMMCHILD

$$\frac{\begin{array}{c} queue(s) = [f, v, itf] \;\#Q \qquad [This.itf, N'.itf'] \in bindings \\ f' \notin RefFutSet(S) \qquad host(f') = N' \qquad subCp' = subCp \leftarrow (Enqueue((subCp \uparrow N'),\; [f',\; v,\; itf'])) \\ s' := s(\!|queue := Q,\; results := results(s)[f \mapsto (V_f, f')]|\!) \end{array}}{\begin{array}{c} S \vdash \mathrm{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \mathrm{Comp}[N_0, itfs, subCp', bindings, s'], \\ [f', N_0] \;\#\; [(f'', N') \mid f'' \;\in\; RefFutSet(v)] \end{array}}$$

COMMPARENT

$$\frac{\begin{array}{c} (subCp \;\uparrow\; N) = C \\ [N.itf',\; This.itf] \in bindings \qquad subCp' = subCp \leftarrow C' \qquad S \vdash\; C \dashv itf', f, v \mapsto_O C' \qquad host(f) = N_0 \end{array}}{\begin{array}{c} S \vdash \mathrm{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R Enqueue(\mathrm{Comp}[N_0, itfs, subCp', bindings, s], \\ [f, v, itf]),\quad [f, N] \;\#\; [(f'', N_0) \mid f'' \in RefFutSet(v)] \end{array}}$$

TRIGGERFUTUREUPDATE

$$\frac{\begin{array}{c} C \in cpSet(S_o) \qquad \text{the state of } C \text{ is } s \qquad results(s)(f) = v \qquad FRL(s)(f) = \{N_i\}^{i \in 1..n} \\ \forall i \in 1..n,\; S_{i-1} \dashv f, v, N_i \mapsto_F S_i,\; RL_i \qquad S' = RemoveResult(f,\; S_n,\; getName(C)) \end{array}}{\vdash S_o \rightsquigarrow regListFutures(S',\; RL_1 \;\#\; RL_2 \;\#\; \ldots \;\#\; RL_n)}$$

RCVRESULTCOMPOSITE(1)

$$\frac{\begin{array}{c} s' = s(\!|results = [f_i \mapsto v_i \mid \exists\, v_i'.\, [f_i \mapsto v_i'] \in results(s) \;\wedge\; v_i = updateFV(v_i',\; f,\; v)], \\ queue = [[f_j,\; v_j,\; itf_j] \mid \exists\, v_j'.\, [f_j,\; v_j',\; itf_j] \in queue(s) \;\wedge\; v_j = updateFV(v_j',\; f,\; v)] \end{array}}{\begin{array}{c} S \vdash \mathrm{Comp}[N, itfs, subCp, bindings, s] \dashv f, v, N \mapsto_F \\ \mathrm{Comp}[N, itfs, subCp, bindings, s'],\quad [(f'',\; N) \mid f'' \in RefFutSet(v)] \end{array}}$$

RCVRESULTCOMPOSITE(2)

$$\frac{N_0 \neq N' \qquad (subCp \;\uparrow\; N) \dashv f, v, N' \mapsto_F C',\; RL \qquad subCp' = subCp \leftarrow C'}{S \vdash \mathrm{Comp}[N_0, itfs, subCp, bindings, s] \dashv f, v, N' \mapsto_F \mathrm{Comp}[N_0, itfs, subCp', bindings, s],\; RL}$$

R-REDUCTION

$$\frac{S \vdash S \rightarrow_R S', RL}{S \rightsquigarrow registerListFutures(S', RL)}$$

Fig. 7. Semantics of the component composition

FRL for the future f. For every component N_i, a future update is triggered; on N_i, this is matched by the *RcvResult* rule.

RCVRESULTCOMPOSITE(1): This rule expresses future update for a composite component which is the destination of the update. At the component N, the state s is updated such that the new value v for the future f, replaces the old value inside both the *results* and *queue*. The values for any futures inside v should be sent to N, this is recorded in the *RL* list.

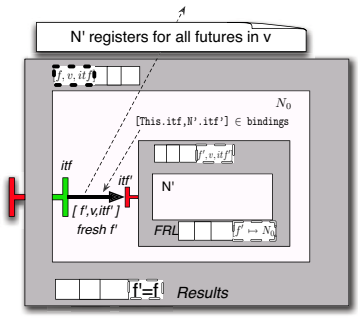

Fig. 8. CommChild rule

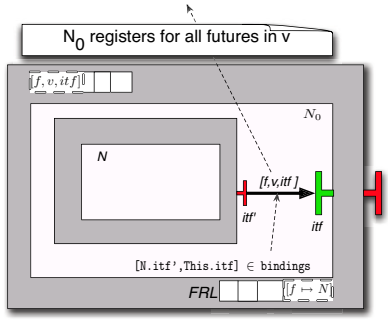

Fig. 9. CommParent

RcvResultComposite(2): This rule ensures that a future update is applied at the component that is the destination of the future update, i.e., only at the component which has the same name as given in reduction parameter $\dashv f, v, N \mapsto_F$. Only the sub-component that contains a component of name N is able to be reduced. This rule navigates in the component hierarchy, finds the component with name N, and applies rule RcvResultComposite(1).

R-reduction: From $\rightarrow_R$, a reduction $\rightsquigarrow$ for the global component system can be performed, after performing all future registrations.

The reduction rules for the primitive components are similar and appear in Appendix A. We only show one of the rules below. RcvResultPrim expresses the future update for a primitive component. All references to the future f are replaced with the new value v. Entries are made in the *FRL* for any futures present inside v. Interaction between the component semantics and the internal state of the primitive is enabled by triggering transitions on the primitive *behaviour*, here *ReceiveResult*.

RcvResultPrim

$$\frac{\begin{array}{c}(s, ReceiveResult\ f\ v, s') \in behaviour(s) \\ s'' = s'(\!|results = [f_i \mapsto v_i \mid f_i \in \mathrm{dom}(results(s)) \wedge v_1 = updateFV(results(s)(f_i), f, v)], \\ queue = [[f_i, v_i, itf_i] \mid [f_i, v'_i, itf_i] \in s.queue \wedge v_i = updateFV(v'_i, f, v)]|\!)\end{array}}{S \vdash \mathrm{Prim}[N, itfs, s] \dashv f, v, N \mapsto_F \mathrm{Prim}[N, itfs, s''], [(f'', N) \mid f'' \in RefFutSet(v)]}$$

5 Formalisation in Isabelle and Properties

Isabelle/HOL is a generic interactive theorem proving framework, that allows implementation of formalised object logic. This section outlines the mechanisation of our component model in Isabelle/HOL, its semantics including eager home strategy, and several formalised proofs.

First, the definition of the component structure and the component semantics are directly translated from the preceding sections as we will show below. Then this section will describe the properties we proved using our formalisation. This is clearly the most innovative part of this paper as it shows that our formalisation is able to handle mechanised proofs entailing reasoning on components, their structure, and futures. While the formalisation represents a few hundreds lines of code, the proofs are much longer (above 5000 lines) and entail reasoning interleaving component structure, semantics, and future registration aspects.

Component model

The data type for components is defined as follows:

```
datatype Component =
Primitive Name (Name↦ Interface)                                    PrimState |
Composite Name (Name↦ Interface) (Component list) (Binding set) CompState
```

Isabelle/HOL support for *finite sets* is much weaker than for list, thus it is easier to reason on lists as compared to sets. Consequently, we utilise lists where possible; for example, the subcomponents of a composite are defined as a list of components, which allows inductive reasoning on the component structure. We model the bindings in composite components as a set because no inductive reasoning is required on bindings. In practice, we only reason on a subset of components, that we call correct components:

```
constdefs CorrectComponent:: "Component ↦ bool"
"CorrectComponent C == CorrectCompStructure C ∧ distinct(RqIdList C)
∧ (RefFutSet C) ⊆ (set(RqIdList C))∧ distinct(map getName(cpList C )
```

`CorrectComponent` states the correctness rule for a component as: component should be constructed correctly (more precisely: bindings are one-to-one, and connect an existing client interface to an existing server one, local behaviour of primitives refer to existing interfaces, ...). Each future should correspond to a unique request in the component hierarchy (*RqIdList* has no duplicate), and each futures referenced by the components should correspond to a request. Finally, names of components in the composition should be unique. Note that `cpList` returns list of all components in the composition, recursively (`cpSet C= set(cpList C)`). The requirement of checking correct referencing throughout the composition hierarchy is stronger than what is needed for most proofs, and can at times be loosened, resulting in a weaker correctness rule(shown as `CorrectComponentWeak` in Isabelle/HOL implementation).

Semantics

The semantics of primitive and composite components, as detailed in Section 4.2 and Appendix A has been entirely specified in Isabelle/HOL. To compare semantic specification in Isabelle to its mathematical equivalent, we show below the CommParent, and compare it to Figure 7. It is easy to see the equivalence of the two specifications (only a few intermediate variables were removed in the Isabelle version),

```
CommParent:
"⟦(subCp↑N) = Some C,⦇src=N.itf',dest=This.itf⦈ ∈ bindings; snd f= N0;S⊢ C⊣f,v,N↦_O C'⟧"
⟹ S⊢Composite N0 itfs subCp bindings s →_R (Composite N0 Itf (subCp<-C') bindings s)
          ← ⦇id=f, param= v,invokedItf= itf⦈,(f,N) # (map (λ id. (id,N0 )) (snd (v))
```

Properties and proofs

The formalisation sketched above and entirely written in Isabelle/HOL is rich enough to allow proofs of various lemmas. Our objective is to have a framework rich enough to address most aspects of distributed components features, but also the framework should be close enough to the existing component framework so that equivalence between the implementation of the framework and the specification is simple and convincing. We believe that our approach is adequate to prove proper-

ties entailing component structures, asynchronous communications, and component behaviours. More specifically in this paper we focused on the implementation of a future update strategy. Consequently, we only present below theorems related to future updates and registration of futures. Of course those properties cannot be proved without relying on numerous other lemmas mainly related to components structure, and navigation inside component hierarchy. Most of the lemmas are proved by induction on the component structure or on the reduction rules.

A first crucial theorem we proved is `UpdatedFutureDisappear`; it assures that when a future has been updated, no reference to this future exist in the updated component. More precisely, when the future `f` is updated, at the component with the name `N` inside the component system `S`, the new component `C` (with the name `N`) inside the reduced system `S2` no longer has a reference to `f` (`LocalRefFutSet` returns the list of futures referenced locally by C, it is similar to *RefFutSet* but does not enter subcomponents).

```
lemma UpdatedFutureDisappear:
"⟦S⊣f,v,N↦F S2, RL; CorrectComponent S; (S2 ↑↑ N) = Some C ; f ∉ set (snd v)⟧
⟹ f ∉ LocalRefFutSet C)"
```

Concerning future registration, the main theorem we proved in Isabelle is the following one (`GlobalRegisteredFuturesComp` checks that all futures are registered in the given component system):

```
theorem FuturesRegistered:
"⟦⊢C1⇝C2; CorrectComponent C1; GlobalRegisteredFuturesComp C1⟧
⟹ GlobalRegisteredFuturesComp C2"
```

It states that after global reduction $\vdash$`C1` $\rightsquigarrow$ `C2`, all futures registered in `C1` are also registered in the reduced system `C2` along with any new futures generated as the result of component communications. Consequently, the proof of the theorem relies on lemmas about transmission of registered futures, and registration of newly created futures. We show two such lemmas below.

The first lemma states that, if a future `f` (in component named `N`) is registered in `C`, and `C` reduces by $\rightarrow_R$ to `C'`, then the `f` is also registered in `C'`.

```
lemma R_maintainsRegFutures:
"⟦S⊢C →R C',RL; CorrectComponent C; RegisteredFuture f N C; C∈cpSet S⟧
⟹ RegisteredFuture f N C'"
```

The second lemma concerns registration of new futures. It states that if in a source configuration, all futures contained in a subcomponent of `C1` are registered in `S` (expressed by `LocalRegisteredFuturesComp`). Let `C2` be obtained by $\rightarrow_R$ reduction from `C1`. Then, a future referenced from a subcomponent `C'` of `C2` is either initially registered in `S` or will be registered because a corresponding entry is in the registration list `RL`.

```
lemma registeredFutures_R:
"⟧S ⊢ C1 →R C2,RL; C1∈cpSet S;∀C∈cpSet C1. LocalRegisteredFuturesComp C S;
    C'∈cpSet C2; f∈LocalRefFutSet C'⟧
⟹ RegisteredFuture f (getName C') S ∨ (f,getName C')∈set RL"
```

Above proofs are almost entirely mechanised: only properties ensuring preservation of `CorrectComponent` by the reduction are left for future works.

The two theorems presented in this section ensure that the eager home future

update strategy is complete, that is it keep track of the future references in all the component system, and then it updates all those references, removing all references to the considered futures. Consequently, the future can safely be garbage collected. This strategy can be thus adopted in the implementation of the GCM; this guarantees safety of the future update implementation.

This work is not particularly tied to the use of Isabelle/HOL: similar formalisation and results could be obtained with another theorem prover.

6 Conclusion

This paper presented a model for distributed components communicating asynchronously using futures. The communication model is based on a request-reply paradigm, where requests are enqueued at target component and invoker receives a future, representing the result. Futures are first class: and consequently future references can be spread across components. When the results are available, they are sent to the relevant components using a future update strategy.

Future update strategies are somewhat neglected in the literature. We believe that even though future update strategies need not be included for studying properties of a language, they are still important for reasoning on the implementation of this language. Consequently, our semantics include formalisation of a future update strategy. Our model is precise and expressive enough to reason about futures and components, and to guarantee correctness properties. The properties shown here are: futures are registered correctly during reduction, and futures values can be safely garbage collected after update. All of our work, the component model specification, its semantics, and proofs of properties, has been mechanised in isabelle/HOL. Those mechanised proofs ensure the correctness of the implementation of future updates in ProActive/GCM.

We now have sufficient formal constructs and tools to express future update strategies and to study their properties. This work showed that it is possible to formally prove completeness and correctness of our future update mechanism, and of the corresponding implementation in ProActive/GCM. The proofs are relatively long due to the numerous reduction rules, and the rich component structure, thus a lot of cases had to be considered. One of the main difficulties was to design the good representation for our model in the Isabelle theorem prover. A crucial point during the specification phase was to find the good Isabelle/HOL abstraction to represent the component structures. We think we found a good balance between expressiveness and abstraction, that allows formal reasoning but is close enough to the component model implementation.

We now intend to further study the update strategies and to establish an equivalence between different strategies. Also this framework seems to be a good basis to study reconfiguration in distributed component systems.

References

[1] Baude, F., D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio and C. Pérez, *Gcm: a grid extension to fractal for autonomous distributed components*, Annales des Télécommunications **64** (2009).

[2] Boer, F. S. D., D. Clarke and E. B. Johnsen, *A complete guide to the future*, in: *Proc. 16th European Symposium on Programming (ESOP'07)*, LNCS **4421** (2007), pp. 316–330.

[3] Caromel, D., 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** (2006), pp. 69–77.

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

[5] Dedecker, J., T. V. Cutsem, S. Mostinckx and W. D. Meuter, *Ambient-oriented programming in ambienttalk*, in: *Proceedings of 20th European Conference on Object-oriented Programming (ECOOP)* (2006).

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

[7] Henrio, L. and F. Kammüller, *Functional active objects: Typing and formalisation*, in: *Proceedings of the International Workshop on the Foundations of Coordination Languages and Software Architecture (FOCLASA)* (2009).

[8] Henrio, L., F. Kammüller and M. Rivera, *An asynchronous distributed component model and its semantics*, in: *FMCO 2008* (2009).

[9] Johnsen, E. B., O. Owe and I. C. Yu, *Creol: a type-safe object-oriented model for distributed concurrent systems*, Theoratical Computer Science **365** (2006), pp. 23–66.

[10] Khan, M. U. and L. Henrio, *First class futures: a study of update strategies*, Research Report RR-7113, INRIA (2009).
URL HAL:http://hal.archives-ouvertes.fr/inria-00435573/en/

[11] Merle, P. and J.-B. Stefani, *A formal specification of the Fractal component model in Alloy*, Research Report RR-6721, INRIA (2008).
URL http://hal.inria.fr/inria-00338987/en/

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

[13] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer-Verlag, 2002.

[14] Ranaldo, N. and E. Zimeo, *Analysis of different future objects update strategies in proactive*, in: *IPDPS 2007: Parallel and Distributed Processing Symposium, IEEE International*, 2007.

[15] Taura, K., S. Matsuoka and A. Yonezawa, *Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation*, in: *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms* (1994), pp. 275–292.

[16] Welc, A., S. Jagannathan and A. Hosking, *Safe futures for java*, SIGPLAN Not. **40** (2005).

[17] Yonezawa, A., E. Shibayama, T. Takada and Y. Honda, *Modelling and programming in an object-oriented concurrent language ABCL/1*, in: *Object-Oriented Concurrent Programming*, MIT Press, 1987 .

A Primitive Component Semantics

This appendix presents the reduction rules expressing the operational semantics of primitive components. Each of the rule updates the internal state of the component according to its behaviour, additionally:
TAU allows internal transitions.
CALL emits a message towards another component. A new future is created to

$$\textsc{Tau}\quad \frac{(\mathit{PintState}(s),\, \mathit{Tau}, s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)], [\,]}$$

$$\textsc{Call}\quad \frac{(\mathit{PintState}(s), \mathit{Call}(i_1, v, f), s_2) \in \mathit{behaviour}(s) \qquad f \notin \mathit{RefFutSet}(S)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv i_1, f, v \mapsto_O \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)]}$$

$$\textsc{EndService}\quad \frac{(\mathit{PintState}(s), \mathit{EndService}(f, v), s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{results} := \mathit{results}(s)\#[f, v]|\!)], [\,]}$$

$$\textsc{ServeNext}\quad \frac{(\mathit{PintState}(s), \mathit{NewService}(v, f), s_2) \in \mathit{behaviour}(s) \qquad \mathit{queue}(s) = [f, v, i]\#Q}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{queue} := Q|\!)], [\,]}$$

$$\textsc{RcvResultPrim}\quad \frac{\begin{array}{c}(s, \mathit{ReceiveResult}(f, v), s') \in \mathit{behaviour}(s) \\ s'' = s'(\!|\mathit{results} = [f_i \mapsto v_i \mid f_i \in \mathrm{dom}(\mathit{results}(s)) \wedge v_1 = \mathit{updateFV}((\mathit{results}(s))(f_i), f, v)], \\ \mathit{queue} = [[f_i, v_i, \mathit{itf}_i] \mid [f_i, v'_i, \mathit{itf}_i] \in \mathit{queue}(s) \wedge v_i = \mathit{updateFV}(v'_i, f, v)]|\!)\end{array}}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv f, v, N \mapsto_F \mathrm{Prim}[N, \mathit{itfs}, s''], [(f'', N) \mid f'' \in \mathit{RefFutSet}(v)]}$$

Fig. A.1. Primitive Component Semantics

represent the result of this call.

EndService terminates a request execution. The produced value v is stored as the result for future f, inside the computed results.

ServeNext serves the next request, in a FIFO order. The request is dequeued.

RcvResultPrim receives a future value, and replaces all references to the future by the new value.

B Index of Notation

Summary of symbols and operations	
$\#$	List Append Operator
$\setminus$	List remove operator
$[a \mapsto b]$	Mapping from a to b
$\uparrow$	Find operator for component list
$\leftarrow$	Replace operator for component list
f	Future: $f ::= (id, N)$
v	Value: $v ::= \{V, f_i\}^{i \in 1..n}$
(V_f, f_0)	Special value indicating value only contains future
R	Request: $R ::= (f,\ v,\ itf)$
s	Component state: $s ::= (CompState \mid PrimState)$
$itfs$	List of component interfaces
subCp	List of subcomponents:$subCp\ ::=\ \{C_i\}^{i \in 1..n}$
queue	Request queue of current state:$queue ::= [R_i]^{i \in 1..n}$
results	Computed results: $results ::= [f_i \mapsto v_i]^{i \in 1..n}$
FRL	Future recipient list: $FRL ::= [f_i \mapsto \{N_j\}^{j \in 1..n}]^{i \in 1..n}$
$currReq$	Lists of requests beings served currently: $currReq ::= [f_i]^{i \in 1..n}$
$refF$	List of referenced futures in current state $refF ::= [f_i]^{i \in 1..n}$
$Enqueue(C, R)$	Enqueues the request R at C, returns a new component
RL	List of registrations to be made: $RL ::= [f_i \mapsto \{N_i\}^{i \in 1..n}]$
$RqIdsSet(S)$	All request id's referenced in S
$RefFutSet(S)$	All futures referenced from S and its subcomponents recursively
$host(f)$	Returns the name of component which will compute the result for this future. $host(f) \triangleq snd(f)$
$cpSet(C)$	Returns the set containing C and all it's sub-components recursively
$cpList(C)$	Returns a list containing C and all it's sub-components recursively
$removeResult(f,\ C,\ N)$	Update the component such that all references to f are removed
$updateFV(v, f, v')$	Update the future f by replacing old value v with v'
$registerListFutures(S,RL)$	Register all entries in RL and return new component system S'
`GlobalRegisteredFuturesComp`	Function to check if all futures are registered
`CorrectComponent`	Correctness rules (constraints) for components