Component Service Promotion: Contracts, Mechanisms and Safety

Pascal André, Gilles Ardourel and Mohamed Messabihi

AeLoS Team - LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière F-44322 Nantes Cedex, France
{FirstName.LastName}@univ-nantes.fr

Abstract. Composition is a core concept of component and service-based models. In hierarchical component composition, promotion is used to make services available at a higher level of the hierarchy without breaking encapsulation. In this article we will study different kinds of promotion of services equipped with contracts, their usefulness, as well as their safety by considering appropriate proof obligations. We introduce several explicit assertion constructs in order to reduce the proof effort. We study the impact of encapsulation and rich state description on these promotions. We illustrate the approach (specification and verification) with the Kmelia component language.

1 Introduction

Composition is a core concept of component models. In hierarchical component composition, *promotion* is used to make an entity available at a higher level of the hierarchy. The exposed entity may vary from one model to another: it can be a port, an interface or a service. In UML2 related component models such as Palladio, KobrA, Java/A [10], the exposed entity is a *port* and the promotion is obtained by a delegation connector which usually does not modify the sub-component features. The approach is similar in BIP [13] or in ADLs like Darwin and Unicon which model architectures as composite components [15]. Sofa [8] and Fractal [7] are component models that expose *interfaces*. Promotion is achieved by special interface bindings (delegate/subsume). Again the promoted interfaces usually remain unchanged except that connectors can be redefined. Last, *service* promotion can be seen as a special kind of service composition [14,12]. In this article we deal with this last category.

The motivation of our approach is to build correct components and composites considering both bottom-up and top-down construction approaches. The two guiding principles are abstraction and encapsulation: components are black boxes in horizontal composition (assembly) and vertical composition (composite). Therefore we aim at avoiding, detecting or correcting errors when assembling components and promoting services at the composite level. In previous works [2,3] we set the basis of building correct components and assemblies. The verification of correctness is done by establishing different proof obligations.

In this article we tackle the issue of the correct vertical composition and especially the problem of correct promotion. We show that promotion can be more flexible than it

appears while still being safe. We present a verification method and show that the proof effort can be reduced by using explicit predicate constructs and capitalising on already-proven properties. We describe a model and a process that support scalability through the abstraction task that makes an assembly be seen as a component: we want to avoid the flattening of composites for both the specification and the verification tasks. As in our previous works, we propose a methodology equipped with tools which is applied in the context of the Kmelia component model [2].

The article is structured as follows. Section 2 introduces a general component model and the promotion correctness obligation is sketched. In Section 3 we give a classification of promotions in which we describe their uses and we distinguish between safe and unsafe promotions. The Section 4 is dedicated to the correctness verification of the kinds of promotion previously described, introducing predicate operators and illustrating their role in the proofs with a specific component language (Kmelia). In Section 5, we study the impact of encapsulation on the promotions. Section 6 is dedicated to related works, and finally Section 7 concludes the article.

2 Hierarchical Composition and Service Promotion

This section introduces a component model equipped with an internal composition operator that allows promotion of services. We then present promotion correctness based on the respect of the Client-Supplier Contract.

2.1 Simple Model and Example

We consider a component model that separates component interface from component body and enables component composition. Each component defines a state space (using typed variables V and invariant properties Inv) and services. A composite is a component that encapsulates other components (called sub-components). We assume here that the component interface exposes services which therefore can be promoted at the composite level. We distinguish between the *provided* services (server point of view) and the *required* services (client point of view) of a component. A service refers to a software functionality, defined here by a signature and pre/post-conditions which are predicates over the state space variables and parameters. The state space of a required service is virtual, vV are the assumed variables.

Figure 1 illustrates the formalism used in this article. The boxes denote components and the grey (resp. white) "funnel" denotes provided (resp. required) services. Three components are considered: a composite cc that includes two components ocA and ocB[1]. ocA provides a service provServ1 that is required by service reqServ1 of component ocB: an assembly link binds both services. ocB provides a service origProvServ, which is promoted at the composite cc level by promoProvServ: a promotion link binds both services.

Service promotion is usually associated with both the operations of making a service available at the composite level and adding it in the composite's interface. In this article

[1] which can be themselves composites.

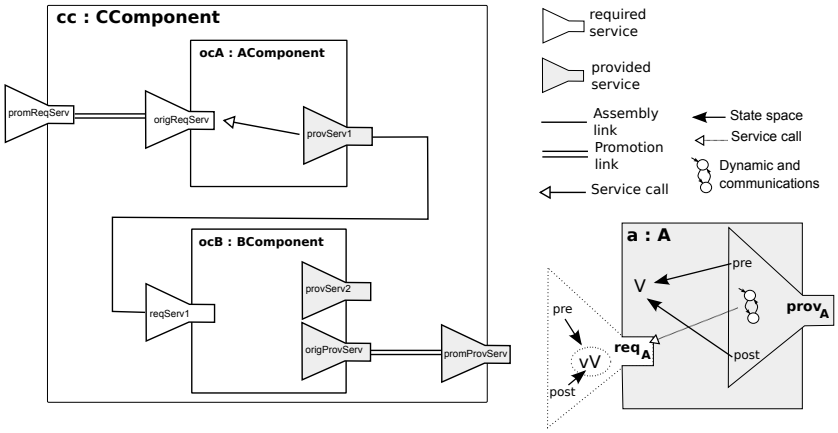

Fig. 1. Component and Composite component

we call promotion the former operation. The latter is only considered when explicitly mentioned.

2.2 Promotion Correctness

We study here the case of the promotion of a service *origin* provided by a component C to a service *promoted* of a composite CC. The contract of the service *origin* ensures that under the pre-condition Pre_{origin} of *origin* and the invariant inv_C of C, the service *origin* will satisfy its post-condition $Post_{origin}$ and preserve the invariant inv_C. Considering the component state $before$ and $after$ values, we express this contract as follows:

Service Client-Supplier Contract
$Pre_{origin}(before, param) \wedge inv_C(before) \Rightarrow Post_{origin}(after, result) \wedge inv_C(after)$.

In such a contract, the pre-condition should be established by the caller of the service (the client), while the post-condition and the invariant are established by the callee (the supplier). Since we only consider promotion in this article, we assume that the above contract holds for the service *origin*[2]. We also assume that the actual service behaviour is consistent with its contract, we ignore the possibility that the service has been underspecified and we do not try to make its pre/post-conditions more precise. Under these hypotheses, the promotion of *origin* to *promoted* is considered to be correct if the *service client-supplier contract* holds for *promoted*.

Whether using a top down or a bottom up approach, the designer of a composite has to select from the set of available components which ones will be useful for making its composite component. The services available in these components were not built with a specific composite designer's needs, even if they fit its purpose, they can range from being quite generic operations with weak pre/post-conditions, to very specialised services with strong and precise pre/post-conditions.

[2] In [2] we showed how to prove such a contract in the Kmelia specification language.

The composite designer has multiple and sometimes conflicting goals:

1. safety, which requires to prove the composition of components to be safe,
2. scalability, which requires higher level abstractions and hiding the subcomponent structure and services,
3. flexibility, which requires taking into account future changes in the composite (for instance when a subcomponent is expected to be replaced by a better one),
4. adaptation to the environment: the services should be tailored to the environment the component is targeting, that might require uniform interfaces (which also enhance readability).

In order to acheive safety in the simplest way, most component models leave pre and post-conditions unchanged when promoting a service. However, it is necessary to change them to take the all the above goals into account. In the following, we will see which changes can be safely applied.

2.3 Changing Predicates

During promotion, the predicates (pre/post-conditions) can be either

- weakened ($Predicate_{origin} \Rightarrow Predicate_{promoted}$),
- unchanged ($Predicate_{origin} \Leftrightarrow Predicate_{promoted}$),
- or strengthened ($Predicate_{promoted} \Rightarrow Predicate_{origin}$).

If none of these possibilities holds, then a part of the *promoted* predicate is not related to the *origin* predicates, and nothing can be concluded except that the predicate makes no sense and is considered to be incorrect.

The signature of a service can be seen as a part of a service's contract: the parameter types are part of the pre-condition and the return type is part of the post-condition. Since being of type T implies being of its super-type U, we can see that using a subtype in a predicate is strengthening it while using a super-type is weakening it.

3 Small Classification of Service Promotion

We study here different combinations: weakening, strengthening and unchanging, in order to capture their intent and to discuss their safety. At first sight the safe changes look like a contravariance policy in the sense of the type systems: restricting the conditions on the parameters and the component state (before) and expanding the conditions of the result and the component state (after) will always be safe. Actually it could work for provided service but we target also covariant cases for the postcondition. Conversely the cases for required services are neither covariant nor contravariant.

3.1 Provided Service Promotion

Table 1 summarises the different changes of predicates during the promotion of a provided service and their safety.

– *Weakening pre-condition* is unsafe in the general case because it allows to break the Service Contract. A potential caller may invoke the *promoted* service in situations where the *origin* service can not ensure its post-condition.
– *Strengthening pre-condition* ensures that the original pre-condition will hold.
– *Weakening post-condition* is safe because the original service will then ensure more than it needs.
– *Strengthening post-condition* is unsafe in the general case. However, this part of the contract being the responsibility of the specifier of *promoted*, there are some cases in which it can put constraints on the execution context of *origin* in order to ensure it. We will characterise these cases in the following.

	Weakened Pre	Unchanged Pre	Strengthened Pre
Weakened Post	Unsafe	Safe	Safe
Unchanged Post	Unsafe	Safe	Safe
Strengthened Post	Unsafe	Generally Unsafe	Generally Unsafe

Table 1. Modifications of predicates when promoting a provided service

Out of the nine combinations derivable from these cases, we will not consider the cases involving the weakening of a pre-condition since they definitely cannot be proven consistent in the context of the component *C* alone.

Safe Kinds of Provided Service Promotion The promotion kinds in this category are always safe provided that the contract at the composite level has been proven (this is discussed in Section 4.2).

1. *Keeping both the pre- and post-conditions* is the standard promotion that preserves the original contracts.
2. *Keeping the pre-condition and weakening the post-condition* is useful when the composite does not need as much precision in the type of a result as offered by the original service. Having a weaker post-condition will allow for easier future service substitutions. From a methodological point of view, it is related to information hiding, in the sense that it hides from the user, specific informations that are not deemed relevant in order to ease internal changes.
3. *Strengthening the pre-condition and keeping the post-condition* is useful when the composite *CC* is designed to evolve in an environment which is more constrained in terms of datatypes (e.g. subtyping, domain restriction...) than the more generic components it contains. In such cases, strengthening the pre-condition to match those of the other services makes a more consistent interface and allows for substitutions with more strict services.
4. *Strengthening the pre-condition and weakening the post-condition* combines the previous goals and uses.

Generally Unsafe Promotion Kinds The promotion kinds in this category are unsafe in the general case. However, some cases exist where they are both safe and useful.

1. *Strengthening both the pre/post-conditions: "strengthening by parameters"* is safe if we can prove that the restriction on the pre-condition implies the promoted post-condition. This case is illustrated in Section 4. While unlikely, the post-condition could also be strengthened by the context (see below).
2. *Keeping the pre-condition and strengthening the post-condition: "strengthening by context"* is safe only when the usage of the components in the composite strengthen the invariant on the state in such a way that it also strengthen a post condition depending on this state. This kind of promotion can only arise using a component model that supports state observability (reported in section 5). Furthermore, the context can only be restricted if the restriction is done before the invocation of the promoted service, otherwise the component invariant can not be strengthened. This means that the strengthening by context can only be used if a protocol (*i.e.*a user guide) or a specific initialisation of the sub-component is used.

We can illustrate the last case by considering a variant of the previous one where the constrained parameter is replaced by a state which is assigned a value either at the initialisation of the component by the composite (if it is supported by the component model) or by using a service which is called before the service *origin* (this can be ensured either by a protocol on the component *C* or by a protocol on the composite).

We saw earlier that a service's signature contains both a pre- and a post-condition. However, the post-condition strengthening will be more rare and harder to prove because type dependency is seldom expressed in post-conditions. To express it one can use either anchor types or an explicit return of a parameter by reference modified by a service; the former requiring some kind of clone operation and the latter being rather unorthodox.

3.2 Required Service Promotion

In the specification of a required service, the pre-condition states what the potential callers inside the component will ensure upon calling the service which will achieve the required service. Conversely, the post-condition states what these callers expect from the required service. In this contract, the client is the caller of the required service (see the service call arrows in the right part of Fig. 1) and the supplier is the service that achieves it. This is the reverse situation of the one described in Table 1, with an additional restriction: the execution context in this case is in the behaviour of the callers and can not be constrained efficiently to enable generally unsafe situations.

1. *Keeping both the pre- and post-conditions* is the standard service delegation.
2. *Keeping the pre-condition and strengthening the post-condition* is requiring more than what was expected by the *origin* service. This can be done for uniformising the component interface.
3. *Weakening the pre-condition and keeping the post-condition* is promising less than the *origin* service. This can be done either for anticipating a change in *origin* service or because of information hiding.

4. *Weakening the pre-condition and strengthening the post-condition* accepts (2) and (3) together.

Table 2 summarises the different changes of assertions during the promotion of a required service and their safety.

	Weakened Pre	Unchanged Pre	Strengthened Pre
Weakened Post	Unsafe	Unsafe	Unsafe
Unchanged Post	Safe	Safe	Unsafe
Strengthened Post	Safe	Safe	Unsafe

Table 2. Modifications of assertions when promoting a required service

3.3 N-ary Service Promotion

In many component models, a set of $origin_i$ services can be promoted to a single *promoted* service. These kinds of promotion can have different semantics: shared, multiple, multi-cast, gather-cast [4]. However, the consequences on the predicates are quite simple since the client service contract has to be verified for each couple ($origin_i$, *promoted*).
For provided services:

- the pre-condition of *promoted* should imply the strongest of the $origin_i$ pre-conditions (if any) or their conjunction (if satisfiable),
- the post-condition of *promoted* should be implied by the weakest of the $origin_i$ post-conditions (if any) or their disjunction.

For required services:

- the pre-condition of *promoted* should be implied by the weakest of the $origin_i$ pre-conditions (if any) or their disjunction,
- the post-condition of *promoted* should imply the strongest of the $origin_i$ post-conditions (if any) or their conjunction (if satisfiable).

4 Verification Methodology of Promotion Correction

We illustrate promotion cases defined in the previous section on a stock Management case study. The specification language is Kmelia [2]. The proofs of correctness are experimented with Atelier B[3] one of the tool support of the B method [1].

The B method is a general purpose proof-based formal method; its input formalisms is based on set-theory and first order logic. An *abstract machine* made of a state space

[3] `http://www.aterlierb.eu`

(which is described by an invariant) and operations, is the unit of B specifications. The B method generates proof obligations to establish the consistency of abstract machines. Refinement of abstract machines to executable codes may also be considered with the B method.

Kmelia is an abstract formal component model dedicated to the specification and development of correct components. A Kmelia *component* is a container of services; it has a state space constrained by an invariant. A *service* is more than a simple operation; it has pre/post-conditions and a behaviour described with a labelled transition system (LTS). A service can be composed from other services but this feature and the LTS are not used in this article. An *assembly* is a set of components linked via their required and provided services with the aim to build effective functionality. A *composite component* is a component that encapsulates an assembly. Kmelia is supported with an Eclipse-based analysis platform called COSTO.

4.1 An Example in Kmelia

The support example is a simplified *Stock Management* application. The system is designed by assembling two components: sm:StockManager and ve:Vendor. The former is the core business component to manage references and storage. The latter is the system access interface. The system specification is given in [2] and we go one step further in the specification by detailing the StockManager. Figure 2 shows that the composite component is made of three components: stock and catalog are instances of a specialised COTS[4] Dictionary, and m an instance of Manager is a controller of the composite. Most of the promoted services come from m.

Listing 1 gives the specification of the StockManager composite. The first part describes the composite as a component while the **COMPOSITION** clause describes the encapsulated assembly (its components and assembly links). The From keyword denotes a promotion link for state variables or services. The ***obs*** keyword characterises observable features, according to the rules given in Section 5. The promotion store−addToEntry is a generally unsafe situation of change: only natural numbers are possible.

Listing 1. Kmelia specification StockManager composite

```
COMPONENT StockManager
INTERFACE
  provides : {newReference, deleteReference, store, order}
  requires : {authorisation}
USES {STOCKLIB}
VARIABLES
  obs catalog From m.keys;
  plabels From catalog.values; //product description
  pstock  From stock.values   //product quantity
INVARIANT
  obs @borned: size(catalog)<=maxRef,
  @referenced: forall ref : Reference | includes(catalog,ref) implies
     (plabels[ref] <> emptyString and plabels[ref] <> noQuantity),
  @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
     (plabels[ref] = emptyString and plabels[ref] = noQuantity)
SERVICES
```

[4] component of-the-shelf

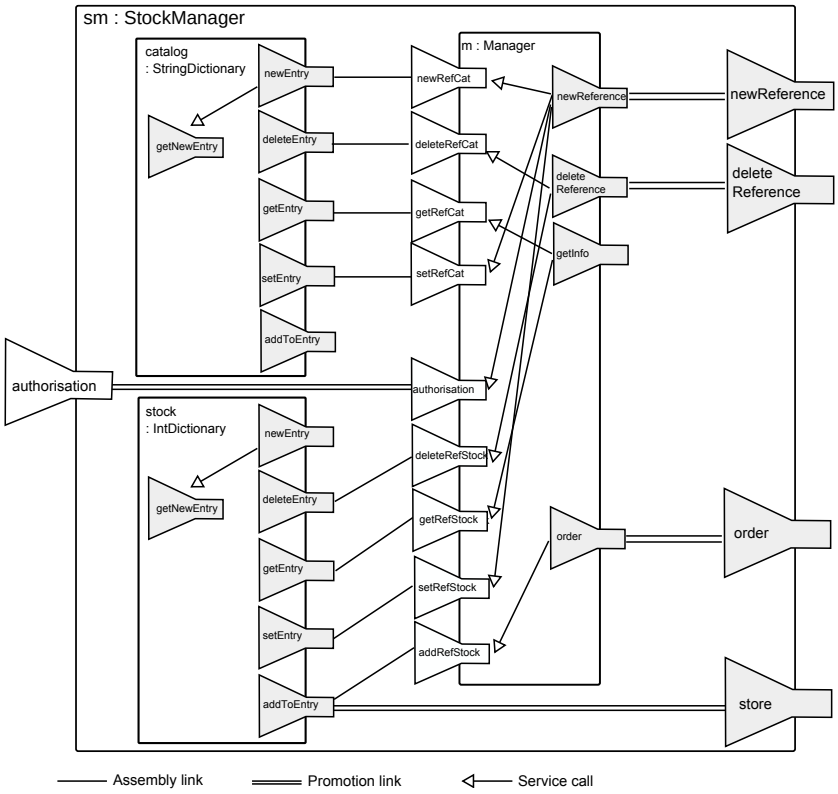

Fig. 2. Stock Manager composite component

```
########### promoted provided services
provided newReference From m.newReference
End
provided deleteReference From m.deleteReference
End
provided order From m.order
End
provided store(pid : Integer; pqty : Integer) : Integer From stock.addToEntry
  RedefinedPre Stronger With (0 < pqty)
  RedefinedPost Stronger With pstock[pid] > old(pstock)[pid]
End
########### required services (partial description)
required authorisation() From m.authorisation
End
END_SERVICES

COMPOSITION
  Assembly
    Components
      catalog : StringDictionary;
      stock : IntDictionary;
      m : Manager;
    Links //////////assembly links/////////
      @nrc: m.newRefCat catalog.newEntry
      @drc: m.deleteRefCat catalog.deleteEntry
```

```
      @grc: m.getRefCat catalog.getEntry
      @src: m.setRefCat catalog.setEntry
      @drs: m.deleteRefStock stock.deleteEntry
      @grs: m.getRefStock stock.getEntry
      @srs: m.setRefStock stock.setEntry
      @ars: m.addRefStock stock.addToEntry
  End // assembly
END_COMPOSITION
```

During the edition of the specification with the COSTO platform, the first steps of analysis are performed on the specification: syntax, type-checking, structure and visibility well-formedness. The proofs on contracts are performed using B tools [16,3]. First, B machines are extracted for each kind of property we target to check, this extraction step is followed by processing proof obligations in AtelierB. In the following we explain the properties to be checked which are related to promotion contracts, and the way we prove them using AtelierB.

4.2 Explicit Promotion Operators

A promotion is *correct* if we can prove the *Client Service Contract* for the *promoted* service. For some kinds of promotion, we can capitalize on the proof of the service contract of *origin* and simplify greatly the problem of promotion correction. However, knowing which kind of promotion is being used in the general case is more difficult than the correction problem itself. In order to make the intent explicit, we propose to use predicate constructors to strengthen or to weaken an *origin* predicate.

Here are the predicate constructors in the form of keywords WEAKER, STRONGER, WITH preceding a predicate.

- WEAKER $Predicate_{promoted}$,
  Introduces $Predicate_{promoted}$ as a weakened form of the original one.
  The associated proof obligation is: $(Predicate_{origin} \Rightarrow Predicate_{promoted})$.
- WEAKER WITH $PredicateW_{promoted}$,
  Constructs $Predicate_{promoted}$ as $Predicate_{origin} \vee PredicateW_{promoted}$
  The associated proof obligation: $(Predicate_{origin} \Rightarrow Predicate_{promoted})$ holds by construction. There is an optional (w.r.t. safety) proof obligation: $\neg\,(Predicate_{origin} \Rightarrow PredicateW_{promoted})$
- STRONGER $Predicate_{promoted}$,
  Introduces $Predicate_{promoted}$ as a strengthened form of the original one.
  The associated proof obligation is: $(Predicate_{promoted} \Rightarrow Predicate_{origin})$
- STRONGER WITH $PredicateW_{promoted}$,
  Constructs $Predicate_{promoted}$ as $Predicate_{origin} \wedge PredicateW_{promoted}$
  The associated proof obligation: $(Predicate_{promoted} \Rightarrow Predicate_{origin})$ holds by construction. There is an optional (w.r.t. safety) proof obligation: satisfiability of $Predicate_{promoted}$

Using these constructors captures the designer intent, reduces the proof effort (when the contract is satisfied by construction or when it is always unsafe) and in some cases allows to automatically detect errors without resorting to a proof assistant (*e.g.*weakening pre-conditions in a provided service, strengthening post-conditions which do not depend on constrained state or parameters). An additional case of WEAKER WITH can

also be used in models that supports predicate name by removing a specific part of a conjunction predicate, which ensures the weakening by construction.

Table 3 summarises the impact of different changes of predicates during the promotion of a provided service and their safety.

Pre	Weakened	Strengthened		Unchanged
Post			With	
Unchanged Post	Unsafe	Safe	Proven Safe	Proven Safe
Weakened Post With	Unsafe	Safe	Proven Safe	Proven Safe
Weakened Post	Unsafe	Safe	Safe	Safe
Strengthened Post	Unsafe	Generally Unsafe	Generally Unsafe	Generally Unsafe

Table 3. Modifications of predicates when promoting a provided service

4.3 Formal Analysis of Service Promotion Correctness

In this section, we rely on the constructors that have been defined in the previous section to characterise the proof obligations in Kmelia. We also show how each of them is proven using Atelier B.

1. **Unsafe case:** the specifier is notified immediately that the promotion is not correct without any proof.
2. **Proven safe case:** the intent of the specifier is captured by strengthening/weakening operators and the proof obligations are satisfied by construction.
3. **Safe case:** as in the previous case, the specifier's intent is captured, but to be sure that the promotion is really safe, we must prove that the predicates (pre/post-conditions) match the intent (proving the strengthening or weakening).
4. **Generally unsafe case:** in this case we must prove that the new predicates satisfy the client contract.

The main idea behind our method for checking the correctness of component service promotion is the following: we encode a promoted service as a B machine, in such a way that the consistency proof of it establishes the correctness of the promoted service.

Practically, we generate the appropriate B specifications such that their proof obligations correspond to the client contract on the promoted pre/post-conditions. At this stage the sub-components are assumed to be proven correct.

Listing 2. B pattern for service promotion verification

```
MACHINE
   C_CC_origin_promoted
CONSTANTS
   /* ORIGINAL */
    /* C variables */
      C_v1, ...
    /* C variables updated values  */
      C_v1_new, ...
    /* parameters  of origin service */
```

```
      o_param1,... , o_result,
   /* PROMOTION */
    /* C variables promoted in CC */
      CC_v1, ...
    /* C variables promoted in CC, updated values */
      CC_v1_new, ...
    /* parameters  of promoted service */
       p_param1, ..., p_result
PROPERTIES
   /* ORIGINAL */
    /* C variables and origin parameters typing */
       C_v1∈T ∧ o_param1∈T ∧ ...
    /* C invariant related to the above C variables and their updated values */
       Inv_C ∧ Inv_C_new...
    /* postcondition of service origin */
       Post_origin ∧ ...
  /* PROMOTION */
   /* C variables promoted in CC and promoted parameters typing */
       CC_v1∈T ∧ p_param1∈T ∧ ...
   /* promotion variable mapping */
       CC_v1=C_v1 ∧ ...
   /* parameter mapping */
       p_result=o_result ∧ ...
   /* precondition of promoted service - hypothese for the client contract */
       Pre_promoted ∧ ...
ASSERTIONS
   /*precondition of the origin service */
      Pre_origin ∧
   /*postcondition of the promoted service */
      Post_promoted
END
```

The **CONSTANTS** clause of the B machine contains variables C_v1,... of the C base component, parameters o_param1,... of the *origin* service and its result variable o_result. It also contains their promoted version, prefixed by CC_ for variables and p_ for parameters and the result variable of the promoted service.

The **PROPERTIES** clause contains typing information for every variable and parameters involved (C_v1∈T ∧ o_param1∈T ∧ ...), mapping predicates for the promoted variables (CC_v1=C_v1 ∧ ...), and the following predicates which are used as axioms:

- the promoted pre-condition Pre_promoted which is supposed satisfied by the client,
- the base component invariant Inv_C ∧ Inv_C_new... which has already been proven,
- the original post-condition Post_origin which is considered satisfied if the original pre-condition holds.

The **ASSERTIONS** clause contains the original pre-condition Pre_origin and the promoted post-condition Post_promoted.

For each promotion of a provided origin to a promoted, we build a B machine as shown in Listing 2. Indeed, for a B machine with properties P_B, invariant I_B and assertions A_B, the B method generates the following proof obligation: $P_B \wedge I_B \Rightarrow A_B$.

4.4 Experimental Results

This section presents the experimentations led on the example of Section 4.1. Once edited and verified with COSTO, the specification is visited to extract one B machine for each service promotion. First the sub-component correctness is checked independently with the principles given in [2,16]. Then the B machines are extracted from the Kmelia

specifications. For this experimentation the extraction was done manually, but we developed B extraction plugins to prove component and assembly correctness [16,3]. The full Kmelia specifications and B machines are given in appendix.

The B machine addToEntry_Store is used to prove the correctness of the Store service in StockManager, which results from the promotion of addToEntry from IntDictionary. The machine contents instantiates the pattern of Listing 2.

Listing 3. StockManager_addToEntry_store extracted B machine

```
MACHINE
    StockManager_addToEntry_store
CONCRETE_CONSTANTS
    /* origin variables */
        o_keys ,
        o_values ,
        o_result ,
        o_noValue ,
    /* promoted variables */
        p_catalog ,
        p_labels ,
        p_stock ,
        p_result ,
    /* origin parameters */
        o_key ,
        o_value ,
    /* promoted parameters */
        p_id ,
        p_qty ,
    /* updated variables*/
        new_o_values ,
        new_p_stock
PROPERTIES
    /* origin typing */
        o_keys ⊆ 1 .. 100 ∧
        o_values ∈ 1 .. 100 → INT ∧
        o_result ∈ INT ∧
        o_noValue ∈ INT ∧
        o_noValue = - 1 ∧
    /* promoted typing */
        p_catalog ⊆ 1 .. 100 ∧
        p_labels ∈ 1 .. 100 → INT ∧
        p_stock ∈ 1 .. 100 → INT ∧
        p_result ∈ INT ∧
    /* origin parameters typing */
        o_key ∈ INT ∧
        o_value ∈ INT ∧
    /* promoted parameters typing */
        p_id ∈ INT ∧
        p_qty ∈ INT ∧
    /* updates variables typing */
        new_o_values ∈ 1 .. 100 → INT ∧
        new_p_stock ∈ 1 .. 100 → INT ∧
    /* origin invariant already proved */
        card ( o_keys ) ≤ 100 ∧
    /* origin post */
        o_result = o_key ∧
        new_o_values(o_key) = o_values(o_key)+
                                    o_value ∧
    /* maping predicat */
        p_result = o_result ∧
        p_catalog = o_keys ∧
        p_stock = o_values ∧
        p_id = o_key ∧
        o_value = p_qty ∧
        new_o_values = new_p_stock ∧
    /* pre promoted */
        ( p_id ∈ p_catalog ∧ p_stock ( p_id ) +
                    p_qty ≥ 0 ∧ p_qty > 0 )

ASSERTIONS
    /*pre origin*/
        o_key ∈ o_keys ∧ o_values ( o_key ) +
                        o_value ≥ 0 ∧
    /*promoted post*/
        p_result = p_id ∧
        p_catalog = p_catalog ∪ { p_id } ∧
        new_p_stock(p_id) = p_stock(p_id)+p_qty ∧
        new_p_stock ( p_id ) > p_stock ( p_id )
END
```

The analysis of this machine generated 6 proof obligations which were all discharged by the AtelierB prover in the *Automatic force (1)* mode. Additionally, in order to better illustrate our approach, we intentionally introduced the following errors in the promotion contracts:

- The pre-condition of promoted service was weakened by deleting the predicate pid $>= 10$ from the original one. In this case, regardless of the post-condition there are still proof obligations that are not discharged by AtelierB prover. This corresponds to the column "Unsafe" in Table 1.

– Keeping the pre-condition unchanged, we have not introduced other restrictions in the invariant of the composite (restriction by context). The AtelierB could not prove the strengthened post-condition of the promoted service because we have no guarantee that pqty $>$ 0 establishes the predicate pstock[pid] $<$ old(pstock)[pid]. This is the first case of "Generally Unsafe" in Table 1. This simple case could have been detected without launching the prover.
– Strengthening both the pre-condition with pqty $>$ 0 and the post-condition with pstock[pid] $<$ old(pstock)[pid] let the new post-condition unsatisfied. Hence the AtelierB proof could not succeed.

The contract expressed by the B pattern in Listing 2 corresponds to the general case of the client service contract for provided services. It contains the proof of pre-conditions strengthening. However, the safety property can be proven without consistency of the designer declarations: a weakened post-condition could have been introduced using a STRONGER keyword. This case and the reverse one (less likely to have ensured the safety proof) requires an additional proof.

5 Encapsulation and Observability Rules

In this section we study the impact of state observability on promotion safety. State observability in component models is achieved using either observable variables, accessing methods or attribute controllers. Abstraction and encapsulation are crucial to the scalability of the component approach. To an outside observer, a composite component should not be distinguished from a primitive component and should not be overly complex. Promoting all the services and variables of its sub-components would run contrary to this goal. When deciding what is observable or what is promoted, the designer makes a trade-off between encapsulation and a precise state description. From the verification point of view, the goal is to achieve both abstraction and capitalisation on previous proofs. In the following we note V^O the **observable** subset of the state variables V.

5.1 Observability of Predicates

Predicates containing non-observable state variables are of no use to potential clients of the component. Consequently, we distinguish between observable predicates which contain only variables in V^O from non-observable predicates which can take their variables in V. The observable predicates Inv^O and the non-observable predicates Inv^{NO} form a partition of the invariant Inv. The decision to make a predicate observable is subject to the following guideline (gl) and two rules (r1, r2):

gl. To get observable predicates as meaningful as possible, there must be few (preferably none) non-observable predicates depending only on observable variables.
r1. The pre-condition of a provided service must not contain non-observable predicates because it would make for an unfair contract: the client would have no way to know if the pre-condition is satisfied.

r2. Conversely, the post-condition of a required service, which expresses what is expected, cannot contain non-observable predicates.

More details about the verification of invariants and well-formedness of predicates w.r.t. observability can be found in [2,3]. In Kmelia, one can define a calculated variable (*e.g.*defining *isEmpty* as $size = 0$). This abstraction mechanism can be used to simplify predicates. If the initial variable is no longer present in observable predicates, it can be safely removed from V^O. This abstraction mechanism is even more useful when building a composite component, since only a subset of the predicates are actually used.

5.2 Variable Promotion

The composite invariant and the pre/post-conditions of its services might depend on the observable variables of its sub-components. In order to preserve encapsulation, these variables have to be seen as variables of the composite (otherwise it would expose the sub-components). In Kmelia the operation of *variable promotion* allows one to define special composite variables that link to observable variables of sub-components.

State variables promotion. An observable variable *vo* from a sub-component $c : C$ can be promoted as a variable *vp* of a composite component (the syntax for that is: vp FROM c.vo). The promoted variables retain their types and are abstractions of the read-only access to the *vo* in their effective contexts.

The consequences of the observability rules for the provided services are:

- Observable variables in Pre_{origin} must be observable by the clients of $promoted$; removing them in the predicate or marking them as non observable would be equivalent as a weakening of the pre-condition, which is unsafe and forbidden.
- Observable variables in $Post_{origin}$ can optionally be made observable in the composite. If they are not, it can be a weakening of the post-condition in the spirit of information hiding.

5.3 Invariant Promotion

In order to qualify correctly the promoted variables, the corresponding observable invariants are to be promoted too. If these predicates make use of non-promoted variables, the designer will face again a trade-off between abstraction (dropping the predicates) and precise qualification of the variables (promoting the variables needed). In the following we call *reachable properties* of a promoted variable the subset of the predicates in the observable invariant of its origin that depends only on promoted variables. The promoted invariant is usually a weakened version of the original one when all variables are not promoted. However, constraints in the initialisation of the sub-components can strengthen it.

5.4 Consequences on the Promotion Process

The rules governing observability and promotion dependencies are quite simple and easy to integrate in a semi-automated process sketched as follows:

1. determining the sub-components of the composite;
2. selecting the services to promote, and automatically promote the variables needed and their reachable properties:
   (a) optionally modifying the services pre/post-conditions,
   (b) updating needed variables according to the new predicates;
   (c) automatically checking the proof feasibility if applicable (see table 3),
3. optionally selecting additional promoted variables and recompute their reachable properties;
4. optionally defining calculated variables and abstracting existing predicates;
5. assigning variables, services and predicate observability according to the rules previously defined through automatic guidance and verification;
6. checking the proof feasibility and generating proof obligations if applicable using the method described in Section 4.

This iterative promotion process stresses the use of encapsulation and abstraction. The verification method remains unchanged but the extraction process takes the selection of predicates and variables into account. The automatic computation of reachable properties and the observability rules guide the designer in establishing the trade-off between encapsulation and a precise state description before proving the promotion correctness.

6 Related Work

The work presented in this article covers a series of topics including aggregation, composition, promotion, sharing, and contracts in the area of components and services. In the following we focus on the approaches which use a notion of contract.

Contracts are helpful to deliver trusted rich information. Contracts for component as described in [18] consider functional and extra-functional contracts (including dynamic behaviours) to provide trust-by-contract components. However the proof of the contracts is not treated at the design level.

Beugnard et al. [5] have investigated types of component contracts and have classified contracts into four levels. *Syntactic contracts (i)*, semantic constraints such as *behavioural contracts (ii)* and *synchronisation contracts (iii)* are encountered in various component models; and finally *quality of service (iv)* which is often used at runtime. Brogi [6] proposed four levels of service contracts : signature, quality of service, ontology for data and protocols. In summary, the word "contract" or "behavioural contract" is often overloaded. We distinguish here functional contracts from synchronisation contracts. In the former category *e.g.* [17,13], the pre/post-conditions are interpreted in terms of call sequences and (observational) equivalence rather than in logical predicates.The latter category is related to the definition of Meyer's contracts and subcontracts, it is the subject of the remaining of the section.

ConFract [9] inherits from Fractal's interface delegation (promotion) support. ConFract contracts are mainly used for composite components and they are located in Fractal membranes. ConFract contracts are independent entities which are associated to several participants of the composite (external and internal views), and support a rely/guarantee mechanism. ConFract uses the executable assertions language CCL-J to

express specifications at interface and component levels. In the case of CCL-J, when a method is called on an interface, the contract controller is then notified and it applies the checking rules. Pre-conditions, post-conditions and method invariants of all contracts "are checked at runtime". A ConFract contract may cover several interfaces and it does not handle promotion, *i.e.* the interface may change by promotion. In Kmelia the promotion contract must conform to the original one and therefore we reuse the proof efforts made on sub-components.

The Service Component Architecture [11,14] is a set of specifications which describes a model for building applications using a Service-Oriented Architecture. The approach, as formalised in [11] is very similar to Kmelia when ports are services but contracts are not handled. A promotion mechanism is proposed to make services visible at the composite level but no transformation is permitted.

Hidden dependencies [12] is another formalism close to Kmelia: it supports contracts for provided and required services and it handles four types of service dependencies. But unlike Kmelia the dynamic compatibility is weak and services are flat operations supporting subtyping. Promotion is defined by delegation dependencies, the promoted services are said to be visible while the original ones are said to be hidden. A fixed-point equation solves the dependency and contract rules. The contents of the dependency relation is not given, and promotion links seem to be governed by an $\wedge$-rule rather than an $\Rightarrow$-rule. No verification support is provided yet.

We compared functional contracts and verification with formal methods in [3]. Therefore, compared with existing works, our approach contributes at the level of correct-by-construction composite components and also at the level of the consistency of component assemblies. Considering component consistency against the services, we handle visibility, encapsulation and personalisation by considering the distinction between observable/local conditions. Considering assembly consistency, we are close to the case of *plugin matching* of [19] with a special case of explicit required service definition.

7 Conclusion

In this article we presented the issue of the correct promotion of services in component models. We described several kinds of promotions, their usefulness and the conditions of their safety. We defined operators on predicates to be used during the promotion of a service to make explicit the intent of the designer and reduce the proof effort. In particular, using these operators allows to easily rule out promotion kinds in systematically unsafe situations, and automatically accept always safe promotions kinds, as well as defining simple heuristics to warn the designer about generally unsafe situations.

We presented and illustrated a verification method that capitalises on previously proven properties [2]. This method is based on the generation of B models from relevant parts of the Kmelia specifications and their analysis using B tools. We then described a model and a process that support scalability through abstraction while taking into account the verification needs.

The COSTO tool currently determine the always safe and always unsafe cases according to the keywords involved, but we aim at adding wizards to assist the specifier in following the process sketched in Section 5 and define simple heuristics for determin-

ing when calculated variables would favour abstraction. Other short term perspectives of this work include its full implementation in COSTO with a new B extraction plugin integrating the promotion language features introduced in this article, as well as a feedback analysis.

A medium term perspective is to extend the current primitives to behavioural contract promotion, as a follow-up of the work on the behavioural compatibility rules already defined in Kmelia.

Note: A separate appendix for the FACS 2010 article is available at the Kmelia website: `http://www.lina.sciences.univ-nantes.fr/coloss/download/facs10_app.pdf`.

References

1. Jean-Raymond Abrial. The B-Book Assigning Programs to Meanings. Cambridge University Press, 1996. ISBN 0-521-49619-5.
2. Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix. Using assertions to enhance the correctness of kmelia components and their assemblies. Electronic Notes in Theoretical Computer Science, 263:5 – 30, 2010. Proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS 2009).
3. P. André, G. Ardourel, C. Attiogbé, and A. Lanoix. Contract-based Verification of Kmelia Component Assemblies using Event-B. In Proceedings of the Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2010), 2010.
4. Françoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In CCGRID, pages 599–610. IEEE Computer Society, 2007.
5. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. Computer, 32(7):38–45, 1999.
6. Antonio Brogi. On the Potential Advantages of Exploiting Behavioural Information for Contract-based Service Discovery and Composition. Journal of Logic and Algebraic Programming, March 2010.
7. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. Software Practice and Experience, 36(11-12), 2006.
8. Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In SERA '06: Fourth IC on Software Engineering Research, Management and Applications, pages 40–48. IEEE Computer Society, 2006.
9. Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite Contract Enforcement in Hierarchical Component Systems. In Software Composition, 6th International Symposium (SC 2007), volume 4829, pages 18–33, 2007.
10. Ivica Crnkovic and Magnus Larsson. Component based software engineering - state of the art. Technical report, Mälardalen University, January 2000.
11. Zuohua Ding, Zhenbang Chen, and Jing Liu. A rigorous model of service component architecture. Electr. Notes Theor. Comput. Sci., 207:33–48, 2008.
12. Daniel Enselme, Gerard Florin, and Fabrice Legond-Aubry. Design by contract: analysis of hidden dependencies in component based application. Journal of Object Technology, 3(4):23–45, 2004.
13. Susanne Graf and Sophie Quinton. Contracts for bip: Hierarchical interaction models for compositional verification. In Proceedings of the 27th IFIP WG 6.1 international conference FORTE '07, pages 1–18. Springer-Verlag, 2007.
14. Bernd J. Krämer. Component meets service: what does the mongrel look like? ISSE, 4(4):385–394, 2008.

15. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, 26(1):70–93, january 2000.
16. Mohamed Messabihi, Pascal André, and Christian Attiogbé. Preuve de cohérence de composants Kmelia à l'aide de la méthode B. In 4ème Conférence Francophone sur les Architectures Logicielles, volume L-4 of RNTI, pages 113–126. Cépaduès-Éditions, 2010.
17. Ralf Reussner, Iman Poernomo, and Heinz W. Schmidt. Reasoning about Software Architectures with Contractually Specified Components. In Component-Based Software Quality, volume 2693 of LNCS, pages 287–325. Springer, 2003.
18. H. Schmidt. Trustworthy components-compositionality and prediction. J. Syst. Softw., 65(3):215–225, 2003.
19. A. M. Zaremski and J. M. Wing. Specification matching of software components. ACM Transaction on Software Engeniering Methodolology, 6(4):333–369, 1997.