Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies

Pascal Andre, Gilles Ardourel, Christian Attiogbé, Arnaud Lanoix

Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies

Pascal André, Gilles Ardourel, Christian Attiogbé,
Arnaud Lanoix

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

Abstract

The Kmelia component model is an abstract formal component model based on services. It is dedicated to the specification and development of correct components. This work enriches the Kmelia language to allow the description of data, expressions and assertions when specifying components and services. The objective is to enable the use of assertions in Kmelia in order to support expressive service descriptions, to support client/supplier contracts with pre/post-conditions, and to enhance formal analysis of component-based system. Assertions are used to perfom analysis of services, component assemblies and service compositions. We illustrate the work with the verification of consistency properties involving data at component and assembly levels.

Keywords: Component, Assembly, Datatype, Assertions, Property Verification

1 Introduction

The Kmelia component model [?] is an abstract formal component model dedicated to the specification and development of correct components. A formal component model is mandatory to check various kind of properties for component-based software systems: correctness, liveness, safety; to find components and services in libraries according to their formal requirements; to refine models or to generate codes.

The key concepts of the Kmelia model are services, component, component assembly and component composition. One important feature of the Kmelia model is the use of services as first class entities. A service has a state, a dynamic behaviour which may include communication actions, an interface made of required, provided and subservices. Component composition is based on the interaction between linked services which form a component assembly. This use of services constitutes a bridge to service oriented abstract models.

In [?] we introduced the syntax and semantics for the core model and language. It has been incrementally enriched later. We mainly focused on the dynamic aspects of composition: interaction compatibility in [?], component protocols with service composition in [?] and multipart interaction with synchronous communication and shared services in [?]. Following this incremental approach, we consider in this article an enrichment of the data and expressions in the kmelia model and its impact on the language syntax, its semantics and the verification of properties. Our guiding objective is twofold: 1) enable the definition of assertions (with invariant, pre/post conditions, and properties of services, components, and compositions), 2) to increase the expressiveness of the action statements so as to deal with real size case studies.

Assertions are useful (i) to define contracts on services; contracts increase the confidence in assembly correctness and they are a pertinent information when looking for candidates for a required service, (ii) to ensure the consistency of components respecting the invariant. The actions implement a functional part of the services which should then be proved to be consistent with the contracts. Therefore the correctness verification aspects of the Kmelia model is enhanced.

Motivations. Modelling real life systems requires the use of data types to handle states, actions and property descriptions. The state of the art shows that most of the abstract components models [?,?,?,?,?] focus mainly on the dynamic features. They enable various verifications of the interaction correctness but they lack expressiveness on the data types and do not provide assertions mechanisms and the related verification rules. As an example, in Wright the dynamic part based on CSP is largely detailed (specification and verification) while the data part is minor [?]. In [?] the data types are defined using algebraic specifications, which are convenient to marry with the symbolic model checking of state transition systems. But this model does not support contracts and assertions.

Contribution. In this work, we enrich the model with data and assertions at service and composition levels in order to deal with safe services, component consistency and assembly contracts. *First*, the Kmelia language is enriched with data and assertions so as to cover in an homogeneous way structural, dynamic and functional correctness with respect to assertions. *Second*, we deal with state space visibility and access through different levels of nested components; in addition to service promotion we define variable promotions and the related *access rules* from component state in *component compositions*. *Last*, feasibility of proving component correctness using the assertions is presented. We show how structural correctness is verified and how the associated properties are expressed with the new data language.

The article is structured as follows. Section **??** gives an overview of the Kmelia abstract model and introduces its new features. In Section **??** a working example is introduced to illustrate the use of data and assertions. The formal analysis issue is treated in Section **??**; we present various analysis to be performed and we focus on component consistency and on checking assembly links. Section **??** concludes the article and draws some discussions and perspectives.

2 The Kmelia Model and its new Features

This section recalls the main features of Kmelia. The core concepts are component, services, component assembly and composition [?]. Now, the Kmelia language allows the description of datatypes, expressions and first order logic predicates. We describe the Kmelia model, focusing on its new features.

2.1 Data types and expressions

To design the Kmelia data language, we have established a trade off between the desired expressiveness of our language and the verification concerns. We tried to encapsulate statements from other formal data languages such as Z, B, OCL or CASL, with the idea to reuse existing tool supports for checking syntax and properties, but this approach was not convincing due to expressiveness, syntax and semantics conflicts between the used languages. To avoid the separation of analysis tools and to work on the same abstract model, we advocate for an approach where both data and dynamic part are integrated in a unique Kmelia language. We enrich the Kmelia language by designing a small but expressive data language. This enables us to deal homogeneously with the expression of the properties related to the component level and to the composition level.

Basic types such as Integer, Boolean, Char, String with their usual operators and standard semantics are permitted. Abstract data types like record, enumeration, range and collection (arrays, sets) are allowed in Kmelia. User-defined record types are built over the above *basic* types. Specific types and functions may be defined and imported from libraries. A Kmelia *expression* is built with constants, variables and elementary expressions built with standard arithmetic and logical operators. An assignment is made of a variable at the left hand side and an expression at the right hand side.

Assertions (pre-/post-conditions and invariants) are first order logic *predicates*. In a post-condition of a service, the keyword old is used to distinguish the before and after variable states. This is close to OCL's **pre** or Eiffel's **old** keywords. Guards in the service behaviour (eLTS) are also predicates. All the assertions are governed by an observability policy described in Section **??**.

2.2 Components

A **component** is one element of a component type. A component is referenced with a variable typed using the component type; for example c:C where c is a variable and C a component type. The access to a state variable v of c is denoted c.v.

A **component type** C is a 9-tuple $\langle \mathcal{W}, Init, \mathcal{A}, \mathcal{N}, \mathcal{M}, \mathcal{I}, \mathcal{D}, \nu, \mathcal{CS} \rangle$ with:

- $\mathcal{W} = \langle T, V, type, Inv \rangle$ the state space where T is a set of types, V a set of variables, $type : V \to T$ the function that map variables to types and Inv an invariant defined on V.
- $Init$ the initialisation of the variables of V.
- $\mathcal{A}$ a finite set of elementary actions.
- $\mathcal{N}$ a finite set of service names. Let $\mathcal{N}^P$ (provided services) and $\mathcal{N}^R$ (required services) be two disjoint finite sets of names[1]: $\mathcal{N} = \mathcal{N}^P \uplus \mathcal{N}^R$.
- $\mathcal{M}$ a finite set of message names.
- $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ the component interface which is the union of two disjoint finite sets of names $\mathcal{I}^P$ and $\mathcal{I}^R$ such that $\mathcal{I}^P \subseteq \mathcal{N}^P \wedge \mathcal{I}^R \subseteq \mathcal{N}^R$.

[1] $\uplus$ denotes the disjoint union of sets

- $\mathcal{D}$ is the set of service descriptions; it includes the provided services ($\mathcal{D}^P$) and the required services ($\mathcal{D}^R$).
- $\nu : \mathcal{N} \to \mathcal{D}$ is the function mapping service names to service descriptions. Moreover there is a projection of the $\mathcal{N}$ partition on its image by ν:
$s \in \mathcal{N}^P \Rightarrow \nu(s) \in \mathcal{D}^P \wedge s \in \mathcal{N}^R \Rightarrow \nu(s) \in \mathcal{D}^R$
- $\mathcal{CS}$ is a set of constraints related to the services of the interface of C in order to control the usage of the services.

Observability of the component state. In order to allow a context-independent design and composition of components, we need the *observability* of component state and we precise the associated rules. Thus in addition to the public interface of a component, we propose its state to be observable by client services and by composite components, through a subset of the component state variables. Therefore the state variables (V) are split into V^O the subset of the **observable** variables and V^{NO} the subset of the non observable variables. The subsets form a partition of V. Particularly, pre-/post-conditions and the state invariant Inv are composed of an observable (Inv^O defined on V^O) and a non-observable part.

2.3 Services

The behaviour of a component relies on the behaviours of its services. A (sub-)service models a functionality *activated* by a call. A service may activate other services during its evolution. Due to dependencies between services and interaction between components, the actions of several activated services may interleave or synchronise. Only one action of an activated service may be observed at time. Formally a *service* s of a component type C[2] is defined by a 4-tuple $\langle \mathcal{IS}, l\mathcal{W}, lInit, \mathcal{B} \rangle$ with:

- The service interface $\mathcal{IS}$ is defined by a 6-tuple $\langle \sigma, \mu, v\mathcal{W}, Pre, Post, \mathcal{DI} \rangle$ where
  - · σ is the service signature $\langle name, param, ptype, res \rangle$ with $name \in \mathcal{N}$, $param$ a set of parameters, $ptype : param \to T$ the function mapping parameters to types and $res \in T$ the service result type;
  - · $v\mathcal{W} = \langle vT, vV, vtype, vInv \rangle$ is a *virtual state space* with vT a set of types, vV a set of variables, $vtype : vV \to vT$ the function mapping context variables to types and $vInv$ an invariant defined on vV;
  - · μ is a set of message signatures $\langle mname, mparam, mptype \rangle$ where $mname \in \mathcal{M}$, $mparam$ and $mptype$ are similar to those of the service signature;
  - · Pre is a pre-condition defined on the union ($\cup$) of the variables in V, vV, and $param$: $V \cup vV \cup param$;
  - · $Post$ is a post-condition defined on $V \cup vV \cup param \cup \{$ result $\}$;
  - · $\mathcal{DI}$ is the *service dependency*; it is composed by services on which the current service depends on. $\mathcal{DI}$ is a 4-tuple $\langle sub,\ cal,\ req,\ int \rangle$ of disjoint sets where $sub \subseteq \mathcal{N}^P$ (resp. $cal \subseteq \mathcal{N}^R$, $req \subseteq \mathcal{N}^R$, $int \subseteq \mathcal{N}^P$) contains the provided services names (resp. the ones required from the caller, the ones required from any component, the internal services) in the scope of s.

[2] and by extension a service of a component $c : C$

- $l\mathcal{W} = \langle lT, lV, ltype, lInv \rangle$ is the local state space where lT is a set of types, lV a set of local variables, $ltype : lV \rightarrow lT$ the function mapping local variables to types and $lInv$ a local state invariant defined on lV (mostly $lInv = true$).
- $lInit$ the initialisation of the variables of lV.
- The behaviour $\mathcal{B}$ of a service s is an *extended labelled transition system* (eLTS), detailed in [?,?,?]. A transition label is a combination of actions; it can be guarded. The actions are either *elementary actions* from $\mathcal{A}$ or *communication actions* (to call/to end a service, to send/to receive a message).

Virtual state spaces. As a required service is an abstraction of a service offered by another component, it is necessary to describe this “imaginary” component. We introduce the notion of a *virtual state space* $v\mathcal{W}$ in order to abstract a service from its definition context which is a component. For a *provided* service this virtual context is always empty.

Observability rules vs. service state space. Let s be a service of a component type C. The distinction between observable and non-observable variables of the component state space is revisited[3] according to the following table:

Service state space	Variables		Invariant	
	Observable part	Non-observable part	Observable part	Non-observable part
`Provided` s	V^O	V	Inv^O	Inv
`Required` s	vV	V	$vInv$	Inv

The pre-/post-conditions of s must respect the well-formedness rules related to the observable, non-observable and virtual contexts according to the following table:

Service Assertions scope	pre-condition		post-condition	
	Observable Pre^O	Non-observable Pre^{NO}	Observable $Post^O$	Non-observable $Post^{NO}$
`Provided` s	$V^O \cup param$	none	$V^O \cup param \cup \{$ result $\}$	$V \cup param \cup \{$ result $\}$
`Required` s	$vV \cup param$	$V \cup param$	$vV \cup param \cup \{$ result $\}$	none

The other cases not detailed in the table are summarised in Figure **??** which describes: an abstract view of the variables of a component, their scopes and the assertion scopes; it also depicts how these contexts are used in assembly and composition.

Fig. 1. State variables scope and assertion scope

[3] it is not a partition here because of the supplementary variables in *param* and *result*

The observable pre-/post-conditions will be used to check the assembly contracts and the promotion contracts. Non-observable pre-conditions (resp. post-conditions) are meaningless for a provided service (resp. required service) because they prevent safe assembly and promotion contracts. The non-observable pre-condition of a required service gives call conditions on the (caller) component state variables. The non-observable post-condition of a provided service should establish the non-observable part of the invariant.

The state space $l\mathcal{W}$ local to a service is used only in the service behaviour $\mathcal{B}$ but not used in the assertions.

2.4 Assembly and Composition

An *assembly* is a set of components that are linked (*horizontal composition*) through their services. An assembly is one element of an *assembly type*. An *assembly link* associates a required service to a provided one. Considering the rich interface of a Kmelia service (see **??**), we need an explicit matching mechanism, to link properly the 6-tuples defining given services; therefore, additionally to signatures and dependency (via sublinks) mapping we now define *context* and *message mappings*. When needed, message or service parameters re-ordering must be handled through adaptation mechanisms [**?**].

Assembly context and message mapping. Consider a required service sr of a component cr of type CR linked to a provided service sp of another component cp of type CP. The virtual state space variables (vV_{sr}) of sr must be "instantiated" using the *observable* variables of sp (V^O_{CP}) by a mapping (total) function $vmap : vV_{sr} \rightarrow exp(V^O_{CP})$ where $exp(X)$ denotes an expression over the variables of X.
Each message name of sr is mapped to a message name of sp by a mapping (total) function $mmap : mname_{sr} \rightarrow mname_{sp}$.

A *composition* is the encapsulation of an assembly into a component (the composite) where some features (variables and services) of the nested components can be promoted to the composite level. *Promotion links* are used to promote provided or required services. The mappings and rules are similar to the ones of assembly, they are not detailed here.

State variables promotion. An observable variable $vo \in V^O_C$ from a component $c : C$ can be promoted as a variable $vp \in V_{CP}$ of a composite component $cp : CP$. Formally, there are a bijection $prom : V^O_C \rightarrow V_{CP}$ which establishes the *variable promotion*, i.e. a bridge between the variable names. In the Kmelia syntax, $(vo, vp) \in prom$, is written vp FROM c.vo. The promoted variables retain their types ($type(vp) = type(vo)$) and are accessed (*read-only* at the composite level) in their effective contexts using a service of the sub-component that defines the variables. This guarantees the encapsulation principle.

Now Kmelia services are equipped with expressive means (pre-/post-conditions, observability, virtual context) to describe contracts. Section **??** illustrates them on a working example. They are used to check services and assemblies correctness as described in Section **??**.

3 A Working Example

The example is a simplified *Stock Management* application including a *vending* process as a main service. This process manages product references (catalog) and product storage (stock). Administrators have specific rights, they can add or remove references under some consistency business rules such as: *a new reference must not be in the catalog* or *a removable reference must have an empty stock level.*

Fig. 2. Simplified Assembly of the Stock Case Study

The system is designed as a general reusable component StockSystem. As shown in Fig. **??** it encapsulates an assembly of two components: a StockManager and a Vendor. The former one is the core business component to manage references and storage. The latter one is the system interface which main service, the vending service, is promoted at the StockSystem level. In this paper we focus on the vending and newReference services, the other services will not be more detailed further. With respect to vending, a user may add a new item in the stock management system; a new reference, and a quantity is required for the added item. In the design system the Vendor component requires a service addItem which will get a new reference and perform the update of the system. This simple functionality may fail if there is no available new reference.

The required service addItem is fulfilled with the provided service newReference. The links and sublinks are explicitly defined in the composition part of a composite component, as detailed in the listing **??**.

The nested services represent the *service dependency* $\mathcal{DI}$. For example, the required service addItem provides a special code subservice [4]. Similarly the provided service newReference requires a ask_code service from its caller (see the ***calrequires*** declaration in the interface of newReference in the listing **??**).

Inside the components, the different arrows represent various kind of calls: function call (with no side effects), service call (according to the service dependency). The newReference service calls the primitive display function (declared in the predefined Kmelia library), an internal service getNewReference [5] and the ask_code service

[4] In Kmelia, a subservice of a service s, is a service that belongs to the interface (*subprovides*) of s.

[5] which is also a subservice because it is not exposed in the *StockManager* component interface

required to its caller.

Data types in Kmelia. The data types are explicitly defined in a **TYPES** clause or in the shared libraries (predefined or user-defined). As an example, the following library (named Stocklib) declares some specific types, functions and constants.

```
TYPES
    ProductItem :: struct {id: Integer; desc: String; quantity: Integer} ;
CONSTANTS
    maxRef : Integer := 100;
    emptyString : String := "" ;
    noReference : Integer := -1;
    noQuantity : Integer := -1
```

This data types in this part are quite concrete; more abstract data types are in the process to be included in the predefined library.

A Kmelia component and observable state. The listing **??** is an extract from the Kmelia specification of the StockManager component. The state of StockManager declares among the other variables, the *observable* variable catalog which can be used for context mapping in the assembly links but also in promoted variables for composite components. Two arrays (plabels and pstock) are used to stock the labels of current references and their available quantity. The invariant states that: the catalog has an upper bound; all references in the catalog have a label and a quantity; the unknown references have no entries in the two arrays pstock and plabels. The assertions in Kmelia are possibly named predicates; the labels in front of the invariant lines are names used in this specification.

Listing 1: Kmelia specification StockManager State

```
COMPONENT StockManager
INTERFACE
  provides : {newReference, removeReference, storeItem, orderItem}
  requires : {authorisation}
USES {STOCKLIB}
TYPES
  Reference :: range 1..maxRef
VARIABLES
  vendorCodes : setOf Integer; //authorised administrators
  obs catalog : setOf Reference; // product id = index of the arrays
  plabels : array [Reference] of String; //product description
  pstock : array [Reference] of Integer //product quantity
INVARIANT
  obs @borned: size(catalog) <= maxRef,
  @referenced: forall ref : Reference | includes(catalog,ref) implies
      (plabels[ref] <> emptyString and pstock[ref] <> noQuantity),
  @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
      (plabels[ref] = emptyString and pstock[ref] = noQuantity)
INITIALIZATION
  catalog := emptySet;
  vendorCodes := emptySet;     //filled by a required service
  plabels:= arrayInit(plabels,emptyString); //consistent with ..
  pstock := arrayInit(pstock,noQuantity);   //..empty catalog
```

A Kmelia service with its assertions. The listing **??** gives the specification of the provided service newReference. It provides a new reference if its running goes well. The pre-condition is that the catalog does not reach its maximal size. The post-condition is decomposed into several observable/non-observable named parts. It states that we may have a result ranging in 1..maxRef or no reference at all, in the latter case the catalog remains unchanged.

Listing 2: Kmelia specification Provided Service with assertions

```
provided newReference () : Integer   //Result = ProductId or noReference
Interface
    calrequires : {ask_code} #required from the caller
    intrequires : {getNewReference}
Pre
    obs size(catalog) < maxRef #the catalog is not full
Variables # local to the service
  c : Integer;      # c : input code given by the user
  res:Reference;
  d : String;            # product description
Initialization
  res := noQuantity;
Behavior
Init i # the initial state
Final f # a final state
{  i -- c := __CALLER!! ask_code() --> e1,
        # gets the password on the ask_code (service) channel
  e1 -- [not(c in vendorCodes)]
        display("adding a reference is not allowed") --> end,
  e1 -- [c in vendorCodes] __CALLER ? msg(d) --> e2,
        # gets the product description
  e2 -- [d = emptyString]
        display("adding an EmptySet description is not allowed") --> end,
  e2 -- [d <> emptyString] res := __SELF!! getNewReference() --> e4,
  e4 -- {catalog := including(catalog,res); //add new reference
         pstock[res] := 0; //default stock is null
         plabels[res] := d //product description is the one provided
        }--> end,
  end -- __CALLER!! newReference(res) --> f
        # the caller is informed from the Result and the service ends.
}
Post
 obs @resultRange:((Result >=1 and Result <=maxRef) or (Result = noReference)),
 obs @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                   and catalog = add(old(catalog),Result)),
 obs @noresultValue:(Result = noReference) implies Unchanged{catalog},
 @refAndQuantity: (Result <> noReference) implies
           (pstock[Result] = 0 and plabels[Result] <> emptyString and
           (forall i : Reference | (i <> Result) implies
```

The behaviour of a service is a set of transitions. A transition is labelled and links two states like in e1 −−−label−−−> e2. A transition label is a combination of actions. A label can be guarded with the notation [guard] action∗. The Kmelia syntax of a communication action (inspired by the Hoare's CSP) is: channel(! | ? | !! | ??) message(param∗). ***_CALLER*** stands for the caller channel, ***_SELF*** stands for an internal channel, _rs stands for a required service rs channel. In this article we will not consider further the behaviour. Nevertheless the actions are necessary to check the consistency of the behaviour with respect to the pre-/post-conditions.

Context and message mappings. The context and message mappings (see **??**) are specified in assembly links. In the listing **??**, variables of the virtual context of addItem are associated with an expression on the variables of the context of newReference i.e. the observable state variables of the component sm. In this example, there are no message mapping because only the predefined overloaded msg message is used.

Listing 3: Kmelia specification StockSystem

```
COMPONENT StockSystem

INTERFACE

  provides : {vending}
```

```
  requires : {authorisation}
SERVICES
END_SERVICES
COMPOSITION
  Assembly
    Components
      sm : StockManager;
      ve : Vendor
    Links ////////////assembly links//////////
      lref: p-r sm.newReference, ve.addItem
        context mapping
          ve.catalogEmpty == empty(sm.catalog),
          ve.catalogFull == size(sm.catalog) = MaxInt
        sublinks : {lcode}
      lcode: r-p sm.ask_code, ve.code
      ...
  End // assembly
  Promotion
    Links ////////////promotion links//////////
      lvend: p-p ve.vending, SELF.vending
      laut: r-r sm.authorisation, SELF.authorisation
END_COMPOSITION
```

In the next section, we show how this Kmelia specification is analysed using our COSTO [6] tool and a specific verification approach using the B method and tools.

4 Formal Analysis and Experimentations

Components, assemblies and compositions should be analysed according to various facets. Tables ?? and ?? give an overview of the verification requirements that we consider to validate a Kmelia specification. Some of them was achieved before, in particular the behavioural compatibility of services and components, treated in [?]: it was achieved using model-checking techniques provided by existing tools (Lotos/-CADP [7] and MEC [8]); the involved parts of the Kmelia specifications were translated into the input languages of these tools and checked.

In this section, we address aspects related to data type checking and assertion checking; the main goal is to analyse parts of a Kmelia specification using its new features such as the assertions. Formal verification tools are necessary to check assertions consistency. Our approach consists in reusing existing tools such as the

6 COmponent Sudy TOolkit dedicated to the Kmelia language

7 http://www.inrialpes.fr/vasy/cadp/

8 http://altarica.labri.fr/wiki/tools:mec_4

B tools and especially the Rodin[9] framework. We design a systematic verification method that enables us to reuse the proof obligations generated by the B tools for our specific purpose.

Analysis	**Status**
Static rules: Scope + name resolution + type-checking	done
Observability rules (see ??)	in progress
Component interface consistency	done
Services dependency consistency: $\mathcal{DI}$ well-formed vs. $\mathcal{I}$ and $\mathcal{D}$ (component) $\mathcal{DI}$ vs. $\mathcal{B}$ (eLTS)	done
Simple constraint checking (parameters, query, protocol, ...)	in progress
Local eLTS checking (deadlocks, guard, subprovides, ...)	in progress
Invariant consistency vs. pre/post conditions: provided services : $Inv^O \wedge Pre^O \Rightarrow Post^O \wedge Inv^O$ $Inv \wedge Pre \Rightarrow Post^{NO} \wedge Inv$ required services : $vInv \wedge Pre^O \Rightarrow Post^O \wedge vInv$	 experimental (a) experimental (b) experimental (c)
Consistency between service assertions and eLTS: eLTS vs. *Post* the post condition should be established required service R calls vs. Pre_R the context must ensure the precondition (local+virtual) eLTS vs. subprovided service SP annotations Pre_{SP} the context must ensure the precondition (local)	not yet

Table 1
Formal analysis of a simple Kmelia component

Analysis	State
Static rules: Scope + name resolution + type-checking	done
Observability rules: promoted variables	done
Link/sublink consistency: assembly and composition signature matching service dependency matching (subprovides, callrequires) context mapping (*cm* function) and observability rules message mapping	done
Assembly Link Contract correctness: $cm(Pre_R^O) \Rightarrow Pre_P^O$ $Post_P^O \Rightarrow cm(Post_R^O)$	 experimental (d) experimental (e)
Provided Promotion Link Contract correctness: PP is at the composite level $cm(Pre_P^O P) \Rightarrow Pre_P^O$ $Post_P^O \Rightarrow cm(Post_{PP}^O)$	 experimental (f) experimental (g)
Required Promotion Link Contract correctness: RR is at the composite level $cm(Pre_R^O) \Rightarrow Pre_{RR}^O$ $Post_{RR}^O \Rightarrow cm(Post_R^O)$	 experimental (h) experimental (i)
eLTS (behaviour) compatibility [?]	done

Table 2
Formal analysis of a Kmelia assembly and compositions

Event-B and Rodin framework. Rodin is a framework made of several tools dedicated to the specification and proof of Event-B models. Event-B [?] extends the classical B method [?] with specific constructions and usage; it is intended to the

[9] http://rodin-b-sharp.sourceforge.net

modelling of general purpose systems and for reasoning on them. Proof obligations (POs) are generated to ensure the consistency of the considered model, i.e. the preservation of the **INVARIANT** by the **EVENTS**. Other POs ensure that a *refined* model is consistent, i.e. the abstract **INVARIANT** is preserved and the refined events do not contradict their abstract counterparts.

POs can be discharged automatically or interactively, using the Rodin provers.

Verifying Kmelia specifications using Event-B. The main idea is, first to consider a part of the Kmelia specification involved in the property to be verified (a service, a component, a link of an assembly, an assembly, etc), then to build from this part of the specification, a set of (Event-)B models in such a way that the POs generated for them correspond to the specific obligations we needed to check the Kmelia specification assertions. Using B to validate components assembly contracts has been investigated in [?,?].

We systematically build some Event-B models, with an appropriate structure as explained below, to check some of the proof obligations presented in Tables ?? and ??.

(i) For each component and its provided services, we generate an Event-B model. The proof of the consistency of this model ensures the proof of the rules (a) and (b) for the invariant consistency at the Kmelia level.

(ii) For each required service (and its "virtual context") we have to generate an Event-B model. Its B consistency establishes the rule (c).

(iii) For each assembly link between a required service req and an provided one prov, we give an Event-B model of the observable part of prov, which *refines* the Event-B model of the required service req previously checked.
- the consistency proof of the Event-B model ensures the rule (a) for the invariant consistency at the Kmelia level;
- the refinement proof establishes both the rules (d) and (e) for the Kmelia assembly correctness.

We are not going to deal in this article with the details of the translation procedure. Kmelia invariant and pre-condition translations are quite systematic, whereas the post-condition concept does not exist into the B language. Therefore we abstract the post-condition by using an **ANY** substitution that satisfies the post-condition (once translated) as proposed in the context of UML/OCL to B translations [?]. Figure ?? depicts the Event-B translation into Rodin of the service newReference of StockManager.

Experimental results. Consider the case study presented in Section ??; applying our method, we obtain the Event-B models structured as depicted in Fig ??. These models are studied within Rodin. We can verify the Kmelia components StockManager and Vendor before checking the assembly StockSystem. The Event-B model StockManager is used to prove the preservation of the invariant assertions by the provided services. The refinement v_addItem_sm_newReference is used to check the assembly link between the services newReference and addItem. The Table ?? gives an idea about the number of POs that are to be discharged to ensure

Fig. 3. Rodin

the correctness of the Kmelia specification.

Studying the example within Rodin, reveals some errors in our initial Kmelia specification. For example, the post-condition of newReference was wrong; one of the associated POs could not be discharged. After the feedback in our Kmelia specifications, the error was corrected.

	Auto.	Manual	Total
StockManager	16	3	19
Vendor_addItem	2	1	3
v_addItem_sm _newReference	22	1	23

Table 3
Rodin Proof obligations

Fig. 4. Event-B Models

In a general manner, the assertions associated to Kmelia services help us to ensure the correctness of the assembly link by considering the required-provided relationship as a refinement from the required service to the provided one. When the assertions are wrong, the proofs fail, which means the assembly link is wrong.

5 Discussion and Conclusion

In this article we have presented enrichments to the Kmelia abstract component model: a data language for Kmelia expressions and predicates; visibility features for component state in the context of composite components; contracts in the composition of services. The formal specification and analysis of the model are revisited accordingly. The syntactic analysis of Kmelia is effective in the COSTO tool that supports the Kmelia model. We have proposed a method to perform the necessary assertions verification using B tools: the contracts are checked through preliminary experimentations using the Rodin framework. We have illustrated the contribution with a complete case study which is specified in Kmelia and verified using Rodin.

Discussion. Our work is more related to abstract and formal component models like SOFA or Wright, rather than to the concrete models like Corba, EJB or .NET. The Java/A [?] or ArchJava [?] models do not allow the use of contracts. We have already emphasized (see ??) the fact that most of the abstract models deal mainly with the dynamic part of the components. Some of them [?,?] take datatypes and contracts into account but not the dynamic aspects. Some other ones [?,?] delay the data part to the implementation level.

In [?] *may/must* constraints are associated to the interactions defined in the component interfaces to define behavioural contracts between client and suppliers. In Kmelia, the distinction between a supplier constraint and the client is done from a methodological point of view rather than a syntactic rule. The use of B to check component contracts has been studied in [?,?] in the context of UML components.

Fractal [?] proposes different approaches based on the separation of concerns: the common structural features are defined in Fractal ADL [?] ; dynamic behaviours are implemented by Vercors [?] or Fractal/SOFA [?] and the use of assertions are studied in ConFract [?]. In ConFract contracts are independent entities which are associated to several participants, not to services and links as in our case; their contracts support a rely/guarantee mechanism with respect to the (vertical) composition of components.

Perspectives. Several aspects remain to deal with regarding assertions and the related properties, composition and correctness of component assemblies. First, we need to implement the full chain of assertion verification especially the translation *KmlToB* which is necessary to automatically derive the necessary Event-B models to check the assertions and the assemblies. Second, we will integrate high level concepts and relations for data types. Especially we plan to integrate some kind of objects and inheritance in the type system but also component types. Assertions in this context are more difficult to specify and to verify.

Another challenging point is the support for interoperability with other component models. We assume that in real component applications, a component assembly is built on components written in various specification languages. When connecting services (or operations) we can at least check the matching of signatures. If the specification language of the corresponding services or components accepts contracts (resp. service composition, service behaviour) we can provide corresponding verification means.

References

[1] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[2] Jean-Raymond Abrial. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[3] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[4] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[5] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[6] Pascal André, Gilles Ardourel, and Christian Attiogbé. Adaptation for hierarchical components and services. *Electron. Notes Theor. Comput. Sci.*, 189:5–20, 2007.

[7] Pascal André, Gilles Ardourel, and Christian Attiogbé. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, volume 4829 of *LNCS*. Springer, 2007.

[8] Pascal André, Gilles Ardourel, and Christian Attiogbé. Composing Components with Shared Services in the Kmelia Model. In *7th International Symposium on Software Composition, SC'08*, volume 4954 of *LNCS*. Springer, 2008.

[9] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.

[10] Tomás Barros, Antonio Cansado, Eric Madelaine, and Marcela Rivera. Model-checking distributed components: The vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182:3–16, 2007.

[11] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.*, 160:75–96, 2006.

[12] 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.

[13] Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý, and Petr Tůma. *CoCoME in SOFA*, volume 5153 of *LNCS*, pages 1–2. Springer, Heidelberg, 2008.

[14] 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.

[15] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya, and Antonio Vallecillo. Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.

[16] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural contracts for a sound composition of components. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *FORTE 2003, IFIP TC6/WG 6.1)*, volume 2767 of *LNCS*, pages 111–126. Springer-Verlag, Berlin, Germany, September 2003.

[17] S. Chouali, M. Heisel, and J. Souquières. Proving component interoperability with B refinement. *Electronic Notes in Theoretical Computer Science*, 160:157–172, 2006.

[18] Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite contract enforcement in hierarchical component systems. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2007.

[19] T. Coupaye, V. Quéma, L. Seinturier, and J.-B. Stefani. *Intergiciel et Construction d'Applications Réparties*, chapter Le système de composants Fractal. InriAlpes, January 2007. sardes.inrialpes.fr/ecole/livre/pub/.

[20] Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In Mario Südholt and Charles Consel, editors, *ECOOP Workshops*, volume 4379 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2006.

[21] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static Checking of System Behaviors using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, 2000.

[22] A. Lanoix and J. Souquières. A trustworthy assembly of components using the B refinement. *e-Informatica Software Engineering Journal (ISEJ)*, 2(1):9–28, 2008.

[23] Hung Ledang and Jeanine Souquières. Integration of uml and b specification techniques: Systematic transformation from ocl expressions into b. In *9th Asia-Pacific Software Engineering Conference (APSEC 2002)*. IEEE Computer Society, 2002.

[24] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, volume 3628 of *LNCS*. Springer, 2005.

[25] H. Schmidt. Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65(3):215–225, 2003.

[26] D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

A The *Vendor* Component Partial Specification

Listing 4: Kmelia specification Vendor

```
COMPONENT Vendor
INTERFACE
  provides : {vending}
  requires : {addItem, removeItem, increaseItem, decreaseItem}
USES {STOCKLIB}
CONSTANTS
obs noID : Integer := -1;
VARIABLES
  obs orders : setOf ProductItem;  # observable user card
  vendorId : Integer               # vendor personal code
INITIALIZATION
  orders := emptySet;
  vendorId := noID
SERVICES
########### provided services
# The main (provided) service is vending.
provided vending ()
Interface
    extrequires : {addItem, removeItem, increaseItem, decreaseItem}
Pre true
Variables # local to the service
  choice : CommandChoice ;    # command choice : addItem, ...
  ref : Integer;              # product reference given by the user
  qty : Integer;              # product quantity given by the user
  desc : String;              # product description given by the user
  pi : Integer;
Behavior // The behaviour is specified as an infinite loop
Init i # i is the initial state
Final f # f is a final state
{ i -- {
       displayMenu();                  # call an internal action
       display("Please enter your choice");
       choice := readCommandChoice()   # call an internal action
       } --> e0,
  e0 --[choice = stop] display("bye bye") --> f,
  //final state = end of vending
  e2 --[choice = add] _addItem!!addItem() --> e10,
  e0 --[choice <> stop] display("Product reference") --> e1,
```

```
e1 — ref:=readInt() —> e2,
e2 —[choice = remove] _removeItem!!removeItem(ref) —> e20,
e2 —[choice = store] {_increaseItem!!increaseItem(ref,readInt())} —> e30,
e2 —[choice = order] _decreaseItem!!decreaseItem(ref,readInt()) —> e40,
//—— add Item
e10 <<code>>, #subservice code is available here
e10 — {desc:=readString(); // product description
        _addItem ! msg(desc)            } —> e11,
e11 — _addItem??addItem(pi) —> e12,
e12 — {if (pi <> noReference)
         then display("New reference : "+asString(pi))
         endif } —> i
```

B The derived Event-B models

B.1 StockLib

CONTEXT StockLib

EXTENDS Default

CONSTANTS

References
MaxRef
NullInt
NoQuantity
NoReference

AXIOMS

axm5 : $References = 1\,..\,MaxRef$
axm1 : $MaxRef = 100$
axm2 : $NullInt = -1$
axm3 : $NoQuantity = -2$
axm4 : $NoReference = -3$

END

B.2 StockManager

MACHINE StockManager

SEES StockLib

VARIABLES

vendorCodes
catalog obs
plabels
pstock
Result_newReference obs

INVARIANTS

inv5 : $vendorCodes \subseteq \mathbb{Z}$
inv2 : $catalog \in \mathbb{P}(References)$
obs
inv7 : $finite(catalog)$
obs
inv3 : $plabels \in 1\,..\,MaxRef \rightarrow String$
inv4 : $pstock \in 1\,..\,MaxRef \rightarrow \mathbb{Z}$
• borned : $card(catalog) \leq MaxRef$
obs
• referenced : $\forall ref1 \cdot (ref1 \in References \land ref1 \in catalog \Rightarrow plabels(ref1) \neq EmptyString \land pstock(ref1) \neq NoQuantity)$
• notreferenced : $\forall ref2 \cdot (ref2 \in References \land ref2 \notin catalog \Rightarrow plabels(ref2) = EmptyString \land pstock(ref2) = NoQuantity)$
inv6 : $Result\_newReference \in \mathbb{Z}$
obs

EVENTS

Initialisation

begin
act1 : $vendorCodes := \varnothing$
act2 : $catalog := \varnothing$
act3 : $plabels := (1\,..\,MaxRef) \times \{EmptyString\}$
act4 : $pstock := (1\,..\,MaxRef) \times \{NoQuantity\}$
act5 : $Result\_newReference := 0$
end

Event *newReference* $\widehat{=}$

any
new_Result
new_catalog
new_pstock
new_plabels
where
grd8 : $card(catalog) < MaxRef$
obs
grd1 : $new\_Result \in \mathbb{Z}$
obs

grd2 : $new\_catalog \in \mathbb{P}(References)$
obs
grd11 : $finite(new\_catalog)$
obs
grd3 : $new\_plabels \in 1 \,..\, MaxRef \rightarrow String$
grd4 : $new\_pstock \in 1 \,..\, MaxRef \rightarrow \mathbb{Z}$
grd5 : $(new\_Result > 0 \wedge new\_Result \leq MaxRef) \vee new\_Result = NoReference$
obs
grd6 : $new\_Result \neq NoReference \Rightarrow$
$new\_Result \notin catalog$
$\wedge\ new\_catalog = catalog \cup \{new\_Result\}$
obs
grd7 : $new\_Result = NoReference \Rightarrow new\_catalog = catalog$
obs
grd9 : $new\_Result \neq NoReference \Rightarrow$
$new\_pstock(new\_Result) = 0 \wedge$
$new\_plabels(new\_Result) \neq EmptyString \wedge$
$(\forall ii \cdot (ii \in 1 \,..\, MaxRef \wedge ii \neq new\_Result \Rightarrow$
$new\_pstock(ii) = pstock(ii) \wedge$
$new\_plabels(ii) = plabels(ii)$
$))$
grd10 : $new\_Result = NoReference \Rightarrow$
$new\_pstock = pstock \wedge$
$new\_plabels = plabels$

then
act1 : $Result\_newReference := new\_Result$
act2 : $catalog := new\_catalog$
act3 : $pstock := new\_pstock$
act4 : $plabels := new\_plabels$
end

END

B.3 Vendor_addItem

MACHINE Vendor_addItem

SEES StockLib

VARIABLES

```
catalogFull
catalogEmpty
Result_addItem
```

INVARIANTS

inv1 : $catalogFull \in BOOL$
inv2 : $catalogEmpty \in BOOL$
• notFullEmpty : $\neg\,(catalogEmpty = TRUE \wedge catalogFull = TRUE)$
inv4 : $Result\_addItem \in \mathbb{Z}$

EVENTS

Initialisation

begin
act1 : $catalogFull := FALSE$
act2 : $catalogEmpty := TRUE$
act3 : $Result\_addItem :\in \mathbb{Z}$
end

Event *addItem* $\widehat{=}$

any
new_Result
$new\_catalogEmpty$
$new\_catalogFull$
where
pre_addItem : $\neg\,(catalogFull = TRUE)$
grd2 : $new\_Result \in \mathbb{Z}$
grd6 : $new\_catalogEmpty \in BOOL$
grd5 : $new\_catalogFull \in BOOL$
Post_addItem : $new\_Result \neq NoReference \Rightarrow$
$new\_catalogEmpty = FALSE \wedge$
$new\_catalogFull \in BOOL$
Post_addItem2 : $new\_Result = NoReference$
$\Rightarrow$
$new\_catalogEmpty = catalogEmpty \wedge$
$new\_catalogFull = catalogFull$
then

```
    addItem_result : Result_addItem := new_Result
    addItem_empty : catalogEmpty := new_catalogEmpty
    addItem_full : catalogFull := new_catalogFull
  end

END
```

B.4 v_addItem_sm_newReference

```
MACHINE v_addItem_sm_newReference

REFINES Vendor_addItem

SEES StockLib

VARIABLES
  catalogEmpty
  catalogFull
  Result_addItem
  catalog

INVARIANTS
  inv1 : catalog ∈ ℙ(References)
  inv6 : finite(catalog)
  borned : card(catalog) ≤ MaxRef
  assemblyEmpty : catalogEmpty = bool(card(catalog) = 0)
  assemblyFull : catalogFull = bool(card(catalog) = MaxRef)

EVENTS

Initialisation
  extended
  begin
    act1 : catalogFull := FALSE
    act2 : catalogEmpty := TRUE
    act3 : Result_addItem :∈ ℤ
    act4 : catalog := ∅
  end

Event newReference ≙

refines addItem
  any
    new_Result
    new_catalog
  where
    pre_newReference : card(catalog) < MaxRef
    grd11 : new_Result ∈ ℤ
    grd64 : new_catalog ∈ ℙ(References)
    grd10 : finite(new_catalog)
    post_newRef1 : ((new_Result > 0 ∧ new_Result ≤ MaxRef)
                     ∨
                    new_Result = NoReference)
    post_newRef2 : new_Result ≠ NoReference ⇒
                        new_Result ∉ catalog
                      ∧ new_catalog = catalog ∪ {new_Result}
    post_newRef3 : new_Result = NoReference ⇒ new_catalog = catalog
  with
    new_catalogEmpty : new_catalogEmpty = bool(card(new_catalog) = 0)
    new_catalogFull : new_catalogFull = bool(card(new_catalog) = MaxRef)
  then
    addItem_result : Result_addItem := new_Result
    addItem_empty : catalogEmpty := bool(card(new_catalog) = 0)
    addItem_full : catalogFull := bool(card(new_catalog) = MaxRef)
    act34 : catalog := new_catalog
  end

END
```