Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 263 (2010) 5–30
www.elsevier.com/locate/entcs

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 systems. Assertions are used to perform analysis of services, component assemblies and service compositions. Additionally we enable the definition of virtual contexts for required services and the corresponding observable state space for the components which provide the services. 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

In the design of service and component-based software systems, formal models are helpful to specify and document components and systems, to find components and services in libraries according to formal search requirements, to check various kind of properties (correctness, liveness, safety) for component certification, to refine models and generate executable components. Formal models are mandatory to build trusted components [24]. In [9], we proposed a formal component model, called Kmelia, where *i*) the services are more than simple operations; they includes contracts, communication interactions, dynamic evolution rules and composition, *ii*) the components are designed independently from their environment by setting assumptions, *iii*) the component assemblies are governed by strict service composability rules, *iv*) the composite components are governed by encapsulation and promotion policies.

doi:10.1016/j.entcs.2010.05.002

The Kmelia model [9] is an *abstract* formal component model dedicated to the specification and development of correct components; *abstract* means that component and assemblies are independent from execution platforms. They can be implemented later in centralised or distributed execution platforms. The formal definition of concepts is necessary to express and to ensure the verification of system specification properties. One key feature of Kmelia is the central role of services. A service specification describes a behaviour that corresponds to some desired functionalities. Assembling two components consists in linking their required and provided services. Linking components by their services in assemblies establishes a possible bridge to Service Oriented Architectures.

In [9] we introduced the syntax and semantics for the core model and language. It has been incrementally enriched later. We focused on the dynamic aspects of composition: interaction compatibility in [9], on component protocols with service composition in [7]. Following this incremental approach, we consider in the current article an enrichment of the data and expressions in Kmelia and its impact on the language syntax, its semantics and the verification of properties. Our objective is twofold:

- Enable the definition of assertions (with invariant, pre/post conditions, and properties of services, components, and composites),
- Increase the expressiveness of the *action* statements so as to deal with real size case studies.

Assertions are useful *i*) to define *contracts* [1] on services (with pre/post-conditions); contracts increase the confidence in assembly correctness (by constraining the pre/post-conditions of the involved services) and enable rich query expression when searching for a component in libraries, *ii*) to ensure the consistency of components with respect to 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 via the use of assertions.

Motivations. Modelling real life systems requires to cover the static and dynamic aspects of components (structure, links, actions, interactions). We want to verify early the development step, whether assemblies are well-formed; it is important to cover structural, dynamic and functional aspects of systems to tackle various kind of applications. The state of the art shows that most of the abstract component models [4,15,28,14] enable various verifications of the interaction correctness but they lack expressiveness on the data types; they do not provide assertions and the related verification rules. As an example, in Wright the dynamic part based on CSP is well detailed (specification and verification) while the data part is less well dealt with [4]. In the proposal of [26] the data types are defined using algebraic specifications, which are convenient to use symbolic model checking of state transition systems but this proposal does not deal with contracts and assertions.

Contribution. In this work, we enrich the Kmelia model with data and assertions in order to cover the whole static and dynamic aspects and hence to deal with safe

[1] Our contract definitions are related to results of works such as *design-by-contracts* [23].

services, consistent components and correct contract-based assemblies. 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 introduced. 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 2 gives an overview of the Kmelia abstract model and introduces its new features. In Section 3 a working example is introduced to illustrate the use of data and assertions. The formal analysis is treated in Section 4; we present various analysis to be performed, the ones that are currently implemented and we focus on component consistency and on checking assembly links. Section 5 concludes the article and draws discussions and perspectives. This work is supported by the COSTO tool which is presented during the tool demo session of the conference; In the appendix A we give an extract of the Kmelia syntax concerning the new data part. An overview of COSTO is given in Appendix B.

2 The Kmelia Model and its new Features

The main concepts of Kmelia are: component, service, assembly and composition. A component is a container of services. A service is a complex entity: it has a state and a dynamic behaviour; a service may also declare required and provided subservices. The service behaviour defines the order in which the service performs its actions. Communication actions are primitives for synchronous interactions between services. In a component assembly, components are pairwise linked through services: a required service is *achieved* by the provided service it is linked to. A composite component encapsulates a component assembly.

In this section we revisit the Kmelia model of [9] which is augmented and restructured. In particular the following features are introduced:

(i) a notion of *observability* similar to a read-only visibility in programming, which allows to make the state information of a provider component available to clients or composite components;

(ii) a refined definition of required services allowing constraints on a *virtual state space*, i.e. assumptions on provider components which are not known when designing the services;

(iii) a more flexible *message naming rule*; now two communicating services can use different service names, which is consistent with the independent design of providers and clients;

(iv) *state and message mappings* in the assembly links to handle the correspondence between the provider components and the required service contexts.

These new features are related to service actions, assertions and contracts. We

designed a small but expressive *data language* to enable the description of datatypes, expressions and predicates (quick overview in appendix A).
In the following we use a mathematical toolkit inspired by the Z notation. $X \leftrightarrow Y$ denotes the relations from X to Y ($x \mapsto y$ denotes a pair (x, y) member of a relation); $X_1 \uplus X_2$ denotes the disjoint union of sets; $X \nrightarrow Y$ denotes the partial functions from X to Y; $X \rightarrowtail\!\!\!\twoheadrightarrow Y$ denotes the partial one-to-one functions from X to Y; id denotes the identity relation; when r is a relation ($r : X \leftrightarrow Y$), $\mathrm{dom}(r)$ and $\mathrm{ran}(r)$ denote respectively the domain and the range of r; $E \lhd r$ and $r \rhd F$ denote respectively the domain and the range restrictions of a relation r where $E \subseteq X$ and $F \subseteq Y$.

Definition 2.1 A *state space* $\mathcal{W}$ defines a set of variables constrained by an invariant and an initialisation: $\mathcal{W} = \langle T, V, type, Inv, Init\rangle$ where T is a set of types, V a set of variables, $type : V \rightarrow T$ the function that map variables to types, Inv an invariant defined on V and $Init$ the initialisation of the variables of V.

The state space concept is used for both components and services. In the following $\mathcal{N}$ is a finite set of *names* and Let $\mathcal{M}$ is the set of **message** names with $\mathcal{M} \subseteq \mathcal{N}$.

2.1 Components

The component definition in [9] has been restructured: the set of actions $\mathcal{A}$ is deferred to Kmelia expressions and the constraint $\mathcal{C}_S$ is now achieved by service properties like the protocols of [7].

A **component (type)** C is a tuple $\langle \mathcal{W}, \mathcal{I}, \mathcal{D}, \nu\rangle$ with:

- $\mathcal{W}$ the component the state space (see definition 2.1).
- $\mathcal{I} \subseteq \mathcal{N}$ the component interface, partitioned in two disjoint finite sets $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ where ^{P} stands for provided and ^{R} stands for required.
- $\mathcal{D}$ is the set of service descriptions, as detailed in Section 2.2. Like $\mathcal{I}$, $\mathcal{D}$ is partitioned along the provided/required criteria: $\mathcal{D} = \mathcal{D}^P \uplus \mathcal{D}^R$.
- ν is a partial function which maps service names to service descriptions ($\nu : \mathcal{N} \rightarrowtail\!\!\!\twoheadrightarrow \mathcal{D}$).

Listing 1: Component structure

```
COMPONENT C1
  INTERFACE
    provides : <ServName list>
    requires : <ServName list>
  //component state space
  TYPES
      <Type Defs>
  VARIABLES
      <Var list>
  INVARIANT
      <Predicate>
  INITIALIZATION
    ...
    // var. assignments
  //component services
  SERVICES
    ...
    // as described below
  END_SERVICES
```

The component interface must be consistent with the service description: (1) provided and required services have distinct names: $\mathrm{dom}(\nu \rhd \mathcal{D}^P) \cap \mathrm{dom}(\nu \rhd \mathcal{D}^R) = \varnothing$ and (2) the services in the interface of the component are a subset of the services described in the component: $\mathcal{I}^P \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P) \wedge \mathcal{I}^R \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^R)$.

Observability of the component state. To preserve the abstraction and encapsulation of components, the state of a component is accessed only through its provided services. Nevertheless to understand the specification of a service (i.e. its contract) we might need to *observe* its context (a part of its component state space). Similarly a composite component needs to observe informations from its components.Thus, we distinguish the observable part V^O of the state variables ($V^O \subset V$), the observable part (Inv^O) of the invariant (Inv^O is defined on V^O) and the pre/post-conditions are written accordingly (with the rules of Section 2.2).

2.2 Services

The behaviour of a component relies on the behaviours of its services. A service describes a functionality and a behaviour using actions combined with a labelled transition system. A service is started when it is called (by a client service), it is then said to be *activated* and should evolve until it reaches its final state.

A service shares the state space of its component with other services of the same component. During its evolution a service s may call other services or communicate with them using messages. All the interacting services of s are defined in the interface of s. Due to dependencies and interactions between services, the actions of several activated services interleave or synchronise, but *only one action of an activated service may be observed at a time.*

Formally a *service* s of a component C is defined by a tuple $\langle \sigma, \mathcal{IS}, Cont, \mathcal{W}^L, \mathcal{B} \rangle$ with:

- $\sigma = \langle s, param, ptype, Tres \rangle$ the service signature where s is the service name, $param$ a set of parameters, $ptype : param \to T$ the function mapping parameters to types and $Tres \in T$ the service result type;
- $\mathcal{IS}$ the service interface as detailed below;
- $Cont = \langle Pre, Post \rangle$ the service contract where Pre is the pre-condition and $Post$ the post-condition;
- $\mathcal{W}^L$ the local state space (definition 2.1) which is used only in the service behaviour $\mathcal{B}$;
- $\mathcal{B}$ the service behaviour; it is an *extended labelled transition system* (eLTS) as defined below.

Listing 2: Service structure

```
provided aService_1
      (<param>) : <ResultType>

  Interface
    subprovides : <Serv list>
    calrequires : <Serv list>
    extrequires : <Serv list>
    intrequires : <Serv list>
    ...

  Pre <Predicate>

  Variables # local state space
     <Var list>
  Initialization
    ... // var. assignments

  Behavior
    Init  <initial state>
    Final <final states>
    //eLTS
    {<transition list>}

  Post <Predicate>

End

required aService_2 ()
 ... //in the same way
```

The service interface $\mathcal{IS}$ is defined by a tuple $\langle \mathcal{DI}, \mu, \mathcal{W}^V \rangle$ where

- $\mathcal{DI}$ is the *service dependency*; it is composed of the services which the current service depends on. $\mathcal{DI}$ is made of four disjoint sets $\langle sub,\ cal,\ req,\ int\rangle$ where $sub \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P)$ (resp. $cal \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^R)$), $req \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P)$, $int \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P)$) contains the provided service names (resp. the ones required from the caller, from any component or from the component itself) in the scope of s.
- $\mu = \langle mname, mparam, mptype\rangle$ is a set of message signatures where $mname \in \mathcal{M}$, $mparam$ and $mptype$ are as in service signature;
- $\mathcal{W}^V$ is a *virtual state space* according to definition 2.1;

The behaviour $\mathcal{B}$ of a service s is a labelled transition system (LTS) *extended* by the use of nested states (via a state annotation function) and nested transitions (by specific labels). Therefore $\mathcal{B}$ is an eLTS defined by a tuple $\mathcal{B} = \langle S, L, \delta, \Phi, S_0, S_F\rangle$ with S the set of the states of s; L is the set of transition labels (possibly guarded combinations of actions `[guard] action*`) and δ is the transition function ($\delta \in S \times L \to S$); S_0 is the initial state ($S_0 \in S$); S_F is the finite set of final states ($S_F \subseteq S$); Φ is a state annotation function ($\Phi \in S \to sub_s$). An *action* is now a Kmelia expression. An *elementary action* (an assignment for example) does not involve other services and does not use a communication channel. A *communication action* is either a *service call/response* or a message *communication*. The full details on defining and verifying services behaviour are provided in references [9,7].

The new following new features are consequences on services of the component observability. In particular a provided service can use an observable variable in its pre condition (e.g. a service *addElement* should not be called when the observable component variable *isFull* is evaluated as true). Consequently, a required service may define a virtual context to set assumptions on what should be a provider component.

Virtual state spaces. A required service of a component is an abstraction of a service provided by another component. Since that component is unknown when specifying the required service, it is necessary to "imagine" its state, which we call a *virtual state space* (namely $\mathcal{W}^V$). For a *provided* service this virtual context is always empty.

Observability vs. service assertions. Let s be a service of a component C. The distinction between observable and non-observable variables of the component state space is revisited [2] according to the following tables. The first table indicates the accessible state spaces for a service.

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`	V^V	V	Inv^V	Inv

The second table indicates how service assertions are splitted in observable/non-observable parts and makes it precise which variables are accessible in each part.

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

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

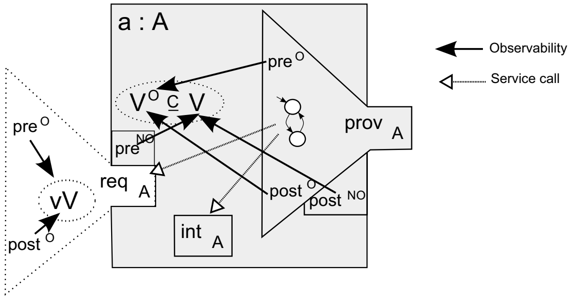

Fig. 1. State variables scope and assertion scope

Fig. 1 summarises the relations between state spaces, observability and service contracts. The box denotes a component a:A. The grey (resp. white) "funnel" denotes provided (resp. required) services. The observable pre/post-conditions of service prov_A (resp. req_A) refer to the observable state V^O of a (resp. the virtual state V^V of req_A). A non-observable pre-condition of a required service req_A gives call conditions on the (caller) component state variables V of a. The non-observable post-condition of a service prov_A locally refers to the whole state V of a and should establish the non-observable part of the invariant of a.

2.3 Assembly and composite

Components are composed through their references by **assembly links** in component assemblies or by **promotion links** in composite components. A link is an abstract communication channel which connects two distinct services. A sublink is a link defined in the context of another link according to the service dependency structure (*cal* and *sub*).

A *component reference* is one element of a component (type). A component reference is denoted by c:C where C is a component. The access to a state variable v of c is denoted c.v. *By murdering the language we will use the term 'component' for both the type and the reference.*

Let $\mathcal{C}$ be a set of component $c_k : C_k$ with $k \in 1..n$ and $C_k = \langle \mathcal{W}_k, \mathcal{I}_k, \mathcal{D}_k, \nu_k \rangle$.

$BaseLink \subseteq (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N})$ is a set of quadruples such that :
(1) $\forall (c_i, n_1, c_j, n_2) : BaseLink \bullet n_1 \in \operatorname{dom} \nu_i \wedge n_2 \in \operatorname{dom} \nu_j$
(2) $\forall c_i : \mathcal{C},\ n_1 : \operatorname{dom} \nu_i \bullet (c_i, n_1, c_i, n_1) \notin BaseLink$
$SubLink : BaseLink \leftrightarrow BaseLink$
(3) $\forall (l_1, l_2) \in SubLink \bullet (l_2, l_1) \notin SubLink^*$

where $\operatorname{dom} \nu_i$ is the set of service names of component C_i and $SubLink^*$ is the transitive closure of the relation $SubLink$. (1) expresses that any basic link relates a service name of a component to a service name of another component; (2) expresses that a service name cannot be linked to itself and (3) there is no circular dependency in the links.

2.3.1 Component assembly

An *assembly* is a set of components that are linked (*horizontal composition*) through their services. An *assembly link* associates a required service to a provided one. A communication channel is established between the interacting services when assembling components. A channel defines a context for the communication actions of the service behaviour. Since a behaviour is written without knowing the component with which it will communicate, one has to know at least the channel dedicated to the communication. A channel is usually named after the required service that represents the context. The placeholder keyword `CALLER` is a special channel that stands for the channel open for a service call. From the point of view of a provided service p, `CALLER` is the channel that is open when p is called. From the point of view of the service that calls p, this channel is named after one of its required service, which is probably named p. The placeholder keyword `SELF` is a special channel that stands for the channel open for an internal service call. In this case, the required service is also the provided service.

The new features are the mappings we have introduced in the links.

Context and message mappings in assembly links. The ultimate goal is to connect a required service defined in its virtual context to a provided service defined in its observable context (the observable state space of its component). The signatures matching and the dependency mapping (via sublinks) were introduced in [9]. Here we add a *context mapping* and a *message mapping*. The former is the consequence of the introduction of the virtual context concept. The virtual state space variables sr of a component $\mathsf{cr}\colon \mathsf{CR}$ must be "instantiated" using the *observable* variables sp of a component $\mathsf{cp}\colon \mathsf{CP}$. The latter enables different message names since the required service communication actions are designed independently from the provided service communication actions. Currently, each message name of sr is mapped to a message name of sp. If more flexibility is needed *e.g.* parameters re-ordering, one can use adaptation mechanisms [6].

Fig. 2 extends Fig. 1 to assemblies and composite components. The boxes denote components (a, b) and composite (c). The conditions given with Fig. 1 are the basis to check the contracts supported by the assembly links and the promotion links. In particular the virtual state V^V of req_B should map with a subset V of a. Non-observable pre-conditions (resp. post-conditions) are meaningless for a provided service (resp. required service) because they prevent safe assembly and promotion contracts.

Formally an *assembly-type* A is a tuple $\langle \mathcal{C},\ alinks,\ subs,\ vmap,\ mmap \rangle$ where

- $\mathcal{C}$ is a set of component references c_k such that $c_k : C_k$ with $k \in 1..n$,

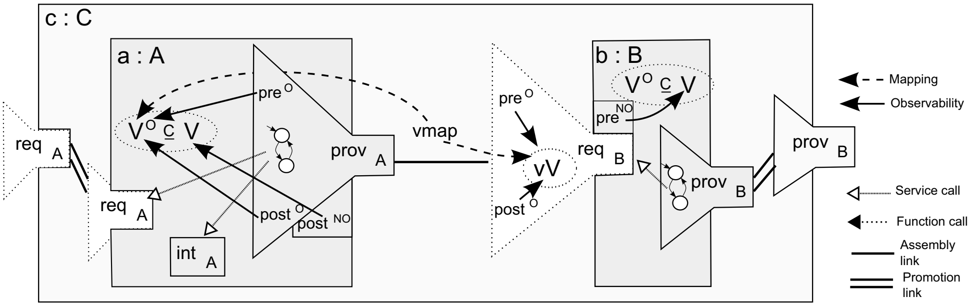

Fig. 2. State variables scope and assertion scope in assemblies

- $alinks$ is a set of assembly links between services of $\mathcal{C}$ such that

$$
\begin{array}{ll}
\multicolumn{2}{l}{alinks \subseteq BaseLink \wedge} \\
(1) & (\forall (c_i, n_1, c_j, n_2) : alinks \bullet c_i \in \mathcal{C} \wedge c_j \in \mathcal{C} \wedge \\
(2) & \quad ((n_1 \in \mathcal{I}_i^P \wedge n_2 \in \mathcal{I}_j^R) \vee (n_1 \in \mathcal{I}_i^R \wedge n_2 \in \mathcal{I}_j^P)))
\end{array}
$$

that is to say: the link components are those of the assembly (1); the linked services have a symmetric nature *required-provided* (2).

- $subs$ denotes links included in other links:

$$
\begin{array}{ll}
\multicolumn{2}{l}{subs \subseteq SubLink \wedge} \\
(3) & (\mathrm{dom}\, subs - \mathrm{ran}\, subs) \subseteq alinks \wedge \\
(4) & (\forall ((c_i, n_1, c_j, n_2) \mapsto (c_k, n_3, c_l, n_4)) \in subs \bullet c_i = c_k \wedge c_j = c_l) \wedge \\
(5) & (\forall (c_i, n_1, c_j, n_2) : \mathrm{ran}\, subs \bullet ((\nu_i(n_1) \in \mathcal{D}^P{}_i) \text{ xor } (\nu_j(n_2) \in \mathcal{D}^P{}_j)))
\end{array}
$$

that is to say: $Sublink$ depends on other links (3) of the same components (4); the required services are linked to the provided one (2),(5).

- $vmap = \langle vmapVar, vmapExp \rangle$ is the context mapping function associated to links such that

$$
\begin{array}{ll}
\multicolumn{2}{l}{vmapVar : BaseLink \leftrightarrow V} \\
\multicolumn{2}{l}{vmapExp : (BaseLink \times V) \twoheadrightarrow exp(V)} \\
(6) & \mathrm{dom}\, vmap \subseteq (alinks \cup subs) \wedge \mathrm{dom}\, vmapExp = vmap \wedge \\
(7) & (\forall (c_i, n_1, c_j, n_2) : \mathrm{dom}\, vmap \bullet vmap(\!| (c_i, n_1, c_j, n_2) |\!) = V^V_{\nu(n2)} \wedge \\
(8) & \quad var(vmapExp(c_i, n_1, c_j, n_2)) \subseteq V^O_{C_i})
\end{array}
$$

where $exp(V)$ denotes an expression over the variables of V and $var(exp(V)) = V$. These formula express that the links of the context mapping are those of the assembly (7). The mapped variables are those of the virtual state space variables of the required service n_2. The mapping expression is built using the observable variables of n_1 (8).

- *mmap* is the message mapping function associated to links such that

$$
\begin{array}{ll}
\multicolumn{2}{l}{mmap : BaseLink \leftrightarrow \mathcal{M} \times \mathcal{M}} \\
(9) & \operatorname{dom} mmap \subseteq (alinks \cup subs) \wedge (\forall (c_i, n_1, c_j, n_2) \in \operatorname{dom} mmap \bullet \\
(10) & \qquad (\forall (m_1, m_2) \in mmap(\!|(c_i, n_1, c_j, n_2)|\!) \bullet m_1 \in \mu_{s_1} \wedge m_2 \in \mu_{s_2}))
\end{array}
$$

The links of the message mapping are those of the assembly (9). The mapped messages are those of the linked services s_1 and s_2 (10).

Listing 3: Assembly links

```
Assembly
  Components
    cp: CP;
    cr: CR
  Links // ----------assembly links-----------
    @la: p-r cp.provServ cr.reqServ
       context mapping
         cr.var1 = expr(cp.varA, cp.varB...),
         cr.var2 = expr(cp.varA, cp.varB...),
       message mapping
         provServ.msg1 = reqServ.msgA
         provServ.msg2 = reqServ.msgB
       sublinks: {lasub}
    // ----------sublinks------------
    @lasub: r-p cp.subReq cr.prov
    ...
End // assembly
```

As an illustration, let consider the assembly example of Listing 3. Two components cp:CP and cr:CR are assembled by an assembly link named @la; this link expresses that the required service reqServ of cr is *implemented* by the provided service provServ of cp. The **p**−**r** prefix denotes the link direction (from **p**rovided to **r**equired). The context mapping associates an expression built on the observable variables varA, varB. . . of the component cp to the var1,var2. . . variables of the virtual context of reqServ. The message mapping associates msg1 of provServ to msgA of reqServ. The sublinks must be consistent with the service dependencies. For example, if the provided service provServ requires the service subReq in its `calrequires` dependency, then a sublink must be associated to a service prov provided by the component cr or in the `subprovides` dependency of the service reqServ.

2.3.2 Composite

A *composite* is a component that encapsulates assemblies or other components. Some features (variables and services) of the nested sub-components can be promoted at the composite level. In a previous version [9], we defined *promotion links* to promote services with possible service renaming. Promotion is extended here to state variables promotion and it permits pre-condition weakening and post-condition strengthening with respect to the state variable promotion. We current apply the same observability schema for assembly clients or composite clients except that observable variables can be promoted at the composite level.

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 accessed in their effective contexts using a service of the sub-component that owns the variables. This guarantees the encapsulation principle.

Formally a *composite-type* is a tuple $CC = \langle C, \mathcal{A},\ plinks,\ pvars\rangle$ where

- $C = \langle \mathcal{W}, \mathcal{I}, \mathcal{D}, \nu\rangle$ is a component type as defined in Section 2.1. The default predefined composite component is SELF: C.
- $A = \langle \mathcal{C},\ alinks,\ subs,\ vmap,\ mmap\rangle$ is an assembly definition as in Section 2.3.1.
- $plinks$ is a set of promotion links between services of A which are unused in $alinks$:

$$\begin{array}{ll} \multicolumn{2}{l}{plinks \subseteq BaseLink \wedge (\forall (c_i, n_1, c_j, n_2) : plinks \bullet} \\ (1) & c_i \in \mathcal{C} \wedge c_j = SELF \wedge \\ (2) & ((n_1 \in \mathcal{I}_j^R \wedge n_2 \in \operatorname{dom} \nu^R) \vee (n_1 \in \mathcal{I}_j^P \wedge n_2 \in \operatorname{dom} \nu^P)) \wedge \\ (3) & (\forall c_k \in \mathcal{C}, n_3 \in \mathcal{I}_k \bullet (c_i, n_1, c_k, n_3) \notin alinks \wedge (c_k, n_3, c_i, n_1) \notin alinks) \wedge \\ (4) & \nu(n_2) = rename(\nu(n_1), n_2)) \end{array}$$

  where $\operatorname{dom} \nu$ is the set of service names of the composite, $rename(s, n)$ is a function that returns the service s renamed by n.
- $pvars$ is the promotion mapping function such that

$$\begin{array}{ll} \multicolumn{2}{l}{pvars : (\mathcal{C} \times V) \leftrightarrow V} \\ (5) & (\forall (c_i, vp) : \operatorname{dom} pvars \bullet vp \in V_{C_i}^O \wedge type(vp) = type(pvars(c_i, vp))) \end{array}$$

A promotion link associates the SELF component to one of the assembly components (1). The promoted services keep their nature (*provided* or *required*) (2). The promoted services are not linked in the assemblies (3). The promoted service definition is the one of the sub-component up to a service renaming (4). The promoted variables are observable variables in their owner sub-component and have the same type in the composite (5).

The newly introduced features (data language, observability, virtual context) are expressive means to describe contracts. Section 3 illustrates them.

3 A Working Example

In this section we illustrate the Kmelia language by specifying a simplified *Stock Management* system ; the new features of Kmelia are also discussed. This system manages product references (catalog) and product storage (stock). The main service models a *vending* process. The system administrators have specific rights, they can add or remove references under some business rules such as: "*a new reference was not in the catalog*" or "*a removable reference must have an empty stock level*".

The system is designed as a reusable component StockSystem providing a (promoted) vending service and requiring an authorisation service. It encapsulates an assembly of two components: sm:StockManager and ve:Vendor as depicted in Fig. 3. The former is the core business component to manage product references and storage. The latter models a vendor user interface. With the vending service, a user may

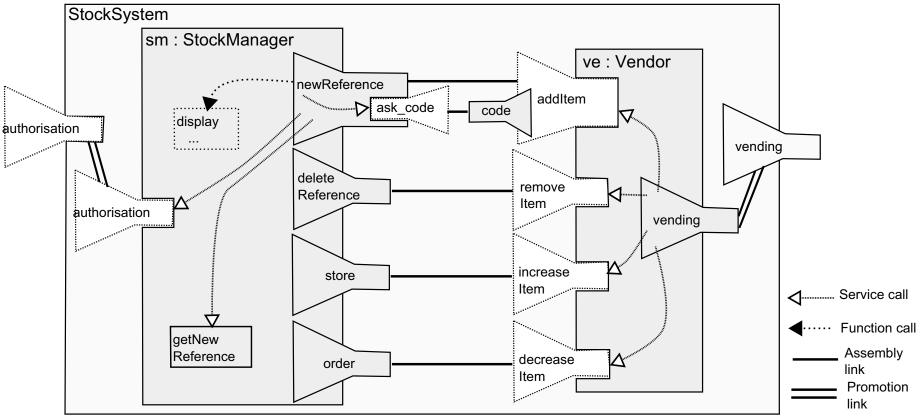

Fig. 3. Simplified Assembly of the Stock Case Study

add a new item in the stock management using the required service addItem. The required service addItem is fulfilled with the provided service newReference which gets a new reference and performs the update of the system if there is an available new reference (see Listing 5).

Links and sublinks are explicitly defined in the composite component, as detailed in Listing 7. The nested services in Fig. 3 represent the *service dependency* $\mathcal{DI}$. As example, the required service addItem provides a subservice named code. This will be detailed in the Listing 4. The different arrows represent various kinds of call: function call (with no side effects) or service call (according to the service dependency $\mathcal{DI}$). The service newReference calls a display function (declared in the predefined Kmelia library), a service getNewReference internally required (from the same component), the service ask_code required from its caller and a service authorisation which is externally required.

Data types in Kmelia. The data types are explicitly defined in a TYPES clause, in the shared libraries (predefined or user-defined). The following library (named Stocklib) declares some specific types, functions and constants. The data types in this part are rather concrete; more abstract data types are in the process to be included in the predefined library.

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

A Kmelia component and observable state. The Listing 4 shows an extract of the Kmelia specification of the StockManager component. The state of StockManager declares an *observable* variable catalog which will be available for a context mapping. Two arrays (plabels and pstock) are used to store the current references labels and available quantities. The invariant is a set of named predicates [obs] [@name]: <pred_expr>, where labels in front of the assertion are (optional) predicate names. The prefix obs means that the predicate belongs to Inv^O. As example @borned states that the catalog has an upper bound; @referenced establishes that all references in the catalog have a label and a quantity; @notreferences expresses that the unknown references have no label and no quantity.

Listing 4: Kmelia specification of 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. Listing 5 shows the specification of newReference, one of the service provided by StockManager. Its pre-condition expresses that the catalog does not reach its maximal size. The prefix obs means that the predicate @resultRange belongs to $Post^O$. The observable post-conditions state that we may have a result ranging in 1..maxRef or noReference ; in the latter case the catalog remains unchanged. The non-observable post-condition (without the prefix obs) indicates how the non-observable state variables plabels and pstock would evolve.

Listing 5: Kmelia specification of the newReference Service

```
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
//{eLTS see figure below}
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
 (pstock[i] = old(pstock)[i] and plabels[i] = old(plabels)[i] ))),
 @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
End
```

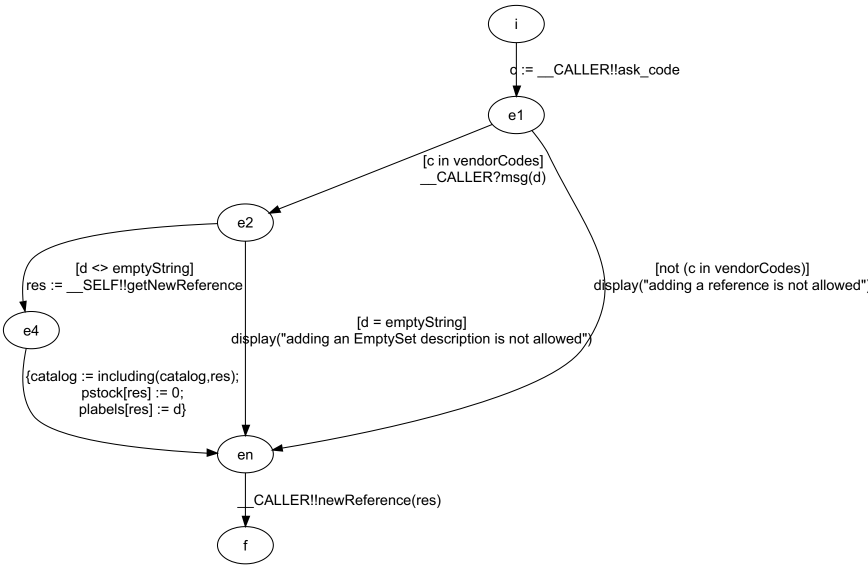

Fig. 4. Behaviour of the newReference service exported by the COSTO tool

The behaviour of the service newReference is not presented in its textual form but its graphical representation is given Fig. 4. It is is obtained by a *dot* translator of the COSTO tool (dedicated to the Kmelia model, Sec.4). Remind that a transition label is a guarded combination of actions. The communication actions are noted `channel(comOp) message(param*)` where the communication operators `comOp` are send (!) or receive (?) a message, call (!!) or result(??) a service. The `channel` _CALLER is used for the caller service, _SELF is used for a service of the same component (internal call), _rs stands for a required service. In Fig. 4, the behaviour of newReference consists to ask a vendor code if the returned code is referenced into the vendorCodes, then get a product description d and add it to the catalog. In any case the service returns the operation status to the caller.

Virtual state space of a required service. The Listing 6 shows the specification of the required service addItem of the component Vendor. A required service set assumptions on its provider by setting a virtual state space (page 6). In the Listing 6, the virtual variables of addItem represent the Vendor view of a catalog: it is only concerned by asking whether it is empty or full. The pre/post-condition are written accordingly.

Listing 6: Kmelia specification of addItem

```
required addItem () : Integer
Interface
  subprovides : {code}
  Virtual Variables
     catalogFull : Boolean;
```

```
    catalogEmpty : Boolean    //possibly catalogSize
  Virtual Invariant not(catalogEmpty and catalogFull)
  Pre not catalogFull
  //No LTS
  Post (Result <> noReference) implies (not catalogEmpty)
End
```

In Listing 7 we describe the composite StockSystem, it is an extract of the textual representation of Fig. 3. The composite StockSystem is defined with an assembly which includes the sub-components sm:StockManager and ve:Vendor. The assembly links and sublinks connect the sub-component services. The promotion links set the services vending and authorisation at the composite level.

Listing 7: Kmelia specification of StockSystem

```
COMPONENT StockSystem
INTERFACE
  provides : {vending}
  requires : {authorisation}
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
```

Context and message mappings. The context and message mappings (see Section 2.3.1) are specified into the assembly links. In Listing 7, the variables of the virtual state space of the required service addItem are associated with an expression of the variables of the context of the provided service newReference *i.e.* the observable state variables of the component sm. In this example, there is no message mapping because both services use the same msg message (declared in the default Kmelia library).

This example is used for the experimentation on the formal analyses described in the next section.

4 Formal Analysis and Experimentations

The formal analysis of a Kmelia specification consists in checking various kind of properties at the Kmelia model. The verification goal is to detect the specification errors: the violation policy is postponed to implementation issues. Some analysis are performed directly in the COSTO tool which supports Kmelia, the others are delegated to appropriate external tools. In this section, we address those aspects related to the new features introduced in Sect. 2. We show how the Rodin framework that supports the Event-B notation is interesting to check the component consistency and the contract correctness.

4.1 Formal analysis

Kmelia concepts are analysed according to various facets such as safe type checking, consistency and correctness, communication integrity, deadlock freeness, assembly compatibility, promotion consistency, etc. The formal analysis individually validate the components before checking the assembly:

(i) each individual component specification must be validated once for all by checking the verification requirements given Table 1. When all the rules are checked, then the component specification is considered as correct, possibly put in libraries, and could be reused within assemblies or composites.

(ii) assembly or promotion links must only consider references to components previously shown as correct. Then the rules given in Table 2 are verified to validate the Kmelia assembly and composite.

Analysis	**Status of the work**
Static rules: Scope + name resolution + type-checking	implemented
Observability rules (see 2.2)	implemented
Component interface consistency	implemented
Services dependency consistency: $\mathcal{DI}$ well-formed vs. $\mathcal{I}$ and $\mathcal{D}$ (component) $\mathcal{DI}$ vs. $\mathcal{B}$ (eLTS)	implemented
Simple constraint checking (parameters, query, protocol, ...)	implementation in progress
Local eLTS checking (liveness, *false* guards, reachability ...) $deadlock-free(\mathcal{B})$	implemented with subprovides without expressions
Invariant consistency vs. pre/post conditions: provided services : $Inv^O_{old} \wedge Pre^O_{old} \wedge Post^O(r) \Rightarrow Inv^O$ $Inv_{old} \wedge Pre_{old} \wedge Post^O(r) \wedge Post^{NO}(r) \Rightarrow Inv$ required services : $Inv^V_{old} \wedge Pre^O_{old} \wedge Post^O(r) \Rightarrow Inv^V$	(a) checked via B tools (b) (see 4.4) (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 pre-condition (local+virtual) eLTS vs. subprovided service SP annotations Pre_{SP} the context must ensure the pre-condition (local)	study in progress

where the $_{old}$ prefix denotes the variable value before the service execution.

Table 1
Formal analysis of Kmelia components

The status *implemented* means that the analysis is implemented in the COSTO tool. The status *checked via L* means that we implemented a translator plugin to generate specications in the L language or tool.

Currently no result is directly inferred from the checking of the individual components to facilitate the check of the assembly. The considered rules are complementary.

The verification process used for the rule numbered (a), (b), ... will be detailed in Section 4.4 after a short analysis example in Section 4.2 and an introduction to the case tool principles in Section 4.3.

Analysis	**Status of the work**
Static rules: Scope + name resolution + type-checking	implemented
Observability rules: promoted variables	implemented
Link/sublink consistency: assembly and composite signature matching service dependency matching (subprovides, callrequires)	implemented
context mapping (*vmap*) and observability rules message mapping (*mmap*)	implemented
Assembly Link Contract correctness: $vmap(Pre^O_R) \Rightarrow Pre^O_P$ $Post^O_P \Rightarrow vmap(Post^O_R)$	 (d) (e)
Provided Promotion Link Contract correctness: PP is in the composite $vmap(Pre^O_{PP}) \Rightarrow Pre^O_P$ $Post^O_P \Rightarrow vmap(Post^O_{PP})$	 (f) checked via B tools (g) (see 4.4)
Required Promotion Link Contract correctness: RR is in the composite $vmap(Pre^O_R) \Rightarrow Pre^O_{RR}$ $Post^O_{RR} \Rightarrow vmap(Post^O_R)$	 (h) (i)
eLTS (behaviour) compatibility eLTS (behaviour) compatibility with *mmap*	checked via CADP and MEC implementation in progress

Table 2
Formal analysis of Kmelia assemblies and composites

The verification process used for the rule numbered (a), (b), ... is detailed in Section 4.4 after a short analysis example in Section 4.2 and an introduction to the case tool principles in Section 4.3.

4.2 Simple static analysis example

Let $depends_k$ be a relation between component services names defined as a part of the service dependency in a component C_k where $sm = \nu_k(m)$:

$$depends_k : \mathcal{N} \leftrightarrow \mathcal{N}$$
$$\forall(n, m) : depends_k \bullet (n \in cal_{sm}) \vee (n \in req_{sm}) \vee (n \in sub_{sm})$$

The *Link/sublink consistency* analysis on an assembly is intended to check whether the following property holds: the services in the sublinks are in the dependencies of the involved services (w.r.t *sublinks*).

$$\forall(l, sl) \in subs \mid l = (C_i, n_1, C_j, n_2) \wedge sl = (C_k, n_3, C_l, n_4) \bullet$$
$$((n_3, n_1) \in depends_i{}^* \vee (n_4, n_2) \in depends_j{}^*)$$

where $depends_k{}^*$ is the transitive closure of the service dependency $depends_k$.

4.3 Automated checking

As mechanisation is a means to assess design and development techniques based on formal methods, we developed an Eclipse-based framework named COSTO (COmponent Study TOolbox [5]) to support all the steps of component analysis and specification (see also Appendix B). COSTO is dedicated to the management of the Kmelia specifications and to the checking of the primary properties (syntax, types, static). It delegates the verification of complex properties such as deadlock freeness, component or assembly consistency to other more efficient tools, as illustrated Fig. 5. According to the kind of property to check we target one tool, in such a way that the property can be expressed and verified; we select the parts of the Kmelia specifications involved in the target property; they are translated into the input formalism of the targeted tool and then checked by the tool.

The behaviour compatibility of services and components was treated in [9] using model-checking techniques provided by existing tools (Lotos/CADP [3] and MEC [4]).

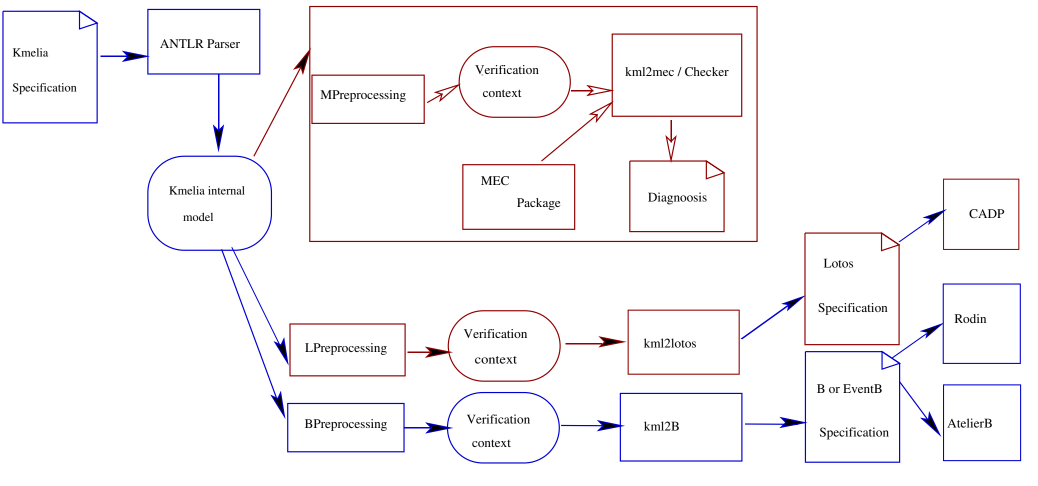

Fig. 5. A View of the COSTO Framework

In this article we focus on the bottom part of Fig. 5, related to the analysis of data and assertion properties.

4.4 Formal analysis of assertions

Our approach consists in reusing existing proof tools such as the B tools and especially the Atelier B [5] and Rodin [6] frameworks. The main issue is to present the verification of the necessary Kmelia elements in an appropriate manner to use efficiently the B provers. We design a systematic verification method that enables us to reuse the proof obligations generated by the B tools for our specific purpose.

[3] http://www.inrialpes.fr/vasy/cadp/
[4] http://altarica.labri.fr/wiki/tools:mec_4
[5] http://www.atelierb.eu/
[6] http://rodin-b-sharp.sourceforge.net

We are currently developing a plugin (named `Kml2B`) in the context of the COSTO tool. A first presentation of `Kml2B` is available in [22]. It extracts B specifications to enable the verification of invariant consistency rules (a, b, c) and to mechanise the proof for assembly link contract rules (d, e, f, g, h, i). In the following we present the systematic verification method and the manual experimentations done with Rodin.

Event-B and Rodin frameworks. Rodin is a framework made of several tools dedicated to the specification and proof of Event-B models. Event-B [1] extends the classical B method [2] with specific constructions and usage; it is used for the 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 for the Kmelia specification assertions. This approach was investigated before in [17,20] on the context of classical B and UML components.

We systematically build Event-B models, with an appropriate structure as explained below, to check a few of the proof obligations presented in Tables 1 and 2. Details and patterns which guide the Event-B models generation are given in [8].

(i) For each component and its provided services, we build 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 write an Event-B model. Its B consistency establishes the rule (c).

(iii) For each assembly link between a required service req and a provided one prov, we build 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 the rules (d) and (e) for assembly correctness.

(iv) For each promotion link between a provided service prov and its promoted one pprov, we build an Event-B model of the observable part of prov, which *refines* the Event-B model of the provided service prov 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 (f) and (g) for the Kmelia promotion correctness.

(v) For each promotion link between a required service req and a promoted one rreq, we build an Event-B model of the observable part of rreq, 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 (h) and (i) for the Kmelia promotion correctness.

We are not going to deal in this article with the details of the translation procedure and result[7]. For short, Kmelia invariant and pre-condition translations are quite systematic, whereas the post-condition concept does not exist in classical B. 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 [21]. In Event-B, translations of the post-conditions are quite systematic. Listing 8 depicts the Event-B translation of the service newReference of the Kmelia component StockManager.

Listing 8: Event-B specification for newReference

```
newReference =
ANY
  new_Result, new_catalog, new_pstock, new_plabels
WHERE
    card(catalog) < MaxRef
  ∧ new_Result ∈ INT
  ∧ new_catalog ∈ ℙ(References)
  ∧ finite (new_catalog)
  ∧ new_plabels ∈ References → String
  ∧ new_pstock ∈ References → INT
  ∧ ( (new_Result > 0 ∧ new_Result ≤ MaxRef) ∨ new_Result = NoReference )
  ∧ ( new_Result ≠ NoReference ⇒ new_Result /∈ catalog ∧ new_catalog = catalog ∪ {new_Result} )
  ∧ ( new_Result = NoReference ⇒ new_catalog = catalog )
  ∧ ( new_Result ≠ NoReference ⇒ new_pstock(new_Result) = 0
            ∧ new_plabels(new_Result) ≠ EmptyString
            ∧ ( ∀ ii . ( ii ∈ References ∧ ii ≠ new_Result ⇒ new_pstock(ii) = pstock(ii)
                   ∧ new_plabels(ii) = plabels( ii ) ) ) )
  ∧ ( new_Result = NoReference ⇒ new_pstock = pstock ∧ new_plabels = plabels )
THEN
    Result_newReference := new_Result
    catalog := new_catalog
    pstock := new_pstock
    plabels := new_plabels
END
```

4.5 Experimental results

Applying our method on the case study presented in Section 3, we obtain the Event-B models structured as depicted in Fig 6. 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 establish the preservation of the component invariant by the provided services. The model Vendor_addItem allows us to check the preservation of the virtual state space by the required service addItem.

[7] The Kmelia and Event-B specifications are available online at http://www.lina.sciences.univ-nantes.fr/coloss/download/facs09_app.pdf

Then the refined model v_addItem_sm_newReference is used to check the assembly link between the required service addItem and the provided one newReference.

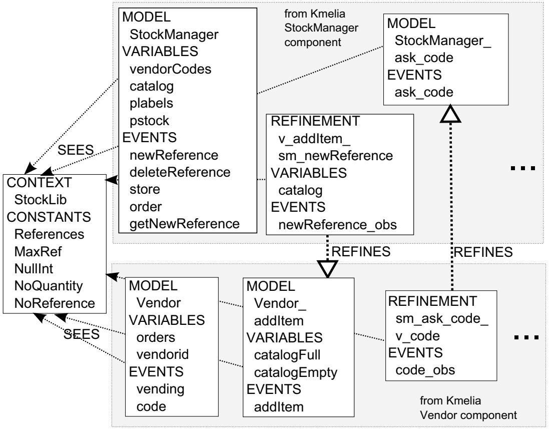

Fig. 6. Event-B Models

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

Table 3
Rodin Proof obligations

We did not experiment the promotion correctness on this example, but it follows the same schema as the assembly correctness where the context mapping *vmap* is replaced by the promotion mapping *pvars*.

Table 3 gives an idea about the number of POs that are to be discharged to ensure the correctness of the Kmelia specification. Studying the example within Rodin revealed some errors in our initial Kmelia specification. As example, the first version of post-condition of the service newReference was wrong; one of the associated POs could not be discharged. We discovered that some conditions were missing in the case we have variables which are not updated. As a feedback in our Kmelia specifications, the error was corrected, and the PO discharged thereafter.

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

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 assertion verification using B tools: the contracts are checked through preliminary experimentations using the Rodin framework. We have illustrated the work with an example which is specified in Kmelia, translated manually but systematically, 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 [11] or ArchJava [3] models do not allow the use of contracts. We have already emphasised (see Section 1) the fact that most of the abstract models deal mainly with the dynamic part of components. Some of them [18,27] take datatypes and contracts into account but not the dynamic aspects. Some other ones [13,15] delay the data part to the implementation level.

In [16] *may/must* constraints are associated to the interactions defined in the component interfaces to define behavioural contracts between clients 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 [17,20] in the context of UML components.

Fractal [19] proposes different approaches based on the separation of concerns: the structural features are defined in Fractal ADL [25] ; dynamic behaviours are implemented by Vercors [10] or Fractal/SOFA [14] and the use of assertions is studied in ConFract [18]. In ConFract contracts are independent entities 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.

In [12] a component is a model in the sense of the algebraic specifications. Dynamic behaviours are associated to components but not to services, which are simple operations. The component provided and required interfaces are type specifications and composing component is based on interface (or type) refinement. In Kmelia components are assembled on their services; therefore the main issue is not to refine types as in [12] but rather to check contracts as in [29]. More specifically our case is more related to the *plugin matching* of [29].

Perspectives. Several aspects remain to be dealt 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 which is necessary to automatically derive the needed Event-B models to check the assertions and the assemblies (a part of this work is already done with the *Kml2B* plugin). Second, we will integrate high level concepts and relations for data types. In particular, we plan to integrate ideas from objects and inheritance in the type system and also in component typing. But, assertions in this context are more difficult to specify and to verify. Another challenging point is the support for interoperability with other component models. In some real component software, 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 enable the use of contracts (resp. service composition, service behaviour) we can provide the 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] Pascal André, Gilles Ardourel, and Christian Attiogbé. A Formal Analysis Toolbox for the Kmelia Component Model. In *Proceedings of ProVeCS'07 (TOOLS Europe)*, number 567 in Technichal Report. ETH Zurich, 2007.

[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. http://www.lina.sciences.univ-nantes.fr/coloss/download/sc07.pdf.

[8] 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)*, ENTCS, 2010. to be published.

[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. http://www.lina.sciences.univ-nantes.fr/coloss/download/sc06.pdf.

[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] Michel Bidoit and Rolf Hennicker. An algebraic semantics for contract-based software components. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, volume 5140 of *LNCS*, pages 216–231. Springer, July 2008.

[13] 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.

[14] Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý, and Petr Tůma. *CoCoME in SOFA*, volume 5153, pages 388–417. Springer-Verlag, 2008.

[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, 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 *Software Composition, 6th International Symposium (SC 2007)*, volume 4829, pages 18–33, 2007.

[19] Thierry Coupaye and Jean-Bernard Stefani. Fractal Component-Based Software Engineering. In *Object-Oriented Technology, ECOOP 2006*, pages 117–129. Springer, 2006.

[20] 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.

[21] 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.

[22] M. Messabihi, P. André, and C. Attiogbé. Preuve de cohérence de composants Kmelia à l'aide de la méthode B. In *4e Conférence francophone sur les Architectures Logicielles (CAL'2010)*, 2010. to be published.

[23] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd edition, 1997. http://www.eiffel.com/doc/oosc/.

[24] B. Meyer. The Grand Challenge of Trusted Components. In *Proceedings of 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.

[25] ObjectWeb Consortium. Fractal adl. [Online]. Available: http://fractal.ow2.org/fractaladl/index.html. [Accessed: Jun. 17, 2009], 2009.

[26] 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.

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

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

[29] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4):333–369, 1997.

A A generic syntax of expressions and assertions

This appendix summarises the syntax of types, expressions and assertions.

Basic types such as Integer, Boolean, Char, String with their usual operators and semantics are permitted. Abstract data types like record, enumeration, range, array and set 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.

$$
\begin{array}{lcl}
TypeDef & ::= & TypeName \\
 & | & \text{"struct" "\{"}\ TypeDecl\ (\text{","}\ TypeDecl)^*\ \text{"\}"} \\
 & | & \text{"array" "["}\ LiteralValue_1\ \text{".."}\ LiteralValue_2\ \text{"]" "of"}\ TypeName \\
 & | & \text{"enum" "\{"}\ KmlExpr\ (\text{","}\ KmlExpr)^*\ \text{"\}"} \\
 & | & \text{"range"}\ LiteralValue_1\ \text{".."}\ LiteralValue_2 \\
 & | & \text{"setOf"}\ TypeName
\end{array}
$$

A Kmelia *expression* is built with constants, variables and elementary expressions built with standard arithmetic and logical operators like $(+, *, mod, <, >=, != , or, and, implies, not, ...)$. An assignment is made of a variable at the left hand side and an expression at the right hand side. In the following, each identifier class is denoted by a non-terminal symbol (C : constants, V : variables, O : operators, T : types). Identifiers are symbols built on letters, digits and the "_" character according to the usual rules. The third rules includes the function calls.

$$
\begin{array}{lcl}
KmlExpr & ::= & LiteralValue \\
 & | & V \mid C \\
 & | & \text{"("}\ KmlExpr\ \text{")"} \\
 & | & O\ KmlExpr \\
 & | & O\ \text{"("}\ KmlExpr_0, ...\ KmlExpr_n\ \text{")"} \\
 & | & KmlExpr_1\ O\ KmlExpr_2
\end{array}
$$

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 are also predicates. All the assertions must conform to the observability rules described in Section 2.2.

$$
\begin{array}{lcll}
Pred & ::= & \text{"@"}\ name\ \text{":"}\ Cond & \text{/* condition */} \\
Cond & | & KmlExpr & \text{/* boolean expression */} \\
 & | & (\text{" exists "} \mid \text{" forall "}) & VarDecl\ \text{"|"}\ Cond
\end{array}
$$

B Tool Demo

The COmponent STudy TOolkit (COSTO) toolbox is a prototype designed to support the Kmelia abstract component model and the associated verifications techniques, either directly or through the exportation of the relevant parts of a Kmelia model according to a verification context into provers or model-checkers.

The COSTO toolbox consists of:

- a core module with an ANTLR-based parser and an API to access the Kmelia (internal) model,
- several verification and exportation modules,
- a set of eclipse plugins.

Figure B.1 shows the Kmelia editor in eclipse, and a sample of the kind of errors (typing, observability, incompleteness of the mapping) that are detected. Besides standard completion, the editor supports smart completion in the case of assembly links. In Figure B.2, only required services defined in the Vendor component type are proposed and the user is warned that some of them do not match the exact signature of the provided service new_reference which is defined in the StockManager component type.

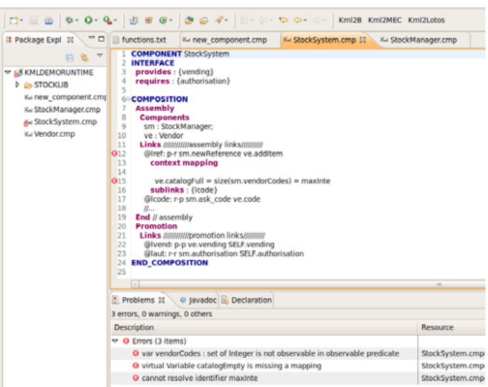

Fig. B.1. Kmelia editor in eclipse

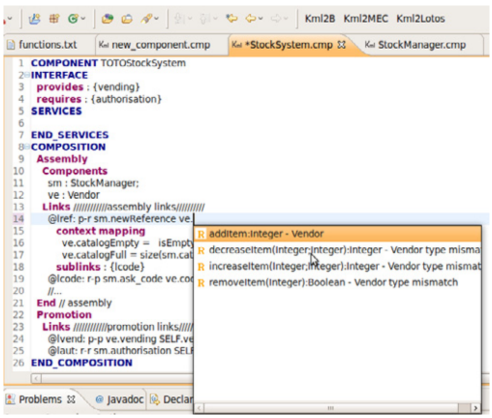

Fig. B.2. Smart completion

Using the exportation buttons on the top of the editor while selecting an assembly link generate models in MEC, LOTOS or B to be verified in external tools.

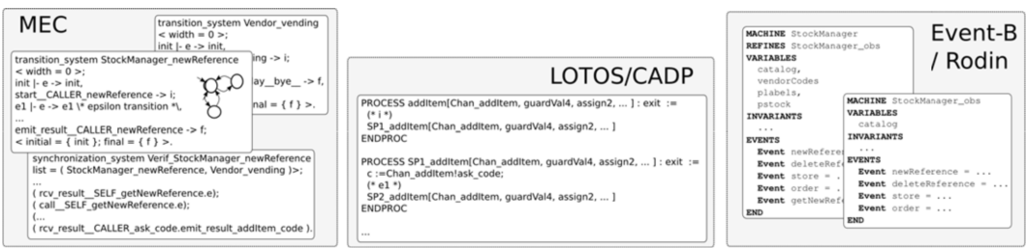

Fig. B.3. Exportations to MEC/LOTOS/Event-B