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 perform 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 \( \langle W, Init, A, N, M, I, D, \nu, C \rangle \) with:

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

\(^1\) \( \sqcup \) 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 \((\mathcal{D}^R)\).

• \( \nu : \mathcal{N} \rightarrow \mathcal{D} \) is the function mapping service names to service descriptions. Moreover there is a projection of the \( \mathcal{N} \) partition on its image by \( \nu \):
  \[
  s \in \mathcal{N}^P \Rightarrow \nu(s) \in \mathcal{D}^P \quad \land \quad s \in \mathcal{N}^R \Rightarrow \nu(s) \in \mathcal{D}^R
  \]

• \( \mathcal{CS} \) 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 \((\mathcal{IS}, \mathcal{IW}, \mathcal{IInit}, \mathcal{B})\) with:

• The service interface \( \mathcal{IS} \) is defined by a 6-tuple \( \langle \sigma, \mu, vW, \text{Pre}, \text{Post}, \mathcal{DI} \rangle \) where
  - \( \sigma \) is the service signature \( \langle \text{name}, \text{param}, \text{ptype}, \text{res} \rangle \) with \( \text{name} \in \mathcal{N} \), \( \text{param} \) a set of parameters, \( \text{ptype} : \text{param} \rightarrow T \) the function mapping parameters to types and \( \text{res} \in T \) the service result type;
  - \( vW = \langle vT, vV, vtype, vInv \rangle \) is a *virtual state space* with \( vT \) a set of types, \( vV \) a set of variables, \( vtype : vV \rightarrow vT \) the function mapping context variables to types and \( vInv \) an invariant defined on \( vV \);
  - \( \mu \) 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;
  - \( \text{Pre} \) is a pre-condition defined on the union \((\cup)\) of the variables in \( V \), \( vV \), and \( \text{param} : V \cup vV \cup \text{param} \); and
  - \( \text{Post} \) is a post-condition defined on \( V \cup vV \cup \text{param} \cup \{ \text{result} \} \);
  - \( \mathcal{DI} \) is the *service dependency*; it is composed by services on which the current service depends on. \( \mathcal{DI} \) is a 4-tuple \( \langle \text{sub}, \text{cal}, \text{req}, \text{int} \rangle \) of disjoint sets where \( \text{sub} \subseteq \mathcal{N}^P \) (resp. \( \text{cal} \subseteq \mathcal{N}^R \), \( \text{req} \subseteq \mathcal{N}^R \), \( \text{int} \subseteq \mathcal{N}^P \)) 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 \( c : C \)
Andre et al.

- $lW = \langle lT, lV, ltype, lInv \rangle$ 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 $B$ 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 $A$ 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 $vW$ 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 according to the following table:

<table>
<thead>
<tr>
<th>Service state space</th>
<th>Variables</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Observable part</td>
<td>Non-observable part</td>
</tr>
<tr>
<td>Provided s</td>
<td>$V^O$</td>
<td>$V$</td>
</tr>
<tr>
<td>Required s</td>
<td>$vV$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Service Assertions scope</th>
<th>pre-condition</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Observable</td>
<td>Non-observable</td>
</tr>
<tr>
<td>Provided s</td>
<td>$V^O \cup param$</td>
<td>none</td>
</tr>
<tr>
<td>Required s</td>
<td>$vV \cup param$</td>
<td>$V \cup param$</td>
</tr>
</tbody>
</table>

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](image-url)
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 $W$ local to a service is used only in the service behaviour $B$ 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 $sp$ ($V_{CP}^{O}$) by a mapping (total) function $vmap : vV_{sr} \rightarrow \text{exp}(V_{CP}^{O})$ where $\text{exp}(X)$ 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 $mmap : \text{mname}_{sr} \rightarrow \text{mname}_{sp}$.

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 $vo \in V_{C}^{O}$ 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 $prom : V_{C}^{O} \rightarrow V_{CP}$ 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 ($\text{type}(vp) = \text{type}(vo)$) 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 DI. For example, the required service addItem provides a special code subservice. 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 and the ask_code service

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

\[\text{5 which is also a subservice because it is not exposed in the StockManager component interface.}\]
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
INARIANT
    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 |
| inrequires : {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 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 @resultInRange: ((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} |
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/-CADP7 and MEC8); 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 COmp Onenent Study Toolkit dedicated to the Kmelia language
7 http://www.inrialpes.fr/vasy/cadp/
8 http://altarica.labri.fr/wiki/tools:mec_4
Andre et al.

B tools and especially the Rodin 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

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static rules:</strong> Scope + name resolution + type-checking</td>
<td>done</td>
</tr>
<tr>
<td><strong>Observability rules</strong> (see ??)</td>
<td>in progress</td>
</tr>
<tr>
<td><strong>Component interface consistency</strong></td>
<td>done</td>
</tr>
<tr>
<td>Services dependency consistency:</td>
<td></td>
</tr>
<tr>
<td>DI well-formed vs. I and D (component)</td>
<td>done</td>
</tr>
<tr>
<td>DI vs. B (eLTS)</td>
<td>done</td>
</tr>
<tr>
<td><strong>Simple constraint checking</strong> (parameters, query, protocol, ...)</td>
<td>in progress</td>
</tr>
<tr>
<td><strong>Local eLTS checking</strong> (deadlocks, guard, subprovides, ...)</td>
<td>in progress</td>
</tr>
<tr>
<td><strong>Invariant consistency vs. pre/post conditions:</strong></td>
<td></td>
</tr>
<tr>
<td>provided services: ( \text{Inv}^O \land \text{Pre}^O \Rightarrow \text{Post}^O \land \text{Inv}^O )</td>
<td>experimental (a)</td>
</tr>
<tr>
<td>( \text{Inv} \land \text{Pre} \Rightarrow \text{Post}^O \land \text{Inv} )</td>
<td>experimental (b)</td>
</tr>
<tr>
<td>required services: ( v\text{Inv} \land \text{Pre}^O \Rightarrow \text{Post}^O \land v\text{Inv} )</td>
<td>experimental (c)</td>
</tr>
<tr>
<td>Consistency between service assertions and eLTS:</td>
<td></td>
</tr>
<tr>
<td>eLTS vs. Post the post condition should be established</td>
<td>not yet</td>
</tr>
<tr>
<td>required service ( R ) calls vs. ( \text{Pre}_R ) the context must ensure the precondition (local+virtual)</td>
<td></td>
</tr>
<tr>
<td>eLTS vs. subprovided service ( SP ) annotations ( \text{Pre}_SP ) the context must ensure the precondition (local)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static rules:</strong> Scope + name resolution + type-checking</td>
<td>done</td>
</tr>
<tr>
<td><strong>Observability rules:</strong> promoted variables</td>
<td>done</td>
</tr>
<tr>
<td><strong>Link/sublink consistency:</strong> assembly and composition</td>
<td></td>
</tr>
<tr>
<td>signature matching</td>
<td>done</td>
</tr>
<tr>
<td>service dependency matching (subprovides, callrequires)</td>
<td></td>
</tr>
<tr>
<td>context mapping (( cm ) function) and observability rules</td>
<td></td>
</tr>
<tr>
<td>message mapping</td>
<td></td>
</tr>
<tr>
<td><strong>Assembly Link Contract correctness:</strong></td>
<td></td>
</tr>
<tr>
<td>( cm(\text{Pre}^O_R) \Rightarrow \text{Pre}_P^O )</td>
<td>experimental (d)</td>
</tr>
<tr>
<td>( \text{Post}_P^O \Rightarrow cm(\text{Post}^O_P) )</td>
<td>experimental (e)</td>
</tr>
<tr>
<td><strong>Provided Promotion Link Contract correctness:</strong> PP is at the composite level</td>
<td></td>
</tr>
<tr>
<td>( cm(\text{Pre}^O_R P) \Rightarrow \text{Pre}_P^O )</td>
<td>experimental (f)</td>
</tr>
<tr>
<td>( \text{Post}^O_P \Rightarrow cm(\text{Post}^O_P P) )</td>
<td>experimental (g)</td>
</tr>
<tr>
<td><strong>Required Promotion Link Contract correctness:</strong> RR is at the composite level</td>
<td></td>
</tr>
<tr>
<td>( cm(\text{Pre}^O_R) \Rightarrow \text{Pre}_R^O )</td>
<td>experimental (h)</td>
</tr>
<tr>
<td>( \text{Post}_R^O \Rightarrow cm(\text{Post}^O_R) )</td>
<td>experimental (i)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>elTS (behaviour) compatibility</strong> [?]</td>
<td>done</td>
</tr>
</tbody>
</table>

Table 1

**Formal analysis of a simple Kmelia component**

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

\[ http://rodin-b-sharp.sourceforge.net \]
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 \textsc{Invariant} by the \textsc{Events}. Other POs ensure that a \textit{refined} model is consistent, i.e. the abstract \textsc{Invariant} is preserved and the refined events do not contradict their abstract counterparts.

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

\textbf{Verifying \textsc{Kmelia} specifications using Event-B.} The main idea is, first to consider a part of the \textsc{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 \textsc{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 \textsc{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 \textit{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 \textsc{Kmelia} level;
- the refinement proof establishes both the rules (d) and (e) for the \textsc{Kmelia} assembly correctness.

We are not going to deal in this article with the details of the translation procedure. \textsc{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 \textsc{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 \texttt{newReference} of \textsc{StockManager}.

\textbf{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 \textsc{Kmelia} components \textsc{StockManager} and \textsc{Vendor} before checking the assembly \textsc{StockSystem}. The Event-B model \textsc{StockManager} is used to prove the preservation of the invariant assertions by the provided services. The refinement \texttt{v_addItem_sm_newReference} is used to check the assembly link between the services \texttt{newReference} and \texttt{addItem}. The Table ?? 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, reveals some errors in our initial Kmelia specification. For example, the post-condition of \texttt{newReference} was wrong; one of the associated POs could not be discharged. After the feedback in our Kmelia specifications, the error was corrected.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Service & Auto & Manual & Total \\
\hline
StockManager & 16 & 3 & 19 \\
Vendor\_addItem & 2 & 1 & 3 \\
v\_addItem\_sm\_newReference & 22 & 1 & 23 \\
\hline
\end{tabular}
\caption{Rodin Proof obligations}
\end{table}

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


A The Vendor Component Partial Specification

Listing 4: Kmelia specification Vendor

```plaintext
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
  # The main (provided) service is vending.
  provided vending ()
  Interface
    extraquires : {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 — { 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,

```
Andre et al.

e1 \rightarrow \text{ref} := \text{readInt()} \rightarrow e2,
e2 \rightarrow \left[ \text{choice} = \text{remove} \right] \_\text{removel} !! \text{removel}(\text{ref}) \rightarrow e20,
e2 \rightarrow \left[ \text{choice} = \text{store} \right] \{ \_\text{increaseIt} !! \text{increaseIt} (\text{ref}, \text{readInt()}) \} \rightarrow e30,
e2 \rightarrow \left[ \text{choice} = \text{order} \right] \_\text{decreaseIt} !! \text{decreaseIt} (\text{ref}, \text{readInt()}) \rightarrow e40.

// add item

\text{e10} <<\text{code>>}, \# \text{subservice code is available here}

\text{e10} \rightarrow \{ \text{desc} := \text{readString()}; \ // \text{product description} 
\_\text{addItem} ! \text{msg(desc)} \} \rightarrow \text{e11},
\text{e11} \rightarrow \_\text{addItem} ?? \text{addItem}(\text{pi}) \rightarrow \text{e12},
\text{e12} \rightarrow \{ \text{if} (\text{pi} <> \text{noReference}) 
\text{then display}("\text{New reference: }\text{asString(\text{pi})}\) 
\text{endif} \} \rightarrow \text{i}
B The derived Event-B models

B.1 StockLib

CONTEXT StockLib

EXTENDS Default

CONSTANTS

References
MaxRef
NullInt
NoQuantity
NoReference

AXIOMS

\[ \text{axm5: } References = 1 \ldots \text{MaxRef} \]
\[ \text{axm1: } \text{MaxRef} = 100 \]
\[ \text{axm2: } \text{NullInt} = -1 \]
\[ \text{axm3: } \text{NoQuantity} = -2 \]
\[ \text{axm4: } \text{NoReference} = -3 \]

END

B.2 StockManager

MACHINE StockManager

SEES StockLib

VARIABLES

vendorCodes
catalog obse
plabels
pstock
Result newReference obse

ININVARIANTS

inv5 : vendorCodes ⊆ Z
inv2 : catalog ∈ P(References)
obse
inv7 : finite(catalog)
obse
inv3 : plabels ∈ 1 \ldots \text{MaxRef} \rightarrow \text{String}
inv4 : pstock ∈ 1 \ldots \text{MaxRef} \rightarrow \mathbb{Z}
• borned : card(catalog) ≤ \text{MaxRef}
obse
• referenced : \forall ref1 \cdot (ref1 ∈ References \land ref1 ∈ catalog \Rightarrow \text{plabels(ref1)} ≠ \text{EmptyString} \land \text{pstock(ref1)} ≠ \text{NoQuantity})
• notreferenced : \forall ref2 \cdot (ref2 ∈ References \land ref2 \notin catalog \Rightarrow \text{plabels(ref2)} = \text{EmptyString} \land \text{pstock(ref2)} = \text{NoQuantity})
inv6 : Result newReference ∈ Z

EVENTS

Initialisation

begin
act1 : vendorCodes := \emptyset
act2 : catalog := \emptyset
act3 : plabels := (1 \ldots \text{MaxRef}) \times \{ \text{EmptyString} \}
act4 : pstock := (1 \ldots \text{MaxRef}) \times \{ \text{NoQuantity} \}
act5 : Result newReference := 0
end

Event newReference \equiv

any
new Result
new catalog
new pstock
new plabels
where
grd8 : card(catalog) < \text{MaxRef}
obse
grd1 : new Result ∈ Z

\text{References}
\text{MaxRef}
\text{NullInt}
\text{NoQuantity}
\text{NoReference}

\text{axm5: } References = 1 \ldots \text{MaxRef}
\text{axm1: } \text{MaxRef} = 100
\text{axm2: } \text{NullInt} = -1
\text{axm3: } \text{NoQuantity} = -2
\text{axm4: } \text{NoReference} = -3
\text{vendorCodes}
\text{catalog obse}
\text{plabels}
\text{pstock}
\text{Result newReference obse}
\text{inv5: } vendorCodes \subseteq \mathbb{Z}
\text{inv2: } catalog \in P(References)
obse
\text{inv7: } \text{finite}(catalog)
obse
\text{inv3: } \text{plabels} \in 1 \ldots \text{MaxRef} \rightarrow \text{String}
\text{inv4: } \text{pstock} \in 1 \ldots \text{MaxRef} \rightarrow \mathbb{Z}
\text{borned: } \text{card}(catalog) \leq \text{MaxRef}
obse
\text{referenced: } \forall \text{ref1} \cdot (\text{ref1} \in References \land \text{ref1} \in catalog \Rightarrow \text{plabels(ref1)} \neq \text{EmptyString} \land \text{pstock(ref1)} \neq \text{NoQuantity})
\text{notreferenced: } \forall \text{ref2} \cdot (\text{ref2} \in References \land \text{ref2} \notin catalog \Rightarrow \text{plabels(ref2)} = \text{EmptyString} \land \text{pstock(ref2)} = \text{NoQuantity})
\text{inv6: } \text{Result newReference} \in \mathbb{Z}
obse
\text{act1: } \text{vendorCodes} := \emptyset
\text{act2: } \text{catalog} := \emptyset
\text{act3: } \text{plabels} := (1 \ldots \text{MaxRef}) \times \{ \text{EmptyString} \}
\text{act4: } \text{pstock} := (1 \ldots \text{MaxRef}) \times \{ \text{NoQuantity} \}
\text{act5: } \text{Result newReference} := 0
\text{end
\text{newReference} \equiv
\text{any
\text{new Result
\text{new catalog
\text{new pstock
\text{new plabels
\text{where
\text{grd8: } \text{card}(catalog) < \text{MaxRef}
obse
\text{grd1: } \text{new Result} \in \mathbb{Z}
obse
Andre et al.

grd2 : new_catalog ∈ P(References)
obs
grd11 : finite(new_catalog)
obs
grd3 : new_plabels ∈ 1 . . MaxRef → String
grd4 : new_pstock ∈ 1 . . MaxRef → Z
grd5 : (new_Result > 0 ∧ new_Result ≤ MaxRef) ∨ new_Result = NoReference
obs
grd6 : new_Result ≠ NoReference ⇒
new_Result ∉ catalog ∧ new_catalog = catalog ∪ {new_Result}
obs
grd7 : new_Result = NoReference ⇒ new_catalog = catalog
obs
grd8 : new_Result ≠ NoReference ⇒
new_pstock(new_Result) = 0 ∧
new_plabels(new_Result) ≠ EmptyString ∧
(∀ ii · ii ∈ 1 . . MaxRef ∧ ii ≠ new_Result ⇒
new_pstock(ii) = pstock(ii) ∧
new_plabels(ii) = plabels(ii))
obs
end

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 ∈ BOOL
inv2 : catalogEmpty ∈ BOOL
• notFullEmpty : ¬(catalogEmpty = TRUE ∧ catalogFull = TRUE)
inv4 : Result_addItem ∈ Z

EVENTS

Initialisation

begin
act1 : catalogFull := FALSE
act2 : catalogEmpty := TRUE
act3 : Result_addItem := Z
end

Event addItem ≡

any
new_Result
new_catalogEmpty
new_catalogFull

where
pre_addItem : ¬(catalogFull = TRUE)
grd2 : new_Result ∈ Z
grd6 : new_catalogEmpty ∈ BOOL
grd5 : new_catalogFull ∈ BOOL
Post_addItem : new_Result ≠ NoReference ⇒
new_catalogEmpty = FALSE ∧
new_catalogFull = BOOL
Post_addItem2 : new_Result = NoReference ⇒
new_catalogEmpty = catalogEmpty ∧
new_catalogFull = catalogFull
then
addItem : 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(card(catalog))
borning : card(catalog) ≤ MaxRef
assemblyEmpty : catalogEmpty = bool(card(catalog) = 0)
assemblyFull : catalogFull = bool(card(catalog) = MaxRef)

EVENTS

Initialization

begin

act1 : catalogFull := FALSE
act2 : catalogEmpty := TRUE
act3 : Result addItem ∈ Z
act4 : catalog := Ø

end

Event newReference ≡

refines  addItem

any

new_Result
new_catalog

where

pre_newReference : card(catalog) < MaxRef
grid11 : new_Result ∈ Z
grid64 : new_catalog ∈ P(References)
grid10 : 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}
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

21