Contract-based Verification of Kmelia Component Assemblies using Event-B

Pascal André[1] Gilles Ardourel[1] Christian Attiogbé[1]
Arnaud Lanoix[1]

COLOSS Team
LINA CNRS UMR 6241
University of Nantes
France

Abstract

Building reliable software systems from components requires to verify the consistency of components and the correctness of component assemblies. In this work, we design a verification method to address the problem of verifying the consistency of component states and the correctness of assembly contracts, using pre-/post-conditions. The starting point is specifications written with Kmelia: a Kmelia component type declares provided and required services which are used to link components in component assemblies. From these Kmelia specifications we generate Event-B models in such a way that we can check the consistency and also the correctness of assembly at the Kmelia level, using Event-B provers. An illustrative example based on a stock management system is used to support the presentation.

Keywords: Components, services, contracts, assembly, formal verification, Event-B

1 Introduction

Component-based software engineering is a practical approach to address the issue of building large software by combining existing and new components. Establishing the correctness of such software systems is still a challenging concern. In this context we proposed an abstract and formal model, named Kmelia [9,6], with an associated language to specify components, their provided and required services and their assemblies. Our main concern is to establish their correctness; preliminary works are presented in [8]. The COSTO framework [4] is dedicated for directly doing some local checkings (as type-checking) and for extracting from the Kmelia specifications the necessary inputs to reuse other verification tools, as illustrated in Fig. 1. Currently LOTOS/CADP and MEC are used to check the behavioural compatibility [9,5].

Among the formal analysis necessary to ensure the global correctness, we consider in this article: (i) the component invariant consistency vs. pre-/post-conditions of each provided/required service; (ii) the Kmelia assembly link contract correctness, that relates

[1] Email: Firstname.Lastname@univ-nantes.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

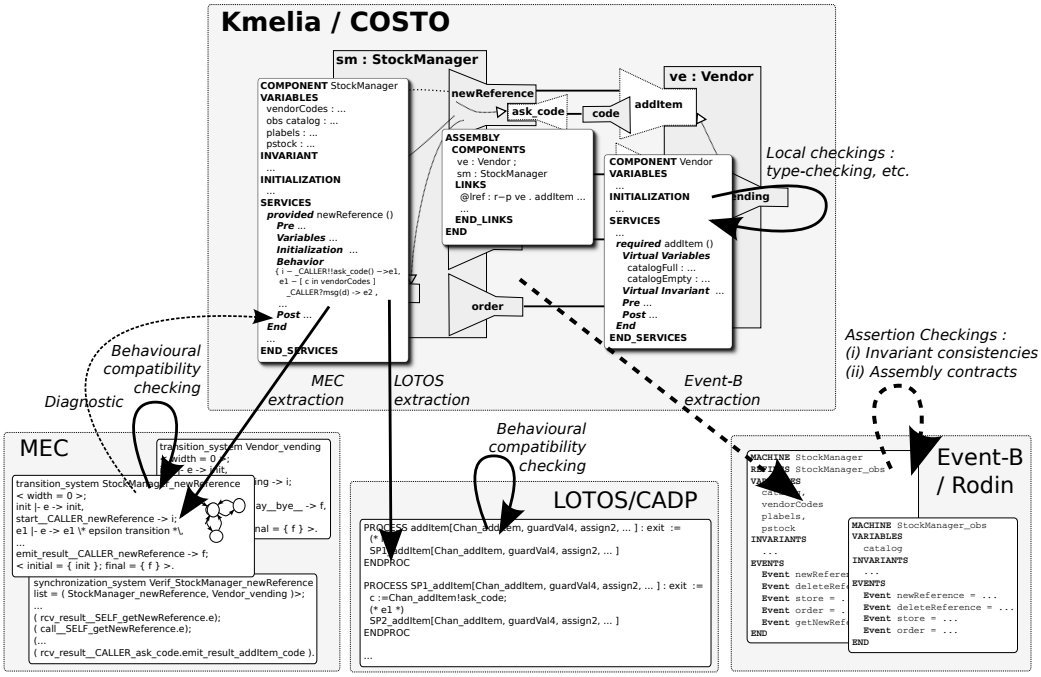


Fig. 1. Kmelia/COSTO framework overview

services which are linked in the assemblies. We successfully apply the *design-by-contracts* paradigm [14]. In [17] several cases of matchings and contracts between (behaviours of) components were considered. Our case is more linked with *plugin matching*: a required service must match with a provided service in an assembly link; it corresponds to the following relation between an existing behaviour S and a given behaviour Q intended to match S: the pre-condition of Q is at least as strong as the pre-condition of S and the post-condition of S is at least as strong as the post-condition of Q.

The main contribution of this work is as follows. We design a verification method that uses Event-B to check the consistency of Kmelia components and the correctness of their assembly contracts. We show how to generate the necessary Event-B models from parts of the Kmelia specifications we want to verify. We design Event-B patterns to guide the extraction and produce the necessary proof obligations. Then, we describe how these proof obligations at the Event-B level are linked with the attempted proofs at the Kmelia level: checking the Event-B invariant preservation is related to Kmelia invariant consistency. Event-B refinements are used to check the correctness of Kmelia component assemblies. The Rodin framework [16] is used to achieve the formal analysis.

Using (Event-)B to validate components assembly contracts was investigated in [10,12]. The work in [10] was devoted to the use of B for component interfaces and assemblies: the component interfaces are specified using classical B and then the provided interface is proved to be a refinement of the required one. Development of adapters are also considered [11,12]. Our approach is quite similar with respect to the use of the refinement to check the assembly, but we start from complete component descriptions and target Event-B to prove properties related to the data part (invariant consistency, service behaviour consistency, assembly correctness). In [15], (web-)services are specified as B abstract machines

and service composition only considers the call contract by simply ordering B operations (sequence, loop, ...) in order to create new operations. In our work we focus on the clientship contract while the call contract that has to be verified locally to a component can be proved once for all.

The article is structured as follows. Section 2 gives an overview about Event-B concepts. In Section 3 we introduce the Kmelia concepts and the consistency of components and services. Section 4 is dedicated to the verification method proposed to establish the Kmelia component consistency using Event-B. Section 5 describes component assemblies and assembly link contracts. Then, Section 6 is devoted to the verification of assembly links contracts. Finally, in Section 7 we illustrate the proposed verification method with an assembly of components for a stock management system. The article is concluded in Section 8 where we give some perspectives.

2 Event-B

The B method [1] is a well known approach for formal specification and development of sequential computer programs. An extension known as Event-B [2,3] covers system modelling, reactive and distributed systems development. Event-B extends the classical B method with specific constructions and usage.

An Event-B abstract specification comprises a static part called the **CONTEXT** (defining **SETS**, **CONSTANTS** and their **AXIOMS**) and a dynamic part called the **MACHINE**. The machine is made of a state defined by means of **VARIABLES** and an **INVARIANTS** clause which contains predicates that constrain the variables. They are supposed to hold whenever variable values change.

```
MACHINE M1
VARIABLES x
INVARIANTS I1(x)
EVENTS
 Event E1 ≙
  any p where G1(p,x)
  then S1(p,x)
  end
END
```

```
MACHINE M2
REFINES M1
VARIABLES  y
INVARIANTS I2(x,y)
EVENTS
 Event E2 ≙
 refines E1
  any q where G2(q,y)
  then S2(q,y)
  end
END
```

Fig. 2. A machine `M1` and a possible refinement `M2`

Besides its state, a machine has an **EVENTS** clause containing a list of events that describe the way the machine evolves. Each event is made of a *guard* and an *action*. When the guard of an event is true, the event is enabled. When the guards of several events are true, the choice of the triggered event is non-deterministic. The action of an event determines the way the state variables evolve when the event occurs. In the event `E1` of Fig. 2, `p` denotes variables (i.e. parameters) that are local to the event[2], `G1(p,x)` is a predicate denoting the guard and `S1(p,x)` denotes the action that updates some variables, it is a collection of *assignments* as follows:

[2] When no local variables are used, the event becomes `E1` ≙ **when** `G1(x)` **then** `S1(x)` **end**

Assignment	**Before-after predicate**	
`x := E1(p,x)`	`x′ = E1(p,x)`	*(Deterministic)*
`x :| Q1(p,x,x′)`	`Q1(p,x,x′)`	*(non-deterministic)*

where `E1(p,x)` denotes an expression and `Q1(p,x,x′)` is a before-after predicate.

The semantics of an Event-B machine is expressed via proof obligations (POs), which must be discharged to ensure the *machine consistency*. For instance, for the non-deterministic assignment, we must prove (see [2]):

- *Feasibility:* `I1(x) ∧ G1(p,x) ⇒ ∃ x′. Q1(p,x,x′)`
- *Invariant preservation:* `I1(x) ∧ G1(p,x) ⇒ I1(x′)`

A refinement process is used to transform the abstract machine into a more concrete one, which **`REFINES`** the abstract machine. New variables can be introduced and the (old) abstract variables can be refined to more concrete ones. This is reflected in the substitutions of the events as well. A concrete event **`refines`** its abstraction 1) when the guard of the former is at least as strong as the guard of the latter (guard strengthening), and 2) when the concrete actions "realise" the abstract ones. New events may be introduced at the refined level. These new events should not prevent forever the old ones from being triggered.

The previous POs (feasibility and invariant preservation) are to be proved on the refined machine. Furthermore, new POs ensure that the refinement is correct, i.e. the refined events do not contradict their abstract counterpart. We have to discharge these POs (see [2]):

- *Guard strengthening:* `I1(x) ∧ I2(x,y) ∧ G2(q,y) ⇒ G1(p,x) ∧ I2(x′,y′)`
- *Action simulation:* `I1(x) ∧ I2(x,y) ∧ G2(q,y) ∧ Q2(q,y,y′) ⇒ ∃x′.Q1(p,x,x′)`

Event-B is accompanied with tool support in the form of a framework called Rodin [16]. POs corresponding to the Event-B models are automatically generated. Then, they can be discharged automatically or interactively, using the Rodin provers.

In this short presentation we omit some Event-B concepts: deadlock freeness POs, variant POs, decomposition of Event-B models, etc. A complete description of Event-B can be found in [2,3].

3 Kmelia Components, Services and Invariant Consistency

In this section first we describe the main features of a Kmelia component type, then we show how to establish its invariant consistency. Note that many details and features of Kmelia are omitted here; we focus only on the contract aspects and not on the behavioural ones. The reader will find detailed descriptions of Kmelia in [9,8].

3.1 Main features of a component

A *component type* is introduced by a **COMPONENT** clause identified by a specific name. The simplified component pattern of Fig. 3 illustrates the Kmelia language and the graphical representation.

The **INTERFACE** of a component declares the services that the component ***provides*** and ***requires***. The **TYPES** clause contains new type definitions. The **VARIABLES** clause includes the component state variables. It distinguishes the *observable* variables o, which

are visible from any client, from the other ones x which are supposed to be non-observable. The **INVARIANT** clause declares a conjunction of predicates that build the component state invariant property. The **INITIALIZATION** clause assigns values to state variables. The partition (observable/non-observable) propagates to the whole state space (*e.g.* invariant, initialisation) and the service assertions. For the sake of simplicity, the different parts of the pattern given in Fig. 3 are made abstract, for example inv(o) represents the invariant built using the variables o, init (o,x) represents the initialisation expression on the variables o and x, etc.

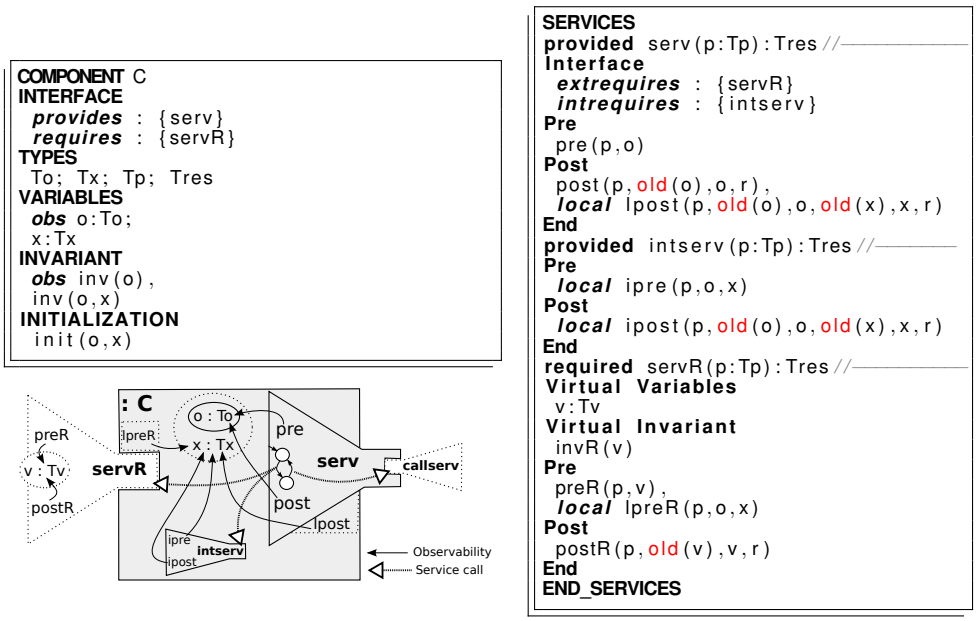


Fig. 3. Kmelia pattern C and its graphical form

The **SERVICES** clause lists the component services whose behaviours describe the functionalities offered by the component. A Kmelia *service* is more than a simple operation because it has (i) a dynamic behaviour defined by a state-transition system (which is formally an *extended Labelled Transition System* detailed in [9,6,7]), (ii) it can require or provide other services, through its interface, i.e. a description of its *service dependency*. That includes the declaration of the services required from any component (***extrequires***), and, it may also include the services locally required from its own component (***intrequires***).

Two kinds of services should be considered in a component:

- The **provided** services implement functionalities achieved and offered by the component; **Pre**-/**Post**-conditions are expressed on the component state variables to specify the effect of the service behaviours.
- the **required** services are abstractions of services provided by other components. The **Virtual** variables and invariant of a required service give the observable state of the "imaginary" component; **Pre** and **Post** denote the requirements about them.

Invariant and pre-conditions are classical first-order predicates with specific variable scopes, constrained by observability rules. Post-conditions are before-after predicates: in Kmelia, old(x) denotes the before value of variable x, when x denotes the after value. In Fig. 3, r denotes the returned result of the service where it is used. Note that ***local* Pre**-

/Post-conditions of a service serv are defined on variables of the whole state space of the component that defines the service serv. The following restrictions are due to observability: a required service has an empty local post-condition, a provided service which is in the provided interface of a component has an empty local pre-condition, whereas a provided service which is not in the provided interface of a component is called an *internal* service *e.g.* intserv.

3.2 Invariant consistency

Among the analysis we have to perform to establish a Kmelia component correctness, one is the component *invariant consistency*: the provided and required services do not break their corresponding component invariant. We assume this preservation at three levels of observability. Hence the following rules.

(i) The observable part of the component must be consistent, i.e. the observable part of the post-condition of a provided service is sufficient to establish the observable part of the invariant:

$$\mid \mathsf{inv(old(o))} \wedge \mathsf{pre(p,old(o))} \wedge \mathsf{post(p,old(o),o,r)} \Rightarrow \mathsf{inv(o)} \qquad \text{(INV/O)}$$

(ii) The complete invariant must be preserved by the provided services:

$$\begin{array}{l} \mid \mathsf{inv(old(o))} \wedge \mathsf{inv(old(o),old(x))} \wedge \mathsf{pre(p,old(o))} \\ \wedge \mathsf{post(p,old(o),o,r)} \wedge \mathsf{lpost(p,old(o),o,old(x),x,r)} \\ \Rightarrow \mathsf{inv(o)} \wedge \mathsf{inv(o,x)} \end{array} \qquad \text{(INV/F)}$$

and also by the internal services:

$$\begin{array}{l} \mid \mathsf{inv(old(o))} \wedge \mathsf{inv(old(o),old(x))} \wedge \mathsf{ipre(p,old(o),old(x))} \\ \wedge \mathsf{ipost(p,old(o),o,old(x),x,r)} \\ \Rightarrow \mathsf{inv(o)} \wedge \mathsf{inv(o,x)} \end{array} \qquad \text{(INV/F')}$$

(iii) The last verification concerns the consistency of the "imaginary" context, i.e. each required service preserves its associated virtual invariant:

$$\mid \mathsf{invR(old(v))} \wedge \mathsf{preR(p,old(v))} \wedge \mathsf{postR(p,old(v),v,r)} \Rightarrow \mathsf{invR(v)} \qquad \text{(INV/V)}$$

These rules should be checked to ensure the invariant consistency at the Kmelia level.

4 Formal Analysis of Kmelia Components using Event-B

In order to verify the rules defined in the previous section, we design a verification method that uses Event-B: we systematically build appropriate Event-B models, by extracting the necessary Kmelia elements in such a way that the Event-B proof obligations (POs) correspond to the specific rules we need to check at the Kmelia level. Then, we can discharge these POs using Rodin and interpret the proof results at the Kmelia level.

Remark 4.1 We are not going to deal in this article with the details of an automatic translation procedure from Kmelia to Event-B. The pattern in the listing of Fig. 4 (and the others) postpones a strong hypothesis about the translation of Kmelia expressions to Event-B; For

example we assume the same type universe (for basic types).

4.1 Verification of INV/O

For each Kmelia component, we generate a first Event-B model as illustrated by the pattern given in Fig. 4. The observable state variables and invariant of Kmelia become variables and invariant of the Event-B model `C_obs`. For each service provided in the component interface, a new event is built: parameters and pre-conditions become local variables and guards of the event, observable post-conditions become a non-deterministic assignment.

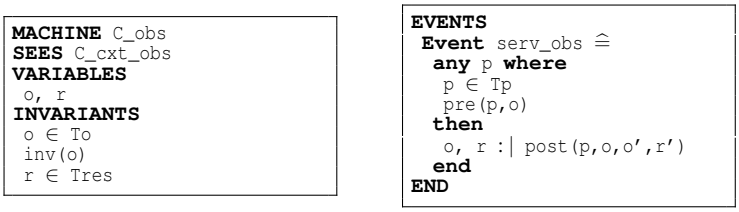


Fig. 4. Pattern `C_obs`

The Kmelia invariant and pre-condition translations are quite systematic. The post-conditions are quite directly translated into before/after predicates using the following procedure: we extract a list of the updated variables, then we build a non-deterministic assignment where the left hand side is these updated variables and the right hand side is the post-conditions viewed as a before-after predicate. We have to switch from the Kmelia notation to the one of Event-B; the before variables old(o) are used in Kmelia whereas the after variables `o'` are used in Event-B. Finally, the result variable r of each service becomes a global variable of the Event-B model; it is processed as the other state variables in the before/after predicate.

Remark 4.2 Another way to deal with post-conditions in Event-B consists to abstract a 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 [13].

Proposition 4.3 (INV/O-Consistency) *The Event-B proof of the consistency of the generated Event-B machines ensures the proof of the rule INV/O for the invariant consistency of the observable part at the Kmelia level.*

Proof. Consider the generated POs [2] for the consistency of `C_obs` by `serv_obs` :

(Feasibility)

$$o \in To \wedge inv(o) \wedge r \in Tres \wedge p \in Tp \wedge pre(p,o) \Rightarrow \exists\, o',r'.\ post(p,o,o',r')$$

(Invariant preservation)

$$o \in To \wedge inv(o) \wedge r \in Tres \wedge p \in Tp \wedge pre(p,o) \Rightarrow o' \in To \wedge inv(o') \wedge r' \in Tres$$

From *(Feasibility)* we deduce `post(p,o,o',r')`. Combined with *(Invariant preservation)*, we obtain exactly the expected rule INV/O and then we establish the proposition 4.3. □

4.2 *Verification of INV/F and INV/F'*

Now, we consider the whole component, not only its observable parts. We build another Event-B model (C) as a refinement of the previous one C_obs. The non-observable variables and invariants are added to the state of the model C. For each provided service, we refine the corresponding event by adding the local post-conditions to the assignment. Each internal service is also translated as a new event. The pattern in Fig. 5 gives an overview of the translation scheme.

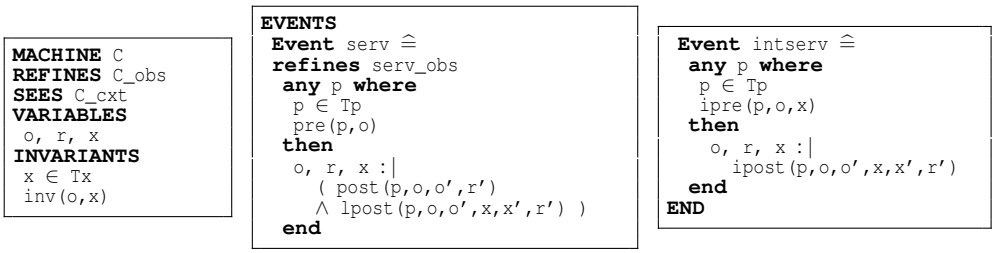


Fig. 5. Pattern C

Proposition 4.4 (INV/F/F'-Consistency) *Proving the refinement on the generated Event-B models results in the proof of the rules INV/F and INV/F' for the invariant consistency of the whole Kmelia component.*

Proof. The Event-B proof obligations concerning the refined event serv of C correspond exactly to the expected rule INV/F. Similarly, the POs concerning the new event intserv introduced by refinement ensure the rule INV/F'. □

4.3 *Verification of INV/V*

For each required service (and its associated "virtual context") we have to generate an Event-B model. The **Virtual** variables and **Virtual** invariant at the Kmelia level become respectively the variables and invariant of the Event-B model whereas the required service becomes the only one event of the model as shown in Fig. 6.

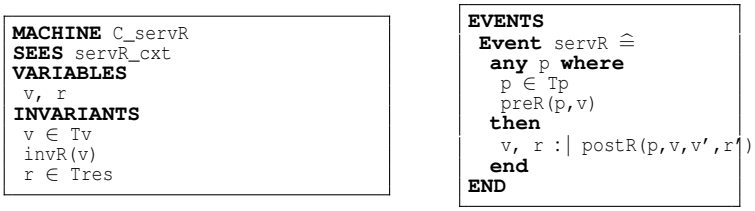


Fig. 6. Pattern C_servR

Proposition 4.5 (INV/V-Consistency) *The Event-B proof obligations of the model C_servR ensure the proof of the rule INV/V for the invariant consistency of the required service servR.*

Proof. As previously, the correspondence between the generated POs about the invariant preservation by C_servR and the Kmelia rule INV/V is obvious. □

5 Kmelia Assembly Links Correctness

In the following we consider Kmelia component assemblies and we show how an assembly contract is established. Again, only the part of Kmelia related to contracts is covered.

5.1 Component Assembly and Links

A Kmelia *assembly* is a set of components that are linked through their services. An assembly link associates a required service to a provided one. In order to introduce the services to be linked in an assembly, we consider another component-type A given in Fig. 7 in addition to the component type C already shown in the Listing of Fig. 3.

```
COMPONENT A
INTERFACE
  provides : {servP}
VARIABLES
  obs o:To
INVARIANT
  obs invP(o)
SERVICES
provided servP(p:Tp):Tres
Pre
  preP(p,o)
Post
  postP(p,old(o),o,r)
End
END_SERVICES
```

```
ASSEMBLY
 COMPONENTS
   c:C ;
   a:A
 LINKS
   r–p c.servR, a.servP
      Context mapping MAP(v,o)
   End
 END_LINKS
```

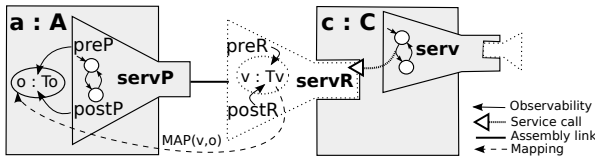


Fig. 7. Kmelia assembly pattern

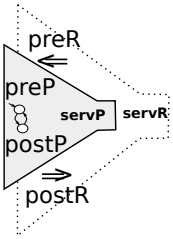
An **ASSEMBLY** is specified by two main clauses, as shown in Fig. 7: the **COMPONENTS** clause gives the instances of component type that we consider in the assembly and the **LINKS** clause indicates the *assembly links*. For the sake of simplicity we assume that the parameters of the two linked services are the same (or they have been renamed to be the same). A **Context mapping** specifies an expression MAP(v,o) which denotes how the variables v of the virtual context of the required service servR are mapped to the observable variables o of the component which offers the provided service servP.

If the linked services have service dependencies, then Kmelia requires the description of *assembly sublinks* according to the interface of services but we are not going to introduce them here.

5.2 Assembly contracts

The correctness of a link is established by: (i) the matching of the service signatures (with parameter renaming), (ii) the service dependency consistency, (iii) the respect of the service contracts and (iv) the behavioural compatibility between the linked services. The first two steps (i) and (ii) are handled by the Kmelia compiler. The behavioural compatibility (iv) definition and analysis is presented in [9].

In this article we focus on the contract part (iii) only. Informally this contract is as follows: whenever the required service servR is called, the provided service servP should apply. It is expressed by two rules PRE and POST given below:



$$invR(v) \wedge invP(o) \wedge MAP(v,o) \wedge preR(p,v) \Rightarrow preP(p,o) \qquad \text{(PRE)}$$

$$invP(old(o)) \wedge MAP(old(v),old(o)) \wedge postP(p,old(o),o,r) \Rightarrow postR(p,old(v),v,r) \qquad \text{(POST)}$$

Similarly to the component invariant consistency, the above rules are the starting point for the effective verification method.

6 Formal Analysis of Kmelia Assembly Links using Event-B

The main idea behind our method for checking the correctness of an assembly link is the following: we consider a provided service as a *refinement* of a required service. Practically, we convert the Kmelia assembly correctness rules to an Event-B refinement proofs; we generate the appropriate Event-B models such that their proof obligations correspond to the intended rules PRE and POST. Here the generated Event-B models concern the links between given required/provided services already checked to be consistent.

For each link between a required service servR and a provided one servP, we build an Event-B model which **REFINES** the Event-B model previously generated for the required service servR (as shown in Section 4). The corresponding refinement pattern is shown in Fig. 8: the observable variables of the provided service are added and the invariant is completed with the mapping `MAP(v,o)`. The pre-condition of the required service is preserved to avoid a guard strengthening. To ensure the PRE rule we add the expression $\forall$ `q. q∈Tp ∧ preR(q,v) ⇒ preP(q,o)` to the invariant; it expresses that the pre-condition of the required service is at least as strong as the one of the provided service in Kmelia assembly [3]. Finally, the only event of the abstract machine is refined using the post-condition of the provided service as assignment.

```
MACHINE c_servR_a_servP
REFINES C_servR
VARIABLES
 v, r, o
INVARIANTS
 o ∈ To
 inv(o)
 MAP(v,o)
 ∀q . q∈Tp ∧ preR(q,v) ⇒ preP(q,o)
```

```
EVENTS
 Event servP ≙
  refines servR
  any p where
   p ∈ Tp
   preR(p, v)
  then
   v, r, o :| postP(p,o,o',r') ∧ MAP(v',o')
  end
END
```

Fig. 8. Event-B pattern for an assembly link

Proposition 6.1 (PRE/POST-Contract) *The refinement proof between the generated Event-B events establishes both the rules PRE and POST for the Kmelia assembly correctness as shown below.*

Proof. The following POs concerning the refinement proof [2] of `servP` establish the proposition 6.1:

[3] This predicate can be simplified if the service has no parameter

(i) If the POs about the *Invariant preservation* are discharged, then
$\forall q \cdot q \in Tp \wedge preR(q,v) \Rightarrow preP(q,o)$ is established
and the rule PRE is ensured.

(ii) The POs about *Action simulation* are:

$$
\begin{array}{l}
v \in Tv \wedge inv(v) \wedge res \in Tres \\
\wedge\ o \in To \wedge inv(o) \wedge MAP(v,o) \\
\wedge\ (\forall q \cdot q \in Tp \wedge preR(q,v) \Rightarrow preP(q,o)) \\
\wedge\ p \in Tp \wedge preR(p,v) \\
\wedge\ postP(p,o,o',r') \wedge MAP(v',o') \\
\Rightarrow \exists\ v\$0' \cdot postR(p,v,v\$0',r')
\end{array}
$$

which obviously corresponds to POST (by substituting $v\$0'$ by v').

□

7 Case Study: a Stock Management System

We report on the experimentation of the proposed method using a case study on a simplified *Stock Management* system. The system manages product references (catalog) and product storage (stock). Administrators have specific rights, they can add or remove references under some business rules such as: *a new reference must not be in the catalog* or *a removable reference must have an empty stock level*. The detailed Kmelia and Event-B specifications are available in appendix at our website [4] .

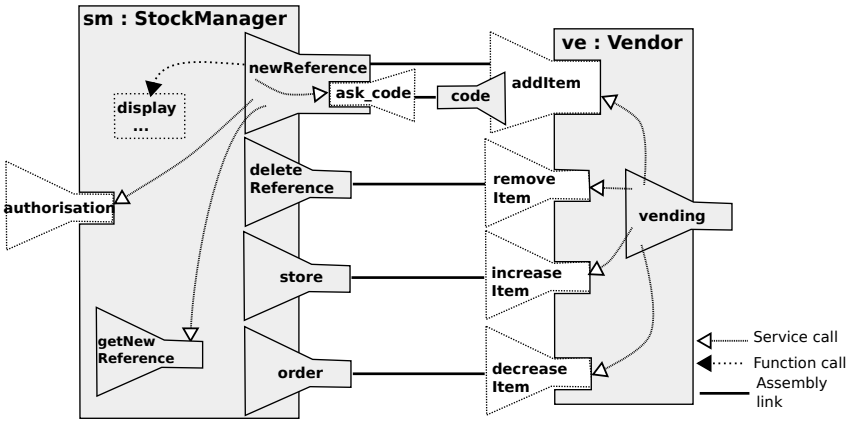


Fig. 9. Assembly of the Stock Management Case Study

As shown in Fig. 9 the system is designed as an assembly of two components sm of type StockManager and ve of type Vendor. The former one is the core business component to manage references and storage. The latter one is the system interface with a main service, the vending service. The Vendor component requires a service addItem which may get a new reference and perform the update of the system. In this paper we focus on the vending and newReference services, the other services will not be further detailed.

[4] http://www.lina.sciences.univ-nantes.fr/coloss/download/fesca10_app.pdf

7.1 *Checking the Invariant Consistency of the Components*

We have to build several Event-B models to check the different levels of component consistencies, as presented in Sections 4 and 6. The figure 10 gives an overview of the necessary Event-B models which should be generated to check the Kmelia specifications.

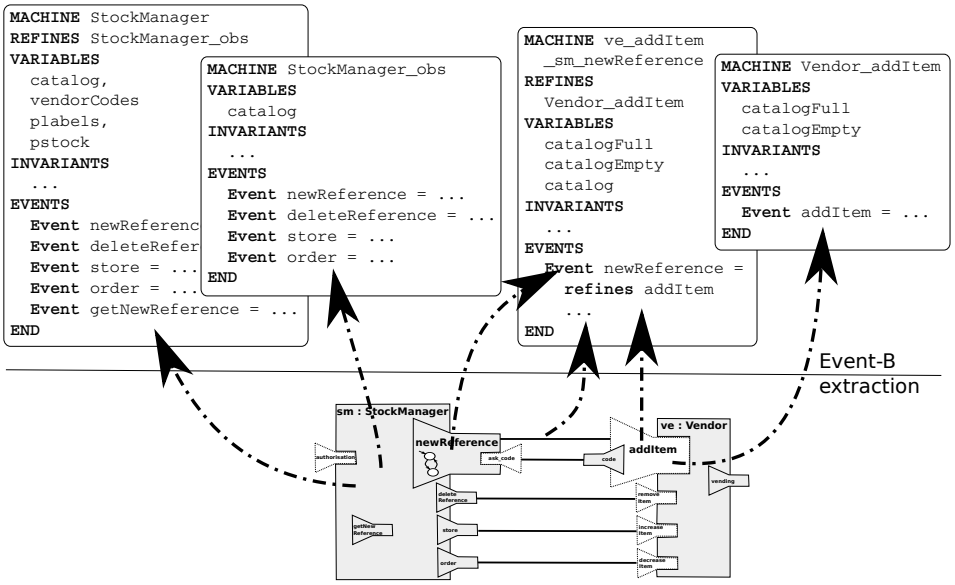


Fig. 10. Overview of the generated Event-B models

First we focus on the Kmelia specification of the StockManager component, which is made of some variables, particularly the *observable* variable catalog which contains the set of product references. Two arrays (plabels and pstock) are used to store the labels of the current references and their available quantities. The component invariant expresses that the catalog has an upper bound; each reference in the catalog has a label and a quantity; the unknown references have no entries in the two arrays pstock and plabels.

The consistency checking of StockManager follows the verification method presented in Section 4:

(i) An Event-B model `StockManager_obs` is generated following the pattern of Fig. 4: only the observable variable catalog of the component becomes a variable of the model. The observable part of invariant is also translated. Finally, the model contains one event per provided service. `StockManager_obs` is used to prove the consistency of the component vs. the current service according to its observable part (proposition 4.3).

(ii) Afterward, an Event-B model `StockManager` is built as a refinement of `StockManager_obs` following the pattern given in Fig. 5. It takes into account the internal part of the component state by adding the non-observable parts through the refinement. The events corresponding to the provided services are refined to include their local post-conditions. Thus, `StockManager` is used to check the Kmelia rules INV/F and INV/F' (proposition 4.4).

(iii) For the required services of `StockManager`, specific Event-B models are generated and checked as explained in Section 4 with the pattern of Fig. 6.

	Total	Auto.	Manual.	Undischarged
StockManager_obs	9	5	4	-
StockManager	13	7	3	3
Vendor_addItem	5	2	3	-
ve_addItem_sm_newReference	19	12	5	2

Table 1
Overview of the Generated/discharged POs with Rodin

The consistency of the component Vendor and of its required/provided services are verified using the same process. We focus particularly on the required service addItem and its corresponding Event-B model `Vendor_addItem` obtained by applying the pattern presented in Fig. 6.

7.2 Checking the Correctness of the Assembly links

The verification method presented in Section 6 is used to validate the assembly links depicted in Fig. 9. Let us consider the link between the service addItem of the Vendor component which is fulfilled with the provided service newReference of StockManager. As described in our method (Section 6), an Event-B refinement (named `ve_addItem_sm_newReference`) is generated to check the correctness of this contract. It refines the previously generated Event-B model `Vendor_addItem`. Its invariant includes the observable invariant of the component StockManager and also a mapping between the virtual variables of addItem and the observable variable catalog of StockManager.

7.3 Investigating the Event-B models and reviewing the Kmelia sources

We check all the generated Event-B models using Rodin. The POs have been generated. The major part of the POs are automatically discharged. The other ones are done manually with the use of the interactive provers, as illustrated Table 1:

- Concerning `StockManager`, 3 POs are not discharged, but they are related to the lack of typing information at the B level.
- Considering `ve_addItem_sm_newReference`, 2 POs are not yet discharged. These remaining POs are related to the proof of the preservation of the initialisation, which are not considered into the required service and its virtual context.

Studying this example within Rodin revealed some errors during the development of the Kmelia specifications. For example, the first version of the post-condition of newReference was wrong; one of the associated POs could not be discharged; looking at it thoroughly, its is related to the fact that this version of the post-condition did not cover all the needed cases.

Altogether, the main found errors were about interval limits (integers, ranges, null values...) and implicit preservation of unchanged component state variables.

8 Conclusion and Perspectives

We introduced a method to check the consistency of Kmelia components and the correctness of their assemblies using the Event-B framework. Our verification method is based on the generation of Event-B models from parts of the Kmelia specifications, and their analysis

using Event-B tools. The Kmelia specifications are equipped with contracts in the form of pre-/post-conditions defined on services and with invariants at the component level. The contracts are exploited to generate appropriate Event-B models suited to validate the components and their assemblies. The refinement technique of Event-B is used to manage both the structuring of the generated Event-B models and also the proofs to be discharged. This work contributes to fill the gap on the way to build correct software from existing components. The current results constitute one more step for rigorously building components and assemblies using the Kmelia framework. We put emphasis on tool assistance as a cornerstone for the adoption of rigorous software methods. The Rodin tools which are suitable for that purpose was selected for experimentations.

Perspectives

We plan to use Event-B and refinement to check the consistency between service assertions and the eLTS describing the service: in order to ensure that the eLTS establishes the post-condition, each transition would be translated to an event of an Event-B model. This model must refine the Event-B model containing the post-condition.

The generation of the Event-B models is currently not fully mechanised. We are implementing a dedicated module in our COSTO framework [4], to support this generation work. An important related aspect is the traceability between the Kmelia specification and the Event-B tools used to manage the consistency checking. Currently the Event-B models are generated in such a way that there is an obvious structuring of the relationship between Kmelia components and services on the one hand and Event-B models and events on the other hand; it remains to exploit the output of the proof tools to go back into the initial Kmelia specifications; this will considerably help in updating the source specifications.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An Open Extensible Tool Environment for Event-B. In *ICFEM 2006*, volume 4260 of *LNCS*. Springer, 2006.

[3] 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.

[4] P. André, G. Ardourel, and C. 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.

[5] Pascal André, Gilles Ardourel, and Christian Attiogbé. Vérification d'assemblage de composants logiciels Expérimentations avec MEC. In *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, pages 497–506. Lavoisier, April 2006.

[6] 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.

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

[8] Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix. Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies. In *6th International Workshop on Formal Aspects of Component Software(FACS 2009)*, LNCS, 2009. to be published.

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

[10] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. *ENTCS*, 160:157–172, 2006.

[11] S. Colin, A. Lanoix, and J. Souquières. Trustworthy interface compliancy: data model adaptation. *Electronic Notes in Theoretical Computer Science*, 203(7):23–35, April 2009. Proceedings of the Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2007).

[12] 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.

[13] 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.

[14] Bertrand Meyer. Applying "design by contract". *IEEE COMPUTER*, 25:40–51, 1992.

[15] Nikola Milanovic. Contract-Based Web Service Composition Framework with Correctness Guarantees. In *Service Availability, Second International Service Availability Symposium, ISAS 2005*, volume 3694 of *LNCS*, pages 52–67. Springer, 2005.

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

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

This separate appendix for the FESCA@ETAPS 2010 article is available at the Kmelia website [5].

A Kmelia specification

A.1 The Kmelia assembly StockSystem

```
ASSEMBLY
COMPONENTS
 ve : Vendor ;
 sm : StockManager
LINKS
@lref: r–p ve.addItem,   sm.newReference
Context mapping
 catalogEmpty == empty(sm.catalog),
 catalogFull == size(sm.catalog) = MaxInt
sublinks : {lcode}
End
@lcode: p–r ve.code, sm.ask_code
...
END_LINKS
```

A.2 The Kmelia component StockManager

```
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
SERVICES
########### required services (partial description)
required ask_code() : Integer
   //default assertion = true and No LTS
End
required authorisation() : setOf Integer   //...
End
########### provided services
provided newReference () : Integer   //Result = ProductId or noReference
Interface
    calrequires : {ask_code} #required from the caller
    intrequires : {getNewReference}
Pre
    size(catalog) < maxRef #the catalog is not full
Variables # local to the service
   c : Integer;      # c : input code given by the user
   res:Reference;
   d : String;           # product description
Initialization
   res := noQuantity;
Behavior
Init i # the initial state
Final f # a final state
{   i — c := __CALLER!!ask_code() —> e1,
        # gets the password on the ask_code (service) channel
   e1 — [not(c in vendorCodes)]
        display("adding a reference is not allowed") —> end,
```

[5] http://www.lina.sciences.univ-nantes.fr/coloss/download/fesca10_app.pdf

```
  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
 @resultRange:((Result>=1 and Result<=maxRef) or (Result = noReference)),
 @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                    and catalog = add(old(catalog),Result)),
 @noresultValue:(Result = noReference) implies Unchanged{catalog},
 local @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] ))
          ),
 local @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
End
//////// internal provided services ///////////////
query provided getNewReference () : Reference
Pre
  size(catalog) < maxRef #possible if the catalog is not full
Variables
  i: Reference;
Initialization
  i := 1 ;
Behavior
Init i
Final f
{  i -- {while (pstock[i] <> noReference) do
           i := i +1
         endwhile} --> res,
  res -- __CALLER !! getNewReference(i) --> f
}
Post
  notIn(catalog,Result) //the reference is really new
End
END_SERVICES
```

A.3 The Kmelia component Vendor

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

```
  //final state = end of vending
  e2 —[choice = add] _addItem!!addItem() —> e10,
  e0 —[choice <> stop] display("Product reference") —> e1,
  e1 — ref:=readInt() —> e2,
  e2 —[choice = remove] _removeItem!!removeItem(ref) —> e20,
  e2 —[choice = store] {_increaseItem!!increaseItem(ref,readInt())} —> e30,
  e2 —[choice = order] _decreaseItem!!decreaseItem(ref,readInt()) —> e40,
  //——— add Item
  e10 <<code>>, #subservice code is available here
  e10 — {desc:=readString(); // product description
         _addItem!msg(desc)          } —> e11,
  e11 — _addItem??addItem(pi) —> e12,
  e12 — {if (pi <> noReference)
          then display("New reference : "+asString(pi))
          endif } —> i
  //——— other choices ...
}
Post obs true
End
provided code() : Integer // ...
/* daemon service that answers the code of the user */
  Pre obs true
Behavior
Init e0
Final f
{ e0 — [vendorId = noID] display("Enter your vendor code") —> e1,
  e0 — [vendorId <> noID] __CALLER!!code(vendorId) —> f,
  e1 — vendorId:=readInt() —> e0
}
Post obs Result <> noID
End
########### required services (partial description)
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
  @ref: (Result <> noReference) implies (not catalogEmpty),
  @noref: (Result = noReference) implies Unchanged{catalogEmpty,catalogFull}
End
END_SERVICES
```

B Event-B Models

B.1 The Event-B context `StockLib`

CONTEXT StockLib

EXTENDS Default

CONSTANTS
`References`
`MaxRef`
`NullInt`
`NoQuantity`
`NoReference`

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

END

B.2 The Event-B model `StockManager_obs`

MACHINE StockManager_O

SEES StockLib

VARIABLES
`catalog`
`Result_newReference`

INVARIANTS
`type1` : $catalog \in \mathbb{P}(References)$
`type2` : $finite(catalog)$
`@borned` : $card(catalog) \leq MaxRef$
`type3` : $Result\_newReference \in \mathbb{Z}$

EVENTS

Initialisation
begin
`act2` : $catalog := \varnothing$
`act5` : $Result\_newReference := 0$
end

Event $newReference \mathrel{\widehat{=}}$
when
`grd1` : $card(catalog) < MaxRef$
then
`act1` : $Result\_newReference, catalog :|$
$($
$((Result\_newReference' > 0 \land Result\_newReference' \leq MaxRef) \lor Result\_newReference' = NoReference)$
$\land$
$(Result\_newReference' \neq NoReference \Rightarrow$
$Result\_newReference' \notin catalog \land catalog' = catalog \cup \{Result\_newReference'\}$
$)$
$\land$
$(Result\_newReference' = NoReference \Rightarrow catalog' = catalog)$
$)$
end

END

B.3 The Event-B model `StockManager`

MACHINE StockManager

REFINES StockManager_O

SEES StockLib

VARIABLES
`catalog`
`Result_newReference`
`vendorCodes`
`plabels`
`pstock`

INVARIANTS
`type5` : $vendorCodes \subseteq \mathbb{Z}$
`type6` : $plabels \in 1..MaxRef \rightarrow String$
`type7` : $pstock \in 1..MaxRef \rightarrow \mathbb{Z}$
`@referenced` : $\forall ref1 \cdot (ref1 \in References \land ref1 \in catalog \Rightarrow plabels(ref1) \neq EmptyString \land pstock(ref1) \neq NoQuantity)$

@notreferenced : $\forall ref2 \cdot (ref2 \in References \land ref2 \notin catalog \Rightarrow plabels(ref2) = EmptyString \land pstock(ref2) = NoQuantity)$

EVENTS

Initialisation
extended
begin
act2 : `catalog` $:= \varnothing$
act5 : `Result_newReference` $:= 0$
act8 : $vendorCodes := \varnothing$
act7 : $plabels := (1 \mathinner{..} MaxRef) \times \{EmptyString\}$
act6 : $pstock := (1 \mathinner{..} MaxRef) \times \{NoQuantity\}$
end

Event *newReference* $\widehat{=}$

refines *newReference*
when
grd1 : $card(catalog) < MaxRef$
then
act2 : $Result\_newReference, catalog, pstock, plabels :|$
$($
$((Result\_newReference' > 0 \land Result\_newReference' \leq MaxRef) \lor Result\_newReference' = NoReference)$
$\land$
$(Result\_newReference' \neq NoReference \Rightarrow$
$Result\_newReference' \notin catalog \land catalog' = catalog \cup \{Result\_newReference'\}$
$)$
$\land$
$(Result\_newReference' = NoReference \Rightarrow catalog' = catalog)$
$\land$
$(Result\_newReference' \neq NoReference \Rightarrow$
$pstock'(Result\_newReference') = 0 \land plabels'(Result\_newReference') \in String \setminus \{EmptyString\} \land$
$(\forall ii \cdot (ii \in 1 \mathinner{..} MaxRef \land ii \neq Result\_newReference' \Rightarrow pstock'(ii) = pstock(ii) \land plabels'(ii) = plabels(ii)))$
$)$
$\land$
$(Result\_newReference' = NoReference \Rightarrow pstock' = pstock \land plabels' = plabels)$
$)$
end

END

B.4 The Event-B model `Vendor_addItem`

MACHINE Vendor_addItem

SEES StockLib

VARIABLES
`catalogFull`
`catalogEmpty`
`Result_addItem`

INVARIANTS
inv1 : $catalogFull \in BOOL$
inv2 : $catalogEmpty \in BOOL$
@notFullEmpty : $\neg(catalogEmpty = TRUE \land catalogFull = TRUE)$
inv4 : $Result\_addItem \in \mathbb{Z}$

EVENTS

Initialisation
begin
act1 : $catalogFull, catalogEmpty :|(\neg(catalogEmpty' = TRUE \land catalogFull' = TRUE))$
act3 : $Result\_addItem :\in \mathbb{Z}$
end

Event *addItem* $\widehat{=}$
when
grd1 : $\neg(catalogFull = TRUE)$
then
act1 : $Result\_addItem, catalogEmpty, catalogFull :|$
$((Result\_addItem' \neq NoReference \Rightarrow$
$catalogEmpty' = FALSE \land catalogFull' \in BOOL)$
$\land$
$(Result\_addItem' = NoReference \Rightarrow$
$catalogEmpty' = catalogEmpty \land catalogFull' = catalogFull)$
$)$
end

END

B.5 The Event-B model `v_addItem_sm_newReference`

MACHINE v_addItem_sm_newReference

REFINES Vendor_addItem

SEES StockLib

VARIABLES
`catalogEmpty`
`catalogFull`
`Result_addItem`
`catalog`

INVARIANTS
`inv1` : $catalog \in \mathbb{P}(References)$
`inv6` : $finite(catalog)$
`borned` : $card(catalog) \leq MaxRef$
`assemblyEmpty` : $catalogEmpty = bool(card(catalog) = 0)$
`assemblyFull` : $catalogFull = bool(card(catalog) = MaxRef)$
`inv7` : $(\neg(catalogFull = TRUE)) \Rightarrow (card(catalog) < MaxRef)$

EVENTS

Initialisation
extended
begin
`act1` : $\texttt{catalogFull}, \texttt{catalogEmpty} : |(\neg(\texttt{catalogEmpty}' = \texttt{TRUE} \wedge \texttt{catalogFull}' = \texttt{TRUE}))$
`act3` : $\texttt{Result\_addItem} :\in \mathbb{Z}$
`act4` : $catalog := \varnothing$
end

Event *newReference* $\widehat{=}$

refines *addItem*
when
`grd1` : $\neg(catalogFull = TRUE)$
then
`act1` : $Result\_addItem, catalog, catalogEmpty, catalogFull : |$
$($
$((Result\_addItem' > 0 \wedge Result\_addItem' \leq MaxRef) \vee Result\_addItem' = NoReference)$
$\wedge$
$(Result\_addItem' \neq NoReference \Rightarrow$
$\quad Result\_addItem' \notin catalog \wedge catalog' = catalog \cup \{Result\_addItem'\}$
$)$
$\wedge$
$(Result\_addItem' = NoReference \Rightarrow catalog' = catalog)$
$\wedge$
$(catalogEmpty' = bool(card(catalog') = 0))$
$\wedge$
$(catalogFull' = bool(card(catalog') = MaxRef))$
$)$
end

END