Multilevel Contracts for Trusted Components

Mohamed Messabihi Pascal André Christian Attiogbé
LINA UMR CNRS 6241
University of Nantes, France
FirstName.LastName@univ-nantes.fr

This article contributes to the design and the verification of trusted components and services. The contracts are declined at several levels to cover then different facets, such as component consistency, compatibility or correctness. The article introduces multilevel contracts and a design+verification process for handling and analysing these contracts in component models. The approach is implemented with the COSTO platform that supports the Kmelia component model. A case study illustrates the overall approach.

1 Introduction

Component-Based Software Engineering (CBSE) using *off-the-shelf* components is one approach to deal with the software complexity. Since the components may be developed by third-parties, assembling them requires means to ensure the correctness of the component behaviours and their interoperability. This requires first that the components have rich interface descriptions and second the availability of a verification process to check the given properties. In our work we tackle the issue of building trusted components by the means of *contracts*.

As a component is usually defined as *"an unit of composition with **contractually** specified interfaces and explicit context dependencies only"* [27], the notion of contract appears to be a natural solution to express and to organise component specification and verification. The contracts are suitable to express properties such as the component's consistency preservation or the component interoperability. By contract we mainly refer to contract-based design [20] which extended the use of Hoare assertions (pre/post-conditions, invariant) at design and programming levels. But contracts may have different meanings depending on the context (component or assembly of components) or the facet (signature, structure, assertions, dynamics) [25, 10, 12]. Therefore we use the term *multilevel contract* in this work.

In order to improve the confidence of the components and their assemblies, it is necessary to make contracts explicit [10]. This demands a strong emphasis on their analyzability early in development process and a way to ensure systematically the correctness of the components with respect to the contracts. However, most of today component-based technologies lack formal analysis tools to ensure the component dependability. Our work contributes in satisfying this need.

The contribution of this article is as follows. We consider a multilevel contract approach which covers a specification activity and an analysis activity realised by verification techniques. We show (i) how the contracts can be defined at different levels (service, component, assembly) of a component model in order to specify several kinds of correctness properties and (ii) how the contracts can be checked. The demonstration applies to our experimental Kmelia component model and its language. The core of the data language of Kmelia is a first order logic, it is extended with user-defined data types and related statements; the behaviour language is based on transition systems. We experiment this proposal with the COSTO (COmponent Study TOolkit), a toolbox associated to the Kmelia model. Yet Kmelia/COSTO enables us to experiment at an abstract level, with medium size systems with intensive data and client-provider interaction style.

The remainder of the article is organised as follows. In Section 2 we introduce the multilevel contract approach; the considered levels and the contracts are made explicit. In Section 3 we overview the global design and verification process based on the multilevel contracts. Our contract approach is implemented in the Kmelia component model; Section 4 describes how contracts are integrated at different levels. The support case study is a simple bank Automatic Teller Machine (ATM). Section 5 illustrates the verification process in the case of Kmelia and the experimentations on the ATM case study. In Section 6 we discuss related approaches. Finally Section 7 concludes the article and describes planed extensions of this work.

2 Using Multi-levels Contracts in Component Models

In this section we assume a general service component model where a component interface is defined by one or several services expressing provided or required functionalities; a component may be assembled with other components via its interface; a component may have invariant properties; a service may have a dynamic behaviour including interactions with other service components. According to [21], *a Trusted Component is a reusable software element possessing specified and guaranteed property qualities.* The notion of contract is helpful to model various kind of properties. Contracts take place in services as assertions (pre/post-conditions), in components as invariants to preserve by the services, or in assemblies as component compatibility properties. These general contracts should be made precise and extended to cope with the expressiveness of the considered component models. In particular the *interoperability* between components should consider the following properties.

- *Static interoperability properties*: the compatibility of interface signatures (naming and typing); does a component give enough information about its interface(s) in order to be (re)usable by other components ?
- *Architectural properties*: the availability of the required components, the availability of the required services, the correctness of the linked component interfaces;
- *Functional properties*: do the components do what they must do? These correctness properties may be checked both on each component and on the component assemblies and compositions.
- *Behavioural compatibility*: the correct interaction between two or more components which are combined. The properties depends on the interaction model features: sequential vs. concurrent, call vs. synchronisations, synchronous vs asynchronous, pair vs. multipart communication, shared data, atomic/structured actions...

In order to cope with different meaning and different context, we introduce the notion of multilevel contract. A *multilevel contract* is a contract defined at different structure level (service, component, assembly, composition) according to different expected properties. The hierarchical vision of the contracts provides a convenient framework to master the incremental building of components and the latter verification process. In the following, we detail the main properties associated to each level.

Service contract A contract at the service level expresses that the service terminates in a consistent state. This contract deals with the *behavioural consistency* and the *functional correctness* properties.

- The *behavioural consistency* property states that the execution of the service actions does not lead to inconsistent states (such as deadlock).
- The *functional correctness* property expresses that a service achieves what it is supposed to do. The functional correctness of a service of a component is defined here using the Hoare-style specification (Pre-condition, Statement, Post-condition) where Statement is the service behaviour. This property should be checked with respect to the requirements of the owner component.

Component contract At the component level the contract states that this component can be reused with confidence. It deals with three properties: the *component consistency*, the *protocol correctness* property and also the *service accessibility.* A component protocol is defined here as the set of all the valid sequences of service invocations.

- The *component consistency* property states that the invariant properties of the component are preserved by all the services embodied in the component. Considering that a component equiped with services is *consistent* if its properties are always satisfied whatever the behaviour of the services is, one can set a consistency preservation contract between the services and their owner component to ensure that property.
- The *protocol correctness* property expresses that the order in which the services are to be invoked by clients is correct with respect to the rules given by the services' specification.
- The *service accessibility* property states that the services defined in the interface of a component are available. This is related to intra-component traceability of service dependency.

Assembly contract In an assembly, made of linked trusted components, each component will contribute to the well-formedness of the links by requiring or ensuring appropriate assertions: this is the coarse-grained contract. The link establishes a client/supplier relation. The assembly contract covers correctness properties with four layers:

- The first layer deals with the *service signature compatibility* among the services of the interfaces of the assembled components. The service call should respect the service signature. The signature matching between the involved services of component interfaces covers at least name resolution, visibility rules, typing and subtyping rules.
- The second layer deals with the *service structure consistency* of the assembled components. Assuming that services can be composed from other (sub)services, connecting services is possible only if their structures are compatible (but not necessary identical).
- The third layer deals with the *service compliance* of the assembled components. If the services use a Hoare-like specification, one has to relate their pre-conditions and post-conditions [29]. The caller pre-condition is stronger than the called one. The called post-condition is stronger than the caller's one. Each part involved in the assembly should fulfil its counterpart of the contract.

- The fourth layer deals with the *behavioural compatibility* between the linked services of the assembled components. Behavioural compatibility is about the correct interaction between two or more components which are combined through their services.

The following table summarises the crossing of levels and properties (layers) covered by the multilevel contracts. Note that the composite level is not dealt with in this paper.

service	**component**	**assembly**	**composite**
behavioural consistency	component consistency	service signature compatibility (ssic)	ssic
functional correctness	protocol correctness	service structure consistency (sstc)	sstc
	service accessibility	service compliance (sco)	sco
		behavioural compatibility (bhc)	bhc

The four layers above are useful to define interoperability levels. A Corba component with IDL interfaces can be compatible only at the first level with other models. The four layers can be augmented with other kind of properties like the quality of service.

In Section 3 we provide the details of the global verification process based on the above levels and contracts. It applies to component models with high level services such as the Kmelia multiservices component model, which serves as a working context for applying the multilevel contract definition (Section 4) and its verification (Section 5).

3 Multi-levels Contract Design and Verification Process

This section presents a component-based design process which takes into account the multilevel contracts. The components and assemblies are assumed to be abstract, meaning that they are independent from execution platforms. They can be refined or implemented later in centralised or distributed execution platforms.

As depicted in Figure 1, the process is divided in two phases: the specification phase made of specification activities and the formal analysis phase made of verification activities. The workflow is presented as a whole but the activities can be performed iteratively in any order. From a practical point of view, the specifier would switch from one phase to the other according to a customised methodology, inspired from top-down or bottom-up approaches, with a component or system orientation. For example the specifier may iterate on the component level only to deliver components off the shelf. This design approach allows the reuse of designed components by making the component descriptions available in a component library.

3.1 Specification phase: making contracts explicit

The specification phase includes three activities: a software system design (assembly/composition), a software component specification and a service specification. In a top-down approach, the *system design* activity starts first. It defines the system as a collection of interacting subsystems and components. If components or assemblies that match the requirements already exist on the shelf, they can be directly integrated in the system design. Otherwise, the *component specification* activity will produce the new component(s). Once the component structure is established, the detailed *service specification* activity proceeds. The main point in this phase is that the contracts must be explicitly expressed at each level in order to be checked.

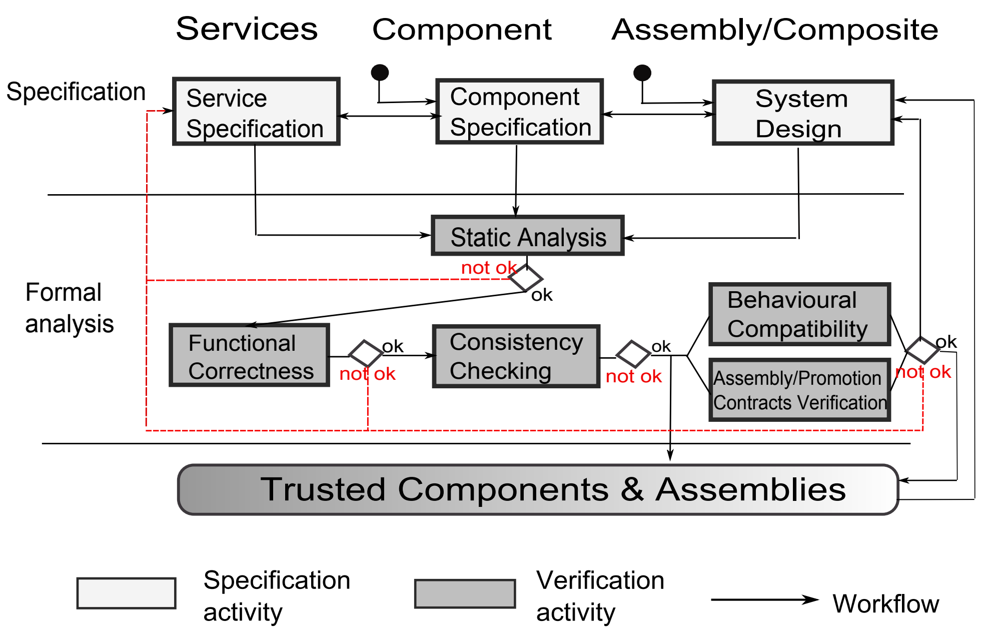


Figure 1: Contract design process

3.2 Formal analysis based on contract checking

The models produced during the specification phase are analysed by checking the contracts. The verification process iterates on five formal analysis activities as depicted in Figure 1, each activity refers to contracts of Section 2.

1. The *Static analysis* activity checks the syntactic correctness at all levels, the service accessibility of the component level, and the *static interoperability* of the assembly level, which itself covers the service signature compatibility and the service structure consistency.

2. The *Functional correctness* activity checks the *behavioural consistency* property at the service level and a part of the *protocol correctness* property at the component level.

3. The *Consistency checking* activity covers the *component consistency* property at the component level.

4. The *Behavioural compatibility* activity checks the *behavioural consistency* property at the service level, a part of the *protocol correctness* property at the component level and the *behavioural compatibility* at the assembly level.

5. The *Assembly/Promotion contracts verification* activity checks the *service compliance* of the assembled components at the assembly level. The promotion consists in making a component feature available at the composite level *i.e.* a component service can be promoted as a composite service possibly with restrictions or extensions. The promotion is treated in the context of composite components, which is out of the scope of this article.

We are not going to deal with all the details of the verification activities. Section 5 provides a concrete material on how the process is put into practice in the context of COSTO/Kmelia.

4 Designing Contracts in the Kmelia Component Model

We introduce here the main features of Kmelia, an abstract and formal component model [7]; an up-to-date formal description of the model can be found in [4]. We illustrate the use of contracts with a simple bank Automatic Teller Machine (ATM).

The key features of Kmelia are:

- *service*: a service describes a functionality; it is more than a simple operation; it has a pre-condition, a post-condition and a behaviour described with a labelled transition system (LTS). Moreover a service may hierarchically give access to other services. The behaviour supports communication interactions, dynamic evolution rules and service composition;
- *component*: a component is a container of services; it is described with a state space constrained by an invariant. A component is designed independently from its environment by setting assumptions such as virtual client components or required service specifications;
- *assembly of components*: an assembly is a set of components linked via their required and provided services with the aim to build effective functionality. Linking components by their services in assemblies establishes a possible bridge to Service Oriented Architectures. The component assemblies are governed by strict service composability rules;
- *composite component:* a composite component is a component that encapsulates assemblies or other components; it is subject to encapsulation and promotion policies.

Let us illustrate Kmelia by an ATM case study. This ATM delivers standard services such as withdrawal via a user interface. Its Kmelia specification is built on an assembly of four components as depicted in Figure 2. The central ATM_CORE handles the ATM bank services; the USER_INTERFACE component controls the user access; the AAC component stands for the bank management and the LOCAL_BANK component holds the local management access. Components are pairwise linked: a required service is *achieved* by the provided service it is linked to. An assembly link is a correspondence between a required service and a provided one according to mapping relations (names, context, messages, subservices). It materialises the support for *assembly contracts.*

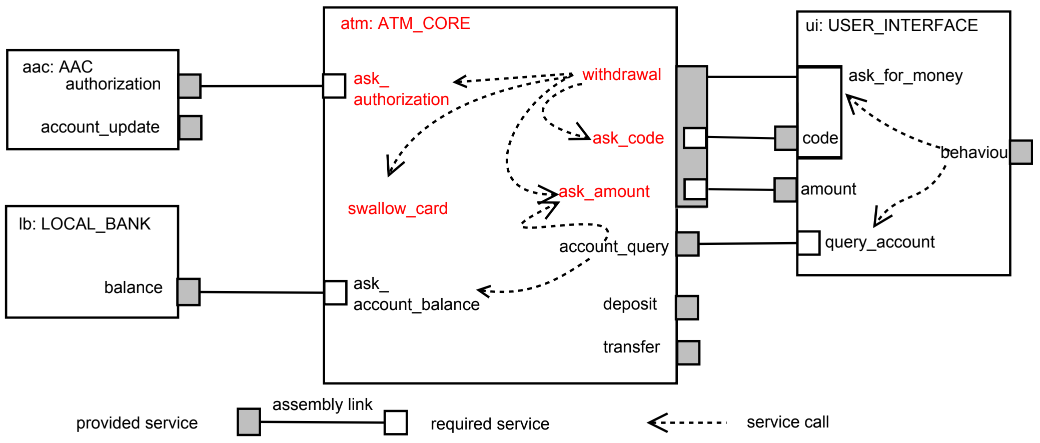


Figure 2: A component assembly for the ATM System

A Kmelia component is described by an interface, a state space and service descriptions. The interface declares the services which are provided or required by the component. The state space is a set of variables constrained by an invariant. In Listing 1 the ATM_CORE state space includes an ATM name, an identifier, a set of swallowed cards and the available notes. The CashCard data type is defined in the user-defined library ATMLIB. The obs prefix denotes a variable with a read-only access for a linked client service. The invariant predicate states that there is enough cash to proceed a transaction and the "bad" card container is not full.

Listing 1: Kmelia specification ATM_CORE

```
COMPONENT ATM_CORE
INTERFACE
  provides : {withdrawal, account_query, deposit, transfer}
  requires : {ask_authorization, ask_account_balance}
USES {ATMLIB}
CONSTANTS
  obs available_cash : Integer := 0; //observable constant
  swallowed_size : Integer := 100 //non-observable constant
VARIABLES
  obs available_notes : Integer;    //observable variable
  name : String;                    //non-observable variables
  ident : Integer;                     //ATM identifier
  swallowed_cards : setOf CashCard  //kept cards
INVARIANT
 @cash_disp: available_notes >= 0 ,
 @card_capacity: size(swallowed_cards) <= swallowed_size
INITIALIZATION
  available_notes := 10000;
  name := "ATM203";
  ident := readInt();
  swallowed_cards := emptySet;
```

A service may be a non-trivial entity with a state and a dynamic behaviour. A service may also declare required and provided subservices. All these elements are involved in the *service contract*. The service behaviour defines via an *extended labelled transition system* (eLTS) the order in which the service performs its actions. Communication actions are primitives for synchronous interactions between services. The withdraw service achieves a withdrawal on a cash card, under some controls. Listing 2 illustrates its declaration.

Listing 2: Kmelia specification of ATM services

```
provided withdrawal (card : CashCard) : Boolean
Interface
   subprovides : {ident} # from caller
   calrequires : {ask_code, ask_amount}#from authr caller
   extrequires : {ask_authorization} #from another cmp
   intrequires : {swallow_card}   #from the myself
Pre
  available_notes >= available_cash  # enough money
Variables                          # local to the service
  nbt,c,m : Integer;    # c, a : input code and amount
  # nbt : number of authorized trials of code entering
  rep : Boolean;   #rep : reply from the authorization
  success: Boolean # success: result of the withdrawal
Behavior
   //see the corresponding LTS figure
Post
  obs @notes: (result=true  implies available_notes
        < old(available_notes))
||(result=false && available_notes == old(available_notes));
// end of service
End
```

The corresponding required service ask_for_money is defined in the USER_INTERFACE component.

```
required ask_for_money (card : CashCard) : Boolean
Interface
  subprovides : {code}
    //provided to the callee

  Virtual Variables
     dispensable : Boolean;
     // assume this observable information

  Virtual Invariant true
  Pre dispensable

  //No LTS

  Post not (Result) implies dispensable
   //dispensable may evolve in the other case

End
```

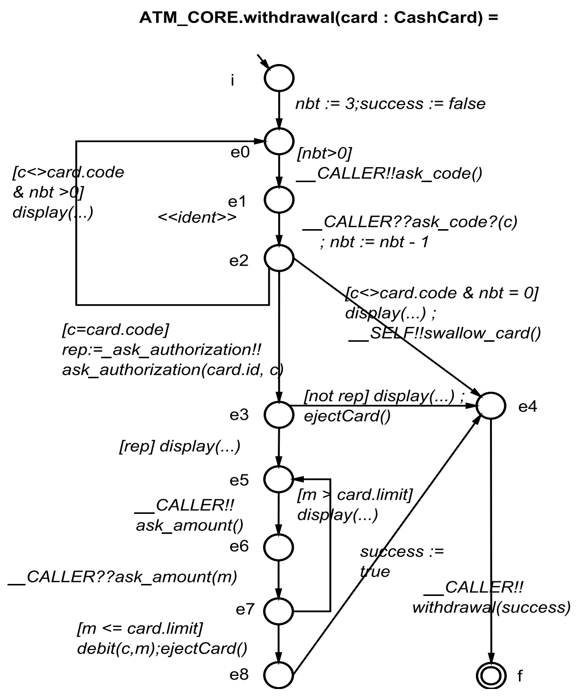
A required service may have a full service specification in Kmelia, especially if it sets assumptions on any provider service via a *virtual context*. This allows to define service contracts separately from assembly contracts and to improve the property verification locality. The context mapping of the `lwith` link in Listing 3, shows how the virtual context of the required service is "instantiated" by an actual context of the provider service.

Listing 3: "ATM Assembly links"

```
Assembly
     Components          atm:ATM_CORE;         ui:USER_INTERFACE
     Links ////////////assembly links//////////
     @lwith: p-r atm.withdrawal ui.ask_for_money
          context mapping  //a kind of explicit adaptation
            ui.dispensable = atm.available_notes >= 0
          sublinks : {lcode,lamount}
```

```
    @lamount: r-p atm.ask_amount ui.amount
    @lcode: r-p atm.ask_code ui.code
      // the service withdrawal of the ATM_CORE is connected to
      // the service ask_for_money required by the USER_INTERFACE
```

The withdrawal behaviour starts with an identification step: card insertion, password control, authentication by ACD/ATM Controller (AAC). If the AAC accepts the transaction, the ATM asks for the amount of cash, otherwise the card is ejected and the withdrawal transaction ends. The given amount is compared with the current card policy limit. When the allowed amount is lower than the requested one or if the current ATM cash is not sufficient, the ATM asks again for the amount of cash. Otherwise the ATM asks the AAC to process the transaction, updates the card limit, delivers the cash and prints a receipt when possible, and the withdrawal transaction ends after a card ejection. Two actions (debitCard, ejectCard) represent functions defined by the specifier in the user-defined ATMLIB library while display is a predefined function in Kmelia.



5 Checking Contracts in Kmelia

The verification process is supported by a set of tools integrated into the COSTO (COmponent Study TOolbox) platform which is a set of Eclipse-based plugins [2] we developed to support the specification and analysis of Kmelia component systems. COSTO manages the Kmelia specifications and handles the verification of the primary properties (syntactic analysis, type checking, static analysis, ...) as depicted in Figure 3. The verifications of complex properties such as deadlock freeness, component or assembly consistency are delegated to other appropriate external tools. Let us assume here that the static verifications (syntactic, type, well-formedness checking) are already performed by the COSTO tool; we show how other verification tools are used to check contracts.

5.1 Checking a service contract: functional correctness

The basic idea is to *evaluate* all the paths of a service behaviour ($\mathscr{B}$) and to determine whether they are compliant with the post-condition or not. Actually this is a non-trivial problem similar to the one of model-checking code. To prove this property we investigated B tools, including *ProB* a model checker for B. We had to turn back to more appropriate tools because B tools needed additional material to prove loop invariants and *ProB* was not powerful enough. In this section we present an investigation using the Key[1] tool [8]. Key accepts JML specifications as

[1]http://www.key-project.org

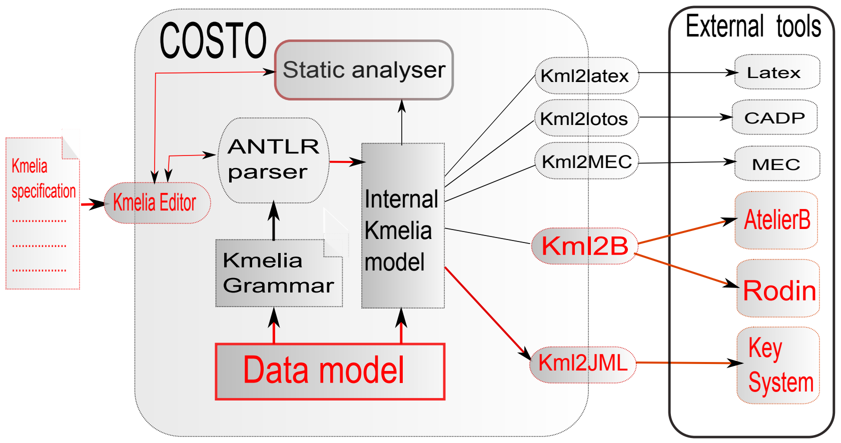


Figure 3: COSTO Framework Overview

input; therefore we defined a process to compute JML specifications from Kmelia services. An ongoing plugin called Kml2Jml implements this process (Figure 3). Each Kmelia component C is translated to a Java class C.java where: each provided service of C becomes a method of the C.java class and each required service of C becomes a method of a *virtual* component class denoted by an instance variable vc : VC in the C class. The LTS that specifies $\mathscr{B}$ is translated in two steps into a Java code. The translation is not straightforward, due to the gap between the LTS structure and the structured programming control structures of Java. The first step introduces the control structure by converting the LTS into a syntax tree, which is an extension of the well-known regular expressions formalism. The second step computes the Java code from the syntax tree.

Algorithm (step1): From LTS to Syntax tree Let $L(\mathscr{B})$ bet the set of possible behaviours of a service *srv*. Since $\mathscr{B}$ can be seen as an automaton, the syntax tree E such that $L(E) = L(\mathscr{B})$ is generated by the algorithm of *McNaughton/Yamada* (cf. *Kleene's theorem* in [24]).

Algorithm (step2): From syntax tree to Java It is straightforward from the previously obtained syntax tree. The main idea is to transform the product (.) as a sequence operator (; in Java), the Union (+) as a conditional structure (if else ...)[2], and the Kleene star ($*$) as a recursive method modeling E_*. The body of this method describes a statement block repeated in the LTS. The resulting Java code annotated with JML specifications is checked using the Key tool.

Example. Let us check the functional correctness of the *withdrawal* provided service of the ATM_Core component. Applied to the Java representation of the *withdrawal* service, the Key tool analysis revealed some errors. As an example, the post-condition of *withdrawal* was not satisfied: the error was due to the addition of amount to availaible_notes instead of subtracting it. After correcting this mistake, and regenerating the Java code, Key proved it correct automatically with 654 *symbolic states* and 18 *path conditions*.

[2]Note that even non-deterministic choice in LTS can be modeled by an if-else construction over an abstract variable that could be refined later by developer

5.2 Checking a component contract: component consistency

Here the deal is to reuse B tools like Atelier-B[3] and Rodin[4] because the B provers are appropriate to prove that kind of property, considering the fact that most of the Kmelia data types and expression are translatable in B. We developed a plugin named `Kml2B` in COSTO (Figure 3) that extracts (Event-)B specifications. For each Kmelia component ***C***, an (Event-)B model called C is built. Its state space is extracted from the component's one. The provided services srv_i in ***C*** are translated into srv_i operations within the C model. The extracted specification is imported and checked in Atelier-B or Rodin. The B tools enables the verification of invariant consistency at the Kmelia level. The full translation procedure is explained in [19].

Example. After extracting (Event-)B models by running the Kml2B plugin, the ATM_Core model is used to prove the preservation of invariant by its provided services. We proved consequently the consistency of the invariant component. However, if the post-condition is modified as $available\_notes <= old(available\_notes)$ then the invariant $available\_notes >= 0$ is not preserved anymore. This error was easily detected with the B tools.

5.3 Checking assembly contracts

Checking an assembly contract engages four verifications steps: (i) the matching of the service signatures (up to parameter renaming), (ii) the service dependency consistency, (iii) the matching of the service pre/post-conditions and (iv) the behavioural compatibility of services. Step (i) and (ii) are performed by checking static interoperability.

5.3.1 Checking the static interoperability

This verification includes: type checking, signature matching, component and service interfaces structure matching, observability rules, and service availability (requirements, subservices). The COSTO tool performs these analysis by using simple correspondence checking algorithms, graph algorithms and standard typing algorithms. Static analysis of these contracts helps to detect some incompatibilities; therefore the component designer may correct its component at design time. This corresponds to the "Static Analysis" step in Figure 1. The reader can find more details of this analysis in [4].

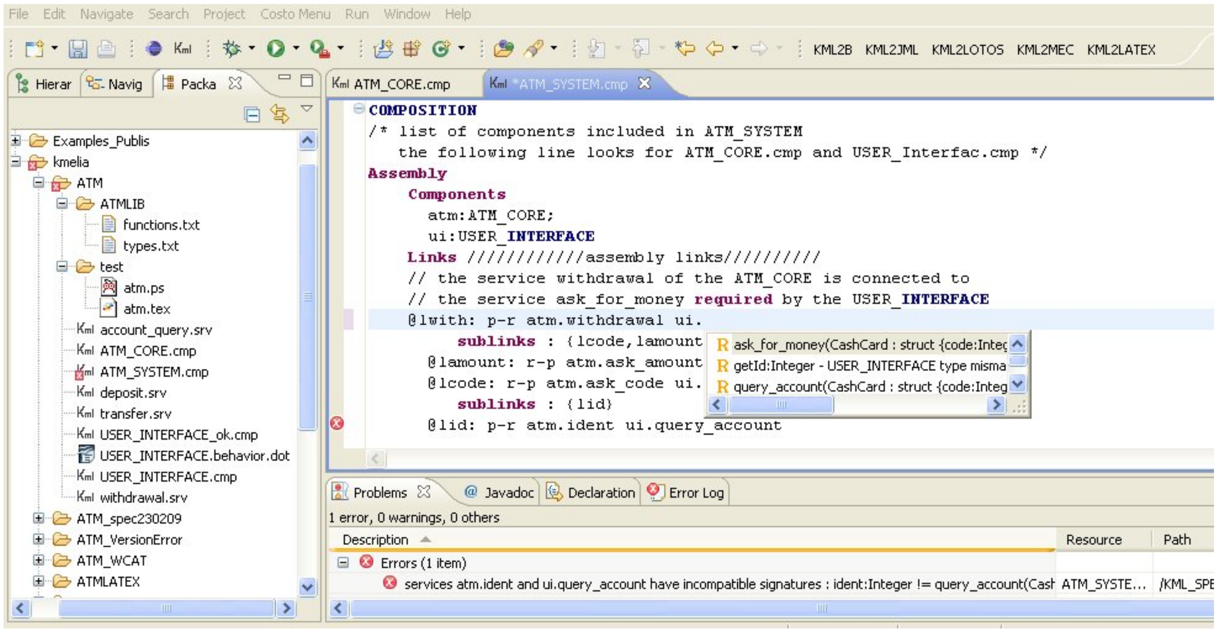
Example. Figure 4 shows the Kmelia editor in the Eclipse IDE, and a sample of the kind of errors (typing, observability, incompleteness of the mapping) that are detected. Besides standard completion, the editor supports smart completion in the case of assembly links. In Figure 4, only required services defined in the User_Interface component type are proposed and the user is warned that some of them do not match the exact signature of the provided service withdraw which is defined in the ATM_Core component type.

5.3.2 Checking the service contracts compliance

Based on an assembly link, the main issue is to decide whether the provided service matches with the required service it is linked to. The matching condition is: *the pre-condition of required*

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

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



```
COMPOSITION
/* list of components included in ATM_SYSTEM
   the following line looks for ATM_CORE.cmp and USER_Interfac.cmp */
Assembly
    Components
      atm:ATM_CORE;
      ui:USER_INTERFACE
    Links ////////////assembly links//////////
    // the service withdrawal of the ATM_CORE is connected to
    // the service ask_for_money required by the USER_INTERFACE
    @lwith: p-r atm.withdrawal ui.
         sublinks : {lcode,lamount
      @lamount: r-p atm.ask_amount
      @lcode: r-p atm.ask_code ui.
         sublinks : {lid}
      @lid: p-r atm.ident ui.query_account
```

```
R ask_for_money(CashCard : struct {code:Integ
R getId:Integer - USER_INTERFACE type misma
R query_account(CashCard : struct {code:Integ
```

```
1 error, 0 warnings, 0 others
Description                                                                                     Resource      Path
Errors (1 item)
  services atm.ident and ui.query_account have incompatible signatures : ident:Integer != query_account(Cash  ATM_SYSTE...  /KML_SPE
```

Figure 4: Error detection and smart completion in COSTO/Kmelia

*service **Req** is stronger than the one of provided service **Prov** and the post-condition of **Req** is weaker than the one of **Prov**.* In term of B proof obligations this property is rewritten as: the provided service refines the required service (considering the adaptation defined in the context mapping of the link). The refinement relation leads to the generation of specific proof obligations in (Event-)B. In practice we reuse the work presented in Section 5.2. We extend it by generating B machines for the required services and for the refinement relation. For each required service ***Req*** in a component ***C***, one (Event-)B model Req is created before checking the consistency of the virtual context of the service ***Req***. The same (Event-)B model is refined by Req_Prov_Ref. The state space of the machine Req is obtained by translating the virtual context of service ***Req***, and the operation req is the translation of the service ***Req***. The full details of the translation schema and the proof obligations are available in [5] while the Kml2B plugin is introduced in [19].

Example. The analysis of the assembly link lwith between the required service ask_for_money and the provided service withdraw_ref with the AtelierB reveals some errors that were introduced intentionally in the specification of ask_for_money for experimental purpose. The post-condition (not(result) implies not (dispensable)) means that if *result* is $false$ then the $available\_notes$ is less than 0 which can not be deduced from the withdraw_ref post-condition. Then the service contract compliance ($Post(waithdraw\_ref) \Rightarrow Post\ (ask\_for\_money)$) is not fulfilled. After correcting the error, the resulting B machines generated 28 proof obligations which were all proved by the AtelierB prover in *Automatic* mode.

5.3.3 Checking behavioural compatibility

This verification focus on the synchronous communication actions between services (start/end of services, send/receive message) defined in the Kmelia model. Checking behavioural compatibility is a widely studied topic [28, 6, 11]. It often relies on checking the behaviour of a (component-based) system through the construction of a finite state automaton. We adopt a pairwise verification approach that avoids state explosion as described in [6]. Hence for each assembly link, including the sublinks that share the same communication channel, the behavioural compatibility verification is applied. Instead of developping the checker we turned to existing model checkers because ensuring dynamic behavioural compatibility is usual target property of communicating processes and transition systems. Currently we target MEC and CADP tools. In order to exploit the CADP tools [16], we encode the Kmelia components into LOTOS processes which are the input of the CADP tools. The behavioural compatibility is based on communication between processes. A plugin named Kml2Lotos have been developed in a previous work [7]. The resulting LOTOS process can be checked using CADP tool. An alternative solution based on MEC model checker have been also experimented.

Example. The experimentations led to detect message inconsistencies. The error made by the specifier was to put a message reception in a loop for one service and a single sent message in the communication service. The deadlock was reached in case of a second pass in the loop. The MEC translator Kml2Mec and a full experimentation with MEC can be found in [3].

6 Related Works

Using contracts for components is not a new topic. However to the best of our knowledge the related approaches do not integrate contracts the way we propose. Contracts for component have been described in [26]. That proposal considers functional and extra-functional contracts and dynamic behaviours to provide trust-by-contract components. However the main issue of that work is software quality; the proof of the contracts is not treated at the design level. Beugnard et al.[10] investigated a typology of component contracts and classified contracts in four levels. Basically *syntactic contracts (i)* are taken into account by all component models. The more relevant semantic constraints such as *behavioural contracts (ii)* and *synchronisation contracts (iii)* are encountered in specific component models. Finally the *quality of service (iv)* is often delegated to runtime models.

In ConFract [14] the contracts are independent entities associated to several participants, while Kmelia attaches them mainly to services and links. The ConFract contracts support a rely/guarantee mechanism with respect to the vertical composition of Fractal components [13]. The executable assertions language CCL-J enables to express specifications at the interface level and the component levels. In the case of CCL-J, when a method of an interface is called, the contract controller is notified and it applies the checking rules. As for the pre-conditions, the post-conditions and the method invariants of all contracts "are checked at runtime". CCL-J is used to validate the contracting mechanisms of ConFract but CCL-J is much simpler than JML in terms of available constructs. In [25] the definition of Meyer's contracts and subcontracts is assumed, which led to rules similar to those of Kmelia. But the interpretation of pre-conditions and post-conditions is done in terms of call sequences rather than in logical predicates. This relies on behavioural contracts rather than functional contracts. In Kmelia, the behavioural contracts are treated separately using behaviour compatibility rules [7]. The SOFA component model and its behaviour protocol formalism [23], based on regular expressions, allow the designer to verify the conformance of a component's implementation to its specification; this verification is done at runtime. But no service contracts compliance is handled.

Architecture Description Languages represent software architectures in terms of components and their overall interconnection structure. Many of these languages support formal notations to specify components and connectors behaviours. For example, Wright [1] and Darwin [17] use CSP-based notations. These formalisms allow to verify correctness of component assemblies, to check properties such as deadlock freedom. However most of the works applying formal verifications in ADLs focus on component interactions, but very few studies addressed the contract issue using pre/post-conditions. Apart from the *syntactic contracts* level (i), the *behavioural contracts (ii)* and the *synchronisation contracts (iii)* are also proved at design time in Kmelia. We do not deal with further constraints such as quality of service, because they depend on data known only at runtime.

Contracts and services have been studied in the context of service composition. From a service composition point of view *e.g.* BPEL, the behavioural aspect is dominant [9]. Considering only the formal models, composition is mainly based on automata, Petri nets and process algebra, as illustrated by the orchestration calculus of Mazzara and Lanese [18]; therefore the contracts focus mainly on dynamic compatibility. Conversely the contracts (in the sense of *design-by-contract*) are taken into account in [22] (using abstract machines) but not the dynamic behaviour. Kmelia cares of both aspects. In [12], the contract is supported at four levels (signature, quality of service, ontology, behaviour) but none of them handle the functional contract. The component architecture (SCA) approaches [15] emphasize the service concept, like Kmelia does; but contract features are not introduced yet in SCA.

7 Conclusion

We have presented how a set of correctness properties of components may be guaranteed by stating and verifying contracts at the level of services, components and assemblies. We have illustrated the idea through the Kmelia model which is equiped with a rich data language that enables to incorporate pre/post-conditions at a service level, invariant at a component level, and behavioural contract at an assembly level. Consequently, property verification is achieved by checking the contracts at the different levels. The automation of the process is undertaken by considering extractions from the Kmelia specification language to generate the specifications in the input language of existing tools such as theorem-provers or model-checkers depending on the targeted properties. The use of a multilevel contracts makes it easy to define interoperability policy. For instance static interoperability exploits low level pre/post-conditions and helps us to check the correctness of assemblies. This may be generalised to assemblies of heterogeneous components, provided that a standard pre/post-condition mechanism is defined and respected.

CBSE lacks standard practices in order to raise a large-scale, open use of components. The road to a wide spread component-based software engineering is simplicity, ease of use, availability of well-defined, standard, free and useful components and interfaces. The Unix operating system is a convincing example that makes the proof of the concept, at a different level; the simple use of Unix .h header interfaces, the simple combination of Unix commands and options, the simple use of unstructured files, the conformance of the standard interfaces including network levels, are recognised as the main points for the development of operating system components that make the success of the Unix software family. We expect that first order logic integrated in high-level programming languages or operating systems as the use of script languages, can play a similar

role of interface standardisation for CBSE. We are working in this direction via the reuse and the extension of existing standard relational database languages (the SQL family) which are already integrated in various operating system features.
Perspectives. A short term perspective of our work is to make the tools used at different levels more integrated with helpful feedback into the Kmelia specifications. We are working on a translation of a subset of Kmelia into the Fractal component model which has a Java execution environment but lacks property verification means. We expect to favour interoperability between the models and also to find some simulation facilities that will be complementary with the formal analysis aspect provided by Kmelia.

Acknowledgements We thank the anonymous reviewers for their comments which helped us to improve the paper, and our colleague Gilles Ardourel for helping in the preparation of this final version.

References

[1] Robert Allen & David Garlan (1997): *A Formal Basis for Architectural Connection. ACM Trans. Softw. Eng. Methodol.* 6(3), pp. 213–249.

[2] Pascal André, Gilles Ardourel & Christian Attiogbé (2007): *A Formal Analysis Toolbox for the Kmelia Component Model.* In: *Proceedings of ProVeCS'07 (TOOLS Europe)*, number 567 in Technichal Report, ETH Zurich.

[3] Pascal André, Gilles Ardourel & Christian Attiogbé (2006): *Vérification d'assemblage de composants logiciels Expérimentations avec MEC.* In: Michel Gourgand & Fouad Riane, editors: *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, Lavoisier, Rabat, Maroc, pp. 497–506.

[4] Pascal André, Gilles Ardourel, Christian Attiogbé & Arnaud Lanoix (2009): *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, pp. –. To appear.

[5] Pascal André, Gilles Ardourel, Christian Attiogbé & Arnaud Lanoix (2010): *Contract-based Verification of Kmelia Component Assemblies using Event-B.* In: *7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010)*, ENTCS, pp. –.

[6] P. Attie & D. H. Lorenz (2003): *Correctness of Model-based Component Composition without State Explosion.* In: *ECOOP 2003 Workshop on Correctness of Model-based Software Composition.*

[7] C. Attiogbé, P. André & G. Ardourel (2006): *Checking Component Composability.* In: Welf Löwe & Mario Südholt, editors: *Software Composition*, *LNCS* 4089, Springer, pp. 18–33.

[8] Bernhard Beckert, Reiner Hähnle & Peter H. Schmitt, editors (2007): *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag.

[9] M.H. ter Beek, A. Bucchiarone & S. Gnesi (2007): *Formal Methods for Service Composition. Annals of Mathematics, Computing & Teleinformatics* 1(5), pp. 1–10.

[10] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau & Damien Watkins (1999): *Making Components Contract Aware. Computer* 32(7), pp. 38–45.

[11] Andrea Bracciali, Antonio Brogi & Carlos Canal (2005): *A Formal Approach to Component Adaptation. Journal of Systems and Software* 74(1), pp. 45–54.

[12] Antonio Brogi (2010): *On the Potential Advantages of Exploiting Behavioural Information for Contract-based Service Discovery and Composition. Journal of Logic and Algebraic Programming* Available at `http://dx.doi.org/10.1016/j.jlap.2010.01.001`.

[13] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma & J.-B. Stefani (2006): *The Fractal Component Model and Its Support in Java. Software Practice and Experience* 36(11-12).

[14] P. Collet, R. Rousseau, T. Coupaye & N. Rivierre (2005): *A Contracting System for Hierarchical Components.* In: George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski & Kurt C. Wallnau, editors: *CBSE*, *Lecture Notes in Computer Science* 3489, Springer, pp. 187–202.

[15] Zuohua Ding, Zhenbang Chen & Jing Liu (2008): *A Rigorous Model of Service Component Architecture. Electr. Notes Theor. Comput. Sci.* 207, pp. 33–48.

[16] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier & M. Sighireanu (1996): *CADP: A Protocol Validation and Verification Toolbox.* In: R. Alur & T. A. Henzinger, editors: *Proc. of the 8th Conference on Computer-Aided Verification (CAV'96)*, *Lecture Notes in Computer Science* 1102, Springer Verlag, pp. 437–440.

[17] Jeff Magee, Jeff Kramer & Dimitra Giannakopoulou (1999): *Behaviour Analysis of Software Architectures.* In: *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, pp. 35–50.

[18] Manuel Mazzara & Ivan Lanese (2006): *Towards a Unifying Theory for Web Services Composition.* In: Mario Bravetti, Manuel Núñez & Gianluigi Zavattaro, editors: *WS-FM*, *Lecture Notes in Computer Science* 4184, Springer, pp. 257–272.

[19] Mohamed Messabihi, Pascal André & Christian Attiogbé (2010): *Preuve de cohérence de composants Kmelia à l'aide de la méthode B.* In: *4ème Conférence Francophone sur les Architectures Logicielles*, *Revue des Nouvelles Technologies de l'Information* RNTI-L-4, Cépaduès-Éditions, pp. 113–126.

[20] Bertrand Meyer (1992): *Applying "Design by Contract". IEEE COMPUTER* 25, pp. 40–51.

[21] Bertrand Meyer (2003): *The grand challenge of Trusted Components.* In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 660–667.

[22] Nikola Milanovic (2005): *Contract-Based Web Service Composition Framework with Correctness Guarantees.* In: Miroslaw Malek, Edgar Nett & Neeraj Suri, editors: *ISAS*, *Lecture Notes in Computer Science* 3694, Springer, pp. 52–67.

[23] Frantisek Plasil & Stanislav Visnovsky (2002): *Behavior Protocols for Software Components. IEEE Trans. Softw. Eng.* 28(11), pp. 1056–1076.

[24] McNaughton R. & Yamada H. (1960): *Regular Expressions and State Graphs for Automata. RE Trans. Electronic Computers 9* , pp. 39–47.

[25] Ralf Reussner, Iman Poernomo & Heinz W. Schmidt (2003): *Reasoning about Software Architectures with Contractually Specified Components.* In: *Component-Based Software Quality*, *LNCS* 2693, Springer, pp. 287–325.

[26] H. Schmidt (2003): *Trustworthy Components-compositionality and Prediction. J. Syst. Softw.* 65(3), pp. 215–225.

[27] Clemens Szyperski (2002): *Component Software: Beyond Object-Oriented Programming.* Addison Wesley Publishing Company/ACM Press. ISBN 0-201-74572-0.

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

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