Multi-levels Use of Contracts for Trusted Components

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

We present in the article a formal approach for handling and analysing contracts in component model early in development process. Contracts are helpful to describe component interoperability levels. This work is founded on the correctness-by-construction methodology with the aim to assist in building correct complex systems. The approach is illustrated on the Kmelia component model and on the COSTO framework, an Eclipse plugin, which supports user friendly editing, and verification of Kmelia contracts by providing various bridges with efficient external tools. A case study is presented as illustration of our approach.

1 Introduction

Component-Based Software Engineering (CBSE) using *off-the-shelf* components is one approach to deal with the complexity of modern software. Since these components are developed by third-parties, assembling components requires means to ensure the correctness of the components behaviour and their interoperability. Therefore building trusted components with rich interface descriptions is an important concern and requires formal support.

As a component is usually defined as *"an unit of composition with* ***contractually*** *specified interfaces and explicit context dependencies only"*[20], the notion of contract appears as a natural solution to express and organise component specification and verification. Contracts are helpful to ensure component consistency and check the interoperability level. Therefore to improve confidence in components and their assemblies, it is necessary to make contracts explicit [9]. This demands a strong emphasis on the analysability early in development process and its automation to ensure the correctness and quality of the final components with respect to the contracts. However, most of today component-based technologies lack the formal analysis tools needed to ensure component dependability. Our work contributes in filling this gap.

In this article we show how various contracts can be integrated and practically used at different levels to ensure correctness properties in component model. The considered multi-levels contract approach deals with specification, analysis, and verification. We describe contracts and the related properties and we show how they are checked in our experimental Kmelia component model. We experiment the proposals with the COSTO (COmponent Study TOolkit) toolbox associated to the Kmelia model.

The remainder of the article is organised as follows: In Section 2 we give the relations between properties to be verified and the appropriate contracts. Section 3 describes how we integrate contracts at different levels in the Kmelia component model. Section 4 describes verification techniques and supporting tools. A simple bank Automatic Teller Machine (ATM) case study is discussed. In Section 5 we discuss related approaches. Finally Section 6 concludes the article and describes planed extensions of this work.

Submitted to:
WCSI10

2 Towards Multi-levels Contractual Component Model

Developing trusted components and services involves the specification and the verification steps. The specification step combines the definition of elementary components and their assemblies. The verification of individual properties and assembly properties crosses the specifications in a round-trip process as shown in Figure 1. To separate the concerns we promote the use of contracts in different contexts and at different levels. This provides a convenient framework to check locally various kind of properties such as functional correctness, component consistency, service and component interoperability, assembly link compatibility. With contracts, component suppliers can offer to the customers an independently issued guarantee for stated functional properties. For software designers, contracts offer a guarantee against unexpected surprises when building the software from components.

Assume in the following that a component interface is defined by one or several services (called protocols in some models), also, that a component may be assembled with other components via its interface services. Contracts may appear at the level of services, of components and of assemblies to ensure desired properties. The syntactic correctness is required at various levels, from service interfaces to the use of components and their interfaces in assemblies. A *syntactic conformity contract* may be used at all levels to guarantee the syntactic correctness of services, components and assemblies: it is the role of the client to respect the expressed typing or syntactic well-formedness when using given services, components or assemblies.

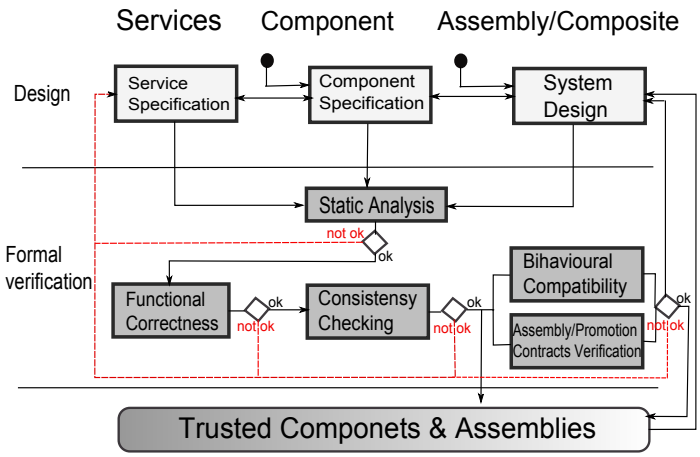


Figure 1: Contracts and verification process

Service contract The contract at the service level deals with *functional correctness* property and *component consistency* property.

- The *functional correctness* property expresses that a service achieves what it is supposed to do. Using the Hoare-style specification (Pre-condition, Statement, Post-condition) where Statement is the service behaviour, we have a definition of the functional correctness of a service. This property is to be checked with respect to a component which is the context of the service.
- The *component consistency* property states that the invariant properties of the component are preserved by 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, one can set a consistency preservation contract between the services and their owner component to ensure that property.

Assembly contract The assembly contract ensures correctness properties at three levels:

- *static interoperability*: it states that a component gives enough information about its interface(s) in order to be (re)usable by other components. It concerns first, the signature matching between the involved services of component interfaces (the client should respect the defined signature); the matching of signatures and interfaces (naming and typing). Second, if the considered component model permits the use of subservices, the subservice interfaces should also be compatible through interface matching.
- the second level is the *services compliancy contracts* of the assembled components. If the services use a Hoare-like specification, one has to relate their pre-conditions and post-conditions [22]. 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 fulfill its counterpart of the contract.
- the third level is 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 (often described as automata). It is a widely studied topic [21, 6, 10]. Checking behavioural compatibility often relies on checking the behaviour of a (component-based) system through the construction of a finite state automaton. However the state explosion limitation is a flaw of this approach [6].

In the following we present the application of multi-level contracts in the context of the Kmelia multi-service component model.

3 Overview of 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 handled by encapsulation and promotion policies.

We consider the ATM case study with standard services (withdrawal, etc) via an user interface. We model it in Kmelia as an assembly with four components (Figure 2): the central

ATM_CORE which handles the ATM bank services, the USER_INTERFACE component which 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. A link is a correspondence between a required service and a provided one up to mapping relations (names, context, messages, subservices). An assembly link materialises the support for *assembly contracts*.

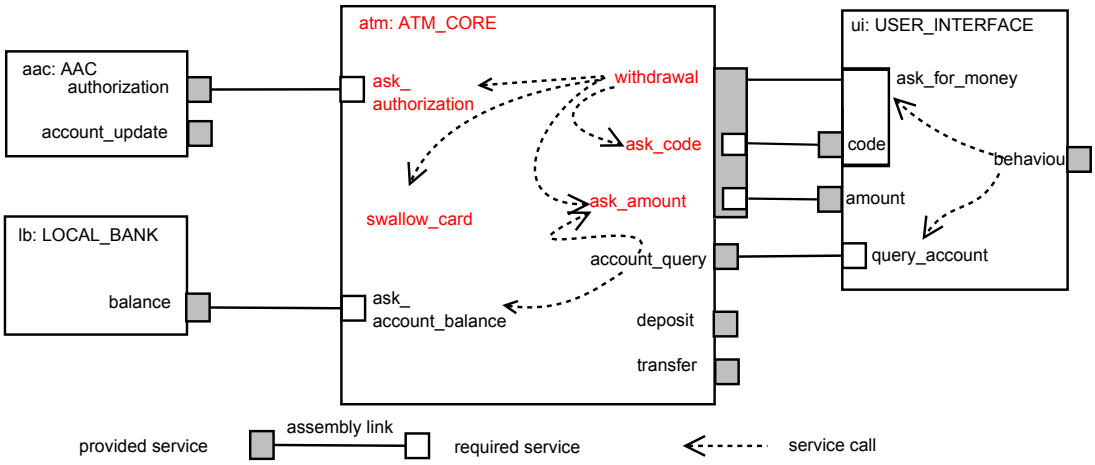


Figure 2: A component assembly for the ATM System

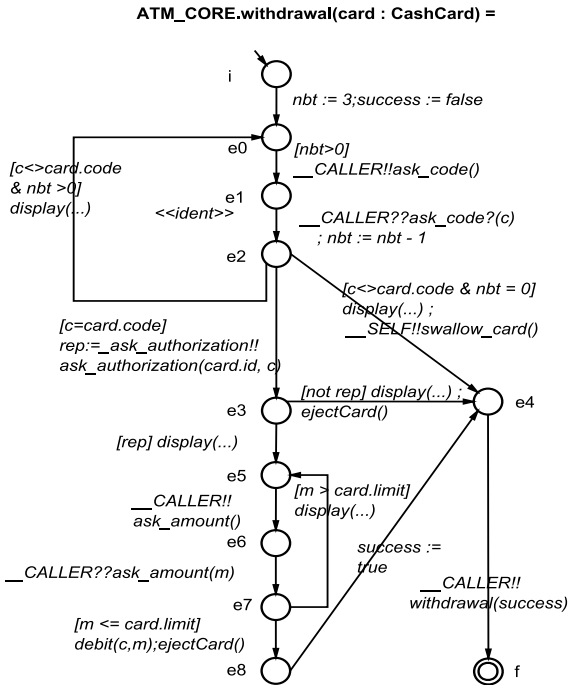
Listing 1: Kmelia specification ATM_CORE

```
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 Kmelia component is described by an interface, a state space and service descriptions. The component interface declares which services are provided or required by the component. The component 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 where 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.

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.

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 predefined in Kmelia.



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));
// fin de 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
```

Required service may have a full service specification in Kmelia, especially by defining assumptions on the provider service via a *virtual context*. This allows to define separately service contracts from assembly contracts and to improve locality of property verification. As an example for the ATM system of Figure 2 the context mapping shows how the virtual context of the required service is "instanciated" by an actual context of the provider service:

```
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
```

4 Checking Contracts in Kmelia

Formal analysis of components is performed according to the different contract levels we have introduced. In this section we overview the formal verifications of contracts that we experimented with COSTO (COmponent Study TOolbox). COSTO is the Eclipse-based plugin [2] we developed to support the specification and analysis of Kmelia component systems. The verifications of the primary properties (syntactic analysis, types checking, static analysis, ...) are integrated in COSTO.

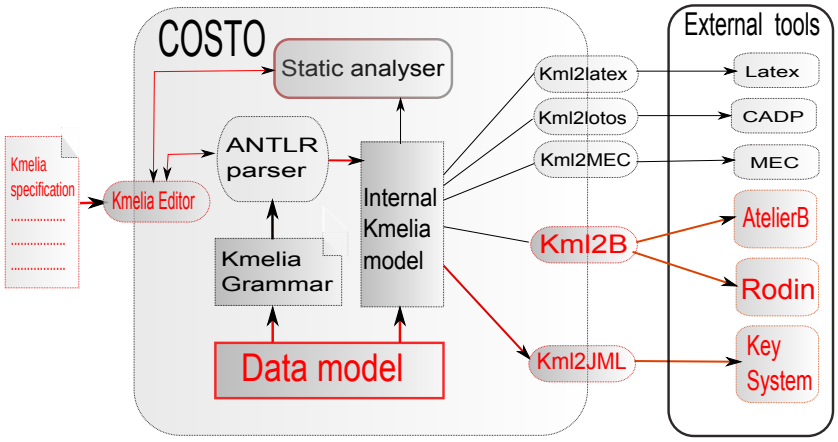


Figure 3: COSTO Framework Overview

We assume at this step that the static verifications (syntactic, type, well-formedness checking) are already performed by the COSTO tool. In this section we show how other verification tools are used to check the contracts.

4.1 Checking service contracts

On the one hand we must verify that the pre/post-conditions establish the component state space invariant consistency, on the other hand we must establish that the behaviour (actions and LTS) is consistent with the pre/post-conditions.

4.1.1 Checking component consistency

Our approach here consists in reusing B tools like Atelier-B[1] and Rodin[2] because the B provers are skill to prove that kind of property and the Kmelia data language is mostly covered by B types. We develop a plugin named `Kml2B` in the COSTO architecture (Figure 3) that extracts (Event-)B specifications. For each Kmelia component C we build an (Event-)B model called C, its state space is extracted from the component's one. The provided services srv_i in C are translated into srv_i operations within C model. The extracted specification is imported and checked in Atelier-B or Rodin. The B tools proof enables the verification of invariant consistency at the Kmelia level. The full translation procedure is formally defined in [15].

[1] `http://www.atelierb.eu/`
[2] `http://rodin-b-sharp.sourceforge.net`

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 have 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 is easily detected with B tools.

4.1.2 Checking functional correctness

The basic idea here is to *evaluate* all paths of a service behaviour (LTS) and to fix whether it is compliant with the post-condition or not. To prove that property we investigate B tools, including *ProB*, a model checker for B, but we had to turn back to more appropriate tools. Actually this is a non-trivial problem similar to the one of model-checking code. In this section we present a solution using the KeY[3] tool [8]. KeY accepts JML specifications as input; therefore we define a process for extracting JML specifications from Kmelia services. We are currently developping a plugin Kml2Jml that implements this extraction (Figure 3). Each Kmelia component C is translated as 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 the dynamic behaviour $\mathscr{B}$ of a Kmelia service is translated in two steps into a Java code. The translation is not straightforward, due to the lack of control structures in LTSs, and especially the difficulty of building loops from LTS. Therefore, we introduced the theoretical grounds to provide an original approach based on regular expression in order to express LTS specifications using Java control structures. This approach can be generalised to translate LTS-based specifications into structured programs.

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 finite state machine, E such that $L(E) = L(\mathscr{B})$ is generated by the algorithm of *McNaughton/Yamada* (cf. *Kleene's theorem* in [17]).

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 ...), and the Kleene start ($*$) as a recursive method modeling E_*. The body of this method describes a statement block repeated in LTS. The resulting Java code annotated with JML specifications is checked using the Key tool.

Example: In what follows, the effectiveness of our approach is illustrated by showing how KeY tool can be used to check correctness of Kmelia components and services presented in Figure 2. We use the ATM_Core component to check functional correctness of its provided *withdrawal*'s service. The analysis of this example with the KeY tool revealed some errors. For example, we introduced error into the *withdrawal* service behaviour so that the post-condition is not satisfied. The error is due to adding amount to availaible_notes instead of subtracting it. Consequently the post-condition is not established because of this error. After correcting this error, the resulting method has been automatically proved using KeY with 654 *symbolic states* and 18 *path conditions*.

[3] `http://www.key-project.org`

4.2 Checking assembly contracts

Checking assembly contract is defined at four levels: (i) the matching of the service signatures (up to parameter renaming), (ii) the service dependency consistency, (iii) the matching of the service contracts pre/post-conditions and (iv) the behavioural compatibility between the linked services. Step (i) and (ii) are performed by checking static interoperability.

4.2.1 Checking static interoperability

This verification deals with type checking, signature matching, component and service interfaces dependency, observability rules, and availability of required or called subservices. The COSTO tool performs this analysis by using simple correspondence checking 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.

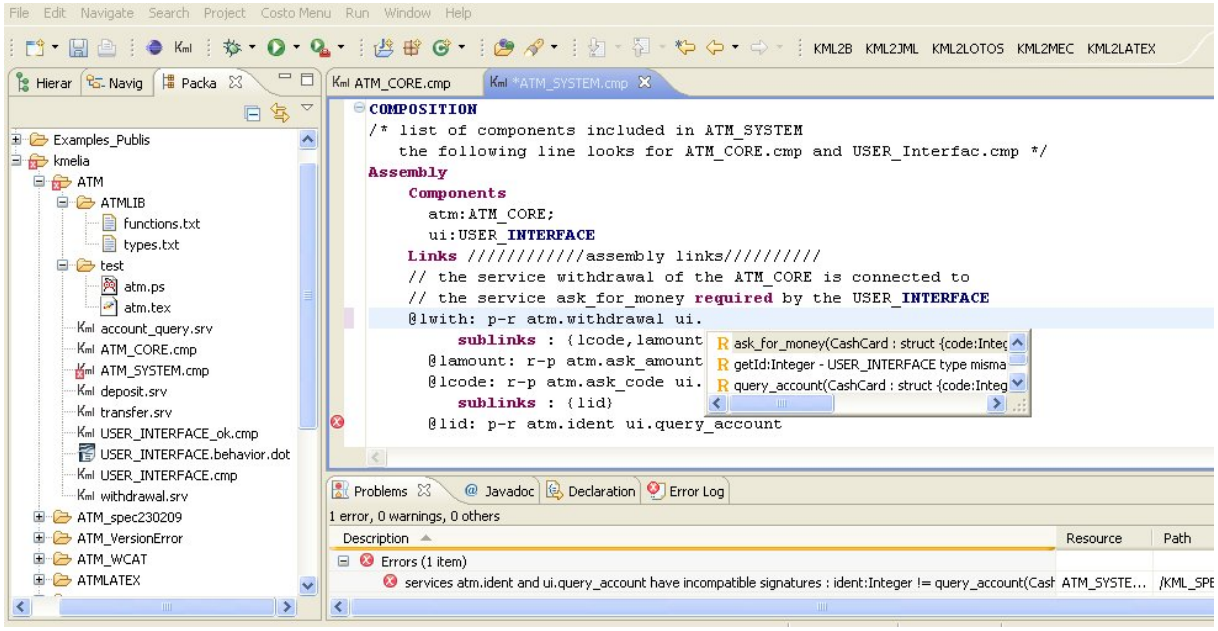


Figure 4: Error detection and smart completion in COSTO/Kmelia

4.2.2 Checking service contracts compliancy

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 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 obligation proofs this property is viewed as if: the provided service refines the required service (considering the adaptation defined in the context mapping of the link). In practice we reuse parts of the work presented in section 4.1.1 and we extend it by generating B machines for the required service, its component and the refinement relation which leads to the generation of specific (Event-)B proof obligations. For each required service ***Req*** in a component ***C***, one (Event-)B model Req is created before to check 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 service ***Req***. The full details of the translation schema and the proof obligations are available in [5] while the Kml2B plugin is introduced in [15].

Example: The analysis of the assembly link between required service ask_for_money and provided service withdraw_ref with the AtelierB reveals some errors that was voluntary introduced in the specification of ask_for_money: as post-condition we have (not(result) implies not (dispensable)). This post-condition 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 compliancy ($Post(waithdraw\_ref) \Rightarrow Post\ (ask\_for\_money)$) is not fulfilled. After correcting the error, the resulting B machines generate 28 proof obligations which are all proved by the AtelierB prover in *Automatic* mode.

4.2.3 Checking behavioural compatibility

At this level we assume that services are not atomic nor executed as transactions, but they communicate and synchronise. As indicated in section 2 to avoid state explosion we work on one link but not the whole assembly. Ensuring dynamic behavioural compatibility is a target property of communicating processes and transition systems, and it is usually checked by model checkers. Currently we target MEC and CADP tools.

In order to exploit the CADP tools [13], 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. The translation procedure is performed as follows; each service state is examined: each outgoing transition of the state corresponds to a LOTOS action, followed by the translation of the reached state. The used channels, the communication actions and the elementary actions are collected to form the current process alphabet. According to these working hypotheses, we define a semantic encoding of the service specifications. The encoding into LOTOS of service specifications is inductively performed by considering: service interface without formal parameters; service interface with formal parameters; service states (initial, final, intermediary and branching) and the transitions related to each service state.

As a mechanisation of this translation, a 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 send inconsistencies. The error made by the specifier was to put a message reception in a loop for one service and a stand alone message send in the communication service. The deadlock was reached in case of second pass in the loop. The MEC translator Kml2Mec and a full experimentation with MEC will be found in [3].

5 Related Work

Contracts are already used at different levels and with various ways in several existing component models; we consider some of them in the following. However to the best of our knowledge there is not an homogenous use of them as we done and shown latter on. Contracts for component have been described in [19]. This proposal considers functional and extra-functional contracts and dynamic behaviours to provide trust-by-contract components. However the main issue of this work is software quality and the proof of the contracts is not treated at the design level. Beugnard et al.[9] have investigated types of component contracts and have classified contracts into four levels. *Syntactic contracts (i)*, which are taken into account by all component models, more important semantic constraints such as *behavioural contracts (ii)* and *synchronisation contracts (iii)* are encountered in various component models; and finally *quality of service (iv)* which is often used at runtime. For example, a Corba component with IDL proposes a contract at level (i) only.

In ConFract [12] contracts are independent entities which are associated to several participants, not to services and links as in our case; their contracts support a rely/guarantee mechanism with respect to the vertical composition of Fractal components [11]. ConFract uses the executable assertions language CCL-J to express specifications at interface and component levels. In the case of CCL-J, when a method is called on an interface, the contract controller is then notified and it applies the checking rules. As for pre-conditions, post-conditions and method invariants of all contracts, they "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 [18] 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, behavioural contracts are treated separately with behaviour compatibility rules [7]. The SOFA component model and its behaviour protocol formalism [16], based on regular expressions, permit the designer to verify the conformance of a component's implementation to its specification; this verification is done at runtime. But no service contracts compliancy is handled.

Architecture Description Languages model software architectures in terms of components and their overall interconnection structure. Many of these languages support formal notations to specify components and connectors behaviors. For example, Wright [1] and Darwin [14] use CSP-based notations. These formalisms allow to verify correctness of component assemblies, checking properties such as deadlock freedom. However most of the work on applying formal verifications in ADLs has focused on component interactions, but very few studies have addressed the contract issue using pre/post-conditions.

Apart from the *syntactic contracts* level (i), *behavioural contracts (ii)* and *synchronisation contracts (iii)* are also expressed in Kmelia and proved at design time. We do not deal with

others constraints such as quality of service, because they depend upon data known only at runtime.

6 Conclusion

We have presented how a set of correctness properties of components may be guaranted by stating contracts at the level of services, components and assemblies. We illustrate the idea through the Kmelia model which is equiped with a rich data language that enables to incorporates pre/post-conditions at service levels, invariant at component level, and pre-post contract at assembly level. Consequently, property verification is achieved by checking the contracts at different levels. The mechanisation of the process is undertaken by considering translation from the Kmelia specification language into the input language of existing tools such as theorem-provers or model-checkers depending on what properties we have to deal with.

The multi-levels use of 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.

It is clear that CBSE lacks of standard practices in order to raise a large-scale, open use of components. According to us, the road to a wide spread component-based software engineering is simplicity, easy of use, availability of well-defined, standard, free and usefull 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 Unix family softwares. 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 which are already integrated in operating system level.
Perspectives A short term perpective 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, some simulation facilities that will be complementary with the formal analysis aspect provided by Kmelia.

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 be published.

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

[10] Andrea Bracciali, Antonio Brogi & Carlos Canal (2005): *A formal approach to component adaptation*. *Journal of Systems and Software* 74(1), pp. 45–54.

[11] 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).

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

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

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