ProVeCS Workshop, TOOLS Europe, Zürich, 2007

A Formal Analysis Toolbox for the Kmelia Component Model

Pascal André, Gilles Ardourel, Christian Attiogbé [1]

LINA CNRS FRE 2729
University of Nantes
France

Abstract

We present in this paper the COSTO toolbox that supports the Kmelia abstract component model. First, an overview of the COSTO toolbox is given. Then the abstract component model Kmelia is presented. One main feature of the toolbox is the connection with existing tools in order to perform the analysis of specification properties. We present this approach for the dynamic aspect analysis; an example of the connection with the CADP toolbox to check Kmelia service behavioural compatibility is used as an illustration.

Keywords: Property Verification Toolbox, Components, Services, Model Checking

1 Introduction

It is an important challenge to deliver correct software components on demand, from various development frameworks and for various problem requirements. Some identified parameters for the success of such enterprise are the availability of reliable, proof-certified and interoperable components. This is tightly related with the availability of tools to help in the design, development and analysis of the components.

Component Based Software Engineering emphasises the development of components and their assemblies to build large scale software. However, in practise the existing component model proposals, both in industry and academia, do not propose provably-correct components and they are quite different and even not interoperable. This motivates our work. For instance a given abstract component model, proved to have desired properties, may be refined into code with respect to various executable platforms. The obtained codes may be used in various software.

This work contributes to assist the user, with a toolbox, in the development of correct components and assemblies from their abstract specifications. Correctness is

[1] pascal.andre@univ-nantes.fr, gilles.ardourel@univ-nantes.fr,christian.attiogbe@univ-nantes.fr

considered from various points of view: in this article we deal with the behavioural interaction between components.

In the article we present the COSTO toolbox which is designed to support the Kmelia abstract component model. The toolbox is currently an experimental prototype not yet publicly available. We give an overview of the modules that compose the toolbox. Instead of presenting in details all the modules we focus on the ones concerned by the verification of dynamic properties and especially the *LOTOS Module*. It illustrates an important principle in COSTO: the use of adequate languages and tools to perform complex verifications.

The article is organised as follows. In Section 2 we give an overview of COSTO, our formal analysis toolbox for components. We present the Kmelia abstract component model based on services in Section 3; this model serves as the component model for property verification. Section 4 is devoted to the analysis of one property related to the dynamic aspect of components: behavioural compatibility. We present the principle of connecting the COSTO toolbox with existing tools by translating Kmelia specifications into targeted formalisms. In Section 5 we illustrate one component of the toolbox: the connection with CADP to check behavioural compatibility. Finally some perspectives are given in Section 6.

2 An Overview of the COSTO Toolbox

Considering that mechanisation is a means to assess design and development techniques based on formal methods, we start the development of a prototype named COSTO (Component Study Toolbox) to support design and analysis of component using the Kmelia Abstract component model. The specification language named Kmelia and the Kmelia abstract component model are briefly described in Section 3.

One of the main features of COSTO is the definition of bridges to existing formal analysis frameworks and their integration in the verification process.

2.1 COSTO Main Modules

The COSTO prototype is composed of several modules written in Java. Most of them can be used in command-line, through their API or using the costo eclipse plugins. Figure 1 shows the main COSTO modules.

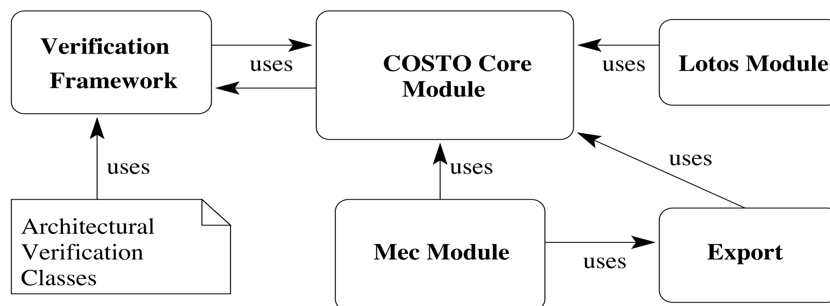

Fig. 1. An overview of the COSTO Toolbox

The **Core module** is the main COSTO module used by all the other modules. It contains a parser for Kmelia textual specifications based on ANTLR, and an API

for manipulating the resulting Kmelia Object Models. Syntax analysis and basic typing checks are done during and after parsing.

The **Verification module** contains a verification framework that is used to define verification processes, execute them on Kmelia Object Models and manipulate verification results. *Architectural properties* analysis such as the correct composition of components according to their services' signatures and interfaces are defined using this framework.

Consistency checks such as *Component Interface and Services consistency* and *Consistency between Services Interfaces and Behaviours* rely on an earlier version of this module and they are currently being integrated in the verification framework to add more flexibility.

The **LOTOS module** contains a translator of Kmelia specifications to LOTOS specifications according to a context (this module is described in section 5). The generated LOTOS specifications can then be checked with CADP (a toolbox with various analysis modules, [16]).

The **MEC module** contains an extractor which selects and transforms parts of a Kmelia behaviour specification in MEC specifications according to a context. It also generates properties to be checked in the MEC model checker. It features a MEC feedback analyser which parses MEC results and generates documentation in order to correct the Kmelia specification. In order to go beyond simple documentation, an automatic integration of this MEC feedback in the verification framework is in the works.

The **Export module** contains generators that help in the documentation of the Kmelia specification. This modules generates LaTeX documentation for components using an export of the Kmelia service behaviours in dot for visual representation.

2.2 COSTO Eclipse Modules

In order to simplify the use of COSTO, several tools have been integrated to the Eclipse Integrated Development Environment as plugins.

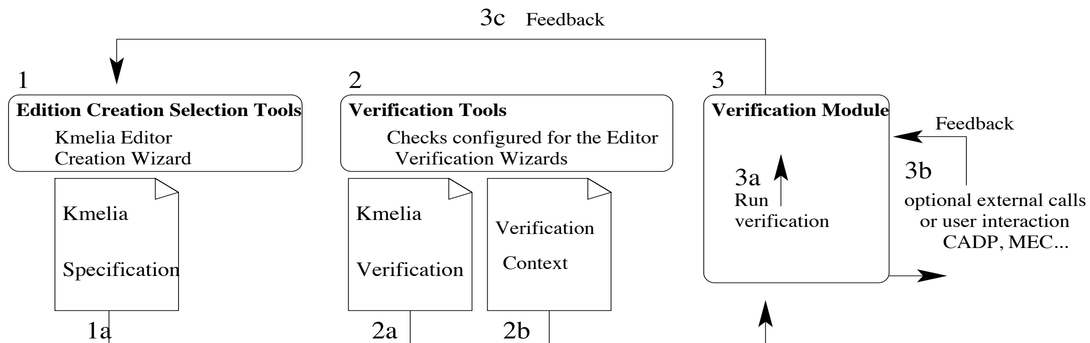

Fig. 2. Using eclipse plugins to verify Kmelia components

The previously described COSTO modules (see Figure 1) are packed into the `coloss.costolib.base` plugin.
Several tools have been built in the `coloss.costolib.ui`:

- A text editor for Kmelia specifications which shows results from syntactic and consistency verifications;

- A tree-based view that outlines a Kmelia specification;
- Wizards for creating Kmelia components and assemblies;
- Menu actions for exporting a Kmelia specification to the various formats supported in the COSTO Exports Module (such as LaTeX);
- Wizards for creating verification contexts and starting the verifications with MEC or LOTOS.

Figure 2 illustrates a scenario of use of the COSTO plugins: starting with a Kmelia specification, generated by a wizard or created with the editor, the user selects or create a verification process and its context with a wizard, then he runs the verification that may or may not rely on an external tool, and finally he gets the verification result.

The Kmelia examples from the following section have been generated or checked with the COSTO prototype.

3 A Multi-Service Component Model: Kmelia

We present here the specification language Kmelia which is central to the COSTO tool. Instead of presenting the language itself (syntax and semantics) we sketch a quick overview of the concepts using examples.

Kmelia is an abstract formal component model based on services [7,3]. Its goal is to describe component systems and to study their properties before any implementation. The interactions between components are described through services and synchronous communications. The dynamic behaviour of services is formally specified by labelled transition systems.

Related component models with dynamic behaviours (or protocols) are SOFA [19], Fractal [10], Tracta [14], Wright [1] and other works [20,6,11,18]. The main difference between Kmelia and the above models and proposals is that Kmelia emphasises the concept of service: (i) components are linked by their services rather than connected by structural ports or gates; (ii) dynamic behaviours are associated to services rather than to components [2] ; (iii) services are functionalities that define contracts; (iv) services can be composed horizontally and vertically.

3.1 Components, Services and Assemblies

A Kmelia abstract component is a mathematical model of an open multi-service system that supports synchronous communication with its environment. A Kmelia component is defined through an abstract state model (made up with variables, an invariant, and an initialisation), an interface (made up with provided and required services) and a constraint definition (logic expressions). Yet, the property language is an *ad hoc* typed first order logic; the planned evolution is to interact with existing theorem provers to check the expressions.

Let us illustrate the model with a simplified real-world problem: a bank Automatic Teller Machine (ATM). Since the case is very common, the details are omitted here. The ATM provides bank services (withdrawal, deposit money, query

[2] Additionally to provided services, Kmelia enables one to specify component protocols as special services.

accounts...) to users. Figure 3 is a textual Kmelia specification of an ATM core component. The component interface provides four usual bank services for exchanging money and requires an external authorisation. The component state model manages the ATM cash data.

```
COMPONENT ATM_CORE
/* The ATM_CORE component is the central component for a bank cashier station.
   The main services of such a system are cash withdrawal, account query, deposit money
   and transfer bank query.
   The current specification focuses only on cash withdrawal. */
INTERFACE
  provides : {withdrawal, account_query, deposit, transfer}
  requires : {ask_authorization, ask_account_balance}
TYPES
  CashCard  :  struct {code:Integer, id:Integer, limit:Integer} // record type
CONSTANTS
  // constants definitions
  available_cash : Integer := 100,
  swallowed_size : Integer := 100
VARIABLES
  // variables definitions
  name : String,
  swallowed_cards : Set,
  available_notes : Integer
PROPERTIES
  // predicates
 cash_disp: available_notes >= 0,
 card_capacity: size(swallowed_cards) <= swallowed_size
INITIALIZATION
  // variables assignments
  name := "ATM203";
  swallowed_cards := emptySet;
  available_notes := 10000;

SERVICES
// services from external files (currently only in the same directory) can be included
  provided external account_query
  provided external deposit
  provided external transfer
  provided withdrawal (card : CashCard)
  // see the service withdrawal in Figure 3
  ...
//required services
   required ask_authorization (id : Integer, code : Integer) : Boolean
   ...
//internal services
   provided debit (c : CashCard, m : Integer)
   ...
END_SERVICES
// end of ATM_CORE specification
```

Fig. 3. Overview of the ATM_CORE component specification

Basically, a Kmelia service encodes a functionality; it is defined with an interface and a behaviour. The *service interface* includes the service signature, the local declarations, the assertions (pre/post conditions) and the *service dependency* (*i.e* the list of services this service depends on). The *service dependency* of a service s_i includes the references to provided subservices (they are the services which are provided in the context of another service) and to required services (those required in the context of s_i). The latter are required from the component itself, from the calling component or from any components.

Figure 4 shows a specification of the `withdrawal` service of the core component for the ATM system in Kmelia. The `subprovides, calrequires, extrequires` clauses in the interface of the `withdrawal` service make explicit the hierarchy and the dependencies between services: the `withdrawal` service provides an `ident` subservice and requires three other services, two of them being required from the component which is calling `withdrawal`.

```
Provided withdrawal (card : CashCard)
/* The service withdrawal is available if there is enough money in the cash dispenser.
   This services requires a bank credit card, a code, an amount to withdraw.
   An authorization is required from the bank consortium.
   This service provides an identification subservice if needed.
*/
Interface
   subprovides : {ident}
   calrequires : {ask_code, ask_amount} //required from the caller
   extrequires : {ask_authorization}
Pre
  //service available if there is enough money
  available_notes >= available_cash
Variables
  nbt : Integer,     // nbt : number of authorized trials of code entering
  c : Integer,       // c : input code given by the user
  a : Integer,       // a : input amount given by the user
  rep : Boolean,     // rep : reply from the authorization request
  success : Boolean // success : result of the withdrawal request
Behaviour
init i     // i is the initial state
final f    // i is a final state
{
  ... see the service behaviour in Figure 5 ..
}
Post
  available_notes <= pre(available_notes)
  // (success && (available_notes = pre(available_notes) - a)) ||
  // ((not success) && available_notes = pre(available_notes))
end
```

Fig. 4. Overview of the Kmelia service syntax

Component assemblies establish the communication channel used by the (service) communication actions. Assembling Kmelia components consists in linking their pairwise services: required services may be linked to provided services. An implicit channel is associated to this link that supports the *communication actions* or messages between the services (see section 3.2). The semantics of the links is not straightforward because it must conform to the service interface hierarchy. Indeed the services that appear in the `subprovides` and the `calrequires` clauses of the service interface dependency must (i) share a common link (they are sublinks) and (ii) their links must conform to the hierarchy levels. This constraint is recursive on service inclusion. A component composition is the encapsulation of an assembly within a component with a projection of services by *promotion links*. Promotion links relate the composite services to the inner component services.

Figure 5 is a graphical view of a Kmelia model for the bank ATM. The `as` component is a composition of an ATM CORE (`ac`) with an ATM user interface (`ui`). The main provided service `behaviour` of the `ui` component drives the user commands. For example, the user can ask for money (required service `ask_for_money`) which is linked to the service `withdrawal` provided by the ATM core `ac` component. According to Figure 5, the `withdrawal` service may call internal services (`debit`,...), external services (`ask_authorization`), external services required from the caller (`ask_code`, `ask_amount`) and it provides the *ident* service in the context of *ask_code*. Note that the amount and code links are sublinks: they share the `ask_for_money-withdrawal` link and its implicit communication channel.

The assembly links support the service interactions specified in the service behaviours.

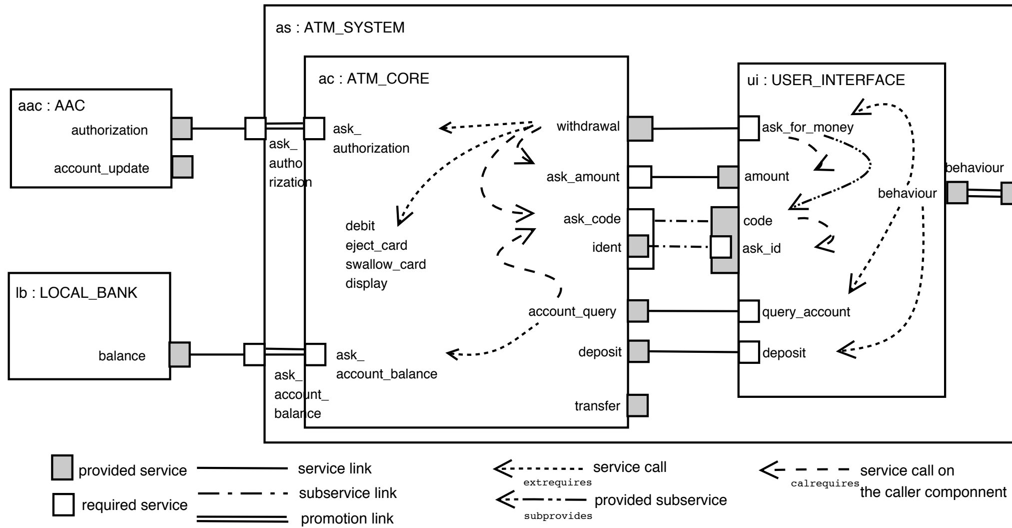

Fig. 5. Assembly for an ATM System

3.2 Service Behaviour Description

In Kmelia, a service behaviour is an extended labelled transition system (eLTS) where the states define the service evolution steps and the transitions are labelled with possibly guarded combination of actions: `[guard] action*`. The actions are either *elementary actions* or *communication actions*. An elementary action (*e.g.* an assignment) does not involve other services; it does not use a communication channel. A communication action is either a *service call/response* or a *message send/ receive*.

The services run concurrently; the communications are synchronous. The communication actions use either the standard communication primitives `!` and `?` for sending/receiving simple messages or their extended forms `!!` and `??` to deal with service calls and service responses. They are prefixed with a communication channel which can either denote the required service (`_service-name`) or the caller (`__CALLER`) or the component itself (`__SELF`). A communication channel that is used in a service behaviour has to be established by a link. For example the `ask_for_money-withdrawal` link (Figure 5) establishes the caller service of the `withdrawal` service: `__CALLER` = `ask_for_money`.

Figure 6 is a visual representation of the `withdrawal` service eLTS. This figure has been produced by the COSTO toolbox. A withdrawal consists in reading the given cash card. The user enters the password. The given password is compared with the card password. If the verification succeeds, the card holder is authenticated otherwise the password is requested again. When the verification fails three times, the card is swallowed. After the card holder identification in the withdrawal service, an authorisation is required from its ACD/ATM controller (AAC), which represents the bank management. If the AAC accepts the transaction, the withdrawal service asks for the amount of cash, otherwise the card is ejected and the transaction ends. The user enters an amount which is compared with the current card policy limit. If the allowed amount is lower than the requested or if the current cash is not sufficient, the amount of cash is asked again. Otherwise, the transaction proceeds. In any positive case the withdrawal transaction ends after a card ejection.

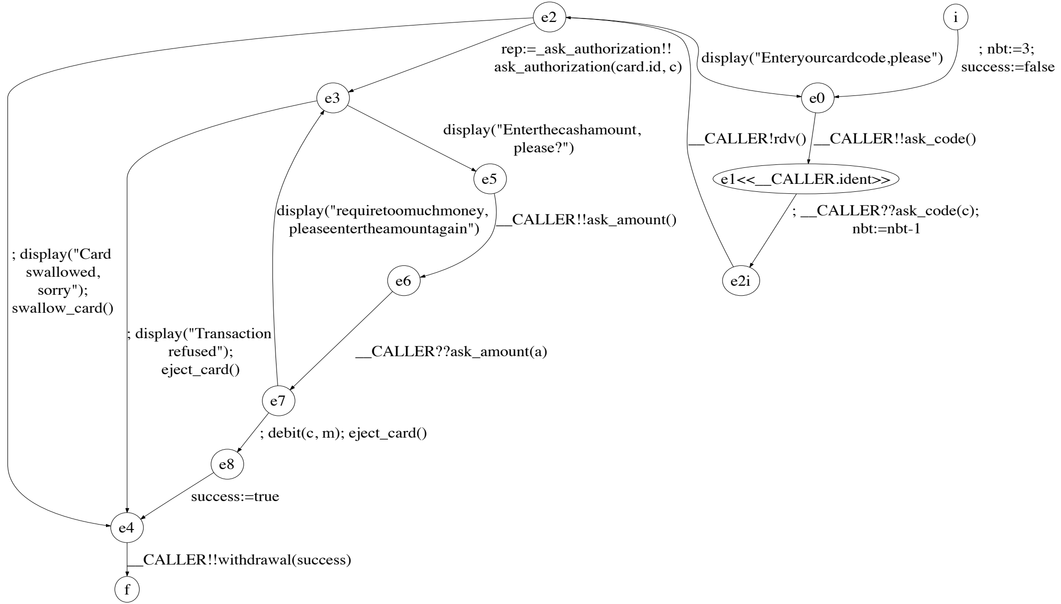

Fig. 6. The `withdrawal` service eLTS (COSTO export)

In Kmelia, service behaviours may contain execution points (states or transitions) where a subservice (declared in the service's interface) can be called. These states or transitions are annotated with Kmelia's vertical structuring operators. For instance, the label of node `e1` in Figure 6 means that the `ident` optional service may be called by a `widthdrawal`'s caller when the running reaches the `e1` node. This label features the `<<>>` operator that denotes an *optional service call*. Kmelia main vertical operators are: *optional service call* `<<>>` , *optional behaviour insertion* `<||>`, *mandatory service call* `[[]]`, *mandatory behaviour insertion* `[||]`. An extended LTS is one in which the states and transitions may be annotated with subservices.

These structuring mechanisms provide a means to reduce the LTS size, to share common services or subservices and to master the complexity of service specification. This hierarchical behaviour structuring is naturally reflected in the service interfaces: this permits a precise description of the use of a subservice in the context of the interaction with a service. Both hierarchies must be consistent. Going one step further, they must be consistent with the link hierarchy of the component assemblies. This point will be one aspect of the compatibility property that will be discussed later in the paper.

The current version of the Kmelia language does not handle *broadcast*; this point and multi-way communications are subject to ongoing works. However we deal with the case where several services run simultaneously, as an interleaved behaviour plus synchronous communications on shared channels.

4 Dynamic Aspects Analysis within COSTO

In this section we focus on one dynamic property of components assemblies: checking that components interact well through their services. We choose this property because it is complex and it illustrates the principle of connecting COSTO to others (external) powerful tools to run the effective verification of the property.

4.1 Analysis of Component Dynamic Aspects

The starting point is an assembly of components. The interaction between the linked services implies a concurrent evolution of the services; this evolution is considered as a dynamic aspect of the component analysis.

Behavioural compatibility is about the correct interaction between two or more components which are combined via some of their services. The behavioural compatibility analysis is performed by considering the correct interaction between the eLTS of the involved services. It is a topic already studied in several approaches [20,12,6,9]. The main concern shared by these approaches is: checking that a given component interacts correctly with another one which may be provided by a third party developer.

In the Kmelia model, the interaction between components results in an interaction between linked services of the components. The interaction between components may involve not only two but many components. But we consider only one caller service and one called service at the same time. Therefore the component interacts correctly with its environment if its services are *compatible* with the other services with which they are linked. The behavioural compatibility analysis is precisely formalised in [7]. We recall here the main idea.

A service is *compatible* with another if either their eLTSs evolve independently, in an interleaved way, or they perform complementary communication actions. That is the basis of our compatibility analysis approach; we check that a given eLTS that models the behaviour of one service matches with a second eLTS that models the linked service behaviour. A complete interaction between the services of several components results in a pairwise local analysis between the eLTS of a caller and that of the called service.

Two eLTS interact from their initial state until a terminal state according to a set of rules that we have defined. The rules indicate the correct evolutions of the eLTS according to the current states of involved services and the labels of their transitions. If the transitions are labelled with independent elementary actions, we have an interleaving of the independent actions of the transition systems. If the transitions are labelled with communication labels involving the same channels, and the actions are complementary (an emission and a reception) then we have a synchronous communication involving the complementary actions. If the labels are communication actions but not complementary, we get an incompatibility.
After a final state of a called service, the caller may continue with its independent transitions or with transitions that involve other (sub)services. After a final state of the caller the called service may continue only with elementary actions or with communication actions that do not need a complementary action from the terminated caller, otherwise we get an incompatibility.

Practically, several points need to be considered to check the behavioural compatibility: various kinds of interactions, synchronous or asynchronous communications, atomic actions or composite ones. Technically, checking the behavioural compatibility often relies on checking the behaviour of a (component based) system through the construction and the analysis of a global finite state automaton. However the state explosion limitation is a flaw of such an approach. We tackle this

problem by considering local pairwise verification of behavioural compatibility: as a component provides several services in its interface, one has to select the service to be checked. Therefore only the links and sublinks of this service will be considered within a compatibility checking. Each service behaviour being encoded with a eLTS, we check only the eLTSs of the involved services. This local verification process iterates for each (linked) pair of services of the component assembly leading to a global checking.

4.2 Principle of Open Property Verification in COSTO

The effective verification of specification properties may require powerful systems such as model checker or theorem prover. Implementing such tools requires much effort and knowledge. One major principle we adopted when building COSTO toolbox was to open it to other languages, tools and environment in order to delegate various computations. The application of this principle to the verification of dynamic properties currently led to the LOTOS module and the MEC module (see Figure 1). This section illustrates this principle.

The analysis of properties which are specific to the Kmelia model are implemented as modules inside the COSTO toolbox; for example the composability of components is specific to Kmelia components, therefore composability checking is implemented as a specific COSTO tool.

But the analysis of general properties or properties that can be translated into general ones is handled with available external tools. For example the behavioural compatibility of two eLTS is a general property that can be checked using existing tools such as MEC [15] or LOTOS [16]. For that purpose, a gateway to these tools is built. The part of the Kmelia specifications to be verified is translated into the input formalism of the targeted tool. The verification is then performed within the target environment. It is suitable that the feedback of such an external analysis be related to the Kmelia specification to help the specifier.

We conducted various experiments with the reuse of existing tools. We used the MEC model checker [5] to deal with behavioural part of Kmelia services. Using MEC we can focus specifically on service behaviours during preliminary analysis (were data are ignored). Some feedbacks from this analysis can help to correct the submitted specification. As far as two linked services (a caller and a callee) are concerned, we translate the LTS of each service into MEC automata, then the MEC synchronous product of both automata is built and then we search for deadlock freeness. The absence of deadlock implies the compatibility of the services. This experimentation with MEC has been reported in [2].

We have also used the LOTOS/CADP toolbox [13,16] as external tool to conduct experiments on Kmelia specifications. We detail the connection between COSTO and CADP in the following section.

5 An External Module to Verify Service Interactions

In this section we explain how the CADP toolbox is connected to COSTO and how behavioural analysis is performed. Considering services to be checked for be-

havioural compatibility, the components that embody these services are first parsed by the Kmelia parser which generates their internal representation; the services involved in the analysis are extracted from the internal representation; then the extracted services are translated as LOTOS processes. To deal with behavioural compatibility we use the LOTOS selective parallel operator `|[...]|` to compose the generated LOTOS processes. This selective parallel composition operator is used because its semantics corresponds to our behavioural compatibility between services (see section 4.1); that is a synchronisation on specific selected actions and the interleaving of the other actions.

The result of the translation and the composition of the processes are used as input of CADP tools.

5.1 LOTOS and CADP

LOTOS [17] is an ISO standard formal specification language. It is initially designed for the specification of network interconnection (OSI) but it is also suitable for concurrent and distributed systems. LOTOS extends the process algebra CCS and CSP and integrates algebraic abstract datatypes. A LOTOS specification is structured with process behaviours. It has the main behaviour description operators of the basic process algebra CCS and CSP. LOTOS uses the "!" and "?" operators of CSP which denote respectively emission and reception. The salient features of LOTOS are: the powerful multi-way synchronisation; the use of communication channels called *gates*; the synchronous interaction of processes; the use of algebraic data types to model data part of systems; the availability of a toolbox (CADP [13,16]). CADP (Construction and Analysis of Distributed Processes) is the toolbox associated to LOTOS; it enables one to apply its various model checking techniques on the described processes which are first compiled into labelled transition systems.

5.2 Translating the Kmelia Services into LOTOS Processes

Remind that the behaviour of each Kmelia service is modelled with an eLTS the transitions of which are labelled with service calls, elementary actions, guarded actions and communication actions. Each state of the eLTS has an identifier. Some of the states are additionally labelled with a list of action names.
An *output transition* of a given state is a transition going from this state to another one. An *input transition* is a transition coming from any one state and entering another considered state.

The general principle of the translation (or encoding) is as follows. The input of the translation is the internal form of the transition system which describes a service. The internal form is obtained from the output of the Kmelia specification parser. The input is translated into a LOTOS process.

We define a set of semantic encoding rules to support the translation of the component services into LOTOS . These semantic rules permit a systematic translation. Three kinds of encoding rules are defined: service interface translation, state translation rules (denoted by the LotosEncoding procedure) and transition label translation rules (denoted by the LotosEncodingL procedure). We do not give a full description of these rules (please see [4]), but we give some illustrations of the used approach.

The translation of the data part of the Kmelia service results in LOTOS data types. To deal with communication, each service has a *default channel* with the same name as the service.

The translation of a transition system is achieved as follows. One main process is associated to the initial state of a transition system of a service and several related sub-processes are associated to the other states of the service. The processes have at least one parameter which is the default channel of the translated service; the used abstract actions are collected as an alphabet that complements the process parameter.

A service without formal parameters (`servName()`) is called by sending its name on the default channel of the service; it is translated by:

```
process servName[servName_chan, ...]: exit :=
   servName_chan? varx: MsgTypeservName;[varx = servName];
```

A service with formal parameters is translated by a process which waits for the encoded service name and its parameters. Thus a service `servName(p1: T1, p2: T2, ...)` is translated with:

```
process servName[servName_chan, ...]: exit :=
   servName_chan? snx: MsgTypeservName;
       ? p1: MsgTypeservName;
          ?p2: msgTypeservName; [snx = servName] ->
            ...
```

From each state of the service there are one or several output transitions.

A state with several output transitions is translated with a non-deterministic choice between the translations of the output transitions; it results in a choice between as many process behaviour as possible in the LOTOS process. The translation into LOTOS of the transitions `S0--act1-->fs1, S0--act2-->fs2` is the encoding of the `S0` state:

LotosEncoding(s0) =

```
 (    LotosEncodingL(act1); LotosEncoding(fs1)
  []  LotosEncodingL(act2); LotosEncoding(fs2) )
```

A state with more than one input transition is translated with a sub-process. Indeed, having more than one input transition means that the state can be reached from several transitions, therefore the sub-process is reused from different state translations.

A state annotated with a list of service names is translated by a non-deterministic choice between several sub-processes. Each sub-process corresponds to the interaction with one of the listed services. Consider the branching state `S0 <<subserv1> ><<subserv2>>`. If the service `subserv1` is called, then the current service proceeds with the initial state in the `subserv1`. The encoding into LOTOS of `S0` is as follows:

LotosEncoding(S0) =

```
 Process SP_Process_S0[...]: exit :=
```

```
(  chan_subserv1?fprm: MsgTypesubserv1 [fprm = subserv1];
       LotosEncoding(initial_State(subserv1))
[] chan_subserv2?fprm: MsgTypesubserv2 [fprm = subserv2];
       LotosEncoding(initial_State(subserv2))
)
Endproc
```

The translation of labels are as follows. *An elementary action* is translated with an abstract action that will be an element of the process alphabet. As far as the *guarded actions* are concerned, first the guard is abstracted as an atomic element and then the guarded action gives a sequence of actions. Activations of service are treated as communication primitives. *Communication actions* are translated with LOTOS communication operators `!` and `?`.

According to the previous statements, we have formalised a specific semantic encoding (namely *LotosEncoding*) of the service specifications. Briefly, 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 annotated) and service transitions.

For the translation of the data part of Kmelia services into LOTOS, we use enumerated types or bytes as data abstractions; the data values are then restricted in order to limit the state explosion problem. For each service, we define a specific LOTOS data type which has a constructor named with respect to the service; this permits the call of the service by sending its name on the convenient channel.

Besides, all the messages which are sent to the default channel associated to a service are used as constructors of the data type associated to this service. The expressions used within actions are translated as abstract actions of the alphabet of the LOTOS process.

After the translation process, we get full LOTOS processes which are used to check behavioural compatibility; they can also be analysed using various CADP verification modules.

5.3 Experimentation Results

The formal analysis using CADP starts after the generation of the LOTOS processes from the parsed Kmelia specifications of the involved components that embody the services.

To check the behavioural compatibility of a pair of services, the LOTOS processes resulting from their translation are composed with the `|[alph]|` operator to form a specific interacting system; `alph` is the action alphabet used for the synchronisation of the processes. The system obtained by composing the service processes is compiled with the CADP compiler CAESAR which also checks for the consistent use of the parallel composition operator. If the compilation is successful then the composition is correct hence the behavioural compatibility. If we have a deadlock from the compilation process due to communication actions mismatch then the processes are not compatible. This result is provided as feedback to the Kmelia specifier.

When there is no communication mismatch, CAESAR generates an internal graph

(corresponding to the LTS) from which various analysis are available. For example the EVALUATOR module of CADP is used to model-check temporal properties (written in μ-calculus) that express safety or liveness properties on the service descriptions. More generally, all the analysis modules provided by the CADP tools are now made available due to the connection we have made through the translation into LOTOS. But in this case the specifier should move to the CADP environment.

6 Conclusion and Perspectives

We have presented an overview of the COSTO toolbox which supports the Kmelia abstract component model. The input of COSTO is Kmelia specifications. Several modules are available within COSTO for parsing, behaviour visualisation, service or component interactions analysis.

The result of the Kmelia specification parsing is either used with the specific tools of COSTO or translated into the input formalisms of external tools. We have emphasised the connection between COSTO and CADP by illustrating with the analysis of the behavioural compatibility analysis with the CAESAR compiler of the CADP toolbox after a translation of Kmelia service specifications into LOTOS processes. The COSTO toolbox is then connected to the CADP analysis framework after the generation of LOTOS processes.

At the current stage of the COSTO development we focus on the use of model checking tools with respect to the behaviours of Kmelia services. Connections are made with the MEC tool and the CADP toolbox.

However, Kmelia component and service specifications are equipped with properties that appear as logical assertions. Therefore we begin a bridging with theorem proving tools such as that of the B Method. For example, we have introduced protocols as user guides in the Kmelia model[3]; they are treated as specific services but their consistency is being studied using the assertions of the services which are to be translated in first order logic and proved correct with theorem proving.

The SOFA [19] framework provides behavioural verification tools but they are specific to SOFA component models. The Vercors platfom [8] has a similar approach to ours; its is yet more mature than the COSTO toolbox. However the input component models of COSTO and Vercors are quite different and they need specific processing; for example, LOTOS specifications are directly used as component behaviour in Vercors whereas LOTOS specifications are generated for the services described inside the Kmelia component models. Moreover the Kmelia model and the related tools consider a correct development of components from their abstract specifications.

The perspectives of this work are: the bridging with the SOFA approach in order to share modules through our toolbox; the bridging with theorem proving tools to complement the property verification aspect for data-intensive systems and the enhancement of the data and assertion language of the Kmelia model for scalability. A methodological analysis process is needed to integrate the various verification modules; for example the combination of a mismatch detection with a module to guide the correction is viewed as the integration of a compatibility analysis tool with an adaptation tool.

Furthermore 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 some simulation facilities that will be complementary with the formal analysis aspect provided by Kmelia.

References

[1] Allen, R. and D. Garlan, *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology **6** (1997), pp. 213–249.

[2] André, P., G. Ardourel and C. Attiogbé, *Vérification d'assemblage de composants logiciels Expérimentations avec MEC*, in: M. Gourgand and F. Riane, editors, *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006* (2006), pp. 497–506.

[3] André, P., G. Ardourel and C. Attiogbé, *Defining Component Protocols with Service Composition: Illustration withe Kmelia Model*, in: *6th International Symposium on Software Composition, SC'07*, LNCS **to appear** (2007), pp. –.

[4] André, P., G. Ardourel, C. Attiogbé, H. Habrias and C. Stoquer, *A Service-Based Component Model: Formalism, Analysis and Mechanization*, Technical Report RR05.08, LINA (2005).

[5] Arnold, A., P. Crubillé and D. Bégay, "Construction and Analysis of Transition Systems with MEC," AMAST Series in Computing: Vol. 3, World Scientific, 1994, iSBN 981-02-1922-9.

[6] Attie, P. and D. H. Lorenz, *Correctness of Model-based Component Composition without State Explosion*, in: *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, 2003, pp. –.

[7] Attiogbé, C., P. André and G. Ardourel, *Checking Component Composability*, in: *5th International Symposium on Software Composition, SC'06*, LNCS **4089** (2006), pp. –.

[8] Barros, T., A. Cansado, E. Madelaine and M. Rivera, *Model-checking distributed components: The vercors platform*, in: *International Workshop on Formal Aspects of Component Software (FACS'06)* (2006).

[9] Bracciali, A., A. Brogi and C. Canal, *A formal approach to component adaptation*, Journal of Systems and Software **74** (2005), pp. 45–54.

[10] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems*, Softw. Pract. Exper. **36** (2006), pp. 1257–1284.

[11] Canal, C., L. Fuentes, E. Pimentel, J. M. Troya and A. Vallecillo, *Adding Roles to CORBA Objects*, IEEE Trans. Softw. Eng. **29** (2003), pp. 242–260.

[12] de Alfaro, L. and T. A. Henzinger, *Interface Automata*, in: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)* (2001), pp. 109–120.

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

[14] Giannakopoulou, D., J. Kramer and S.-C. Cheung, *Behaviour Analysis of Distributed Systems Using the Tracta Approach.*, ASE **6** (1999), pp. 7–35.

[15] Griffault, A. and A. Vincent, *The Mec 5 Model-checker*, in: *CAV: International Conference on Computer Aided Verification*, Lecture Notes in Computer Science **3114** (2004), pp. 488–491.

[16] Hubert Garavel, R. M., Frédéric Lang, *An overview of cadp 2001*, (Also in European Association for Software Science and Technology (EASST) Newsletter) RT-254, INRIA (2002).

[17] LOTOS, I., "A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour," International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1988.

[18] Pavel, S., J. Noye, P. Poizat and J.-C. Royer, *Java Implementation of a Component Model with Explicit Symbolic Protocols*, in: *4th International Symposium on Software Composition, SC'05*, LNCS **3628** (2005), pp. 115–124.

[19] Plasil, F. and S. Visnovsky, *Behavior protocols for software components* (2002), iEEE Transactions on SW Engineering, 28 (9), 2002.
URL citeseer.ist.psu.edu/plasil02behavior.html

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