 Electronic Notes in Theoretical Computer Science 182 (2007) 155–170 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

An Adaptation-based Approach to Incrementally Build Component Systems

# Pascal Poizat

*IBISC FRE 2873 CNRS - Universit´e d’E´vry, France ARLES Project, INRIA Rocquencourt, France Email:* [Pascal.Poizat@inria.fr](mailto:Pascal.Poizat@inria.fr)

# Gwen Salau¨n

*VASY Project, INRIA Rhˆone-Alpes, France Email:* [Gwen.Salaun@inria.fr](mailto:Gwen.Salaun@inria.fr)

# Massimo Tivoli

*POPART Project, INRIA Rhoˆne-Alpes, France Universita` degli Studi dell’Aquila, Italy*

*Email:* [tivoli@di.univaq.it](mailto:tivoli@di.univaq.it)

Abstract

Software components are now widely used in the development of systems. However, incompatibilities between their behavioural interfaces may make their composition impossible. The objective of software adaptation is to compensate such incompatibilities building as automatically as possible corrective connec- tors or components. Constructing component-based systems from scratch is difficult, in particular when components cannot be used directly since they have to be adjusted with respect to their mates. Incremental construction methods are therefore essential because they make it possible to build systems step by step and therefore to master the complexity of their adaptation. In this paper, we propose an incremental approach to build component-based systems which relies on the generation of adaptors to overcome behavioural in- compatibilities. The adaptation stage can be automated being given an abstract mapping formalising the properties of the system to be adapted.

*Keywords:* Software Components, Behavioural Mismatch, Adaptation, Incremental Construction.

# Introduction

Software components are now widely used in the development of systems, including embedded systems, Web services and distributed applications. The main challenges of Component-Based Software Engineering are composition, adaptation and verifi- cation of component applications. *Software adaptation* [[16](#_bookmark24),[7](#_bookmark19)] aims at generating as automatically as possible component *adaptors*. Their role is to compensate incom- patibilities appearing in a system constituted of communicating entities. It is now

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2006.09.037

being accepted that components, and in particular their public interfaces – most of the time the only observable parts of components due to their black-box nature – have to take into account dynamic behaviours. In this paper, we deal with adaptors fixing incompatibilities at this behavioural level, *e.g.,* message name mismatch or deadlocking protocols.

Building a software system from scratch is a difficult task even if one of the promises of the component-based approach is to make the reuse of existing software entities easier. Moreover, composing components is a task which must take also into account adaptation of incompatible components. We propose an approach to build *incrementally* component-based systems, and where the generation of adaptors is fully automated if the software architect gives an abstract description of the properties of the system to be adapted, *i.e.,* an *adaptation mapping*.

In this context, the notion of Software Architecture assumes a key role since it represents the reference skeleton used to compose components and let them interact. The architecture proposed in our approach associates an adaptor to each component. If the component does not require adaptation, the adaptor (called a *no-op* adaptor) will reproduce from an external point of view exactly the same behaviour as the component. We build incrementally a system in such a way that it is able to evolve to architectural changes such as component addition or suppression. This architecture is very close to distributed systems and can be implemented using adaptive middlewares [[1](#_bookmark13),[12](#_bookmark25)]. However our objective here is the automatic retrieval of the needed behavioural adaptor protocols, and not their implementation.

Our adaptor generation process is based on a former work using an expressive notation for the mapping (regular expressions of synchronous vectors), algorithms and tools [[8](#_bookmark20)]. Our focus here is on the incremental construction of a system made up of components and adaptors when the architect adds or removes one component. Our approach is interactive and the architect is informed on the state of the system- to-be before applying a modification (stable, addition of a new service, suppression of a service, etc). Most of the steps are completely automated, for instance to check the compatibility of interfaces, generate the adaptor, evaluate the correctness of the adaptor. Moreover, global approaches [[16](#_bookmark24),[13](#_bookmark26),[5](#_bookmark17),[15](#_bookmark27),[8](#_bookmark20)] which generate a global adaptor for the whole system have a main drawback being that recomputing the global adaptor every time something changes costs a lot. Accordingly incremental construction of adaptor-based systems is a solution to work them out.

The rest of the paper is organized as follows. Section [2](#_bookmark0) presents the formal model of components, and the architectural style we rely on. Section [3](#_bookmark2) focuses on adaptation, and presents mechanisms to check if adaptation is needed, to compute adaptors, and to assess them. In Section [4](#_bookmark5), our approach for incremental construc- tion involving components and adaptors is presented. We describe how the system is updated as automatically as possible when one component is added or removed. Section [5](#_bookmark6) illustrates our incremental approach on several case studies. We end in Section [6](#_bookmark12) with concluding remarks.

# Component Systems

* 1. *Component Interfaces*

Component interfaces are given using a signature and a behavioural interface. A *signature* Σ is a set of operation profiles. This set is a disjoint union of *provided* operations and *required* operations. Behavioural interfaces are also taken into ac- count through the use of *Labelled Transition Systems* (LTSs). A LTS is a tuple (*A, S, I, F, T* ) where: *A* is an alphabet (set of event labels), *S* is a set of states,

*I* ∈ *S* is the initial state, *F* ⊆ *S* are final states, and *T* ⊆ *S* × *A* × *S* is the transition

function. The alphabet of the LTS is built on the signature. This means that for

each provided operation *p* in the signature, there is an element *p*? in the alphabet, and for each required operation *r*, an element *r*!. Communication between two LTSs involves one event with complementary actions *p*?/*p*!.

Expressive behavioural languages such as process algebras can be used to de- fine behavioural interfaces in a more concise way. For instance, the part of the CCS notation restricted to sequential processes is adequate to describe behavioural inter- faces [1](#_bookmark1) : P ::= 0 | a?.P | a!.P | *τ* .P | P1+P2 | A, where 0 denotes a correct termination state, a?.P a process which receives a and then behaves as P, a!.P a process which sends a and then behaves as P, *τ* .P a process which evolves with the internal action *τ* and behaves as P, P1+P2 a process which may act either as P1 or P2, and A denotes the call to a process defined by an agent definition equation A

= P. As process algebras do not enable to define initial and final states, we extend this CCS notation to tag processes with initial [i] and final [f] attributes. 0 and 0[f] being equivalent, we only use 0 in such a case.

* 1. *Architectural Style*

Components communicate by message passing in a peer-to-peer style. Connectors between components are simple communication channels defining a required and provided signature interface too. The required (*resp.*, provided) interface of a com- ponent may be connected to the provided (*resp.*, required) interface of one or more connectors.

We will also distinguish between two kinds of entities: components and adaptors. Components implement the system’s functionality, and are the primary computa- tional constituents of a system. Adaptors, on the other hand, route messages by following different coordination policies that depend on the adaptation to be per- formed whose properties are specified in an abstract way by the software architect. Our architectural style considers systems in which distributed adaptors appear:

it is referred as *Distributed Adaptor-Based Architecture* (DABA). A system is defined as a set of components each of them directly connected to its local adaptor; each adaptor is connected to other adaptors (one or many), through connectors, in a peer-to-peer fashion. Obviously a DABA may include the extreme case of an empty

1 CCS descriptions can be translated into LTS models, which is especially necessary for computation purposes (see Section [3.2](#_bookmark3) about automatic generation of adaptors).

architecture where no component and, hence, no adaptor and connector are present. Such an empty architecture is the starting point of our incremental approach for building component-based systems.

We point out that connectors, and then the underlying communication model, are *synchronous*: two components synchronize on one event (rendez-vous) and then continue their own evolution. This notion is slightly different from the one existing in certain component-based development frameworks, such as COM/DCOM archi- tectures [[15](#_bookmark27)] or BPEL for Web services [[2](#_bookmark14)]. In such models which inherit their communication features from object-oriented programming, communication is basi- cally a method call or a remote procedure call (RPC). Therefore, the caller is waiting for the callee to terminate the required processing before continuing its own evolu- tion. This is described in our model with two explicit messages, one for the request and another one corresponding to the acknowledgement. Asynchronous commu- nication can be modelled describing message queues using additional components which interact synchronously with the components they represent.

# Adaptation

Before defining methods to add and remove components, we present mechanisms to automatically check if a system needs adaptation (behavioural mismatch), the adaptor generation process, and means to evaluate the impact of the adaptation performed on the system.

* 1. *Behavioural Mismatch*

Various definitions of behavioural mismatch have been proposed in the field of software adaptation and Software Architecture analysis. We build on the most commonly accepted one, namely deadlock-freedom. Intuitively, a system made up of several identified components is deadlock-free, and therefore does not need any adaptation, if its synchronous product has no deadlock.

In a DABA style, components are viewed through their adaptors, consequently behavioural mismatch is computed taking adaptors as input instead of components. It is not necessary in case of addition to consider the added component since interac- tions between this component and its adaptor are ensured correct by the algorithm used to build this adaptor.

(*Ai, Si, Ii, Fi, Ti*), *i* ∈ {1*,... , n*}, is the LTS (*A, S, I, F, T* ) such that: Definition 3.1 [Synchronous Product] The *synchronous product* of *n* LTSs *Li* =

* + - *A* ⊆ Π*i*∈{1*,...,n*}*Ai*, *S* ⊆ Π*i*∈{1*,...,n*}*Si*, *I* = (*I*1*,... , In*),
    - *F* ⊆ {(*s*1*,... , sn*) ∈ *S* |

*i*∈{1*,...,n*}

*si* ∈ *Fi*},

* + - *T* is defined using the following rule:

∀(*s*1*,... , sn*) ∈ *S*, ∀*i, j* ∈ {1*,... , n*}*,i < j* such that

∃(*si, a, s*') ∈ *Ti,* ∃(*sj, a*¯*, s*' ) ∈ *Tj*, then

*i j*

(*x*1*,... , xn*) ∈ *S* and ((*s*1*,... , sn*)*,* (*l*1*,... , ln*)*,* (*x*1*,... , xn*)) ∈ *T* , where

∀*k* ∈ {1*,... , n*}, *lk* = { *a* if *k* = *i, a*¯ if *k* = *j, ε otherwise* }

*xk* = { *s*' if *k* = *i, s*'

if *k* = *j, sk otherwise* }

*i j*

The overline function on labels is defined as: *e*?= *e*!, and *e*!= *e*?.

Definition 3.2 [Behavioural Mismatch] An LTS *L* = (*A, S, I, F, T* ) presents a be- havioural mismatch if there is a *deadlock state s*, *i.e.,* a state *s* in *S*, not in *F* and without outgoing transitions.

In practice, behavioural mismatch can be computed (i) encoding the set of LTSs in the EXP.OPEN input format [[11](#_bookmark23)], (ii) computing the product, and (iii) checking the absence of deadlocks on the resulting automaton. Note that to distinguish final states and real deadlocks within EXP.OPEN LTSs, we first add specific loop transitions labelled with accept over final states. Point (i) has been encoded in ADAPTOR, a prototype tool under development dedicated to the adaptation of software components. Points (ii) and (iii) are computed automatically calling CADP

[[9](#_bookmark21)] which is a toolbox to validate and verify concurrent systems.

Example 3.3 Let us suppose three simple components: a client posting requests, a server receiving these requests and interacting with a counter every time a request is managed.

Client[i,f] = req!.args!.ack?.Client Server[i,f] = req?.ack!.count!.Server Adder[i,f] = add?.Adder

The product is computed and a deadlock is found out after the first transition (req!,req?,*ε*) because the client wants to send arguments whereas the server wants to send him an acknowledgement.

* 1. *Adaptors*

In this section, we follow the adaptor generation process proposed in [[8](#_bookmark20)]. To check if a system made up of several components presents behavioural mismatch, the synchronous product [[3](#_bookmark15)] of their LTS behavioural interfaces is computed and then the absence of deadlocks is checked on it. The protocol of an adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. For this to work, the adaptor has to preempt all the component communications. Therefore, prior to the adaptation process, component message names may have to be renamed prefixing them by the component name, *e.g.*, c:message!.

In [[8](#_bookmark20)], we have proposed a mapping notation based on regular expressions of synchronous vectors as an abstract and simple description of the adaptor to be generated. Synchronous vectors express not only synchronization between processes on the same event names, but more general correspondences between the events of the process involved. A synchronous vector (or *vector* for short) for a set of

*n* components LTSs *Li* = (*Ai, Si, Ii, Fi, Ti*), *i* ∈ {1*,... , n*}, is a tuple *<e*1*,... , en>* with ∀*j* ∈ {1*,... , n*} *ej* ∈ *Aj* ∪{*ε*}; *ε* meaning that a component does not participate in a synchronization. Given *n* LTSs *Li* = (*Ai, Si, Ii, Fi, Ti*), and a set of vectors *V* ,

a (vector) regex for these LTSs can be generated by the following syntax: R ::= v

*(vector)* | R1.R2 *(sequence)* | R1+R2 *(choice)* | R\* *(iteration)*, where R, R1, R2

are regex, and v is a vector in *V* .

Using such a mapping and a list of component behavioural interfaces, an adaptor can be generated automatically using algorithms presented in [[8](#_bookmark20)] where we propose two kinds of adaptation, namely adaptation *with* or *without reordering*. Reordering (changing the order of events) is needed to ensure a correct interaction when two communicating entities have protocol messages which are not ordered as required. We emphasize that other algorithms can be used for adaptor generation purposes such as [[5](#_bookmark17),[13](#_bookmark26),[15](#_bookmark27)].

In case of adaptation without reordering, a synchronous product is computed from the LTS encoding the mapping regular expression (an LTS with vectors on transitions which recognizes the regex language) and component interfaces. Then, paths leading to deadlocks are removed [[15](#_bookmark27)], messages are mirrored (inputs in place of outputs, and vice-versa) to make communications with the adaptor possible, and finally, all the possible interleavings (starting by receptions then emissions) for every synchronisation described by a vector are generated to obtain the final LTS constituting the adaptor.

In case of adaptation with reordering, the idea is to encode, into a Petri net, mirrored component interfaces and correspondences between messages described in vectors as well as restrictions on the application order of vectors induced by the mapping regular expression. Then, the LTS of the adaptor is computed from the Petri net marking graph (non-recursive adaptors) or from its cover graph (recursive adaptors). Finally, paths to deadlock are removed, and the *τ* actions added during the Petri net encoding are suppressed using behavioural reductions.

Note that our algorithms are completely automated into a tool, ADAPTOR, which is under development. This tool relies on external tools, namely TINA [[4](#_bookmark16)] for marking and cover graph computation, and CADP [[9](#_bookmark21)] for synchronous product computation and behavioural reductions.

Example 3.4 The behavioural mismatch detected in our simple client/server ex- ample is worked out by the mapping (v1.v2.v3.v4)\* where

v1=<c:req!,s:req?,a:*ε*> v2=<c:args!,s:*ε*,a:*ε*>

v3=<c:ack?,s:ack!, a:*ε*> v4=<c:*ε*,s:count!,a:add?>

The resulting adaptor computed using the algorithm without reordering sketched above is:

A[i,f] = c:req?.s:req!.c:args?.s:ack?.c:ack!.s:count?.a:add!.A

* 1. *Assessment*

Once a new adaptor is generated, we propose different means to assess the new system and make the architect sure that this system still contains all the expected services. Being given a set of components and their adaptors, the synchronous product is computed. Then, we propose several techniques to assess the adapted system.

First, the architect can check the system for services that have been achieved (internal, synchronised) or are still available (external, observable). Achieved ser-

vices are deduced from labels (*l*1*,... , ln*) where there are at least two *li* different of *ε*. Available services are deduced from labels (*l*1*,... , ln*) where there is only one *li* different of *ε*.

Deadlock freedom can be checked on the adapted system as well. Let us recall that the generated adaptor is deadlock-free by construction. However, deadlock- freeness is preserved only with respect to the components with which the adaptor interacts, whereas the full system can still contain deadlocks. Consequently, when the last adaptor is computed using only a part of the components involved in the system, it is worth checking the absence of deadlocks on the full system (see Section [5](#_bookmark6) for illustration purposes).

# Incremental Construction of Systems

In this section, we focus on the two main cases appearing while building a system, namely addition and suppression of components.

* 1. *Addition of a Component*

The method we propose can be viewed as an interactive assistant for the architect since it helps him step by step as a guide to build the system-to-be.

Algorithm 1 (Add) *Inputs: system S* = ⟨*C*1*,... , Cn, A*1*,... , An*⟩*, component C*

*to be added*

1. *check behavioural mismatch on* ⟨*C, A*1*,... , An*⟩
   1. *no deadlock* → *generate a no-op adaptor, and go to step* (iv)
   2. *deadlock* → *adaptation needed, go to step* (ii)
2. *get mapping M*
3. *compute adaptor A from mapping M, component C and connected adaptors*

*Aj*∈*I,I*⊆{1*,...,n*}

1. *assess system S*' = ⟨*C*1*,... , Cn, C, A*1*,... , An, A*⟩
   1. *validated* → *go to step* (v)
   2. *erroneous* → *return to step* (ii)
2. *add C and A in S, and connect A to C and connected adaptors Aj*

A first remark concerns both checks (behavioural mismatch and system assess- ment). They are not mandatory and are computed only if requested by the architect. The deadlock check may give back to the architect some blocking paths computed by CADP to help him to understand the problem and then write down the adaptation mapping needed to correct it.

The addition of a component does not always need protocol adaptation, *e.g.,* when adding the first component or when the test of mismatch does not detect a deadlock. However, our approach associates an automatically generated no-op adaptor to every component as in [[15](#_bookmark27)]. Such an adaptor basically reproduces from an external point of view the same behaviour as its component, and then routes

messages from other adaptors to the component and vice-versa. More formally, the no-op adaptor construction for a process referred by p transforms respectively receptions and emissions as follows: a? ~ p:a?.a!, a! ~ a?.p:a!.

Step (iv)(b) takes into account two slightly different cases, since an erroneous system can be worked out either proposing a new mapping instead of the former one and then replacing the last adaptor, or keeping the last adaptor and proposing an additional mapping to build another adaptor to be connected on top of the previous one. We will illustrate such a situation in Section [5.2](#_bookmark9). Last, mappings have to be kept while building the system since in case of suppression, they may be modified and their corresponding adaptors updated.

* 1. *Suppression of a Component*

Removing a component induces the suppression of its corresponding adaptor, but also the possible update of all the components and adaptors interacting with it. Note that, in the worst case, this corresponds to recompute all adaptors which is as costly as the regular case in a non incremental approach where the centralized adaptor is recomputed for all components.

Algorithm 2 (Remove) *Inputs: system S* = ⟨*C*1*,... , Cn, A*1*,... , An*⟩*, compo- nent Ck,k*∈{1*,...,n*} *to be removed*

1. *detect all the adaptors Aj*∈*I,I*⊆{1*,...,n*} *connected to the adaptor Ak of component*

*Ck*

1. *update the mappings Mj of connected adaptors Aj: suppress all Ck labels into vectors and modify mappings Mj*
2. *generate new adaptors A*'

*j*

*off-line*

1. *assess system S*' = ⟨*C*1*,... , Ci,i*/=*k,... , Cn, A*1*,... , Ai,i*/=*k,... , An*⟩[*A*' */Aj*]*:*

*j*

* 1. *validated* → *go to step* (v)
  2. *erroneous* → *return to step* (ii)

1. *remove Ck, Ak, and replace Aj by A*'

*j*

Step (i) is computed traversing all the mappings and detecting from the vectors the adaptors with which the component/adaptor to remove interacts. The two solutions proposed in step (ii) are complementary: in a first step, *ε* replace all the concerned labels in mappings, then the architect can update these mappings.

As regards complexity results, the most costly step in both previous algorithms is the computation of the adaptor which is exponential [[8](#_bookmark20)]. However, our approach can be applied to non-trivial systems as shown in Section [5](#_bookmark6).

# Application

In this section we show our approach at work on three examples concerning respec- tively a Video-on-Demand system, the Dining Philosophers problem and a Music Player system.

* 1. *The Video-on-Demand System*

Our first example is a simplified version of the component-based Video-on-Demand (VOD) system described in [[6](#_bookmark18)]. We consider a client-server system formed by two kinds of components, one server component, called VOD, and one or more clients of VOD. The server VOD offers services to watch movies. If a client has not registered before, he can only preview movies. If a client has registered, he can view movies. In case of registered clients, the movie can be played directly or recorded (*i.e.*, the movie is first stored and played later on). Our objective, here, is to use adaptors to take into account the differences between registered and unregistered clients. The behavioural interfaces of VOD and its clients are the following:

VOD[i,f] = search?.list!.VOD + preview?.stream!.VOD

+ view?.(play?.stream!.VOD + record?.stream!.VOD) Client[i,f] = menu!.info?.(watch!.data?.Client+store!.data?.Client)

Now, let us show how to build the VOD system incrementally by starting from an

empty architecture (*i.e.*, no component and connector) and adding step by step new components. In Figure [1](#_bookmark8) we show the three steps of the incremental construction process that the architect wants to perform in order to incrementally build a VOD system formed by one VOD server and two clients. In the following we discuss each step in detail.

EMPTY ARCHITECTURE

|  |  |  |
| --- | --- | --- |
| VOD | | |
|  | ... |  |
| A\_VOD | | |

|  |  |  |
| --- | --- | --- |
| VOD | | |
|  | ... |  |
| A\_VOD | | |

Step1: addition of the VOD server



...

...

...



Step2: addition of the C1 client

|  |  |  |
| --- | --- | --- |
| VOD | | |
|  | ... |  |
| A\_VOD | | |
|  | ... |  |

Step3: addition of the C2 client

Fig. 1. Incremental construction of the VOD system

|  |  |  |
| --- | --- | --- |
| A\_C1 | | |
|  | ... |  |
| C1 | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A\_C1 | | |  | A\_C2 | | |
|  | ... |  | | | ... |  |
| C1 | | |  | C2 | | |

Initially, the system has no component and, hence, no connector. Let us suppose that the architect of the VOD system decides to add the VOD server. At this stage, no test of behavioural mismatch is required since the system is formed by a single component and, hence, a no-op adaptor is automatically generated. Let us denote this adaptor by A VOD. It is worth mentioning that A VOD has a strictly sequential input-output behaviour, *i.e.*, each input is followed by the corresponding output. Moreover, we recall that it will reproduce from an external point of view exactly the same behaviour as the component. In the following we show a portion of the A VOD behavioural interface:

A VOD[i,f] = vod:search?.search!.list?.vod:list!.A VOD

+ vod:preview?.preview! ...

The adaptation evaluation (see Section [3.3](#_bookmark4)) is skipped since there is only one

component.

Let us suppose that an unregistered client is added, C1. It has the same be- havioural interface as Client (shown above), *i.e.*, C1 = Client. In this situation, VOD and C1 are incompatible since they use different action names (*e.g.*, search and menu actions do not match). Thus, the architect provides the following mapping that is used to generate A C1, *i.e.*, the adaptor for C1:

M1 = (<c1:menu!,vod:search?> + <c1:info?,vod:list!> +

<c1:watch!,vod:preview?> + <c1:store!,vod:preview?> +

<c1:data?,vod:stream!>)\*

By referring to Section [3.2](#_bookmark3), the mapping M1 is a regular expression of synchro- nized vectors. It defines correspondences between the names of the actions per- formed by C1 (*i.e.*, the ones with “c1:” as prefix) and by A VOD. The actions vod:view?, vod:play? and vod:record? have no counterpart in the mapping, consequently no correspondence is specified for them. M1 is a regular expression describing only a non-deterministic choice of vectors since there is no message or- dering to be performed. From A VOD, C1 and M1 the adaptor A C1 is automatically synthesized (using Section [3.2](#_bookmark3) mechanisms):

A C1[i,f] = c1:menu?.vod:search!.vod:list?.c1:info!. (c1:watch?.vod:preview!.vod:stream?.c1:data!.A C1

+ c1:store?.vod:preview!.vod:stream?.c1:data!.A C1)

and (A C1 | A VOD), where “|” stands for the synchronous product of LTSs, and Next, the architect performs the external behaviour comparison between A VOD can observe that only actions vod:view?, vod:play? and vod:record? remain

externally observable. Therefore, the architect confirms the addition of C1 (and its adaptor) in the system.

Let us suppose, now, that a C2 = Client registered client is added. Analogously to the addition of C1, the architect can observe that C2 is incompatible with respect to VOD and, hence, the test of behavioral mismatch is skipped. The architect gives the following mapping to either match watch and play or store and record:

with vectors:

M2 = (v1.v2.(v3.v5.v7 + v4.v6.v7))\*

v1=<c2:menu!,vod:search?> v2=<c2:info?,vod:list!> v3=<c2:watch!,vod:view?> v4=<c2:store!,vod:view?> v5=<c2:*ε*,vod:play?> v6=<c2:*ε*,vod:record?> v7=<c2:data?,vod:stream!>

From A VOD, C2 and M2, an adaptor is automatically synthesized:

A C2[i,f] = c2:menu?.vod:search!.vod:list?.c2:info!.

(c2:watch?.(vod:view!.vod:play!.A C2’ + vod:play!.vod:view!.A C2’)

+ c2:store?.vod:view!.vod:record!.vod:stream?.c2:data!.A C2) A C2’ = vod:stream?.c2:data!.A C2

By referring to the behavioural interface of A C2, shown above, it is worth

noticing that an adaptor does not always correspond to the specified mapping. In fact the mapping represents an abstract description of the adaptor that, once synthesized, can result in a more complex behaviour due to message interleav- ing (see, for instance, the two message sequences vod:view!.vod:play! and vod:play!.vod:view!). For this reason, the adaptor synthesis process deserves

dedicated algorithms and tools such as the ones described in [[8](#_bookmark20)]. The archi- tect performs again the external behaviour comparison between A VOD and (A C2

| A VOD), and can see that the single action visible from an external point of view

is vod:preview? which is correct. By referring to Section [3.3](#_bookmark4), we recall that when

the adaptor is computed using only a part of the components involved in the sys- tem, the adaptation evaluation process should also verify the presence of possible deadlocks in the full system. Thus, before adding C2 and its adaptor, the deadlock

check on (A VOD | A C1 | A C2) is performed. Since no deadlock is detected, C2 is

added and A C2 is connected to the other components. Note that for this example

component suppression is meaningless: either the architect removes a client (hence its corresponding adaptor) and it does not have any consequence on the system, or he/she removes the server and nothing works anymore.

* 1. *The Dining Philosophers Problem*

In this section, we consider a component-based system simulating the classical din- ing philosophers problem instantiated to the case of two philosophers and two forks. In order to eat, each philosopher needs both forks. In our model, we consider three kinds of components: Fork, Phil1 and Phil2. The behavioural interface of Fork is the following:

Fork[i,f] = fork?.ok!.release?.Fork

Phil1 and Phil2 have different behavioural interfaces (they are given below). In the classical formulation of this problem, philosophers share the same behaviour which may cause a deadlock when they both interact to access the two forks. In this example we have simplified the classical formulation by isolating only the behaviour of each philosopher that may cause a deadlock, when both of them are present. The behavioural specifications of Phil1 and Phil2 are the following:

Phil1[i,f] = fo1!.ok1?.fo2!.ok2?.rel1!.rel2!.Phil1

Phil2[i,f] = fo2!.ok2?.fo1!.ok1?.rel2!.rel1!.Phil2

Without an adaptor the system (f1:Fork | f2:Fork | Phil1 | Phil2) dead- locks because action names do not match. Although an adaptor matching the

different names can be inserted into the system, the two philosophers have also a mismatching interaction protocol. The system deadlocks whenever, *e.g.*, Phil1 asks for and obtains the access to the first fork and then, Phil2 asks for and obtains the access to the second fork. At this stage, both philosophers are blocked waiting for their complementary fork that will never be assigned to them.

The addition of the two Fork components F1 = f1:Fork and F2 = f2:Fork is done analogously to what we have done in Section [5.1](#_bookmark7) for the addition of the VOD server. The no-op adaptors of F1 and F2 are A F1 and A F2, respectively.

In Figure [2](#_bookmark10) we show the two steps of the incremental construction process that the architect wants to perform in order to incrementally build a Dining Philosophers system made up of the two already added forks and two philosophers (*i.e.*, Phil1 and Phil2). In the following we discuss these steps in detail.

2−FORKS ARCHITECTURE

F1 F2

... ...

A\_F1 A\_F2

... ...

A

... ...

A\_Phil1 A\_Phil2

... ...

Phil1 Phil2

F1

...

A\_F1

...

...

addition of Phil2

A\_Phil1

...

Phil1

|  |  |  |
| --- | --- | --- |
| F2 | | |
|  | ... |  |
| A\_F2 | | |

|  |  |  |
| --- | --- | --- |
| F1 | | |
|  | ... |  |
| A\_F1 | | |
|  | ... |  |

|  |  |  |
| --- | --- | --- |
| F2 | | |
|  | ... |  |
| A\_F2 | | |
|  | ... |  |

addition of Phil1

Fig. 2. Incremental construction of the Dining Philosophers system

Let us suppose that a Phil1 component is added. Phil1 is incompatible with respect to F1 and F2. Thus, the following mapping M1 is given:

M1 = (<phil1:fo1!, f1:fork?, f2:*ε*> + <phil1:ok1?, f1:ok!, f2:*ε*> +

<phil1:rel1!, f1:release?, f2:*ε*> + <phil1:fo2!, f1:*ε*, f2:fork?> +

<phil1:ok2?, f1:*ε*, f2:ok!> + <phil1:rel2!, f1:*ε*, f2:release?>)\*

From A F1, A F2, Phil1 and M1 an adaptor A Phil1 is computed:

A Phil1[i,f] = phil1:fo1?.f1:fork!.f1:ok?.phil1:ok1!. phil1:fo2?.f2:fork!.f2:ok?.phil1:ok2!. phil1:rel1?.f1:release!.phil1:rel2?.f2:release!.A Phil1

is stated that the system (A F1 | A F2 | A Phil1 | phil1:Phil1) has no service Analogously to what we have done in Section [5.1](#_bookmark7), the adaptor is assessed and it lost (*wrt.* the services provided by F1 and F2). Thus, Phil1 and its adaptor are

added.

Now, Phil2 has to be added. The addition of Phil2 is carried out analogously to the addition of Phil1 hence giving a mapping M2 and generating an adaptor A Phil2 connected to Phil2, A F1 and A F2. This example points out an interesting application scenario of our approach. When the architect performs the deadlock

check on the entire system, it is found out that (A F1 | A F2 | A Phil1 | A Phil2)

may deadlock. This deadlock comes from the mismatching interaction among Phil1

and Phil2 we mentioned in the beginning of this section. In this case the architect, before confirming the addition of Phil2 (and its adaptor), has to generate a new adaptor on top of both A Phil1 and A Phil2 in order to solve that deadlock. This adaptor is generated by taking into account A Phil1, A Phil2, A F1, A F2 and the mapping M3:

M3 = (<a phil1:phil1:fo1!, a f1:phil1:fo1?, a f2:*ε*, a phil2:*ε*> +

<a phil1:phil1:fo2!, a f1:*ε*, a f2:phil1:fo2?, a phil2:*ε*> + ... )\*

M3 can be automatically generated because the interface signatures of A Phil1, A Phil2, A F1 and A F2 match and no particular adaptation is required. In fact, it is only required to prune the traces leading to deadlocks in the global system (see Section [3.2](#_bookmark3)). Let us denote the last generated adaptor by A. At this point the

construction and hence (A F1 | A F2 | A Phil1 | A Phil2 | A) is deadlock-free as deadlock check is not needed since the architect knows that A is deadlock-free by well. The architect confirms the addition of Phil2 and A Phil2, and A is finally

connected to A F1, A F2, A Phil1 and A Phil2.

* 1. *The Music Player System*

Last, we consider a component-based Hi-Fi system formed by four components: HF, TR, PDA1 and PDA2. HF controls an Hi-Fi station which can be asked to (i) play mp3 files, *i.e.*, read?, (ii) stop reading, *i.e.*, halt?, (iii) stop reading temporarily, *i.e.*, pause? and (iv) resume a temporary stop, *i.e.*, resume?. Its behavioural interface is defined as follows:

HF[i,f] = read?.HFRead

HFRead = pause?.resume?.HFRead + halt?.HF

TR implements a translator which can (i) read an ogg file, *i.e.*, inogg? and

(ii) convert it into a mp3 file, *i.e.*, outmp3!. Its behavioural interface is defined as follows:

TR[i,f] = inogg?.outmp3!.TR

PDA1 is a PDA which can, among other possible actions, ask the music system to (i) play a chosen mp3 file, *i.e.*, play!, (ii) stop playing, *i.e.*, stop!, (iii) stop the player temporarily, *i.e.*, pause! and (iv) resume a temporary stop, *i.e.*, resume!. Its behavioural interface is:

PDA1[i,f] = play!.PDA1PLAY

PDA1PLAY = stop!.PDA1 + pause!.resume!.PDA1PLAY

PDA2 is a different PDA which can, among other possible actions, ask the music system to (i) play a chosen file by means of the suitable player, *i.e.*, playmp3! or playogg! and (ii) stop playing, *i.e.*, stop!. We define its behavioural interface as:

PDA2[i,f] = playmp3!.stop!.PDA2 + playogg!.stop!.PDA2

The architect adds successively HF and TR. Two no-op adaptors, A HF and A TR, are generated. In Figure [3](#_bookmark11) we show the two remaining steps of the incremental construction process that the architect wants to perform in order to incrementally build a music player system formed by the two already added servers (*i.e.*, HF and TR) and two PDA clients (*i.e.*, PDA1 and PDA2).

PDA1

...

A\_PDA1

...

A\_HF

...

HF

|  |  |  |
| --- | --- | --- |
| TR | | |
|  | ... |  |
| A\_TR | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HF | | |  |
|  | | ... |  | |
|  | A\_HF | | |  |
|  | | ... |  | |
| A\_PDA1 | | | |  |
|  | | ... |  | |
|  | PDA1 | | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | TR | | |  |
|  | | ... |  | |
|  | A\_TR | | |  |
|  | | ... |  | |
| A\_PDA2 | | | |  |
|  | | ... |  | |
|  | PDA2 | | |  |

2−SERVERS ARCHITECTURE

...

addition of PDA1

|  |  |  |
| --- | --- | --- |
| TR | | |
|  | ... |  |
| A\_TR | | |
|  | ... |  |

|  |  |  |
| --- | --- | --- |
| HF | | |
|  | ... |  |
| A\_HF | | |
|  | ... |  |

addition of PDA2

Fig. 3. Incremental construction of the Music Player system

Let us suppose that PDA1 is added. PDA1 deadlocks when interacting with HF. Thus, the following mapping is given:

M1 = (v1.(v3.v4)\*.v2)\*

with vectors:

v1=<pda1:play!,hf:read?> v2=<pda1:stop!,hf:halt?> v3=<pda1:pause!,hf:pause?> v4=<pda1:resume!,hf:resume?>

From A HF, PDA1 and M1 an adaptor is computed, A PDA1:

A PDA1[i,f] = pda1:play?.hf:read!.A PLAY

A PLAY = pda1:pause?.hf:pause!.pda1:resume?.hf:resume!.A PLAY

+ pda1:stop?.hf:halt!.A PDA1

The adaptation evaluation is performed, the system (A HF | A PDA1) has no service lost and, hence, the architect confirms the addition of A PDA1 and PDA1 in

the system.

Now, PDA2 is added. In the case of ogg files, TR has to be used to convert an ogg file into a mp3 file to be played using HF. The addition of PDA2 is carried out analogously to the addition of PDA1 hence giving a mapping M2 and generating an adaptor A PDA2 connected to PDA2, HF and TR:

with vectors:

and adaptor:

M2 = (v1.v4 + v2.v3.v4)\*

v1=<pda2:playmp3!, hf:read?, tr:*ε*> v2=<pda2:playogg!, hf:*ε*, tr:inogg?> v3=<pda2:*ε*, hf:read?, tr:outmp3!> v4=<pda2:stop!, hf:halt?, tr:*ε*>

A PDA2[i,f] = pda2:playmp3?.PLAY

+ pda2:playogg?.tr:inogg!.tr:outmp3?.PLAY PLAY = hf:read!.pda2:stop?.hf:halt!.A PDA2

The external behaviour comparison is performed and it indicates that (A HF |

A TR | A PDA2) has as observable actions only hf:pause? and hf:resume?. The architect confirms the addition of PDA2 and its adaptor.

As regards suppression, if either PDA1 or PDA2 is removed, its suppression is straightforward since there is no consequence on the system. If HF is removed, nothing works anymore since both PDA1 and PDA2 need to use it to accomplish their tasks. The only interesting case is when TR is removed since one required service of PDA2 will become unprovided by its environment. In this case, PDA2 can only play mp3 files and, hence, its adaptor has to be changed in order to not receive requests of playing ogg files anymore. This is done by replacing M2 by a new mapping M2’ that computes an adaptor that does not perform the playogg action:

with vectors

M2’ = (v1.v4)\*

v1=<pda2:playmp3!, hf:read?> v4=<pda2:stop!, hf:halt?>

the alphabet difference *A*A PDA2\*A*A PDA2' , the architect is informed that the playing Let us denote by A PDA2’ the adaptor built by taking into account M2’. Checking of ogg files on PDA2 is now observable. Depending on the system requirements this

might be acceptable or not. If it is acceptable the architect confirms the suppression of TR and its adaptor, if not he/she may consider disconnecting PDA2 too. The

the A PDA2’ addition the deadlock check is performed on the system (A HF | A PDA1 A PDA2 adaptor is replaced by A PDA2’ which connects PDA2 to HF. Before confirming

| A PDA2’) to be sure that no deadlock is introduced. A PDA2’ is added since this check succeeds.

# Concluding Remarks

In this paper, we have presented an interactive method to build incrementally systems made up of several communicating components viewed through their be- havioural interfaces. This method is supported by a specific Software Architecture which avoids costly computation steps of building global adaptors when reconfig- uring the system, if possible. In addition, most of the process steps are computed automatically: behavioural mismatch, adaptor generation, adaptor evaluation, up- dates of the system in case of suppression.

To the best of our knowledge, the closer works to ours are dedicated to incremen- tal protocol enhancement. In [[14](#_bookmark28)] it is shown how to compose component wrappers to augment connector behaviour. In [[15](#_bookmark27)], the authors have revisited [[14](#_bookmark28)] providing approach automation. In [[15](#_bookmark27)], the starting point is a centralized adaptor that is always generated. Conversely to this, we try to solve mismatches by only producing local adaptors and we produce a centralized one only when it is unavoidable (*e.g.,* Dining Philosophers problem).

The main perspective of this work is to apply our approach to existing imple- mentation languages and frameworks such as COM/DCOM architectures [[10](#_bookmark22)] or BPEL for web services [[2](#_bookmark14)].

# References

1. Special Issue on Adaptive Middleware. *Commun. ACM*, 45(6):30–64, 2002.
2. T. Andrews et al. *Business Process Execution Language for Web Services (WSBPEL)*. BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, February 2005.
3. A. Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994.
4. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.
5. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
6. A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 2006. To appear.
7. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L’Objet. Special Issue on Software* *Adaptation*, 12(1):9–31, 2006.
8. C. Canal, P. Poizat, and G. Salau¨n. Synchronizing Behavioural Mismatch in Software Composition. In

*Proc. of FMOODS’06*, volume 4037 of *LNCS*, pages 63–77. Springer-Verlag, 2006.

1. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
2. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications.

*Journal of Systems and Software*, 65(3):173–183, 2003.

1. F. Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In *Proc. of IFM’05*, volume 3771 of *LNCS*, pages 70–88. Springer-Verlag, 2005.
2. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software.

*IEEE Computer*, 37(7):56–64, 2004.

1. H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS’02*, pages 213–229. Kluwer Academic Publishers, 2002.
2. B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proc. of ICSE’03*, pages 374–384. ACM Press, 2003.
3. M. Tivoli and M. Autili. SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors. *L’Objet. Special Issue on Software Adaptation*, 12(1):77–103, 2006.
4. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.