Conformance notions for the coordination of interaction components

Rodrigo Ramos *, Augusto Sampaio, Alexandre Mota

Centro de Informática, Universidade Federal de Pernambuco, P.O. Box 7851, CEP 50732970, Recife, Brazil

A R T I C L E   I N F O

Article history:
Received 26 April 2008
Received in revised form 25 April 2009
Accepted 9 September 2009
Available online 5 December 2009

Keywords:
Software components
Frameworks
Coordination patterns
Composition predictability
Conformance verification

A B S T R A C T

In component-based development, software components are taken as units of composition. Nevertheless, to achieve the widely disseminated status of components being plugged together as simple 'Lego Pieces', the integration of components must be carefully designed, systematised and verified; only this can ensure reliable architectures. In this work, we propose some conformance notions to predict the precise nature of some forms of composition, considering coordination patterns used in the integration. These notions are formalised in terms of the denotational semantics of the process algebra CSP, and assist the designer in common activities during integration, such as the substitution of component specifications by implementations, contract adaptations, and system extensions. To support mechanical verifications using FDR (a model-checker for CSP), we derive test characterisations from the denotational definitions of conformance. We illustrate the application of these notions through a systematic composition strategy of software frameworks, and we mechanically verify the preservation of behavioural component properties in these compositions. Moreover, we characterise the well-formedness of a coordination pattern used in this strategy at the design stage, before components are assembled.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Although component-based software development [1] has been around for a long time [2], over the last decade it has re-emerged as a promising paradigm to deal with the ever increasing need for mastering complexity, evolution and reuse in the design of computer based systems. The basic motivation for this paradigm is to replace conventional programming with the composition and configuration of reusable and independent units, called components.

In practice, however, concurrent, distributed and heterogeneous software components do not fit together like ‘Lego Pieces’, or just using a simple glue code. Critical issues for software construction are related to the design of the communication-based interaction mechanisms that permit components to work together [3]; for instance, some simple mechanisms have services that convert data or facilitate the interaction among components. The correct design of these elements is critical because otherwise the system may malfunction in subtle ways or may not work at all. This concern is even more acute when connectors are put together to coordinate a group of components to accomplish a collective set of tasks [4]. Particular problems arise when coordinating the integration of heterogeneous and concurrent software components [5,6]. Integration solutions are often developed in an ad hoc manner, in which incompatibilities are not discovered until their side-effects emerge during implementation [5]; some common problems are differences in the component design concerning communication styles, data representation, protocols, synchronisation models, etc.

Therefore, it is crucial to verify whether Component Based Systems (CBS) satisfy some desired properties. Instead of verifying the entire system, after all components have been assembled, a more promising direction is to conceive a
constructive approach that defines properties to be preserved by individual components as well as for patterns of component composition. Such an approach would offer support for reasoning about the composition of components, to guide the composition in order to meet the specification of a larger system, and to predict the precise nature of any composite, so that the composite can in turn be used as a unit for further composition. The predictability of component composition depends on the nature of the component properties intended to be preserved in the composition [7]. Behavioural properties are related to the architecture (the way components are disposed) and to the environment in which the system is used. A constructive approach tends to generate more manageable proof obligations of the component and composition elements (like coordinators) in isolation, predicting emergent properties [8] that depend on the interaction of all components within the system.

Most approaches to composition [9,10] concentrate only on systematising the integration or reuse of components, lacking either formalism or considering complex mismatches in the behaviours of components in the composition; they usually neither define conformance notions for components in isolation, nor for compositions. On the other hand, solid efforts to define conformance notions [11,12] do not provide test characterisations that can be mechanically performed in verification tools.

This article is a significant extension of [13], which considers some conformance notions for composing frameworks. Here, we formalise a component model for interaction components. The constructive constraints for this model are based on the notions of substitutability and compatibility of software components. These notions allow checking whether a given component can successfully replace another in a particular application, and whether the behaviours of two components are compatible for them to interoperate, respectively. These basic notions support conformance notions for component composition. The systematic strategy for composition addressed in [13] is revisited here as a particular case study to illustrate the conformance notions.

The entire approach is based on the CSP process algebra [14], which allows us to formally address property characterisation and preservation. Every conformance notion is defined in terms of the CSP denotational semantics, which is followed by a testing characterisation based on the failures model of CSP and process refinement notions. This allows automated verification using the FDR tool [15], a model checker for CSP. Our component model focuses on software components whose contracts are described by their dynamic behaviour, interfaces, and protocols, and that repeatedly present the same behaviour to the environment. This covers a wide range of modern applications, such as Session Enterprise JavaBeans™ and transactional conversational services. We analyse the behavioural conformance of components in compositions, implementations and substitutions of interaction components, some of the main activities in Component-Based Development [1]. As far as we are aware [16–18,11,13,12], no previous work has formally addressed similar conformance notions, nor provided such a test characterisation for analysing coordination patterns in the integration (or composition) of components, as we do.

In Section 2 we present a motivating example and raise important demands that guide us throughout subsequent sections of this paper. The next section gives an introduction to CSP based on our working example, and formally defines software components. Section 4 presents some conformance notions for software components in isolation. Based on the basic notions for components, Section 5 introduces conformance notions for composition. Section 5 also presents an elaborate integration pattern for software components, which is used in our strategy for framework composition presented in Section 6. The application of notions for composition conformance is presented in Section 7, which is illustrated by a client/server case study. Related work is considered in Section 8, and a summary of our results as well as topics for further research are presented in Section 9.

2. Motivating example: Coordination patterns

As motivation, some component integration patterns [19,20] are presented in this section. Patterns express common solutions to recurring problems. More abstract patterns, such as the architectural ones we present here, are implemented in terms of more basic design patterns [21,22]. The patterns differ in the way they deal with specific characteristics of the domain problem, which in our case are integration concerns [5,6]. While we present these patterns for coordination, we raise some questions about the preservation of their desired properties during the system evolution.

*Fig. 1* presents two abstract patterns using the workflow notation [23]. As in [24], we use this notation because coordination is about managing dependencies among activities, and workflow is one of the most well known abstract notations to represent them. This is also evident in the workflows describing standards of orchestration of service-oriented systems (as, for instance, WS-BPEL) [25].

The pattern in Fig. 1.a represents an abstract solution for managing shared resources, one of the most common coordination problems [5,24]. Whenever multiple activities (A and B in the figure) share some limited resource (for example, storage space, or server time), a coordinator that controls the access to the resource is needed. The coordination process must serialise the incoming concurrent requests, and select the request to be served, by using a pre-defined access policy, such as first-come/first-serve. In the abstract pattern in Fig. 1.a one of the branches (A or B) is chosen by the deferred choice c, based on interaction with the operating environment. There is no explicit choice, but rather a race between different branches. Desired properties of such a pattern are that all clients will eventually obtain the resource (fairness), or, depending on the nature of the resource, that interactions of one client will not interfere with interactions of another.
The pattern in Fig. 1.b represents an abstract translator. Whenever one activity produces some information that is used by another activity, a coordinator that controls the transfer and mapping of information between activities is needed. It continuously receives a sequence of input data \((rcv\ ev)\) from the producer, and sends the respective sequence of data \((snd\ ev')\) to a consumer component. The exclusive choices specify an internal mechanism that selects the following task depending on the outcome of a preceding task. In other words, these choices decide how long the coordinator will wait for the sequence of inputs, or how many events are output as a result. The coordinator must guarantee the physical transfer of information between the entities, control their synchronisation, and eventually control the replication of information in case of a replicated transfer (multicast or broadcast). Particular situations in which this abstract pattern occurs in distributed systems are the integration of heterogeneous components and the reduction of the number of direct service calls of a remote node (the pattern data transfer object \([19]\)). Similar patterns are also used to adapt/reuse legacy or third-party components during system evolution.

These patterns can be used, for instance, to compose two heterogeneous components (or frameworks) \(CLIENT\) and \(SERVER\) (see Fig. 2). In the example, \(SERVER\) might already be interacting with other applications (generically represented by the environment). The purpose is to introduce the \(CLIENT\) component as a new client of \(SERVER\). The problem, however, is twofold. Firstly, in general, \(CLIENT\) and \(SERVER\) are heterogeneous components that use different services and data types in their communication. Secondly, we assume that \(SERVER\) is currently at its maximum capacity of clients (which are currently part of the system environment); in other words, the contract of \(SERVER\) does not give any guarantee about extra communications (the new communication with \(CLIENT\), in the example). Furthermore, it is necessary to translate this communication, coordinate the sharing of \(SERVER\) between its different clients (the \(CLIENT\) component and the environment), and guarantee the preservation of important properties of this composition. For instance, it is essential to ensure that the environment will not see any difference in its communication with \(SERVER\) after the composition.

A possible solution for coordinating the integration of \(CLIENT\) and \(SERVER\) is to compose the resource manager and the translator patterns in Fig. 1. The new coordination pattern would be formed of a resource manager (see Fig. 1.a) whose activity \(A\) is a translator (following the pattern behaviour described in Fig. 1.b). More details of this pattern are given in Section 5. All these patterns are useful to specify the design of abstract patterns of services to serve coordination needs of unspecified components. They become more concrete only later in the life-cycle, on the assembly with the components (\(CLIENT\) and \(SERVER\) in our example), by relying on the component interfaces and contracts \([26]\). As a consequence of coordinator life-time, some immediate questions to ask about a coordinator are:

- How to preserve the coordination purpose (abstraction) in the concrete pattern solution?
- Which properties emerge from the domain analysis during the coordination design?

Overall, as well as other authors \([18,11,27]\), we have identified the need to evaluate to what extent a final implementation satisfies the initial coordination purpose. In order to accomplish this, we introduce conformance notions that allow us to guarantee these properties both at the coordination design level and in its final implementation. The questions raised in this section are addressed during the paper, using the formal models of the workflows illustrated here.
3. Components in CSP

The basic concepts in Component-Based Software Engineering (CBSE) originate from different areas of software engineering and computer science, such as object-oriented programming, reuse, software architecture, and formal specification [28]. A consequence of this situation, together with distinct efforts of the industry related to component technology, is that CBSE uses concepts that are still not fully formalised [29]. For instance, the terms component, interface and contract, although widely discussed, have no consensual meaning. The meaning of these concepts subtly differs in the way components are observed during development. In this work, we focus on component-driven architectures, which have been widely promoted by Architectural Description Languages (ADLs) [30], and have influenced several changes in the OMG Unified Modelling Language (UML) [31].

We formalise several concepts in CBSE: interfaces, dynamic behaviour, component contracts, and communication protocols. We focus on the interaction points (represented here by interfaces) of black box components and their runtime behaviour. We are not concerned with how components are implemented. The component external behaviour is represented by a CSP process, denoted by a component contract. Actually, we use the CSP process algebra [14] to formalise our entire approach. CSP allows the description of system components in terms of synchronous processes that operate independently, and interact with each other through message-passing communication. The relationship between processes is described using process algebraic operators from which elaborate concurrency and distributed patterns can be constructed. Moreover, CSP offers rich semantic models that support a wide range of process verifications, and comparisons. The operators and semantic models of CSP are explained as the need arises.

3.1. Component interfaces

In general, components are described by means of interfaces, which define their services and capabilities independently from any particular implementation. Interfaces typically present this information in terms of the signature of services provided or required by the component. We assume that interfaces simply consist of input and output events, which may express pairs of request–response or individual synchronisation events of a service. For the following definition of interfaces, only the CSP concept of alphabet (the set of all events used by a process) is required. We use \( \Sigma \) for the set of all possible events in the system. Given a process \( P \), the expression \( \alpha P \) gives its alphabet.

**Definition 1 (Interface).** An interface \( I \) is a pair of input and output events with signature \((\Sigma, \Sigma)\), where

1. Syntactically, \( I\text{.inputs} = \text{first}(I) \) and \( I\text{.outputs} = \text{second}(I) \), where \( \text{first} \) and \( \text{second} \) yield the first and the second element of a pair, respectively;
2. inputs and outputs are disjoint, \( I\text{.inputs} \cap I\text{.outputs} = \emptyset \);
3. The complete alphabet of \( I \) is \( \alpha I = I\text{.inputs} \cup I\text{.outputs} \).

At this level of abstraction, an input and an output event can be interpreted as representing, for instance, a method invocation of a component in the object-oriented paradigm.

To help intuition, consider the communication between the CLIENT and the SERVER components (see Fig. 2). These components encapsulate services of two frameworks in a way that SERVER might provide services required by CLIENT. The interfaces of these components are the places where frameworks are customised (hot spots) by fulfilling their required services and, furthermore, the interfaces specify the points of composition. For instance, \( HSCL \) and \( HSV \) represent a required and a provided interface of CLIENT and SERVER, respectively.

\[
\begin{align*}
HSCL &= (HS\text{CL}\text{inputs}\ , \ HS\text{CL}\text{outputs}) \\
HSV &= (HS\text{SV}\text{inputs}\ , \ HS\text{SV}\text{outputs}) \\
HS\text{CL}\text{inputs} &= \{ \text{obtainResponse}, \text{validData}, \text{invalidData} \} \\
HS\text{CL}\text{outputs} &= \{ \text{processData}, \text{validateData} \} \\
HS\text{SV}\text{inputs} &= \{ \text{receiveRequest}, \text{validInformation} \} \\
HS\text{SV}\text{outputs} &= \{ \text{sendResponse}, \text{validInformation}, \text{invalidInformation} \}
\end{align*}
\]

The interface \( HSCL \) contains the events for requiring data validation and processing, and the interface \( HSV \) offers the events for processing the requests and for validating information. The data types communicated by each event are presented in the next section.

3.2. Dynamic behaviour

In practice, interfaces allow us to sort out several interaction issues when putting components together. However, it is widely recognised that this kind of (signature) interoperability is not sufficient for ensuring the correct development of component-based applications [16,32]. Apart from a static representation provided by interfaces, a complementary one is to express components in terms of their dynamic behaviour [32]. This describes how a component reaches different states during its execution, based on both internal (and external) events and actions.
As an example, we present the dynamic behaviour of CLIENT and SERVER in Fig. 3, using UML State Diagrams [31,33]. Input events are represented as annotations on the arrows, while output events are placed inside the edges. The CLIENT component (see Fig. 3.a) is able to receive data from the environment, validate such data, request data processing from another component, and wait for a response. The data validation and processing are not provided by the component itself. This data treatment is expected to be performed by another component, through the events in the interface HSCL. The SERVER component (see Fig. 3.b) is able to receive a request, perform some processing, validate information and send a response to the requester.

Using CSP, we adopt a component model semantics complementary to other existing ADLs [30], such as Wright [16], rCOS [34] UML-RT [33]. By dealing with components as black-boxes at composition, we present only the semantics related to the observational dynamic behaviour of a component. This is, for instance, a subset of what is defined in [34,33], which also considers internal actions.

Considering the CLIENT and SERVER example, the following is the specification of the CLIENT dynamic behaviour in CSP and the data types it uses. In our specification, we assume that a request event takes the form ch?x, where ch is the name of a channel and x acts as an input variable. The notation ch!v is used for response events, where v is an expression. This component uses the ClientData, ClientError, and ClientResponse types to represent the data being processed, validation errors, and actual processing results, respectively. The specification starts with the declaration of the channels that can communicate values of those data types.

```
channel getUserData, validateData, processData : ClientData  
channel invalidData, displayErrorMsg : ClientError  
channel validData  
channel obtainResponse, displayResponse : ClientResponse

CLIENTCOM = getUserData?data → validateData!data → 
( (invalidData?error → displayErrorMsg!error → SKIP) ∩ 
  (validData → processData!data → obtainResponse?resp → displayResponse!resp → SKIP) )

CLIENT = CLIENTCOM ▦ CLIENT
```

The process CLIENT repeatedly behaves as CLIENTCOM. The sequential composition operator ▦ composes two processes: P ▦ Q. The data processing starts by acquiring data from the environment using the channel getUserData. Next, the prefix operator (→) states that the event validateData!data takes place, representing the validation hot spot invocation. Then, CLIENTCOM offers two choices non-deterministically (∩): it engages either in the events validData or invalidData, for valid or invalid data, respectively. This completes the validation hot spot. The actual data processing is started by the occurrence of the processData event and can be completed by the obtainResponse event. The events displayErrorMsg and displayResponse inform the environment about the operation result: a validation error or the result of the processing, respectively. SKIP is a primitive process that stands for a successful termination. It behaves like P until it terminates successfully (SKIP); when it behaves like Q.

Note that some channels declared above are not in the HSCL interface presented in the previous section. The reason is that HSCL is an interface for communication with the server, whereas the events getUserData, displayErrorMsg and displayResponse would be part of an interface with the user.

Now we present the SERVER dynamic behaviour in CSP. The types ServerInformation, ServerRequest and ServerResponse represent the information to be validated, the requests to be processed, and the processing responses used by SERVER, respectively.
The process SERVER repeatedly performs the process \( \text{SERVER}_{\text{COM}} \). This process offers a deterministic choice (\( \Box \)) between a processing and a validation service. The request for processing is represented by the occurrence of the communication \( \text{receiveRequest}\ req \). The next step is the processing execution, \( \text{processRequest}\ req \). The processing ends with the response being sent back to the caller, \( \text{sendResponse}\ !\text{resp} \); the value of the response is internally chosen by SERVER itself (\( \Box \text{resp} : \text{ServerResponse} \)). The validation operation starts when the process engages in the communication \( \text{validateInformation}\ ?\text{info} \). The choice between returning whether the information is valid (\( \text{validInformation} \)) or invalid (\( \text{invalidInformation}!\text{msg} \)) is nondeterministic and marks the termination of the validation interface.

### 3.3. Component contract

We represent a component contract by its interfaces and dynamic behaviour, as follows:

**Definition 2 (Contract).** A component contract \( \text{Ctr} \) is a pair (interfaces, behaviour) where

1. \( \text{Ctr}.\text{interfaces} \) is a set of interfaces;
2. \( \text{Ctr}.\text{behaviour} \) is a CSP process whose alphabet is defined by \( \text{Ctr}.\text{interfaces} \):
   
   - (a) \( \text{Ctr}.\text{inputs} = \bigcup_{I \in \text{Ctr}.\text{interfaces}} I.\text{inputs} \);
   - (b) \( \text{Ctr}.\text{outputs} = \bigcup_{I \in \text{Ctr}.\text{interfaces}} I.\text{outputs} \);
   - (c) \( \alpha\text{Ctr}.\text{behaviour} = \text{Ctr}.\text{inputs} \cup \text{Ctr}.\text{outputs} \).

In the definition above, the alphabets of input and output events of the component are delimited by the component interfaces. The expression \( \bigcup SS \), for a set of sets \( SS \), yields the distributed union of all sets in \( SS \). In our example, the dynamic behaviour of CLIENT and SERVER with their respective interfaces define their contract.

### 3.4. Communication protocols

Naturally, specifications of the component behaviour at different abstraction levels are desirable depending on the circumstances. For instance, it is convenient to express component communications using protocols that specify allowed execution traces of the component services (accessed via interfaces), with an exclusive focus on events. Dynamic behaviour and protocol have a semantic relationship in nature, being commonly specified in a similar notation, but at different granularity levels. A protocol is in fact a projection of the entire dynamic behaviour over the corresponding interface. For instance, the protocol in Fig. 4.

Consequently, we consider that a component may have several protocols and interfaces that are partitioned in the number of its interactions with other components in the system [12]; each interface has a corresponding protocol that can be automatically obtained by projection, as formalised in *Definition 3*. In this definition, the operator \( P \upharpoonright X \) stands for the projection of \( P \) over the alphabet \( X \); only events within \( X \) are visible in \( P \upharpoonright X \).

**Definition 3 (Protocol).** Let \( \text{Ctr} \) be a component contract and \( I \) an interface, such that \( I \in \text{Ctr}.\text{interfaces} \). The protocol of the interface \( I \) (denoted by \( \text{Prot}_{\text{Ctr}}(I) \)) is defined as

\[
\text{Prot}_{\text{Ctr}}(I) = \text{Ctr}.\text{behaviour} \upharpoonright \alpha I
\]

The restriction operator \( P \upharpoonright X \) can be defined in terms of the CSP operator \( P \setminus Y \), where all events within \( Y \) are hidden from \( P \); only events within \( \alpha P \) and not in \( Y \) are visible in \( P \setminus Y \). For example \( Q = (a \rightarrow b \rightarrow \text{SKIP}) \setminus \{a\} \) is the same as \( Q = b \rightarrow \text{SKIP} \), which is also the same as \( Q = (a \rightarrow b \rightarrow \text{SKIP}) \upharpoonright \{b\} \).
Fig. 4. The projection of the client dynamic behaviour into a protocol.

The protocols of CLIENT and SERVER through the interfaces HSCl and HSv are expressed by the processes PROTCl and PROTsv, respectively.

\[
\text{PROTCl} = \bigcap \text{data} : \text{ClientData} \bullet \text{validateData!data} \rightarrow (\text{invalidData?error} \rightarrow \text{PROTCl}) \\
\bigcap (\text{validData} \rightarrow \text{processData!data} \rightarrow \text{obtainResponse?resp} \rightarrow \text{PROTCl}))
\]

\[
\text{PROTsv} = \text{receiveRequest?req} \rightarrow \bigcap \text{rsp} : \text{ServerResponse} \bullet \text{sendResponse!rsp} \rightarrow \text{PROTsv}
\]

\[
\bigcap \text{validateInformation?info} \rightarrow (\text{validInformation} \rightarrow \text{PROTsv})
\]

\[
\bigcap \left(\prod \text{msg} : \text{ServerResponse} \bullet \text{invalidInformation!msg} \rightarrow \text{PROTsv}\right)
\]

Note that both PROTCl and PROTsv repeatedly perform the actions of CLIENT and SERVER defined in HSCl and HSv, respectively. They accept all events communicated by CLIENT and SERVER in those interfaces, and refuse to communicate any events that their component originally refused to communicate within these interfaces.

3.5. Semantic models

The static and behavioural aspects introduced in this section consider components in isolation. However, CBSE is described not only by static or behavioural aspects of the components, but also by component interactions. As a consequence, common design activities (such as compositions, updates and refinements) are defined in terms of component relationships. In this work, these relationships can be formalised in terms of the CSP semantic models.

The process algebra CSP has semantic models based on: traces, failures and failures-divergences [14]. In the traces model, a process is represented by the set of finite sequences of communications it can perform. In the failures model, a process is represented by its traces, as in the traces model, and also by its failures. A failure is a pair \((s, X)\), where \(s\) is a trace of the process and \(X\) is the set of events the process can refuse to perform after \(s\) is performed. Finally, in the failures-divergences model, a process is represented by its stable failures together with its divergences. A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions (a livelock). Like the failures model, the stable failures also record pairs in the form \((s, X)\); however, the sequences \(s\) are those that reach a stable state where no transition is chosen nondeterministically. In other words, stable states are those in which there are no choices between external and internal actions. The stable failures of a process \(P\) is denoted by \(\text{failures}_{s}(P)\).

Divergences in a CSP process may result from unguarded recursions as, for example, the process \(P = P\), or by hiding external actions. For instance, the process \(Q = (a \rightarrow Q) \setminus \{a\}\) converts the external event \(A\) into an internal action \(\tau\). Therefore, \(Q\) indefinitely performs internal actions, which leads to a divergence. As a consequence, \(Q\) and \(P\) have the same behaviour in the failures-divergences model.

It is commonly accepted that the failures-divergences model gives us the most satisfactory representation for analysing liveness and safety properties of a CSP process. However, when we look into the mathematical theory of how divergences are calculated, it turns out that seeing accurately what a process can do after it has already been able to diverge is very difficult, and not really worth the effort [14]. Combining traces with stable failures (which is in fact the failures part of the failures-divergences model) makes it possible to see beyond any divergence by ignoring divergences altogether. Moreover, it is sometimes advantageous to analyse a divergence-free process \(P\) by placing it in a context in which it may diverge as the
result of hiding some set of actions; this only works when the traces and stable failures in this context are not influenced by these divergences.

For instance, the process \( P = (a \rightarrow P \uplus b \rightarrow P) \setminus \{b\} \) diverges in its initial state. The hiding operation converts the external choice (\( \uplus \)) into an internal choice (\( \cap \)). Therefore, the process internally chooses between the external event \( A \) and an internal action resulted from hiding \( b \). As a consequence, \( P \) may indefinitely perform internal actions, which in the failures-divergences model leads to divergence.

As we discuss in the next section, in our formalisation of some conformance notions, it is not convenient that certain hidden events result in divergence. For example, our intention is that the communication protocols of divergence-free components are also divergence-free processes, even after hiding all events not in the protocol interface. Therefore, we assume in this work software components that are divergence-free, and use the stable failures model to perform verifications on these components.

4. A component model for interaction components

All component technologies are developed to achieve specific design goals and some are used to specify and reason about the behaviour of software systems. However, as envisaged in [35], it is necessary to consciously design component technologies to enable automated and trustworthy analysis and prediction of system behaviour. This is achieved by imposing design and implementation rules on component developers and application integrators (assemblers) through a component model or an architectural style [36]. Both define component types, patterns of interaction, and other design constraints. The significance of this observation is that a system property (for instance, reliability) strongly correlates to its architectural structure. If all these constructive constraints are satisfied, an assembly can be constructed, that is, its components can be integrated, deployed, and so forth.

To achieve that, we define in this work a component model for interaction components. The constructive constraints for this component model are based on the notions of substitutability and compatibility of software components. These notions allow one to check whether a given component can successfully replace another in a particular application, and whether two components are compatible for them to interoperate, respectively. In the next sections we formally define the interaction component type and these notions.

4.1. Interaction components

In this work, we focus on components that repeatedly present the same behaviour to the environment. Such a recurring behaviour is called here an interaction process, which is itself defined in terms of interaction patterns [37]. Each interaction pattern consists of a finite sequence of events (which represent component services) that when performed leads the component (interaction process) back to its initial state. In this manner, the component repeatedly offers these sequences of events, similar to possible transactions (including compensating actions) performed against a database management system. These patterns cover a wide range of applications, like transactional stateful components found in several technologies such as, for instance, Session Enterprise JavaBeans™ and transactional conversational Web Services.

Observing components defined as an interaction process, we note that additional properties should be considered in this domain. In particular, its similarity with transactional components suggests that ACID properties [38] might be relevant. ACID is a set of properties to guarantee that database transactions are processed reliably. Similarly to an interaction pattern, a transaction represents a single logical operation, which might consist of multiple individual events. For instance, a database transaction could have one of the following actions: connect, insert or select entries and commit all changes; or connect, change some entries and rollback. In both cases, we have patterns that may repeat in the component life-time. Except for the Durability property, concerned with transaction persistence, the following ACID properties are relevant to guarantee the reliability of interaction patterns.

Atomicity refers to the guarantee that either all the actions of an interaction pattern or none of them are performed.

Consistency ensures that the component (interaction process) remains in a consistent state before the start of an interaction pattern and after it is complete.

Isolation refers to the ability of the component to perform actions in an interaction pattern that cannot be interfered with by actions in another, possibly concurrent, interaction pattern.

To present the interaction patterns of a process \( P \) (\( \text{InteractionPatterns}(P) \)), we use the CSP operator \( P/s \). If \( s \in \text{traces}(P) \) then \( P/s \) (pronounced ‘\( P \) after \( s \)’) represents the behaviour of \( P \) after the trace \( s \) is performed. So, \( \text{InteractionPatterns}(P) \) is the set of traces that lead the process to its initial state.

**Definition 4** (Interaction Patterns). Let \( P \) be a CSP process. The interaction patterns of \( P \) (denoted \( \text{InteractionPatterns}(P) \)) are:

\[
\{ s : \text{traces}(P) \mid \text{failures}_\perp(P) = \text{failures}_\perp(P/s) \}
\]

**Definition 4** is characterised in terms of the CSP stable failures semantic model, as introduced in Section 3.5. It defines the set of traces after which the process presents the same failures; these are precisely the interaction patterns of \( P \). To guarantee Atomicity and Consistency we define a process \( P \) in terms of interaction patterns (an interaction process). Its traces must be a prefix (\( \leq \)) of an interaction pattern or of a combination of them. Note that in either case they belong to \( \text{InteractionPatterns}(P) \).
Definition 5 (Interaction Process). A divergence-free CSP process $P$ is an interaction process if, and only if:

$$\forall s \in \text{traces}(P) \bullet \exists p : \text{InteractionPatterns}(P) \bullet s \preceq p$$

Based on the above definition, we are able to define components that behave as interaction processes.

Definition 6 (Interaction Component). Let $C$ be a component with contract $\text{Ctr}$. Then $C$ is an interaction component if, and only if, \text{Ctr}.\text{behaviour} is an interaction process.

We consider that an interaction pattern may present default interactions or compensating interactions (when faults occur). We do not focus on differentiating between them, nor on the detection of fault events. We are concerned with whether the complete pattern of services has been performed or not (Atomicity), and whether after one interaction pattern finishes, the process returns to a state where it can initiate other interaction patterns (Consistency). Observe that defining interaction patterns as above, only one interaction pattern can be performed at a time; in other words, all patterns are serialised (the simplest way of obtaining Isolation). These properties help us in further verifications in this paper.

Observe that interaction components have their dynamic behaviour and, as a consequence, their protocols defined as interaction processes.

Theorem 7 (Interaction Protocol). Let $\text{Ctr}$ be a component contract of an interaction component, and $I$ an interface in $\text{Ctr}.\text{interfaces}$. Then $\text{Prot}_{\text{Ctr}}(I)$ is an interaction process.

Proof Sketch. From Definition 3, $\text{Prot}_{\text{Ctr}}(I) = \text{Ctr}.\text{behaviour} \uplus I.\text{inputs}$. As $\text{Ctr}.\text{behaviour}$ is an interaction process and $\text{Prot}_{\text{Ctr}}(I)$ simply restricts the events of $\text{Ctr}.\text{behaviour}$ to $I.\text{inputs}$, then $\text{Prot}_{\text{Ctr}}(I)$ is an interaction process.

Overall, in practice we can simply represent interaction processes solely in terms of the set of their interaction patterns. As a result, any interaction process $P$ can be defined as a recursive process of the form $P = Q \triangleright P$, where the traces of the finite process $Q$ represent interaction patterns of $P$. A default implementation for a set of interaction patterns $T$ of an interaction component with contract $\text{Ctr}$ is $P_{\text{DefaultImp}}(T, \text{Ctr})$ defined as follows.

Theorem 8 (Default Behaviour Implementation). Let $\text{IS}$ be a set of interfaces, $T$ a set of interaction patterns, and $P_{\text{DefaultImp}}(T, \text{IS})$ be given by:

$$P_{\text{DefaultImp}}(T, \text{IS}) = P_{\text{Imp}}(\emptyset, \bigcup_{I \in T}.I.\text{inputs}, \bigcup_{I \in T}.I.\text{outputs}, T) \uplus P_{\text{DefaultImp}}(T, \text{IS})$$

$$P_{\text{Imp}}(s, is, os, T) = (\sqcap a_i : \text{enabled}(s, is, T) \bullet a_i \rightarrow P_{\text{Imp}}(s \uparrow \langle a_i \rangle, is, os, T))$$

$$\sqcup s \in (T - \{\}) \& \text{SKIP}$$

$$\text{enabled}(s, X, T) = \{a : X | \exists t : T \bullet s \uparrow \langle a \rangle < t\}$$

Then $\text{Ctr} = (\text{IS}, P_{\text{DefaultImp}}(T, \text{IS}))$ is a contract for an interaction component.

Proof Sketch. We have to show that $P_{\text{DefaultImp}}(T, \text{IS})$ is an interaction process. Analysing the process $P_{\text{Imp}}(s, is, os, T)$, we observe that this process is divergence-free; it forbids any loop of internal action in its definition. There are two loops in $P_{\text{DefaultImp}}(T, \text{IS})$. The first one recursively invokes the process $P_{\text{Imp}}(\ldots)$, and the second one performs all traces within an interaction pattern; the process $\text{enabled}(\ldots)$ verifies whether an event belongs to the interaction pattern that is currently being performed. The second loop only finishes when an interaction pattern is completely performed, in which case it behaves like SKIP. The process $P_{\text{Imp}}(\ldots)$ represents all interaction patterns. After an entire interaction pattern is performed (end of the second loop), the component returns to its initial state (first loop). As a consequence, we conclude that $P_{\text{DefaultImp}}(T, \text{IS})$ is an interaction process.

In the next subsections we explore relevant properties of interaction components.

4.2. Substitutability

Behavioural subtyping is a strong form of relationship between two (component) types. It requires instances of a subtype and of a supertype to fulfil the principle of type substitutability [39]:

1. An instance of the subtype should be usable wherever an instance of the supertype is expected, without a client being able to tell the difference.

This suggests the use of some form of refinement [14] to formalise behavioural subtyping. Refinement guarantees substitutability in an even stronger form: a system can always be replaced by its refinement without any noticeable difference. For subtyping, we want only that a replacement be unnoticeable at places where a supertype is expected. This is a weaker form of substitutability, but that nevertheless can be characterised in terms of refinement [40]. Refinement definitions vary according to the semantic model adopted. As we discussed in Section 3.5, for capturing substitutability we
adopt the stable failures semantic model. A process \( C \) is a stable failures refinement [14] of \( A \) (denoted \( A \preceq_{sf} C \)) if, and only if, its traces are contained in \( A \)'s, and it presents fewer stable failures; it refuses fewer communications.

\[
\text{failures}_{\perp}(C) \subseteq \text{failures}_{\perp}(A) \land \text{traces}(C) \subseteq \text{traces}(A)
\]

This characterises that the process \( C \) can be more deterministic than \( A \). Refinement is an appropriate subtyping relation only when there is no extension of functionality. However, it must be emphasised that using refinement together with additional CSP operators, it is possible to characterise other subtype relationships, and thus extend functionality.

For instance, consider the process \( \text{CLIENT} \) presented in Section 3.2. At a certain point, after requesting a data validation (\text{validateData} event), it nondeterministically chooses between accepting either the response \text{invalidData} or \text{validData} (and refuses the other). Both events are possible responses. By taking the decision of accepting one and refusing the other, this process might present communication problems after the request for validation. This is a common integration problem [27], called communication deadlock. A solution is to use a more deterministic process \( \text{CLIENT}' \) (such that \( \text{CLIENT}' \preceq_{sf} \text{CLIENT} \)) defined below. It simply replaces the nondeterministic choice operator in \( \text{CLIENT} \) with a deterministic one.

\[
\text{CLIENT}'_{\text{COM}} = \text{getUserData?data} \rightarrow \text{validateData!data} \rightarrow
\]

\[
((\text{invalidData}?error \rightarrow \text{displayErrorMsg!error} \rightarrow \text{SKIP})
\]

\[\square
\]

\[
(\text{validData} \rightarrow \text{processData!data} \rightarrow \text{obtainResponse?resp} \rightarrow
\]

\[
\text{displayResponse!resp} \rightarrow \text{SKIP})
\]

\[
\text{CLIENT}' = \text{CLIENT}'_{\text{COM}} \triangleright \text{CLIENT}'
\]

A possible implementation subtyping [40] relation in CSP for the extension of functionality is: the additional events in the implementation are hidden and afterwards the implementation is compared with the abstract specification. The additional events may either be new operations of the component or services of other component instances called by the component. As it extends functionalities, we explicitly state that all traces are guaranteed.

\textbf{Definition 9 (Implementation Subtyping).} Let \( C \) and \( A \) be CSP process, such that \( \alpha A \subseteq \alpha C \), and \( N = (\alpha C - \alpha A) \). \( C \) is an implementation subtype of \( A \) (denoted \( A \preceq_{\text{int}} C \)) if, and only if:

\[
\text{failures}_{\perp}(C) \setminus N \subseteq \text{failures}_{\perp}(A) \land \text{traces}(C \setminus N) \subseteq \text{traces}(A) \land
\]

\[
\forall t : \Sigma^*, a : N, b : \alpha A \bullet \{t\leftarrow(a), t\leftarrow(b)\} \subseteq \text{traces}(C)
\]

The hiding operation on the stable failures set is defined as follows.

\[
\mathcal{F}_{\perp} \setminus N = \{(s, X) \mid \exists(s', Y) \in \mathcal{F}_{\perp}, s = s' \setminus N \land X \subseteq Y \land N \subseteq Y\}
\]

Hiding some events makes them internal to the component. The refusals of a trace \( s \) (in which the events within \( N \) are hidden) are equal to the refusals of the original trace \( s' \). This is only analysed for the states where no outgoing transition triggered by events in \( N \) is allowed (expressed by \( N \subseteq Y \)); these are stable states. So, hiding makes states that offer events within \( N \) unstable (and thus no refusal is computed for these states). The expression that checks all traces (\( \forall t : \Sigma^* \ldots \)) states that Definition 9 is valid only for processes that forbid choices between events in \( N \) and other events in the process alphabet. The reason is that events in \( N \) represent internal actions of the process, and permitting such choices would possibly introduce deadlocks in the component. As a consequence of this condition, the component with hidden events is also an interaction process.

\textbf{Lemma 10.} Let \( C \) and \( A \) be CSP processes such that \( C \) is an implementation subtype of \( A \) (denoted \( A \preceq_{\text{int}} C \)), and \( A \) is an interaction process. Then \( C \) is an interaction process.

\textbf{Proof Sketch.} From Definition 9, no choice between new events in \( N \) and old events in \( \alpha A \) is specified in \( C \). This neither introduces possible divergences after hiding events in \( N \), nor prevents interaction patterns from repeatedly leading the component (interaction process) to its initial state.

The implementation subtype relation is useful when a component \( A \) is replaced with a component \( C \) in a context where the new operations introduced by \( C \) are not used. For instance, when the operations are internal actions of the component that specialises old services. Nevertheless, in a context where the new operations are used by another component, the implementation subtype relation is unable to capture an appropriate notion of substitutability as, for example, when the new operation leads the component to a state unexpected by other components.

Consider a component \( A \) as on the left-hand side of Fig. 5. It communicates with two independent parts of the system (the environments \( \text{Env}_1 \) and \( \text{Env}_2 \)). These two distinct environments do not communicate, unless through such a component. We want to replace this component with another one that supports a new environment \( \text{Env}_3 \) (assumed to be independent of \( \text{Env}_1 \) and \( \text{Env}_2 \)); see the right-hand side of Fig. 5. All communications of the environment \( \text{Env}_1 \) with the component are new and, therefore, must not interfere with the communications of \( \text{Env}_1 \) and \( \text{Env}_2 \). The implementation subtype relation is unable to capture such a substitution, since, for instance, by hiding communications of \( \text{Env}_3 \) in the context of \( \text{Env}_1 \) we do not forbid the environment to lead the component to unexpected states by \( \text{Env}_1 \).

A common substitution relationship that considers sharing, as in the above scenario, is based on the use of substitution functions [39,40]:
2. The execution of new operations may only lead the component to expected states.

Substitution functions allow us to relate communications in Env₂ with communications in Env₁ and Env₃ in order to show that all communications in the new environment (Env₃) do not lead the component to an unexpected state. Therefore we characterise the new communication context (with Env₃) as a new interface \( Iₙ \), and explain all interaction patterns in this interface by interaction patterns communicated with the environment Env₁. To facilitate our characterisation all communications with Env₁ are also represented by an interface \( I \). All new communications in \( Iₙ \) are observed by Env₂ as interactions of \( I \). All new communications in \( Iₙ \) are observed by Env₁ as empty traces. To achieve that the new component must manage the communication with these environments, avoiding interaction patterns in \( Iₙ \) and in \( I \) executing concurrently (Isolation). In this way, we also avoid that an environment leads the component to a state unexpected by the other ones. Therefore, informally, the new subtyping relation that we need can be summarised as follows.

3. All interaction patterns on new interfaces should be observed as either empty traces or interaction patterns in older interfaces.

The following operation on stable failures is important for the formalisation of this substitutability definition. The traces are represented using a substitution function \( F_{Iₙ \rightarrow I} \), which maps sequences of traces of \( C \) (behaviour) to sequences of traces of \( A \) (behaviour); see the components \( A \) and \( C \) in Fig. 5. This function behaves as an identity for events in the original alphabet \((\alpha A)\) and as a partial function for sequences of new events (communicated via \( Iₙ \)). Using a partial function, we are able to describe the unstable states that are not defined in the original component: those in which new events are pending in order to map a sequence of events of \( Iₙ \) into events of \( I \). In the definition below assume that \( N = \alpha Iₙ = \alpha C - \alpha A, \Sigma = \alpha C, \alpha t \subseteq \alpha A \). Moreover, consider \( \mathcal{F} = \text{failures}_{⊥}(C \text{.behaviour}) \)

\[
\mathcal{F} \setminus F_{Iₙ \rightarrow I} N = \{(a, X) \mid \exists (a', Y) \in \mathcal{F} \land a' \in \text{dom} F_{Iₙ \rightarrow I}, a = F_{Iₙ \rightarrow I}(a') \land X \subseteq Y \cup N\}
\]

where \( F_{Iₙ \rightarrow I} : \Sigma^* \rightarrow (\Sigma - N)^* \), is defined as follows:

\[
F_{Iₙ \rightarrow I}(s^{-t}) = s^{-F_{Iₙ \rightarrow I}(t)}, s \in (\Sigma - N)^* \\
F_{Iₙ \rightarrow I}(n^{-t}) = s^{-F_{Iₙ \rightarrow I}(t)}, n \in N^* \land s \in \alpha I^+
\]

Basically, the operator above states that any interaction pattern of \( A \) may be performed by \( C \), whereas new interaction patterns are only allowed when Env₃ chooses them. New events of \( C \) in such new patterns are explained by events of \( A \), and new interaction patterns are explained by interaction patterns of \( A \). All states in which new sequences of events are not explained by events of \( I \) are unstable states as well. The function \( F_{Iₙ \rightarrow I} \) explains all new events in \( C \) by events of \( A \). This function is abstractly defined in the operator above, since it is specified by the user for each pair of components \( A \) and \( C \).

The subtyping relation based on the substitution function is defined as follows.

**Definition 11** (Interaction Subtyping). Let \( C, A \) be interaction processes, \( Iₙ \) and \( I \) interfaces, and \( F_{Iₙ \rightarrow I} \) a substitution function, such that \( N = \alpha Iₙ = \alpha C - \alpha A, \Sigma = \alpha C, \alpha t \subseteq \alpha A \). \( C \) is an interaction subtype of \( A \) with respect to \( F_{Iₙ \rightarrow I} \) (denoted \( A \subseteq_{\text{intst}} C \)) if, and only if:

\[
\begin{align*}
(failures_{⊥}(C) \setminus F_{Iₙ \rightarrow I} N) &\subseteq failures_{⊥}(A) \land \text{traces}(C \setminus N) \subseteq \text{traces}(A) \land
(\forall t : \Sigma^*, a : N, b : \alpha I \mid \{t^{-}(a), t^{-}(b)\} \subseteq \text{traces}(C) \land
\quad (t, \{\alpha I\}) \notin failures_{⊥}(C) \land (t, \{\beta I\}) \notin failures_{⊥}(C) \land
(\forall t : (\Sigma^* - (\langle \rangle)) \mid t \in \text{InteractionPatterns}(C) \land
\quad ((t \setminus \alpha I = t) \lor (t \setminus \alpha N = t)) \land
\quad ((t \setminus N \neq (\langle \rangle)) \lor (t \setminus \alpha I \neq (\langle \rangle)) \lor (t \setminus (\alpha A - \alpha I) \neq (\langle \rangle)))
\end{align*}
\]

\( C \) is an interaction subtype of \( A \) if, and only if: (1) all sequences of new events in \( C \) can be explained to external observers as events in \( A \) (line 1); (2) Choices between events of \( Iₙ \) and \( I \) are deterministically chosen by the environment (line 3); (3) No interaction pattern contains both events of \( Iₙ \) and \( I \) (line 6); (4) No interaction pattern is formed only of events communicated with either Env₁, Env₂ or Env₃ (line 7). As a result, every time a new interaction pattern of \( Iₙ \) is performed, events of \( I \) are
blocked until the pattern is performed. This explains why pattern of I and \( I_A \) do not interfere in the execution of each other (Isolation). They are serialised and, therefore, not performed at the same time. All states that communicate events in \( I_A \) and do not enable events in I are unstable. The stable states that offer events of I and \( I_A \) are exactly those that need a (external) decision on which patterns to perform: either new interaction patterns of \( I_A \) or the patterns of I.

As we define our substitutability notion in the stable failures, we need to derive a testing characterisation that can be performed in practical model-checkers, such as FDR. These tools perform refinements in the traces, stable failures and failures-divergences models of CSP. The testing characterisation is obtained by constructing a tester process from the sub-type C, and checking the refinement between the supertype A and the tester of C. The tester is represented by the following process \( Tester \), which repeatedly synchronises C with the patterns (old or new).

\[
\begin{align*}
\text{Tester}(C, I, I_N, F_{I_N \rightarrow I}) &= ((C[] \alpha I | \text{New2Old}(C, F_{I_N \rightarrow I}) \setminus \alpha I_N) \setminus \alpha I_A)(R^-) \\
\text{New2Old}(C, F_{I_N \rightarrow I}) &= P_{\text{Imp}}((\{}), \text{InteractionPatterns}(C), F_{I_N \rightarrow I}) \\
\text{ChkPerfNew}(s, a, F_{I_N \rightarrow I}) &= \text{ChkPerfNew}(s, a, F_{I_N \rightarrow I}) \setminus \{\} \\
\text{PerfNew}(s, T, F_{I_N \rightarrow I}) &= (\square a : \text{enabled}(s, T) \bullet \text{ChkPerfNew}(s, a, F_{I_N \rightarrow I}) | R^-) \\
\end{align*}
\]

\( R \) and \( R^- \) are injective renaming functions, \( R(a) = a \) if \( a \notin \alpha I \), \( R(a) = a' \) if \( a \in \alpha I \); \( I' = \{ev' | ev \in \alpha I \} \). \( R^- \) is the inverse function of \( R \).

In \( Tester \), the process C is synchronised with \( \text{New2Old}(C, F_{I_N \rightarrow I}) \), which synchronises in any interaction pattern of C. \( \text{New2Old}(C, F_{I_N \rightarrow I}) \) is defined in terms of the process \( P_{\text{Imp}} \) which performs all interaction patterns in T (similar to the auxiliary process \( P_{\text{Imp}} \) in Theorem 8). As \( \text{New2Old} \) is aimed to synchronise with \( C \), all choices in \( P_{\text{Imp}} \) are deterministic. Every time an event is performed, the processes \( \text{ChkPerfNew} \) and \( \text{PerfNew} \) are invoked in order to check whether the last new events can be explained in terms of old ones. To avoid undesirable synchronisations the renaming functions \( R \) and \( R^- \) are used. In fact, \( R^- \) is the inverse of \( R \) and undoes all renamings of \( R \). Finally, all new events in \( N \) are hidden from the tester process. Note that only well-formed substitution functions would not deadlock this tester process.

\[
\begin{align*}
\text{ChkPerfNew}(t, a, F_{I_N \rightarrow I}) &= \text{PerfNew}(t, F_{I_N \rightarrow I}) \setminus a \in N \setminus \text{SKIP} \\
\text{PerfNew}((), F_{I_N \rightarrow I}) &= \text{SKIP} \\
\text{PerfNew}(n, a, F_{I_N \rightarrow I}) &= (\text{Perform}(F_{I_N \rightarrow I}(n)) \setminus n \in \text{dom} F_{I_N \rightarrow I} \setminus \text{SKIP}, n \in N^*) \\
\text{PerfNew}(s, a, F_{I_N \rightarrow I}) &= (\text{Perform}(F_{I_N \rightarrow I}(n)) \setminus n \in \text{dom} F_{I_N \rightarrow I} \setminus \text{SKIP}, n \in N^* \land a \notin N \\
\text{Perform}(()) &= \text{SKIP} \\
\text{Perform}(a) &= a \rightarrow \text{Perform}(s) \\
\end{align*}
\]

The process \( \text{ChkPerfNew} \) checks if a sequence of new events has been completely performed. The verification is done using the condition operator \( P \setminus \text{cond} \setminus Q \). If \( \text{cond} \) is evaluated true then \( P \) is performed. Otherwise, \( Q \) is performed. \( \text{PerfNew} \) represents the new events using the substitution function \( F_{I_N \rightarrow I} \). Every time a sequence of new events in \( N \) is performed, \( \text{PerfNew} \) performs events in P that represents such new events. The auxiliary process \( \text{PerfNew} \) receives a sequence of events and generates a process whose maximum trace is this sequence.

The following theorem relates the denotational definition of interaction subtype with the one based on process failures refinement, which can be mechanically carried out by practical model checkers. Note that the Tester process introduces all events in \( \alpha A \) that explain new events of C. When these events are introduced, Tester is concerned only with the mapping of traces. As a result, \( A \sqsupseteq P_{\text{intst}} C \) if \( A \) is refined by the Tester process in the traces model. The refusal can be directly verified from refinement relations of A and C in each context A is expected (\( Env_1 \) and \( Env_2 \)). The hiding and the restriction operators are used below to define such contexts.

**Theorem 12.** Let A and C be interaction processes, and IS the set of interfaces of A, such that \( aA \sqsubseteq AC, N = AC - aA \). Then A is an interaction subtype of C (\( A \sqsubseteq C \)):

\[
\begin{align*}
A &\sqsubseteq I \quad \text{Tester}(C, I, I_N, F_{I_N \rightarrow I}) \wedge \\
A &\sqsubseteq I \quad C : I \wedge \\
A &\sqsubseteq I \quad C : I \\
\end{align*}
\]

**Proof Sketch.** Definition 11 states that: all sequences of new events in C can be explained to external observers as events in A. The proof of this statement is divided into two parts. First we check the traces of all processes involved. \( \text{Tester}(C, I, I_N, F_{I_N \rightarrow I}) \) performs the same traces as those of A, considering that the substitution function \( F_{I_N \rightarrow I} \) consistently represents all new sequences of events of C in terms of events of A (line 1, Definition 11). As a result,

\[
\text{traces}(\text{Tester}(C, I, I_N, F_{I_N \rightarrow I})) \subseteq \text{traces}(A)
\]

For the failures set, we consider the assumption that the environment is partitioned in \( Env_1 \) and \( Env_2 \) (see Fig. 5), and that only comparisons of failures concerning these partitions are necessary (lines 2-6, Definition 11). We need to show that

- \( (\text{failures}_1(C) \setminus F_{I_N \rightarrow I} \setminus \alpha I_N) \setminus (\alpha C - (\alpha A - \alpha I)) = \text{failures}(C \setminus (\alpha C - (\alpha A - \alpha I))) \)
- \( (\text{failures}_1(C) \setminus F_{I_N \rightarrow I} \setminus \alpha I_N) \setminus (\alpha C - \alpha I) = \text{failures}(C \setminus (\alpha C - \alpha I)) \)
Using the hiding operator, the communication is restricted to the alphabet used by Env$_1$ or by Env$_2$, also hiding all events of $I_N$. All hidden events are internal. As C is assumed to be divergence free, the set failures$_L(C)\backslash_{\beta N_{\infty}}$ contains all states that offer events to Env$_1$ and Env$_2$. Moreover, as there is no interaction pattern composed only of events communicated either with Env$_1$, Env$_2$, or Env$_3$ (line 7, Definition 11), no unstable state is introduced after hiding these events. Therefore, the expressions in the stable failures semantic model are equivalent to the ones in the failures semantic model. 

With this theorem we have a mechanical way of checking substitutability of components.

### 4.3. Compatibility

While substitution is related to the update of a component in an architecture, compatibility checks the relation of two connected components in the architecture. As a consequence, instead of considering the entire specification of the components, compatibility checks the protocols and interfaces used in the communication. Before presenting the compatibility notion for protocols, some relevant concepts that underly our notion are required: dual interfaces and safety acceptance processes. We say that an interface $I$ is dual to an interface $J$ if all output events of $I$ are inputs of $J$, and vice-versa. Moreover, the respective sets of inputs and outputs of these interfaces are disjoint.

**Definition 13** (Dual Interfaces). Let $I$ and $J$ be interfaces. $I$ is dual of $J$ if, and only if:

$$I\text{.outputs} \subseteq J\text{.inputs} \land J\text{.outputs} \subseteq I\text{.inputs} \land$$

$$(I\text{.inputs} \cap J\text{.inputs}) \cup (I\text{.outputs} \cap J\text{.outputs}) = \emptyset$$

The acceptance notion relates processes $Q$ and $R$ with respect to the subsets $Y$ of their alphabets. The definition states that: every time an event within $Y$ is enabled in $R$ it is also enabled in $Q$. The safety acceptance is a stronger notion that includes acceptance. It relates $Q$ with respect to $Y$ and another subset $X$ of their alphabet. In addition to acceptance, the definition states that $Q$ never deadlocks waiting for an event within $X$ that does not come from $R$.

**Definition 14** (Acceptance). Let $Q$ and $R$ be CSP processes, $Y$ be a set of events, such that $Y \subseteq \alpha Q$. Then $Q$ matches the traces of $R$ with respect to $Y$ (denoted by $Q \text{ accepts}_{Y} R$) if, and only if:

$$\forall t : \alpha R^*, a : Y \mid t^\ast(a) \in \text{traces}(R) \bullet (t, \{a\}) \notin \text{failures}(Q)$$

**Definition 15** (Safety Acceptance). Let $Q$ and $R$ be CSP processes, and $X$ and $Y$ sets of events, such that $X \subseteq \alpha R$, $Y \subseteq \alpha Q$ and $Y \subseteq X$. Then $Q$ matches the traces of $R$ with respect to $X$ and $Y$ (denoted by $Q \text{ safaccepts}_{X,Y} R$) if, and only if:

1. $Q \text{ accepts}_{Y} R$
2. $\forall t : \alpha X^*, a : X \mid t^\ast(a) \in \text{traces}(Q) \bullet (t, X) \notin \text{failures}(R)$

For a component $C$, $C \text{ accepts}_{Y} R$ and $C \text{ safaccepts}_{Y} R$ mean that $C$ never refusal input event that is enabled in $R$. For instance, we say that a component $C$ obeys input acceptance if it satisfies $C \text{ behaviour accepts}_{\alpha C\text{.inputs}}$. This means that $C$ never refuses an input event defined in its traces.

Based on these concepts, we can define an important notion of compatible protocols: protocols whose communication (resulting from their synchronisation) is deadlock-free; they always accept communications from each other.

**Definition 16** (Protocol Compatibility). Let $I$ and $J$ be two dual interfaces of two distinct components with contracts $Ctr_1$ and $Ctr_2$, respectively. Their protocols $Prot_{Ctr_1}(I)$ and $Prot_{Ctr_2}(J)$ are compatible if, and only if, each one safely accepts the output traces of the other.

$$Prot_{Ctr_1}(I) \text{ safaccepts}_{I\text{.inputs},J\text{.outputs}} Prot_{Ctr_2}(J)$$
$$Prot_{Ctr_2}(J) \text{ safaccepts}_{J\text{.inputs},I\text{.outputs}} Prot_{Ctr_1}(I)$$

This is an effective way of ensuring that the communication between two components is deadlock-free. This notion is related to Stuck-freedom conformance for CCS processes [41], concerning events in $X$, but we consider only synchronous communication. In general, we only synchronise components that obey input acceptance, since they refuse input events that could be accepted in their protocols.

A testing characterisation of this notion via refinement can be defined upon a special process, called dual protocol [12]. We define the dual of a protocol $Q$ as a process whose sets of inputs and outputs are equal to $Q\text{.outputs}$ and $Q\text{.inputs}$, respectively, and whose interaction patterns are both given by $InteractionPatterns(Prot_{Ctr_2}(I))$. Moreover, it safely accepts $Q\text{.outputs}$ and it is totally non-deterministic with respect to the other events ($Q\text{.inputs}$).

**Definition 17** (Dual Protocol). Let $I$ be an interface of a component contract $Ctr$. The dual protocol of $Prot_{Ctr}(I)$ is defined as the process $DProt_{Ctr}(I)$, such that:

$$traces(DProt_{Ctr}(I)) = traces(Prot_{Ctr}(I)) \land$$
$$DProt_{Ctr}(I) \text{ accepts}_{I\text{.outputs}} Prot_{Ctr}(I) \land$$
$$\forall s : traces(DProt_{Ctr}(I)) \mid b \in I\text{.inputs} \bullet (s, \{b\}) \in \text{failures}(DProt_{Ctr}(I))$$
A systematic way to obtain the dual protocol of the default one of an interface $I$ is by replacing the choices (internal or external) within $Prot_{Ctr}(I)$ with internal (nondeterministic) ones in places where input communications are involved, and with external choices where only output communications are involved. This strategy allows implementations that communicate with the protocol to decide how they provide these inputs. Alternatively, a dual protocol can also be obtained by changing the parameters of the default implementation of the interaction patterns of a protocol. Suppose $T$ is the set of interaction patterns of $Prot_{Ctr}(I)$, then $DProt_{Ctr}(I) = P_{inp}(\langle \rangle, I.outputs, I.inputs, T)$ (see Theorem 8). In our example (see Section 3.4), the dual protocols of $HS_{CL}$ and $HS_{SV}$ are respectively expressed by the processes $DUAL_{PCL}$ and $DUAL_{PSV}$, as follows:

$$DUAL_{PCL} = (\text{validateData?data} \rightarrow (\square \text{err} : \text{ClientError} \; \&\& \text{invalidData!err} \rightarrow \text{SKIP} \; \&\&$$

$$\square \text{validData} \rightarrow \text{processData.data} \rightarrow (\langle \square \text{resp} : \text{ClientResponse} \; \&\& \text{obtainResponse!resp} \rightarrow \text{SKIP} \rangle) \; \&\&$$

$$DUAL_{PSV} = (\langle \square \text{r} : \text{ServerRequest} \; \&\& \text{receiveRequest!r} \rightarrow \text{sendResponse?x} \rightarrow \text{SKIP} \rangle \; \&\&$$

$$\square \text{info} : \text{ServerInformation} \; \&\& \text{validateInformation!info} \rightarrow (\langle \text{validateInformation} \rightarrow \text{SKIP} \rangle \; \&\&$$

$$\text{invalidInformation?msg} \rightarrow \text{SKIP} \rangle) \; \&\&$$

The following theorem addresses the test characterisation of the compatibility definition given in terms of a denotational semantics (Definition 16).

**Theorem 18.** Let $I$ and $J$ be two interfaces of distinct component contracts $Ctr_1$ and $Ctr_2$, such that all protocols of $Ctr_1$ obey input acceptance. Then $Prot_{Ctr_2}(J)$ is compatible with $Prot_{Ctr_1}(I)$ if $DProt_{Ctr_1}(I) \subseteq_{PCL} Prot_{Ctr_2}(J)$.

**Proof Sketch.** From Definition 17, $DProt_{Ctr_1}(I)$ accepts all output events of $Prot_{CTR_1}(I)$, and traces($DProt_{CTR_1}(I)$) = traces($Prot_{CTR_1}(I)$). As a result, the process $DProt_{CTR_1}(I)$ never deadlocks waiting for an output event that does not come from $Prot_{CTR_1}(I)$. Furthermore, the dual protocol safety accepts all outputs of the default protocol ($DProt_{CTR_1}(I)$ safaccepts outputs $Prot_{CTR_1}(I)$). The same is obtained from communications from $Prot_{CTR_1}(I)$ to $DProt_{CTR_1}(I)$. As $Prot_{CTR_1}(I)$ obeys input acceptance, $Prot_{CTR_1}(I)$ safaccepts inputs $DProt_{CTR_1}(I)$. As a result, $Prot_{CTR_1}(I)$ and $DProt_{CTR_1}(I)$ are compatible. If $DProt_{CTR_1}(I) \subseteq_{PCL} Prot_{CTR_2}(J)$, then $Prot_{CTR_2}(J)$ is also compatible with $Prot_{CTR_1}(I)$.

With this test characterisation defined in terms of refinement, verifications can be mechanically carried out by the FDR model checker. However, refinement checking captured by Theorem 18 cannot be directly applied to verify all possible communications. The reason is that some components may accept more inputs than necessary in a communication, and despite being compatible with a protocol $Prot_{CTR_1}(I)$ they do not refine the dual protocol $DProt_{CTR_1}(I)$. So, we need a mechanism that restricts the events communicated to the ones used by the protocol.

A useful process to help in verification would be a deadlock-free process that represents all possible communications between a protocol $Prot_{CTR_1}(I)$ and another process compatible with it. We call this process communication context process of $Prot_{CTR_1}(I)$. To define this process we need to consider only the protocol $Prot_{CTR_1}(I)$ and its counter-part $DProt_{CTR_1}(I)$. As the input events of $Prot_{CTR_1}(I)$ are outputs of $DProt_{CTR_1}(I)$, we need only to specify the communication context process with respect to $Prot_{CTR_1}(I)$.

**Definition 19 (Communication Context Process).** Let $I$ be an interface of a component contract $Ctr$, and $Prot_{CTR_1}(I)$ its default protocol. A communication context process of $I$ (denoted by $CTX_{CTR_1}(I)$) is defined as:

$$\text{traces}(CTX_{CTR_1}(I)) = \text{traces}(Prot_{CTR_1}(I)) \land$$

$$CTX_{CTR_1}(I) \text{ accepts}_{I.inputs} Prot_{CTR_1}(I) \land$$

$$CTX_{CTR_1}(I) \text{ accepts}_{I.outputs} Prot_{CTR_1}(I)$$

A systematic way to define the communication context process of $Prot_{CTR_1}(I)$ is by using external choice for all outgoing transitions in states of $Prot_{CTR_1}(I)$. All events must be written in the form $ch?data$, where $ch$ is the name of a channel in the interface and $data$ is an identifier to hold possible data values associated with the interface. Alternatively, a communication context process can also be obtained by changing the parameters of the default implementation of the interaction patterns of a protocol. Suppose $T$ is the set of interaction patterns of $Prot_{CTR_1}(I)$, then $CTX_{CTR_1}(I) = P_{inp}(\langle \rangle, aI, \emptyset, T)$. Analysing the communication events of the protocols of $HS_{CL}$ and $HS_{SV}$ in our example (see Section 3.4), we note that their communication contexts can be expressed by the following processes $CTX_{CL}$ and $CTX_{SV}$, respectively.
Definition add external features, such as buffering and polling, to components in order to facilitate their interaction with other components. They act often on the architectural structure, mediating the exchange of events without changing the communication behaviour. They typically introduce orthogonal concerns to the component. Examples are interceptors [21] and extension interfaces [43].

5. Component integration patterns

As revealed in many patterns and integration solution proposals [21,22,42,43], there are three categories of component integration solutions: Extender, Translator and Controller [5].

Extenders add external features, such as buffering and polling, to components in order to facilitate their interaction with other components. They act only on the architectural structure, mediating the exchange of events without changing the communication behaviour. They typically introduce orthogonal concerns to the component. Examples are interceptors [21] and extension interfaces [43].

Translators perform data transformation, marshalling, and the mapping of data among components. They also act only on the architectural structure. Examples are adapters [22] and converters [21].
Controllers coordinate and mediate the control flow among components [42]. They execute a predefined decision-making heuristic, which allows the determination of which source input information to pass, modify, or discard and to which target components valid information is transferred. A controller may have different purposes [21,22] depending on the conflict that it resolves as, for instance, serving a structural purpose when providing a façade, or a behavioral purpose when mediating multiple components. Examples are load balance and resource sharing managers [5].

Many integration solution strategies combine different classes of integration. Some consider them as a first-class architecture entity with specific functions [44,3]. This means that integration solutions may specify patterns of services with abstract and generic interfaces at the design level, and become more concrete when assembling components [26]. We consider in this paper all these integration solutions more broadly as exogenous coordinators [4,17], intended to mean ‘coordination from outside’. Exogenous coordination separates computation from the coordination itself; this simplifies system specification, understanding, construction, evolution and validation of properties.

As an example of a coordinator, we present the Composition and Synchronisation Component in Fig. 6. It combines the resource manager (a controller) and the translator patterns, presented in Section 2 (see Fig. 1).

The CSC coordinator is a resource manager, managing the access to services of a component (see pattern description in Section 2) connected to the interface HotSpotC (see Fig. 6a). It is designed to offer these services to two clients connected to the interfaces HotSpotA and HotSpotB. It defines the context where the component instances can communicate with the component on HotSpotC. Moreover, it controls possible synchronisation and translation between these interfaces. Therefore the CSC is also a translator. The interfaces HotSpotB and HotSpotC have the same alphabet, and have, in fact, dual protocols. Everything received on HotSpotB is sent through HotSpotC, and vice-versa. From an external point of view, components on these two interfaces are synchronised. The part of CSC responsible for this synchronisation is called CSCsync. On the other hand, HotSpotA and HotSpotC are heterogeneous. CSC uses a translator pattern to map events and data values from one interface to another (see pattern description in Section 2). The part responsible for the translation is called CSCcom.

To possibly reuse CSC to compose different components instances, the coordination purpose of CSC is abstractly defined based on patterns of services of unspecified components. At the implementation level, it is specialised with the interaction patterns of the services of each component instance it manages. As the protocol of HotSpotC is a dual of protocol of HotSpotC, only the interaction patterns of HotSpotA and HotSpotB are necessary. Abstractly, we could define CSC as follows.

\[
\text{CSC} = (\text{CSC}_{\text{com}}(\text{InteractionPatterns}(\text{Prot}_A), \text{InteractionPatterns}(\text{Prot}_C)), \text{CSC}_{\text{sync}}(\text{InteractionPatterns}(\text{Prot}_C)) \triangleright \text{CSC}
\]

More details about its behaviour are given in the next section.

6. Application: Formal framework composition

Before applying conformance notions on component coordination, we briefly present a composition strategy we developed in an earlier work [27], which aims to overcome some typical problems during framework integration presented in [6]. Such problems arise when frameworks need to be assembled with each other and with legacy components [45]. As a consequence, even at the specification level, framework composition may give rise to a number of problems, such as incoherent control flow composition and entities overlap [6,45]. These problems are typically related to communication incompatibilities, related to their protocols (incoherent control flow) or to the data types used in the communication (entities overlap).

A framework can be defined as a set of classes that incorporates an abstract design to solve problems of a specific domain [6]. A framework customisation in each domain can be performed either by extension, through specialisation of their classes, or by composition, similarly to components, with other frameworks or applications. In [27], specialisations are verified using ordinary CSP process refinements, like, for instance, checking the specialisation of CLIENT by CLIENT’ in Section 3. We focus on the customisation of frameworks by composition, which is less vulnerable to internal changes [9]. As a consequence, the interfaces of components that encapsulate the frameworks are the places where they are customised (hot spots) and, furthermore, where they are composed. The strategy developed in [27] is also based on CSP, which allows us to deal with property preservation of such compositions.
Table 1
Steps of the systematic strategy to compose framework.

<table>
<thead>
<tr>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Checking properties of the original frameworks</td>
</tr>
<tr>
<td>2. Identifying composition points</td>
</tr>
<tr>
<td>3. Identifying matchings of sequences of events</td>
</tr>
<tr>
<td>4. Communication mapping</td>
</tr>
<tr>
<td>5. Composing hot spots</td>
</tr>
<tr>
<td>6. Specifying the Communication and Synchronisation Component</td>
</tr>
<tr>
<td>7. Composing the frameworks</td>
</tr>
</tbody>
</table>

Table 2
Processing and Validation Hot spot events.

<table>
<thead>
<tr>
<th>Occurred events</th>
<th>→</th>
<th>Enabled events</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; processData &gt;</td>
<td>→</td>
<td>&lt; receiveRequest &gt;</td>
</tr>
<tr>
<td>&lt; sendResponse &gt;</td>
<td>→</td>
<td>&lt; obtainResponse &gt;</td>
</tr>
<tr>
<td>&lt; validateData &gt;</td>
<td>→</td>
<td>&lt; validateInformation &gt;</td>
</tr>
<tr>
<td>&lt; validInformation &gt;</td>
<td>→</td>
<td>&lt; validData &gt;</td>
</tr>
<tr>
<td>&lt; invalidInformation &gt;</td>
<td>→</td>
<td>&lt; invalidData &gt;</td>
</tr>
</tbody>
</table>

Table 3
Matching data type in the example.

<table>
<thead>
<tr>
<th>Domain Type</th>
<th>Generates</th>
<th>Target type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientData</td>
<td>→</td>
<td>ServerRequest</td>
</tr>
<tr>
<td>ServerResponse</td>
<td>→</td>
<td>ClientResponse</td>
</tr>
<tr>
<td>ClientData</td>
<td>→</td>
<td>ServerInformation</td>
</tr>
<tr>
<td>ServerResponse</td>
<td>→</td>
<td>ClientError</td>
</tr>
</tbody>
</table>

This strategy represents a precise solution to the problems of control-flow composition and entities overlap. The former problem is addressed by matching interaction patterns of one component into patterns of the other and checking the absence of possible deadlock situations. The latter, the modelling of different abstractions of real world entities, is addressed by a mapping between entities.

Part of the solution to these problems is achieved through the adoption of the Communication and Synchronisation Component (CSC) in our strategy. As we have shown in Section 5, CSC is responsible for effectively integrating the frameworks, as well as coordinating the interaction between the composed frameworks and the environment. The concrete version of CSC can be automatically generated, provided its parameters are defined for the composition.

Table 1 shows the steps of our strategy to compose frameworks. An overview of the strategy is presented in the rest of this section through the composition process of CLIENT and SERVER (see Section 2), and the creation of a concrete version of CSC. The strategy determines how heterogeneous components that encapsulate such frameworks are composed with CSC, and how the CSC parameters can be derived from the component contracts.

6.1. Composition strategy

An obvious initial step of the strategy is to identify the possible communication (or composition) points. The composition goal of our example is to delegate the implementation of the hot spots of CLIENT to SERVER. This interaction is controlled by the Communication and Synchronisation Component (CSC), which is responsible for, among other things, matching the corresponding sequences of events and values involved in the communication between the different interfaces (see Fig. 2).

Our example falls into the simpler case where each communicated event from one framework corresponds to one event of the other. However, our strategy deals with the more general case of many-to-many mappings.

Table 2 captures information for the composition of the processing and validation hot spots of our example. For instance, when CLIENT (see specification in Section 3) engages on the event validateData, the CSC must synchronise with the event validateInformation of SERVER. This table shows the mapping of occurred events (those events that belong to the hot spot that require some services) into events enabled by the CSC in the target framework.

Observing the mapping between events in Table 2, we note the need for matching their corresponding values (parameters) in the communication. For our example, the necessary mapping between parameters of the events in Table 2 is informally described in Table 3.

When composing two frameworks, we must also guarantee that the events enabled by the CSC are accepted by the target framework; otherwise the composition would present new deadlock situations. In order to avoid this, the strategy requires the refinement of frameworks with nondeterministic hot spots by deterministic versions of them. From a practical point of view, the refinement can be justified by the delegation of the implementation of the hot spot by a more deterministic version of the framework (see Section 4.2). In our example, the CLIENT framework is the one to be refined. The refined version of
CLIENT is expressed by CLIENT’, which presents an external choice operator instead of the internal choice operator on the validation hot spot, as presented below.

The purpose of the CSC is to define situations in which the frameworks can communicate (synchronising on some events) and those in which they can communicate with the environment, independently. As already explained (see Fig. 6), the part of CSC responsible for communicating these frameworks is named CSC_com. In addition to the set of events in the hot spots, CSC_com is parametrised by information extracted from event sequences in the event and data mapping tables (see Tables 2 and 3). Here, for the sake of brevity, we present only the expanded version of CSC_com, introduced below as the process CSC_com(CLIENT’, SERVER).

\[
\text{CSC}_\text{com}(\text{CLIENT’}, \text{SERVER}) = \text{HDSHK}_\text{CLISERV} \parallel \text{COMM}_\text{CLISERV} \\
\text{HDHK}_\text{CLISERV} = \text{validateData}?\text{data} \rightarrow \\
\text{validateInformation}!\text{ClientData} \rightarrow \text{ServerInformation} \quad \text{(data)} \rightarrow \text{SKIP} \\
\text{COMM}_\text{CLISERV} = \\
\quad \text{validInformation}! \rightarrow \text{validateData} \rightarrow \text{COMM}_\text{CLISERV} \\
\quad \square \text{processData}?\text{dt} \rightarrow \text{receiveRequest}’!\text{M} \quad \text{ClientData} \rightarrow \text{ServerRequest} \quad \text{(dt)} \rightarrow \text{COMM}_\text{CLISERV} \\
\quad \square \text{invalidInformation}?\text{msg} \rightarrow \\
\quad \text{invalidData}!\text{M} \rightarrow \text{ServerResponse} \rightarrow \text{ClientError} \quad \text{(msg)} \rightarrow \text{SKIP} \\
\quad \square \text{sendResponse}’\text{resp} \rightarrow \text{obtainResponse}!\text{M} \rightarrow \text{ServerResponse} \rightarrow \text{ClientResponse} \quad \text{(resp)} \rightarrow \text{SKIP}
\]

Initially, the process CSC_com(CLIENT’, SERVER) synchronises on the sequence of events (validateData?data) that initiate the communication between the frameworks (HDHK_CLISERV). Then it accepts the events of the sequence (COMM_CLISERV), one by one, until it synchronises on an event that ends the communication. After the occurrence of output events by one of the frameworks, events of the other are enabled according to Table 2. To facilitate the management of enabled SERVER events, instead of directly using these events, we use events that internally represent them in the composition; for each event \( ev \) an event \( ev’ \) is created. The purpose of these renamed events is to allow non-interference between CLIENT and SERVER and that between CLIENT and the environment; further details are explained later on. The mapping of parameters of the occurred events into parameters of the enabled events are represented by a mapping \( M_{T_a \rightarrow T_b} \), where the placeholders \( T_a \) and \( T_b \) correspond to types in the domain and in the target specification, respectively, as presented in Table 3.

To allow SERVER to be free to communicate with the environment when it is not committed with CLIENT (no communication has been initiated), we define the process CSC.sync(SERVER). Here, for the sake of brevity, we present only the expanded version of CSC.sync, introduced below as CSC.sync(SERVER); for the complete specification of CSC_com and CSC.sync see [27]. Similarly to CSC_com, CSC.sync(SERVER) first synchronises on the events that initiate every communication (HDHK_SERV). Then it communicates intermediate events, and finally those (COMM_SERV) that finalise it. Observe that CSC.sync behaves like CSC_com, mapping, for instance, SERVER events into their internal counterparts.

\[
\text{CSC}_\text{sync}(\text{SERVER}) = \text{HDHK}_\text{SERV} \parallel \text{COMM}_\text{SERV} \\
\text{HDHK}_\text{SERV} = \text{receiveRequest}?\text{inf} \rightarrow \text{receiveRequest}’!\text{inf} \rightarrow \text{SKIP} \\
\text{COMM}_\text{SERV} = \\
\quad \square \text{validateInformation}?\text{inf} \rightarrow \text{validateInformation}’!\text{inf} \rightarrow \text{SKIP} \\
\quad \square \text{sendResponse}’\text{resp} \rightarrow \text{sendResponse}’!\text{resp} \rightarrow \text{SKIP} \\
\quad \square \text{validInformation} \rightarrow \text{validateInformation}’ \rightarrow \text{SKIP} \\
\quad \square \text{invalidInformation}?\text{msg} \rightarrow \text{invalidData}’!\text{msg} \rightarrow \text{SKIP}
\]

The CSC recursively chooses between CSC.sync and CSC_com. It continuously offers, as illustrated in Fig. 2, the possibility of synchronisation between CLIENT and SERVER, and between SERVER and the environment. The corresponding CSP process that represents CSC in our example is presented below:

\[
\text{CSC}(\text{CLIENT, SERVER}) = (\text{CSC}_\text{com}(\text{CLIENT’}, \text{SERVER}) \\
\quad \square \text{CSC}_\text{sync}(\text{SERVER}) \parallel \text{CSC}(\text{CLIENT, SERVER})
\]

The general structure of the composition is the parallel combination (\( \parallel \)) of CLIENT, SERVER and CSC(CLIENT, SERVER), synchronising on events within the hot spots HS_F and within HS_S: the frameworks have to synchronise on the hot spot events involved in the communication but may execute all other events independently. Besides that, the communication hot spots are confined after the composition, using the hiding operator (\( \backslash \)).

**Definition 22.** Let \( F_1 \) and \( F_2 \) be two frameworks that require and provide services through the hot spots HS_F and HS_S, respectively. Then the Communication and Synchronisation Component (CSC) that coordinates their integration, and their composition using CSC are given by the processes CSC(F_1, F_2) and CSC(F_1, F_2), respectively:

\[
\text{C}(F_1, F_2) = (F_1 \parallel_{\text{HS}_F} \text{BackEnd}(F_1, F_2)) \setminus \text{HS}_F
\]

where

- \( \text{BackEnd}(F_1, F_2) = (\text{CSC}(F_1, F_2) \parallel_{\text{HS}_S} \text{F}(R)) \setminus \text{HS}_S' \)
- \( R \) is an injective renaming function, \( R(a) = a \) if \( a \notin \text{HS}_S, R(a) \in \text{HS}_S' \) if \( a \in \text{HS}_S, \) and \( \text{HS}_S' = \{ ev' \mid ev \in \text{HS}_S \} \).
Definition 22 uses a function $R$ to rename events in $HS_{F2}$ to events in $HS'_{F2}$. The objective behind renaming these events ($F_2[R]$) and then hiding ($\ldots \setminus HS'_{F2}$) them is to avoid external interferences. All events within $HS_{F2}$ are made available by $BackEnd$ after being mapped by $CSC\_com$ and $CSC\_sync$. In our example, this allows us to distinguish between an interaction of $SERVER$ with the environment from an interaction of $SERVER$ with $CLIENT$, since each external interaction uses a different alphabet, which is also different from the events used by $SERVER[R]$ (renamed process). We replace $SERVER$ by $SERVER[R]$, where $R$, according to Definition 22, maps each event $ev$ of the hot spot $HS_{SV}$ into an internal event $ev'$; for instance, $receiveRequest$ becomes $receiveRequest'$. The expansion of $SERVER[R]$ results in the process $SERVER'$.

$$
SERVER'_{COM} = 
\begin{align*}
receiveRequest'\?req & \rightarrow \text{processRequest}.req \rightarrow \\
\neg & \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \n
\end{align*}
$$

Since the interaction of the environment with $SERVER$ has a synchronisation set $HS_{SV}$, we preserve this communication in $CSC\_sync(\SERVER)$. On the other hand, the interaction between $CLIENT$ and $SERVER$ is intermediated by the process $CSC\_com(CLIENT', SERVER)$, which involves a mapping between events in both frameworks, so that they can communicate anonymously. Therefore, we can easily instantiate $CSC\_com$ in such a way that $CLIENT$ events are now mapped into the renamed events. In practice, $CSC$ offers the same services to the environment and to $CLIENT$, but using different alphabets. This allows us to reason independently about each interaction. The framework composition in our example then becomes:

$$
E(CLIENT', SERVER') = (CLIENT'\parallel CSC\_FrontEnd)␦S_{CL}\setminus HS_{CL}
$$

where,

- $BackEnd_{CLSV} = (CSC(CLI\_NT, SERVER)\parallel HS_{SV}, SERVER')\setminus HS'_{SV}$
- $HotSpot_{SV}' = \{[\neg validateInformation', \neg invalidInformation', \neg validInformation', \neg receiveRequest', \neg sendResponse']\}$

With this characterisation of framework composition, we can exemplify preservation of conformance notions in a modular way.

7. Conformance notions on compositions

In this section we lift the notions for components in Section 4 to characterise conformance notions for component compositions. These conformance notions are the guidelines that help us to verify whether the integration solution preserves certain behaviours of individual components. Even better, we can use these guidelines to design abstract coordinators that satisfy domain and architectural properties (defined in the component model of Section 4) by construction, independently from the components assembled to them.

We present three conformance notions for component compositions: Composition Compatibility, Service Conformance and Substitutability with Sharing of Services. The first two notions capture general properties in compositions, and can be applied to all integration solution categories presented in Section 5, while the last one is specific for controllers that manage shared resource. The first notion checks the compatibility of the design entities used in the composition. The second notion considers the composition from outside, allowing one to check the compatibility of services not involved in the composition with the environment. The last notion captures the resource sharing of services with the composition and with the environment, verifying its reliability under the sharing of its services. Altogether, these notions allow one to show whether the composition preserves the original component services to the environment. This is especially useful in a system evolution scenario, as we presented in Section 2.

7.1. Composition compatibility

The notion of compatibility on compositions is based on the concept of communication compatibility of components, presented in Section 4.3. It states that the components to be composed must be compatible in order to avoid deadlock. As a consequence, if the components to be composed are deadlock-free, then the composition is deadlock-free as well.

In compositions that have a format similar to the one presented in Fig. 7, we need to check the compatibilities of the component $A$ with $BackEnd$ and of $B$ with $FrontEnd$. In fact, a simpler way would be just verifying the compatibility of the glue code (coordinator) with both components. As coordinators are usually abstractly defined at a design level, we need to restrict their communications to the one used by the protocols of the components under consideration ($A$ and $B$).
Consider a glue code (for instance CSC(F₁, F₂)) related to the communication through interfaces I₁ and I₂ of two components C₁ and C₂, respectively. We intuitively say that the coordinator safely maps service interactions of C₁ into C₂, if service interaction patterns of C₁ are mapped into valid interaction patterns provided by C₂. This notion is captured formally by guaranteeing that the control flow and data mapping between the two components C₁ and C₂ related by the glue code are compatible with the protocols of both components.

**Definition 23.** Let C₁ and C₂ be two components with contracts Ctr₁ and Ctr₂, I₁ and I₂ be interfaces, and GLUE_com(C₁, C₂) be a communication component which maps sequences of events from I₁ into I₂, such that I₁ ∈ Ctr₁.interfaces and I₂ ∈ Ctr₂.interfaces. Then GLUE_com(C₁, C₂) is compatible with C₁ and C₂ if it is compatible with C₁ and compatible with C₂.

Note that we do not need to consider the entire components to check whether the mapping of services is consistent. Only the projections of these components’ behaviours (protocols) in the interfaces involved in the composition are used. To perform the test characterisation of Definition 23 for component compositions we apply Theorem 21 twice. Each time to check the compatibility of the coordinator with one of the components it communicates.

According to Theorem 21, the dual protocol of each interface must be refined by the synchronisation of the coordinator (CSC in our example) with two communication context processes of these interfaces. So, the synchronisation of these processes is only successfully performed (deadlock-free) if the coordinator safely maps the entire communication between the two components. In our example, the test would be characterised in the following way:

\[
\begin{align*}
\text{DUAL}_{\text{PCL}} & \subseteq \text{NAMESPACE} \quad \text{GLUE}_{\text{CTX}} \cdot | \alpha_{\text{HS}_{\text{CL}}} \\
\text{DUAL}_{\text{PSV}}[\text{R}] & \subseteq \text{NAMESPACE} \quad \text{GLUE}_{\text{CTX}} \cdot | \alpha_{\text{HS}_{\text{SV}}}
\end{align*}
\]

where
- \( \text{GLUE}_{\text{CTX}} = \text{CSC}_\text{com}(\text{CLIENT' }, \text{SERVER}) \parallel |_{\alpha_{\text{HS}_{\text{CL}}} \cup \alpha_{\text{HS}_{\text{SV}}}} \text{CTX}_{\text{CLSV}} \)
- \( \text{CTX}_{\text{CLSV}} = \text{CTX}_{\text{CL}} \parallel | \text{CTX}_{\text{SV}} \)
- \( \text{R} \) is an injective renaming function that satisfies \( \text{R}(a) = a \text{ if } a \notin \text{HS}_{\text{SV}} \).

As expected, we have verified that CSC_com safely maps service interaction patterns of CLIENT’ into SERVER, by checking that the process resulted from Definition 23 is deadlock-free, using FDR.

### 7.2. Service conformance

Service conformance is relatively simple to capture. A composition conforms to the services of the original components if services not directly involved in the composition are preserved. We capture this notion with a refinement expression which requires that the observed behaviour of the composed components refines the behaviour of the original components, hiding all interfaces involved in the composition.

**Definition 24.** Let C₁ and C₂ be two component with contracts Ctr₁ and Ctr₂ that require and provide services through the interfaces I₁ and I₂, respectively, and \( \mathcal{P}(C₁, C₂) \) a composition that integrates these two components via I₁ and I₂. Then \( \mathcal{P}(C₁, C₂) \) conforms to the services of C₁ and C₂ if, and only if:

\[
\begin{align*}
\text{Ctr₁}.\text{behaviour} & \setminus \alpha₁ \sqsubseteq_{\text{int}} \mathcal{P}(C₁, C₂) \setminus (\alpha₁ \cup \alpha₂) \\
\text{Ctr₂}.\text{behaviour} & \setminus \alpha₂ \sqsubseteq_{\text{int}} \mathcal{P}(C₁, C₂) \setminus (\alpha₁ \cup \alpha₂)
\end{align*}
\]

As the events of C₁ and C₂ not included in the interfaces I₁ and I₂ do not participate in the framework integration, the conformance notion of these services can be directly checked with the full behaviour that remains externally observed in the composition. On the other hand, the conformance related to the services of I₁ and I₂ have to be checked independently. This is correlated to the conformance notion in the last section. This is because the coordinator might change the way the services in these interfaces are observed.

In our example, the elements C₁, C₂, I₁ and I₂ in Definition 24 can be replaced by CLIENT’, SERVER, HS_{CL} and HS_{SV}, respectively. The composition is represented by \( \mathcal{C}(\text{CLIENT’ }, \text{SERVER}) \).

\[
\begin{align*}
\text{CLIENT’} & \setminus \alpha_{\text{HS}_{\text{CL}}} \sqsubseteq_{\text{int}} \mathcal{C}(\text{CLIENT’ }, \text{SERVER}) \setminus (\alpha_{\text{HS}_{\text{CL}}} \cup \alpha_{\text{HS}_{\text{SV}}}) \\
\text{SERVER} & \setminus \alpha_{\text{HS}_{\text{SV}}} \sqsubseteq_{\text{int}} \mathcal{C}(\text{CLIENT’ }, \text{SERVER}) \setminus (\alpha_{\text{HS}_{\text{CL}}} \cup \alpha_{\text{HS}_{\text{SV}}})
\end{align*}
\]

As expected, the refinements in the definition applied to our example hold, as mechanically checked using FDR.
7.3. Substitutability with sharing of services

The assembly of the server component with the coordinator (BackEnd) (see Fig. 6) must not offer services when the component is not ready to, neither must offer any service that is ready to be offered. The notion guarantees that the interaction of the server with the client component or with the environment does not restrict certain states of the server component. As we mentioned in Section 3, no other communication can initiate while an interaction pattern is being performed; in fact, any user that tries to engage in a new communication is blocked until the end of the current one. After the communication ends, the shared (server) component returns to a state where any user (either the environment or the client component) can start a communication.

To check whether a coordinator (CSC in our example) always makes the services of the server component available to the client component and to the environment, we make use of the interaction subtyping relation (Definition 11). This definition establishes when a new process should be used wherever an original process was expected, without any existing client that shares its service being able to tell the difference. This definition is especially useful when the new process extends the services of the original process considering multiple clients, new and existing ones. Abstractly, the idea of the definition is that all new services, which are offered to other users, only lead to states that are reachable with a sequence of original services. Moreover, the communication with new clients is isolated from communication with existing clients that explain them. This definition closely matches the idea of Liskov and Wing’s extension maps [39], which requires that any new service has to be explained in terms of the original services.

Definition 25. Let \( C_1 \) be a component with contract \( \text{Ctr}_1 \) that requires and provides services through the interface \( I_1 \), Manager a component which encloses \( C_1 \) and manages the sharing of services of \( I_1 \) between \( I_1 \) and an interface \( I_2 \), and \( F_{I_1 \rightarrow I_2} \) a substitution function. Then Manager safely manages the service sharing of \( C_1 \) if, and only if:

\[
\text{Ctr}_1.\text{behaviour} \sqsubseteq_{\text{int}} \text{Manager}
\]

Looking at the process CSC, we observe that the new services are those provided to the client framework, while the original services are those originally provided to the environment. In our example, the former services are contained in \( H_{\text{CSL}} \), and the latter in \( H_{\text{SV}} \). The mapping from the new services into the original services are, therefore, expressed by the mapping that parametrises the process \( \text{CSC}_{\text{com}} \) (see Section 6). In what follows we capture the conformance notion with the process BackEnd (see Definition 22), which synchronises CSC and the server framework.

\[
\text{SERVER} \sqsubseteq_{\text{init}} (\text{BackEnd}_{\text{clsv}} \parallel H_{\text{CSL}} \parallel H_{\text{SV}} \parallel \text{CTX}_{\text{clsv}})
\]

where \( \text{CTX}_{\text{clsv}} = \text{CTX}_{\text{CL}} \parallel \text{CTX}_{\text{SV}} \).

In the test characterisation above, the communication context of CSC with CLIENT and with the environment is expressed by the synchronisation of the process \( \text{BackEnd}_{\text{clsv}} \) with a process that represents the communication context of CLIENT and of the environment (\( \text{CTX}_{\text{clsv}} \)). In this way, it is possible to show that when the CSC engages in a communication, which respects the associated protocols, it can provide services to the environment and to CLIENT appropriately. The mappings performed by \( \text{CSC}_{\text{com}} \) simply replace an interaction pattern with another, which is semantically equivalent, behaving as the implementation of the substitution function that explains the interaction subtyping. We have checked that the assembly of the server framework with the glue code (\( \text{BackEnd}_{\text{clsv}} \)) obeys Definition 25, using FDR.

8. Related work

The Wright [16,3] and rCOS [34,46,47] ADLs present sound component models based on the CSP notation. They propose distinct solutions for coordination [46,47,3]. In both [16,3] and in [34], concrete connector implementations have a representation similar to components. However, at the design level, in [3] coordinators are represented as parametrised CSP processes, connector wrapper templates, and in [46] coordination patterns are represented by global constraints in the software architecture. An interesting strategy is given in [47] to verify the relation of a coordinator purpose and its implementation in a programming language. These works are complementary to our approach, since we represent the external dynamic behaviour of components in a similar way, in terms of CSP processes. However, we make a clear distinction of the points of interaction of a component (via interfaces and protocols). Although Wright represents interfaces via ports, no rule is given for checking protocol compatibility in component compositions; in rCOS, protocols are used to represent the entire external behaviour of a component, without distinguishing points of interaction. Moreover, none of them ensures predictability by restricting component types in their component models, neither addresses formal verification of coordination pattern solutions, as we have presented with the CSC.

There are several verification efforts in the CSP community [34,40] related to updates and reconfigurations of component architectures. Similar to our work, rCOS [34] differentiates input and output events in order to give a proper definition of communication compatibility. However, the component communication mechanism model in rCOS is specific for method invocation [36]. The work reported in [40] presents general substitutability notions for CSP processes, but without checking the covariance of input events and the contravariance of output events [39]. The substitutability notions presented in [40] are related to the interaction subtype notion we propose. We consider these notions often too strong (optimal and safe subtype) or too weak (weak subtype) covering practical application. We believe that our substitutability notion for interaction component
covers a wider range of practical applications, satisfying the same properties as optimal subtype do (when the environment obeys the assumptions we made in Section 4). Abstractly, optimal subtype only permits cyclic transitions triggered by new events in a labelled transition system. We allow with the interaction subtype notion the introduction of entire interaction patterns, without the environment noticing any difference. This is observed even when the component services are shared by several other components, since we precisely define how the environment communicates with the component (partitioning the environment and ensuring compatibility in the communication). Our notion also extends safe subtype by considering substitution functions that explain new sequences of events from existing sequences of events.

Other approaches for composition [10,9] tend only to systematise the integration or reuse of components, lacking either formality or considering the complex mismatches in the framework behaviour. In Section 6, we present an integration pattern solution (represented by CSC), which considers the integration of control-flow and state transitions, and is supported by our conformance notions introduced in Section 7. While the connector matching in [10] always assumes one-to-one mappings, we address many-to-many relationships between events.

Some other interesting approaches related to our efforts should be mentioned. These works [18,11] provide formal methodologies and tools to synthesise suitable coordinators for solving behavioural mismatches between heterogeneous interaction components. In [18], service mappings are extracted from MSC specifications of the black-box components in the system. In [11], service mappings are extracted from protocol definitions with different access rights and their respective available services. These strategies are similar to our previous work [27], which also considers many-to-many mappings. Conformance notions are closer to measures for component adaptation [48]. These measures evaluate to which extent an adapter satisfies the initially requested integration. Our conformance notions would be interpreted as strict measures about behaviour preservation in these integrations.

In our work, all verifications and notions analyse partitions of the component (and composition) behaviour in space (protocols) and time (interaction patterns). This approach combines the advantages of the approaches presented in [12] and in [37], where physical and temporal partitions are realised, respectively. Protocols are observed as a particular type in [12], which permits the verification of compatibility and substitutability. However, concerns about the entire component behaviour are ignored in the definitions of [12]. Interaction patterns are also defined in [37], however without defining any conformance notion for components or compositions. None of these works defines test characterisations that can mechanically be performed in verification tools.

9. Conclusion

Although component-based software development provides mechanisms and tools for constructing systems by plugging components together, the safe construction of these systems is still a research challenge. Conformance notions are required during several development activities, such as safe composition of third-party components, correct adaptation of library components and the consistent update of versioned components in component-based architectures. Most of these activities involve coordination patterns, which are used to integrate components or adapt them. To increase the range of components they integrate, coordinators are abstractly defined at the design level, usually in a parametrised form. Therefore, the conformance notions for compositions must include properties of coordinators in order to achieve an effective approach to verification by construction.

In this work, we have defined a component model for interaction components, with some basic conformance notions that apply to components in isolation. We have then lifted these notions to characterise three conformance notions for component compositions: composition compatibility, service conformance, and substitutability with sharing of services.

We show that some of these notions can be applied to coordination patterns from their abstract specification to a more concrete representation, after components are assembled. To deal with the abstract specifications of coordinators, we use an important definition of communication context, which is incorporated in testing characterisations of our notions. The first two conformance notions are general and can be applied to any coordinator pattern. Indeed they cover the main aspects of patterns classified as Extender or Translator [5], or combinations of them. Composition compatibility is a new conformance notion that generalises the idea of compatibility of transactions presented in [13]. As a particular case of behaviour preservation, these patterns together also guarantee preservation of deadlock-freedom in the compositions that use them. The third conformance notion is exclusive to a subset of control coordinators [5], called resource sharing managers. Using the three notions, we are able to verify whether a coordinator can safely manage the sharing of services of a server component (originally designed to provide these services to a unique invoker) without collapsing the component. Together, they guarantee a consistent extension of the number of heterogeneous clients of the server component, without the environment noticing any difference. Moreover, they allow the verification of well-formedness of the coordinator that integrates all these communications.

We defined these notions in terms of the denotational semantics of CSP. In particular, we have adopted the stable failures model which allows us to avoid undesirable divergences resulting from hiding events. From the denotational definitions, we have derived some test characterisations that allow us to verify the relevant properties mechanically, using FDR. All verifications and notions analyse partitions of each component (and composition) behaviour in space (protocols) and time (interaction patterns), which might dramatically reduce the cost of the verifications in practical systems.

The application of these notions has been illustrated with a systematic strategy for formal composition of frameworks [27] based on the process algebra CSP. An interesting result of our effort to define conformance notions is
that this has revealed the need for modifying the definition of composition originally proposed in [27]. The modification (based on renaming internal events of the server) allowed us to distinguish the origin of different client interactions with the server framework and permitted the control of possible interferences.

As future work we plan to propose composition rules which ensure, by construction, that coordination patterns preserve such properties. For example, in the case of the CSC, a desirable proposition would state that our composition strategy ensures conformance with respect to the original frameworks, checking each conformance notion for compositions defined in this paper. We also plan to exercise other component patterns of integration, as well as working on mechanical support for component composition. Finally, we aim to investigate other types of control coordinators, and additional relevant properties for component composition.

References


