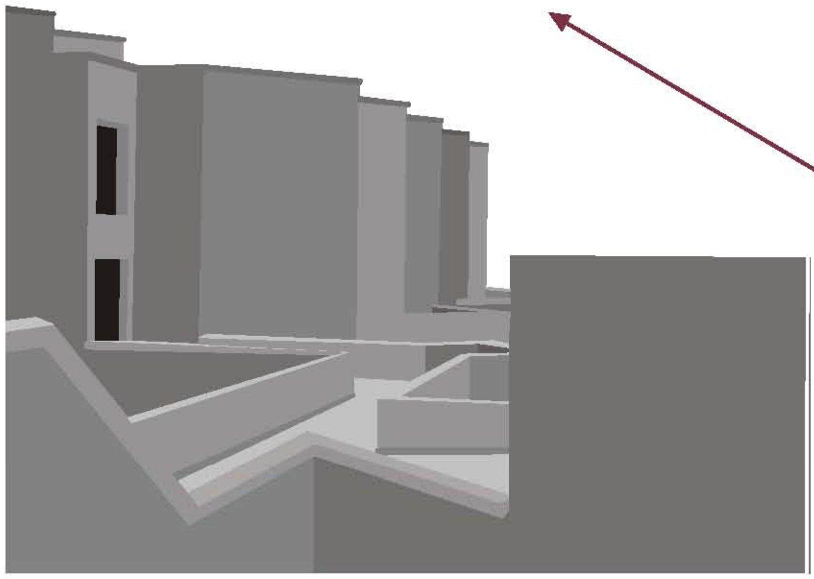

FUNCTIONAL SPECIFICATION OF COMPOSITE COMPONENTS

Perla Velasco Elizondo & Mbe K. Christophe Ndjatchi

Comunicación del CIMAT No I-10-05/24-06-2010
(CC/CIMAT)

Functional Specification of Composite Components

Perla Velasco Elizondo and Mbe K. Christophe Ndjatchi

Abstract.

The availability of the specification of a component is fundamental to its successful use during system development. Unfortunately, current practice of constructing new specifications for composite components does not give much weight to doing so systematically and consistently considering both their constituents' specifications and the semantics of their composition. This technical report presents our progress on developing an approach to derive interface specifications for composite components taking into consideration the former aspects. We focus on deriving the specification of functional properties. Our composites are constructed within the context of a new component model where first-class connectors are utilised. The presented approach is based on a set of connector-specific functions, which allow deriving functional specifications in a systematic and consistent manner.

Centre for Mathematical Research
Software Engineering Group
Jalisco S/N, Colonia Valenciana
36240, Guanajuato, Gto., México

Telephone + 52 473 732 7155 / 735 0800
Fax +52 473 732 5749

Contents

List of Figures

Chapter 1

Introduction

Component-based Development (CBD), is a paradigm in Software Engineering based on the idea of building systems by composing *pre-existing* software components instead of starting from scratch. The idea of constructing *composite components* has been recognised as a good practice in CBD because it is an alternative not only to abstract complex behaviour but also to maximise reuse.

A component's specification defines the component and serves as the sole medium for the component's understanding and use. Therefore, to enable its reuse the availability of a composite component's specification is very important. In this chapter we discuss this issue and how it is supported in CBD approaches that allow the generation of composite components. We will focus on the specification of functional properties and the limitations found in this context.

1.1 Specification of Software Components

As stated before, a *component's specification* defines the component and serves as the sole medium for the component's understanding and use by answering questions such as: What services are offered and requested by the component? How can these services be used? What quality characteristics do the offered services fulfill? And so on. There is a common agreement on the information elements that an atomic component specification should include [10, 30, 13, 2, 12]:

(1) The *instantiation* mechanisms.

(2) The *functional* properties.

(3) The *non-functional* properties.

(4) The information about the *deployment environment*.

Independently of the form that these information elements take, all of them are crucial for defining, validating and setting up a component composition.

The idea of constructing composite components has been recognised as good practice in CBD because it is manner not only to abstract complex behaviour and complex structure but also to maximise reuse [16]. When we talk about composite components we refer to general purpose reusable components made up of an assembly of two or more atomic components. In general, the issue of being a composite should be transparent for a component user as a composite component should be utilised in the same manner as an atomic one.

Given this context, the availability of the specification of a composite component is very important. In an ideal scenario, the information of a composite's specification should be systematically and consistently generated during the component composition process based on the specifications of its constituent components and the semantics of their composition. By systematically and consistently we mean enabling the reuse of a well-defined composition process during different developments. Unfortunately, current practice of constructing new specifications for composite components does not give much weight to doing so in this manner. In the following section we discuss this issue.

1.2 Composite Components in CBD Approaches

As introduced before, the idea of constructing composite components has been recognised as a good practice in CBD. However, producing reusable composite components is not a trivial task. We believe that a fundamental issue to facilitate composite component generation in CBD is the availability of a *constructive approach to composition*. That is, an approach that when taking two or more components and putting them together in some way, it results in a new entity that preserves some of the properties of its constituents. The lack of such an approach may be the reason why, although component composition is supported, only in some CBD methods component composition enables the construction of reusable composite components [20].

To illustrate the former, consider component composition in JavaBeans [22]. In JavaBeans *beans* (i.e. atomic components) are Java classes which adhere to a set of design and syntactic conventions. These conventions make their storage and retrieval possible as well as their automatic composition via a visual development environment (i.e. the BeanBox). When beans are composed, the resulting "composition" takes the form of an *adaptor class*.[1] However, this class does not preserve the properties of the bean class. For example, it does not correspond to an entity that can be specified in terms of its properties, its events and its methods as it can be done for its constituent beans. As a corollary, the adaptor class cannot be stored, retrieved and (re)used as a bean class can.

There are CBD approaches such as Koala [23], PECOS [33] and UML 2 [1] that are closer to our notion of composite components. In these approaches the composite components are specified and composed in the same manner as atomic components. However, there are some issues that make it difficult to systematically and consistently specify them.

Koala is a CBD approach used for the construction of softwares in consumer electronics. Koala supports the construction of composite components from pre-existing component's specifications. In this model, specifications are written in the form of *interfaces* utilising some sort of Interface Definition Language (IDL). Interfaces can be stored in a repository and later on retrieved and reused for composite component definition. These definitions can be compiled to get composites' implementations in a specific programming

[1]In JavaBeans, an adaptor class is wrapper class utilised to wire the composed components.

language. The interfaces in Koala support the specification of various elements relevant to a component. In this example we focus on the provided and required services (i.e. the functional properties) as they are the target of the work presented in this paper. Fig. 1.1 shows an outline of the code for (a) an *interface* and (b) a *component* specification in Koala. As can be seen, (a) defines a set of semantically related services in the form of functions, while (b) defines the actual functionality that a particular component type (i.e. `CTunerDriver`) offers and requires in terms of a set of pre-existing interfaces (e.g. `ITuner`).

(a)
```
interface ITuner{
  void SetFrequency(int f);
  int GetFrequency(void);
}
```

(b)
```
component CTunerDriver{
  provides ITuner ptun;
           IInit pini;
  requires II2c ri2c;
}
```

Figure 1.1: (a) An interface and (b) a component specification in Koala.

Fig. 1.2 shows an outline of the code of the specification of the composite component `CTVPlatform`, as well as its graphical representation in Koala notation. As can be seen, this composite component is defined as an assembly of the components `CFrontEnd`, `CTunerDriver` and `CMemDriver`.

(a)
```
component CTVPlatform{
  provides IProgram pini;
  provides IMem pos;
  requires II2c fast;
  contains
    component CFrontEnd cfre;
    component CTunerDriver ctun;
    component CMemDriver cmem;
  connects
    pini = cfre.pini;
    pos = cmem.pos;
    cfre.rtun = ctun.ptun;
    cmem.rif = ctun.pini;
    ctun.ri2c = fast;
}
```

(b)

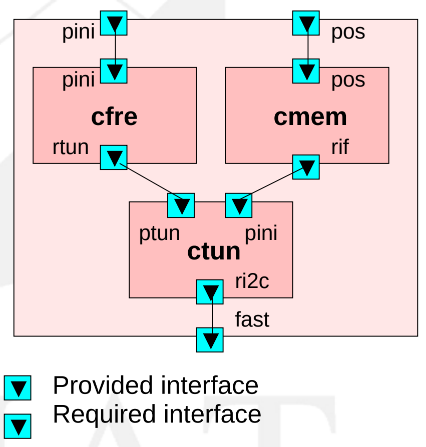

Figure 1.2: (a) A composite component specification and (b) its graphical representation in Koala.

We already mentioned the idea of systematically and consistently generating composites' functional specifications from the information in the functional specifications of their constituent components as well as the semantics of their composition. We consider that it is not achieved in Koala at all. The composite `CTVPlatform` is specified in the same manner in which its constituents are (i.e. it defines the functionality that it offers and requires in terms of a set of pre-existing interfaces). However, the exposed interfaces `IProgram`, `IMem` and `II2c` result from *manually* forwarding them from the inner components to the enclosing one according to the system developer's needs rather than from systematically deriving them based on the semantics of the components' composition.

This manner of forwarding interfaces follows the semantics of delegation connectors in UML 2 [1].[2]

We also observe that, because of the manner in which they are generated, the possibility of reusing these composites in a different development is limited. Highly-reusable composites should be useful to build a number of different systems within an application domain. An alternative to generate a highly-reusable composite is that of providing a mean to invoke a number of valid sequences of operations offered by its constituents [16]. By adopting a composition approach as the one depicted in Fig. 1.2, not all the constituents' services are available to invoke in the resulting composite if the constituents' interfaces have not been forwarded, e.g. the `ptun` interface. Although it is useful for the construction of certain types of composites, this *ad hoc* manner of hiding and exposing the constituents' interfaces could represent a shortcoming for the construction of some others. Note that by forwarding all the constituents' interfaces, to make their services available to invoke, one could violate the composition semantics as it could enable invoking invalid sequences of services.

This process to composite component generation and specification is also adopted by PECOS [33] and UML 2 [1]. Therefore, they present the same shortcomings discussed in this section.

1.3 Summary

The idea of constructing composite components has been recognised as good practice in CBD because it is an alternative not only to abstract complex behaviour and complex structure but also to maximise reuse. Component specifications play an important role when composing software components as the information they contain is used to define, validate and set up a component composition within a specific context. Therefore, to facilitate their reuse the availability of a composite component specification is also very important.

We have already mentioned that the functional specification of a composite component should be directly derivable from the functional specifications of its constituents. While it is in some degree achievable in some component models, we believe that it is not addressed in a systematic and consistent manner.

Despite the fact that composite components generated via approaches such as Koala, PECOS and UML 2 preserve the same shape than its constituents, the approach to composition is not systematic and requires a lot of human intervention to decide which interfaces must be forwarded. Besides that, the generated composites are not enough generic to allow the execution of any valid execution sequence involving the operations in their constituents.

In the following chapter we introduce how a new approach to composition, based of first-class connectors, can help not only to improve the reusability of the generated composite components but also to develop and specify them in a systematic and consistent

[2]In UML 2, a delegation connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behaviour by the component's parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port that will be passed on to that target for handling.

manner.

Chapter 2

A Set of Connectors for Composing Software Components

As mentioned previously, the concept of composition encapsulates the notion of taking two or more components and putting them together in some way. Composition should be constructive. That is, it should result in a new entity that has some of the properties of its constituents. As discussed in Chapter 1, this view of composition has been neglected in current CBD approaches. In this chapter we discuss how our connectors fill this gap and as a corollary, help to make more systematic and consistent the issue of composite components specification.

2.1 A New View of Connectors

The lack of a constructive approach to component composition makes it difficult the generation of reusable composite components. We have demonstrated that a new notion of connectors can tackle this problem [32, 17, 16, 19]. This new notion has been described in a new component model [18]. In this model *components* are passive, general-purpose and do not request services from other components. Instead, components perform their provided services only when invoked by *connectors*. Connectors encapsulate communication schemes; many of them are analogous to well-known patterns [24, 8]. The components have an *interface specification* and an *implementation*. The interface describes the component's *provided services* (i.e. the functional properties) in terms of a name, the types of its input parameters and the types of its output parameters. Additionally, this interface describes the *non-functional* properties and the *deployment context dependencies* related to these services. The implementation corresponds to the services of the component coded in a programming language.

Fig. 2.1 shows a system architecture in the new component model. It consists of a hierarchy of connectors (`K1`−`K5`) representing the system's communication and coordination, sitting on top of components (`C1`−`C6`) that provide the services performed by the system. As can be seen, any connector works as a *constructive composition operator*. That is, when applied to components it yields another component so that the resulting component can in turn be a subject of further composition (see the inner dotted boxes representing composite components in Fig. 2.1).

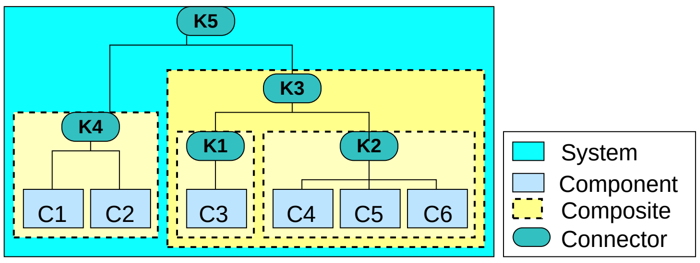

Figure 2.1: A system architecture in the new component model.

In [32] a catalogue of connectors to allow component composition according to this approach has been presented. Although these connectors are meant to be used to construct component-based systems (i.e. development with reuse), they also allow the generation of composite components (i.e. development for reuse). When using these connectors to support development for reuse it is possible to generate composite components which are sufficiently general to cover the different aspects of their use. For example, consider the situation in which the atomic components C1 and C2 in Fig. 2.1 offer a set of services common for the construction of sales applications, e.g. billing and shipping services. It is sensible and convenient to compose these components together in a composite component as it is illustrated in Fig. 2.1. This composite can be reused for the construction of different systems in the same domain. It offers a means to invoke any of the operations in its constituents, as well as a means to specify the deployment information of the particular system under construction, e.g. the location of the clients and shipping fares databases.

In this report, we do not explain all the connectors in our catalogue. Detailed descriptions of these connectors and their implementation can be found in [32]. However, for clarification purposes, we describe the connectors we will use for the construction of the composites in the case studies in Chapter 4. The descriptions are in terms of the notation on the right-hand side of Fig. 2.2. The dotted boxes represent the resulting assemblies. The boxes with the computation label represent the computation in the composed/adapted component. An edge connecting two diagram elements represents the control flow along the direction of the edge. Any data required in the assembly to perform the corresponding computation is denoted as the *input* label. The result generated by the assembly execution is denoted as the *output* label.

The table on the left-hand side of Fig. 2.2 lists the connectors in the catalogue. They are organised in (i) *adaptation connectors*, (ii) *basic composition connectors* and (iii) *composite composition connectors*. Adaptation connectors are *unary* connectors which adapt the component in the sense that before any computation takes place inside the component, the execution of the control scheme encapsulated by the connector is executed first. Basic composition connectors are n-*ary* connectors used to support component composition. We call them "basic" to differentiate them from composite composition connectors, which are complex communication schemes. While basic connectors provide only one type of communication scheme, composite connectors combine many types.

Both the *Sequencer* and *Pipe* composition connectors can be used to compose a set of two or more components, so that the execution of each one of them is carried out in a *sequential* order, see Fig. 2.2 (a) and (b). In the case of the *Pipe* connector, it also models

Category	Type
Adaptation Connector	Guard
	Condition−controlled Loop
	Counter−controlled Loop
	Delay
Basic Composition Connector	Sequencer
	Pipe
	Selector
Composite Composition Connector	Observer
	Chain of Responsibility
	Exclusive Choice Sequencer
	Exclusive Choice Pipe
	Simple Merge Sequencer
	Simple Merge Pipe

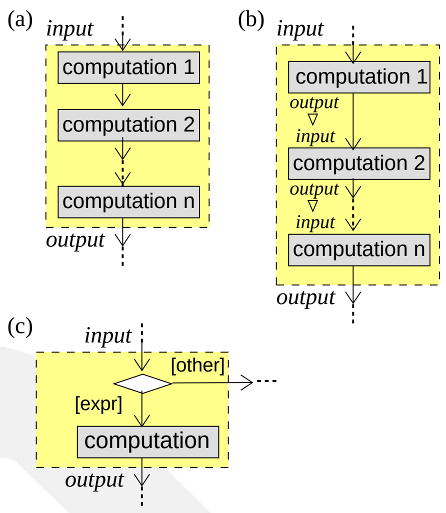

Figure 2.2: The elements in the connector's catalogue and the behaviour of the assemblies resulting from the (a) *Sequencer*, (b) *Pipe* and (c) *Guard* connectors.

internal data communication among the composed units, so that the *output* generated by a component's execution becomes the *input* to the next one in the chain. Fig. 2.2 (c) depicts the case of the *Guard* adaptation connector. Any computation in the adapted component is conditional upon the value of a Boolean expression (expr) being *true*.

2.2 Summary

In this chapter we described a constructive approach to composition, which enables the generation of composite components. This approach is described in a new component model [18]. In this model *components* are passive, general-purpose and do not request services from other components. Instead, components perform their provided services only when invoked by *connectors*. Connectors encapsulate communication schemes; many of them are analogous to well-known patterns [24, 8].

The composites generated via this approach are worth of consideration as they are systematically and consistently constructed and have bigger possibilities of reuse than a composite generated via approaches as the ones presented in Chapter 1. This claim is supported by the notion that we can generate composites offering a mean to invoke a wider combination of the operations in their constituents according to a fixed communication and coordination scheme.

Now that we have defined the generalities of our component composition approach, in the following sections we focus on presenting a set of connector-specific functions to derive the functional specifications of composites components in a systematic and consistent manner.

Chapter 3

Functional Specification of Composite Components

We have outlined a new view of composition where first-class connectors are utilised as operators for composing software components. In general, if a set of atomic components are composed by using our approach, then the user of the resulting composite should be able to execute a number of operation sequences in terms of the operations offered by the atomic components. The nature and number of all possible sequences should be determined from both the functional specifications of the atomic components and the semantics of the connector utilised.

In order to facilitate its reuse, the resulting composite should provide a functional specification informing about these operation sequences. Ideally, the composite's functional specification should be offered in the same form as that of an atomic component. That is, as a set of provided services. Although in this case, these services should be abstractions denoting the operation sequences valid to invoke within the composite's constituents and their corresponding requirements (e.g. their input parameters) and outcomes (e.g. their output parameters).

We have realised that our composition approach makes it easier to develop a set of connector-specific functions to systematically and consistently generate the composites' functional specifications as outlined above. The semantics of these functions is based on simple set operations on the elements of the composites' functional specifications. Next, we introduce such functions.

3.1 Basic Formalism and Assumptions

As stated before, the functional specification of a component informs about the services it provides. In most CBD approaches these services are specified as *operation signatures.* An operation signature is defined by an *operation name* and a number of *parameters.* Each one of these parameters is defined by a *parameter name* and a *parameter type.* We will denote as *Param* the parameters in an operation signature. According to the *role* that a parameter takes, it is possible to partition the elements in *Param* to distinguish among *input* and *output parameters.* Based on the former, we define an operation signature as a tuple

$\langle \mathit{InParam}, \mathit{OutParam} \rangle$

where *InParam* and *OutParam* represent input and output parameters respectively. For simplicity, from now on we will use the following abbreviations to denote an operation signature (*Sig*) and the functional specification (*FSpec*) of a component (i.e. a set of operation signatures):

$\mathit{Sig} == \langle \mathit{InParam}, \mathit{OutParam} \rangle$
$\mathit{FSpec} == \mathbb{P}\, \mathit{Sig}$

Note that in these definitions we do not make the operation name of the signature explicit. However, we assume that each operation signature in a *FSpec* is associated to an operation name which works as an identifier for it. Thus, a functional specification *FSpec* could contain identical tuples $\langle \mathit{InParam}, \mathit{OutParam} \rangle$ as long as these tuples are associated to different operation names. For example, the

credit $\langle \{\mathit{cardId}, \mathit{amount}\}, \{\mathit{errorCode}\} \rangle$ and
debit $\langle \{\mathit{cardId}, \mathit{amount}\}, \{\mathit{errorCode}\} \rangle$.

operation signatures have identical tuples $\langle \mathit{InParam}, \mathit{OutParam} \rangle$ but not operation names.

In this basic formalism we will treat *Param* and its partitions *InParam* and *OutParam* as *bags* instead of sets to allow for duplication of parameters. A composite component could be generated from a set of instances of the same component type. For example, consider the case of composing three instances of a Dispenser Component (e.g. one instance for water, one instance for milk and one instance for coffee) into a Coffee Dispenser composite component. This results in a composition scenario involving a number of functional specifications *FSpec* with the same operation signatures. As we will show later on, when composing these specifications via certain type of connectors, it could be required to have multiple occurrences of the same parameter. Note however that, we assume that operation signatures are well formed. That is, an operation signature $\langle \mathit{InParam}, \mathit{OutParam} \rangle$ cannot have the same parameter both as input and output, i.e. $\mathit{InParam} \cap \mathit{OutParam} = \oslash$. In this work, we also assume that parameters with the same name and type are semantically equivalent.

Considering this basic formalism and assumptions, we have defined a set of helper functions to systematically and consistently generate the functional specifications of composite components

3.1.1 Parameter Complement

The function parameter complement (*param_comp*) maps the elements in a set of parameters to their complementary ones taking into consideration its *role*, i.e. either input or output.

$$\begin{array}{|l} \mathit{param\_comp} : \mathit{Param} \rightarrow \mathit{Param} \\ \hline \mathit{param\_comp} = p_1, p_2, \ldots p_n \in \mathit{Param} \bullet \bigcup_{i=1}^{n} \sim p_i \end{array}$$

3.1.2 Signature Input and Output Parameter

The functions signature input parameter and signature output parameter (*sig_in* and *sig_out* respectively) return the set of input and output parameters of an operation signature respectively.

$$\begin{array}{|l} \mathit{sig\_in}, \mathit{sig\_out} : \mathit{Sig} \rightarrow \mathit{Param} \\ \hline s : \mathit{Sig} \bullet \mathit{sig\_in}(s) = \mathit{InParam} \wedge \mathit{sig\_out}(s) = \mathit{OutParam} \end{array}$$

3.1.3 Signature Match

The function signature (*sig_match*) verifies whether there are common elements among the output parameters of one operation signature and the input parameters of another.

$$\begin{array}{|l} \mathit{sig\_match} : \mathit{Sig} \times \mathit{Sig} \rightarrow \mathit{Boolean} \\ \hline \mathit{sig\_match} = s_1, s_2 : \mathit{Sig} \bullet \mathit{sig\_out}(s_1) \cap \mathit{sig\_in}(s_2) \neq \oslash \end{array}$$

3.1.4 Signature Concatenation

The function signature concatenation (*sig_concat*) works on a set of operation signatures to generate one whose input and output parameters result from the concatenation of the input and output parameters on the participating signatures. Note that to specify the issue of having duplicated elements in *InParam* and *OutParam*, we use the $\uplus$ operator.

$$\begin{array}{|l} \mathit{sig\_concat} : \mathit{Sig} \times \ldots \times \mathit{Sig} \rightarrow \mathit{Sig} \\ \hline \mathit{sig\_concat} = s_1, s_2, \ldots, s_n : \mathit{Sig} \bullet \langle \biguplus_{i=1}^{n} \mathit{sig\_in}(s_i), \biguplus_{i=1}^{n} \mathit{sig\_out}(s_i) \rangle \end{array}$$

3.1.5 Add Input and Output Parameter

functions add input parameter and add output parameter (*add_in* and *add_out* respectively) add input and output parameters to an operation signature respectively.

$$\begin{array}{|l} \mathit{add\_in}, \mathit{add\_out} : \mathit{Param} \times \mathit{Sig} \rightarrow \mathit{Sig} \\ \hline p : \mathit{Param}; \\ s : \mathit{Sig} \bullet \\ \qquad \mathit{add\_in}(p, s) = \langle \{p \cup \mathit{sig\_in}(s)\}, \mathit{sig\_out}(s) \rangle \wedge \\ \qquad \mathit{add\_out}(p, s) = \langle \mathit{sig\_in}(s), \{p \cup \mathit{sig\_out}(s)\} \rangle \end{array}$$

3.1.6 Signature Bound

The function signature bound (*sig_bound*) works on a set of operation signatures and results in one consisting of the union of the given signatures, but with the parameters in the participating signatures that are complementary removed.

$$
\begin{array}{|l}
\mathit{sig\_bound} : \mathit{Sig} \times \ldots \times \mathit{Sig} \rightarrow \mathit{Sig} \\
\hline
\mathit{sig\_bound} = s_1, s_2, \ldots, s_n : \mathit{Sig} \bullet \\
\qquad \langle \{ \mathit{sig\_in}(s_1) \quad \uplus \\
\qquad (\mathit{sig\_in}(s_2) \setminus \mathit{param\_comp}(\mathit{sig\_out}(s_1))) \quad \uplus \\
\qquad (\mathit{sig\_in}(s_3) \setminus \mathit{param\_comp}(\mathit{sig\_out}(s_2))) \quad \uplus \\
\qquad \ldots \quad \uplus \\
\qquad (\mathit{sig\_in}(s_n) \setminus \mathit{param\_comp}(\mathit{sig\_out}(s_{n-1}))) \}, \\
\qquad \mathit{sig\_out}(s_n) \rangle
\end{array}
$$

3.1.7 Signature Union

The function signature union (*sig_union*) works on a set of operation signatures and results in a signature consisting of a set of parameters by keeping only one occurrence of input or output parameters with identical names. As can be seen, we specify it by using the *set union* ($\bigcup$) operator.

$$
\begin{array}{|l}
\mathit{sig\_union} : \mathit{Sig} \times \ldots \times \mathit{Sig} \rightarrow \mathit{Sig} \\
\hline
\mathit{sig\_union} = s_1, s_2, \ldots, s_n : \mathit{Sig} \bullet \langle \bigcup_{i=1}^{n} \mathit{sig\_in}(s_i), \bigcup_{i=1}^{n} \mathit{sig\_out}(s_i) \rangle
\end{array}
$$

Now that we have presented these helper functions, in the following sections we define the connector-specific ones corresponding to each one of the connectors presented in Chapter 2. As we will see, the functional specification of a composite component is estimated based on the information contained the functional specifications of its constituents.

3.2 Functional Specification Functions for Adaptation Connectors

As introduced in Chapter 2, the *Guard* connector is an adaptation connector utilised to guard the execution of an operation in a component upon the value of a Boolean expression being *true*. We also introduced the *Condition* and *Counter Loop* connectors which provide a looping control scheme. In the case of the *Condition-Controlled Loop* connector, the iterative execution of computation in a component is performed until a Boolean expression is not satisfied. In the case of the *Counter-Controlled Loop* the computation is executed repeatedly a fixed number of times.

We have defined the *guard_composite_fspec*, *conditionLoop_composite_fspec* and *counterLoop_composite_fspec* functions to determine the elements in the functional specification of a composite component generated via a *Guard*, *Condition Loop* and *Counter Loop* respectively:

$$\begin{array}{|l}
guard\_composite\_fspec, \\
conditionLoop\_composite\_fspec, \\
counterLoop\_composite\_fspec : InParam \times FSpec \to FSpec \\
\hline
guard\_composite\_fspec, \\
conditionLoop\_composite\_fspec, \\
counterLoop\_composite\_fspec = \\
\qquad\qquad s_1, s_2, \ldots, s_n : Sig; \\
\qquad\qquad f : FSpec; \\
\qquad\qquad p : InParam \mid \#p = 1; \\
\qquad\qquad s_1, s_2, \ldots, s_n \in f \bullet \bigcup_{i=1}^{n} add\_in(p, s_i)
\end{array}$$

Note that besides the functional specification (*FSpec*) of the adapted component, these functions also take one input parameter (InParam). This parameter represents the value to be evaluated by the connectors Boolean expressions. This parameter is added to the input parameters of each one of the operation signatures of the adapted component via the *add_in* helper function.

Now that we have presented the functions corresponding to the adaptation connectors, next we present the ones defined for the composition connectors.

3.3 Functional Specification Functions for Composition Connectors

The *Sequencer* connector is a composition connector utilised to compose a set of two or more components so that the execution of an operation in each one of them is carried out in a sequential order. We have defined the *seq_composite_fspec* function to determine the elements in the functional specification of a composite component generated via a *Sequencer*:

$$\begin{array}{|l}
seq\_composite\_fspec : FSpec \times FSpec \times \ldots \times FSpec \to FSpec \\
\hline
seq\_composite\_fspec = \\
\qquad\qquad s_1, s_2, \ldots, s_n : Sig; \\
\qquad\qquad f_1, f_2, \ldots, f_n : FSpec; \\
\qquad\qquad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \\
\qquad\qquad\qquad \bigcup_{(s_1,\ldots,s_n) \in \prod_{i=1}^{n} f_i} sig\_concat(s_1, s_2, \ldots, s_n)
\end{array}$$

The *Pipe* connector also supports the composition of a set of two or more components so that the execution of an operation in each one of them is carried out in a sequential order. However, in contrast to the *Sequencer*, it also models internal data communication among the composed units, so that the output generated by a component's execution becomes the input to the next one in the chain. Thus, we define the *pipe_composite_fspec* function to the determine the functional specification of the resulting composite when it has been generated via a *Pipe* connector:

$$
\begin{array}{|l}
\mathit{pipe\_composite\_fspec} : \mathit{FSpec} \times \mathit{FSpec} \times \ldots \times \mathit{FSpec} \to \mathit{FSpec} \\
\hline
\mathit{pipe\_composite\_fspec} = \\
\qquad s_1, s_2, \ldots, s_n : \mathit{Sig}; \\
\qquad f_1, f_2, \ldots, f_n : \mathit{FSpec}; \\
\qquad \forall\, s_i \in (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \mid \mathit{sig\_match}(s_i, s_{i+1}) = \mathit{true} \bullet \\
\qquad \displaystyle\bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} \mathit{sig\_bound}(s_1, s_2, \ldots, s_n)
\end{array}
$$

The *Selector* connector is a composition connector utilised to compose a set of two or more components so that the execution an operation in only one of them is carried out based on the evaluation of a Boolean expression. The set of operation signatures in the functional specification of the resulting composite component can be determined by using the *sel_composite_fspec* function which is defined as follows:

$$
\begin{array}{|l}
\mathit{sel\_composite\_fspec} : \mathit{InParam} \times \mathit{FSpec} \times \mathit{FSpec} \times \ldots \times \mathit{FSpec} \to \mathit{FSpec} \\
\hline
\mathit{sel\_composite\_fspec} = \\
\qquad s_1, s_2, \ldots, s_n : \mathit{Sig}; \\
\qquad f_1, f_2, \ldots, f_n : \mathit{FSpec}; \\
\qquad p : \mathit{InParam} \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \\
\qquad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \\
\qquad \displaystyle\bigcup_{i=1}^{n} \mathit{add\_in}(ip_i, s_i) \mid ip_i \in p \wedge s_i \in \bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} \mathit{sig\_union}(s_1, s_2, \ldots, s_n)
\end{array}
$$

As in the *Guard* connector, we have considered a set of input parameters to represent the value to be evaluated in the *Selector* conditions.

As described in Chapter 2, we have composite composition connectors made up from specific arrangements of the *Sequencer*, *Pipe*, *Selector* and *Guard* connectors. Accordingly, we have defined a set of functions in terms of the defined before to determine the information in the functional specification of the resulting component when the *Observer*, *Chain of Responsibility*, *Exclusive Choice Sequencer*, *Exclusive Choice Pipe*, *Simple Merge Sequencer* and *Simple Merge Pipe* connectors are applied. Next we list such functions. The function for the *Observer* connector is defined as follows:

$$
\begin{array}{|l}
\mathit{obs\_composite\_fspec} : \mathit{FSpec} \times \mathit{FSpec} \times \ldots \times \mathit{FSpec} \to \mathit{FSpec} \\
\hline
\mathit{obs\_composite\_fspec} = \\
\qquad f_1, f_2, \ldots, f_n : \mathit{FSpec} \bullet \\
\qquad \mathit{pipe\_composite\_fspec}(f_1, \mathit{seq\_composite\_fspec}(f_2, \ldots, f_n))
\end{array}
$$

The function for the *Chain of Responsibility* connector is defined as follows:

$$cr\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} cr\_composite\_fspec = \\ \qquad f_1, f_2, \ldots, f_n : FSpec; \\ \qquad p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \bullet \\ \qquad seq\_composite\_fspec(sel\_composite\_fspec(p, f_1, \ldots, f_n)) \end{array}$$

The functions for the *Exclusive Choice Sequencer* and the *Exclusive Choice Pipe* connectors are defined as follows:

$$ecSeq\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} ecSeq\_composite\_fspec = \\ \qquad f_1, f_2, \ldots, f_n : FSpec; \\ \qquad p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \bullet \\ \qquad seq\_composite\_fspec(f_1, sel\_composite\_fspec(p, f_2, \ldots, f_n)) \end{array}$$

$$ecPipe\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} ecPipe\_composite\_fspec = \\ \qquad f_1, f_2, \ldots, f_n : FSpec; \\ \qquad p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \bullet \\ \qquad pipe\_composite\_fspec(f_1, sel\_composite\_fspec(p, f_2, \ldots, f_n)) \end{array}$$

Finally, the functions for the *Simple Merge Sequencer* and *Simple Merge Pipe* connectors are defined as follows:

$$smSeq\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} smSeq\_composite\_fspec = \\ \qquad f_1, f_2, \ldots, f_n : FSpec; \\ \qquad p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \bullet \\ \qquad seq\_composite\_fspec(sel\_composite\_fspec(p, f_1, \ldots, f_{n-1}), f_n) \end{array}$$

$$smPipe\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} smPipe\_composite\_fspec = \\ \qquad f_1, f_2, \ldots, f_n : FSpec; \\ \qquad p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \bullet \\ \qquad pipe\_composite\_fspec(sel\_composite\_fspec(p, f_1, \ldots, f_{n-1}), f_n) \end{array}$$

3.4 Summary

In this chapter we presented a set of functions to systematically and consistently generate the functional interface of the composite components generated via our connectors. In general, these functions work on a a set of functional specifications (*FSpec*) which inform about the *services* that the composed components provide. The services of a component are specified as a set of *operation signatures*. An *operation signature* can be specified by a *operation name* as well as a set of *parameters*. Each parameter in the set is defined by a *parameter name*, *parameter type* and a *parameter role*.

In next chapter we present two examples to illustrate the use of these functions.

Chapter 4

Examples

In order to show the usefulness of the functions defined in the previous chapter, in this one we discuss the design of a pair of component based systems by using a set of composite components. The first case is a Drink Selling Machine System and the second one is a Robotics System.

4.1 The Drink Selling Machine System

Consider the case of constructing a set of systems to control a Drink Vending Machine from a set of pre-existing components. To simplify this example, the Drink Vending Machine is limited to two general functions: (1) *to sell a drink* and (2) *to maintain the dispensers*. (1) involves receiving the customer request and payment as well as delivering the drink. (2) involves filling and emptying the dispensers of the drinks' ingredients.

4.1.1 The Proposed Architectures

From the former definition, it seems sensible to organise the system design into three main subsystems: a *Cashier Subsystem*, a *Drink Maker Subsystem* and a *Maintenance Subsystem*. The *Cashier Subsystem* and the *Drink Maker Subsystem* will deal with the drink payment and drink preparing issues respectively to support the function (1). A *Maintenance Subsystem* will support the function (2).

Fig. 4.1 shows (on the left-hand side) the proposed composites to use in the development of the Drink Vending Machine systems and (on the right-hand side) examples of their behaviour using the notation introduced in Chapter 2. Fig. 4.1 (a) depicts a Coffee Card Cashier composite component, which is made of the Card Reader (CR) and Billing Component (BC). The Card Reader component is responsible for reading a chip embedded in a coffee card to get its identifier. The Billing Component is responsible for debiting and crediting coffee cards. By composing these components with a *Pipe* (P) and a *Guard* connector (G), as depicted on the left-hand side of Fig. 4.1 (a), we can generate a composite able to behave as shown on the right-hand side of Fig. 4.1 (a). That is, once a coffee card has been inserted in the card slot of the Coffee Machine and the amount to debit to it has been specified (e.g. *amt*), the composite could try to retrieve the card's identifier by executing the corresponding service in the Card Reader Component (e.g. getCardId). The result of this operation execution can be passed up via de *Pipe* to the *Guard* connector to check if the identifier is retrieved (e.g. *if cardId* $!=$ *null*). If so, then the card could be debited by executing the corresponding service in the Billing Component (e.g. debit) and the resulting execution value will be returned (e.g. *errorCode*).

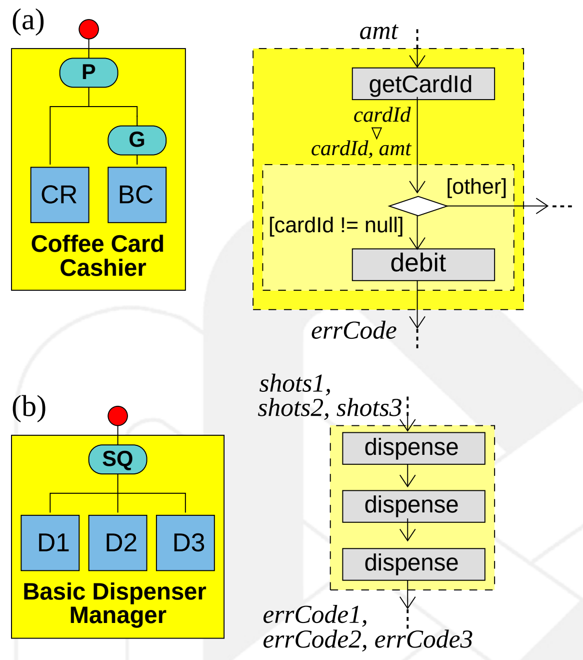

Figure 4.1: Useful composites for the Drink Vending Machine systems and examples of their behaviour.

The Basic Dispenser, shown on the left-hand side Fig. 4.1 (b), is a composite component made up of a set of three instances of the dispenser component and one *Sequencer* connector (SQ). In this composite a Water Dispenser (D1), a Coffee Dispenser (D2) and a Milk Dispenser (D3) have been considered. Thus, this composite allows the sequential execution of one service in each one of these components. For example, the dispense service as depicted on the right-hand side of Fig. 4.1 (b). In this figure, *shots1*-*shots3* denote the number of shots to be dispense in each one of the dispensers, while *errorCode1*-*errorCode3* denote the resulting execution values in each one of the performed executions.

On the other hand, Fig. 4.2 shows three forms in which the above defined composites, together with some other architectural elements, can be utilised to define three different versions of the Drink Vending Machine system. In all the depicted architectures there is a *Selector* connector SL which allows the system to decide which branch of the architecture to execute. Two main branches emerge from the *Selector* connector, which correspond to the main system functions (1) and (2).

Fig. 4.2 (a) shows a version of the system in terms of the three subsystems identified. The *Cashier Subsystem* comprises the Coffee Card Cashier composite and the Payment Manager Component (PMgr) –which manages the drinks menu. The *Drink Maker Subsystem* comprises the Basic Dispenser composite and the Recipe Manager Component (RMgr) –which manages the drinks recipes. The *Maintenance Subsystem* comprises the Basic Dispenser composite only. If the function (1) is required, then the *Selector* connector SL

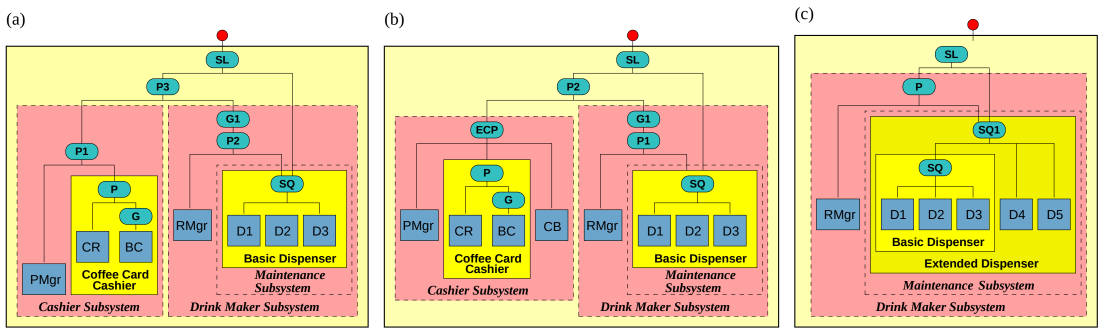

Figure 4.2: Three alternative architectures for the Drink Vending Machine system.

will call the *Pipe* connector P3 and it in turn will call the *Cashier Subsystem* to deal with the drink payment. In this subsystem the *Pipe* connector P1 will first retrieve the drink's price by executing the corresponding operation in the Payment Manager Component PMgr. Then, it will pass up this information when calling the required operation in the Coffee Card Cashier composite. After all these payment issues have been performed, the *Pipe* connector P3 will pass up the resulting data to the *Drink Maker Subsystem*. Internally in this subsystem the *Guard* connector G1 will allow the execution of the computation down on the hierarchy if the drink payment has been processed. Otherwise it will stop it. If there was a successful payment, then the *Guard* connector G1 will call the *Pipe* connector P2. This connector will first retrieve the drink's recipe by executing the corresponding operation in the Recipe Manager component (RMgr) and then call the Basic Dispenser composite to perform the dispense of ingredients accordingly. On the other hand if the function (2) is required, then *Selector* connector SL will call the corresponding operation in the Basic Dispenser composite.

Fig. 4.2 (b) shows a version of the system where the *Cashier Subsystem* has been modified to allow the customer to pay drinks by using either coffee cards or cash. The first payment option is supported by the Coffee Card Cashier composite, while the second one is supported by the Coin Box component (CB). When the function (1) is required, internally in the *Cashier Subsystem* an *Exclusive Choice Pipe*[1] connector ECP will first retrieve the drink's price from the Payment Manager Component PMgr and then, based on the payment method selected by the customer, it will direct the execution of the charge to either the Coffee Card Cashier or the Coin Box component CB. As in the first version, after all these payment issues have been performed, the *Pipe* connector P2 will pass up the resulting data to the *Drink Maker Subsystem* where the drink preparing issues are managed. Both the drink preparing and the function (2) are performed in the same manner as the version of the Drink Vending Machine described before.

Finally, Fig. 4.2 (c) shows a version of the system in which the customer can get drinks for free. That is the reason why the *Cashier Subsystem* is not considered. Additionally, the Basic Dispenser has been further composed to create an Extended Dispenser composite component. This new composite includes the additional dispenser instances D4 and D5

[1]The Exclusive Choice Pipe is a composite composition connector that allows executing a computation in a predecessor component and then, the generated output is passed as input data for the computation of only one component in a set of successor components.

for dealing with more ingredients. In the design of the *Drink Maker Subsystem* the *Guard* connector at the top level has been removed as no validation for the success of the payment is required in this version of the system. The function (2) is performed in the same manner as the systems described above.

Note that that in these examples the defined composites are used more than once for the construction of different systems. The Coffee Card Cashier was utilised two times, while the Basic Dispenser was utilised six times. Thus, they are highly-reusable composites within this domain context.

Now that we have presented the required assemblies and the manner in which they can be composed to define the Drink Selling Machine system, in the following section we demonstrate how the functional specification of these assemblies can be achieved by utilising the functions defined in Chapter 3.

4.1.2 Composite Component Generation

As stated before, the functional specification of a component as a set of provided services which are specified as operation signatures. These signatures in turn are defined in terms of a name and a set of parameters. For clarity purposes, at this starting point we describe the functional specifications of the atomic components required for composite generation in a kind of IDL syntax. In this syntax, the keywords **in** and **out** denote the *role* of a parameter in the operation. In the following sections, we will describe these specifications using the basic formalism described in Section 3.1.

cardReader

getCardId (**out** *int cardId*);

BillingComponent

credit (**in** *int cardId*, **in** *int amount*, **out** *int errCode*);
debit (**in** *int cardId*, **in** *int amount*, **out** *int errCode*);
getBalance (**in** *int cardId*, **out** *int amount*);

Dispenser

emptyDispenser ();
setTemperature (**in** *int temp*, **out** *int errCode*);
add (**in** *int shots*, **out** *int errCode*);
dispense (**in** *int shots*, **out** *int errCode*);

In the following sections we illustrate the use of the defined functions in the the process of composite generation.

Functional Specification of the Coffee Card Cashier Composite

As shown in Section 4.1.1, the Coffee Card Cashier composite is an assembly comprising the Card Reader (CR) and Billing (BC) components as well as the *Guard* and *Pipe* connectors. Because of the nature of our composition approach, we will resolve the functional specification of this composite in a bottom-up manner starting with the assembly involving the *Guard* connector and the Billing component. We will consider "***in** int cardId*" as the parameter to be evaluated by the *Guard* connector (e.g. *if cardId* ! = *null*). Thus, and adopting the notation introduced in Section 3.1, let f be the BillingComponent's functional specification and p be the input parameter to be evaluated by the *Guard*'s Boolean expression:

```
-- Billing Component Instance CB, operations:
-- credit (in int cardId, in int amount, out int errCode)
-- debit (in int cardId, in int amount, out int errCode)
-- getBalance (in int cardId, out int amount)
```

$$\begin{aligned} f = \{ & \langle\{cardId, amount\}, \{errCode\}\rangle, \\ & \langle\{cardId, amount\}, \{errCode\}\rangle, \\ & \langle\{cardId\}, \{amount\}\rangle\} \end{aligned}$$

$$p = \{cardId\}$$

By using the *guard_composite_fspec* function we can derive the following set of tuples, which represent the operation signatures in the functional specification of this assembly:

$$\begin{array}{|l} guard\_composite\_fspec : InParam \times FSpec \rightarrow FSpec \\ \hline guard\_composite\_fspec, \\ conditionLoop\_composite\_fspec, \\ counterLoop\_composite\_fspec = \\ \qquad\qquad s_1, s_2, \ldots, s_n : Sig; \\ \qquad\qquad f : FSpec; \\ \qquad\qquad ip : InParam \mid \#ip = 1; \\ \qquad\qquad (s_1, s_2, \ldots, s_n) \in f \bullet \bigcup_{i=1}^{n} add\_in(ip, s_i) \end{array}$$

$$\begin{aligned} guard\_composite\_fspec = \{ & \langle\{cardId, amount\}, \{errCode\}\rangle, \\ & \langle\{cardId, amount\}, \{errCode\}\rangle, \\ & \langle\{cardId\}, \{amount\}\rangle\} \end{aligned}$$

As can be implied, these tuples denote the signatures of the "guarded" versions of the *debit*, *credit* and *getBalance* operations in the Billing component. In Section 3.1 we mentioned we assume that parameters with the same name and type are semantically equivalent. Note that the input parameter *ip* is semantically equivalent to the input parameter *cardId* of the operations signatures in f. Therefore, the *add_in* helper function in the *guard_composite_fspec* definition kept in only one occurrence. However, internally in the assembly the parameter *cardId* is meant to be utilised in two different manners: as the variable to be evaluated in the *Guard*'s Boolean expression (e.g. $if\ cardId\ != null$) and as the input parameter of the operation signatures provided by the composite.

Using the ADL notation introduced before, the resulting functional specification of this first composition can be written as follows:

GuardedBillingComponent

$op1\ (\textbf{in}\ int\ cardId, \textbf{in}\ int\ amount, \textbf{out}\ int\ errCode);$
$op2\ (\textbf{in}\ int\ cardId, \textbf{in}\ int\ amount, \textbf{out}\ int\ errCode);$
$op3\ (\textbf{in}\ int\ cardId, \textbf{out}\ int\ amount);$

Once that the functional specification of this Guarded Billing Component has been obtained, it can be used together with the one of the CardReader component to generate the functional specification of the Coffee Card Cashier composite. Let f_1 and f_2 be the Card Reader Component and the Guarded Billing Component functional specifications respectively:[2]

```
-- Card Reader Component Instance, operations:
-- getCardId (out int cardId)
```
$f_1 = \{\langle \varnothing, \{cardId\}\rangle\}$

```
-- Guarded Billing Component Instance, operations:
-- op1 (in int cardId, in int amount, out int errCode)
-- op2 (in int cardId, in int amount, out int errCode)
-- op3 (in int cardId, out int amount)
```
$$f_2 = \{\langle\{cardId, amount\}, \{errCode\}\rangle, \\ \langle\{cardId, amount\}, \{errCode\}\rangle, \\ \langle\{cardId\}, \{amount\}\rangle\}$$

Then, we will use the function corresponding to the *Pipe* connector to derive the set of operation signatures in the functional specification of the Coffee Card Cashier composite:

$$pipe\_composite\_fspec : FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{aligned} pipe\_composite\_fspec = \\ & s_1, s_2, \ldots, s_n : Sig; \\ & f_1, f_2, \ldots, f_n : FSpec; \\ & \forall\, s_i \in (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \mid sig\_match(s_i, s_{i+1}) = true \bullet \\ & \bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_bound(s_1, s_2, \ldots, s_n) \end{aligned}$$

According to this function, we must first compute the cartesian product of functional specifications f_1 and f_2, which results in:

[2] We use $\varnothing$ symbol to denote both no input parameters and no output parameters.

$$\begin{aligned}f_1 \times f_2 = \{&\{\langle\varnothing, \{cardId\}\rangle, \langle\{cardId, amount\}, \{errCode\}\rangle\},\\ &\{\langle\varnothing, \{cardId\}\rangle, \langle\{cardId, amount\}, \{errCode\}\rangle\},\\ &\{\langle\varnothing, \{cardId\}\rangle, \langle\{cardId\}, \{amount\}\rangle\}\}\end{aligned}$$

After this cartesian product has been computed, the function *sig_bound* is applied to each one the tuples of the cartesian product. The union of the resulting tuples as required in $\bigcup\limits_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_bound(s_1, s_2, \ldots, s_n)$ is then:

$$\begin{aligned}pipe\_composite\_fspec(f_1, f_2) = \{&\langle\{amount\}, \{errCode\}\rangle,\\ &\langle\{amount\}, \{errCode\}\rangle,\\ &\langle\varnothing, \{amount\}\rangle\}\end{aligned}$$

Using the ADL notation introduced before, the resulting functional specification of the Coffee Card Cashier composite can be written as follows:

CoffeeCardCashier

*op*1 (**in** *int amount*, **out** *int errCode*);
*op*2 (**in** *int amount*, **out** *int errCode*);
*op*3 (**out** *int amount*);

In this functional specification, *op*1, *op*2 and *op*3 are signatures abstracting the valid sequences of operations to invoke within the composite's constituents, i.e. *getCardId-credit*, *getCardId-debit* and *getCardId-getBalance* respectively. Note that, the requirements (i.e. the input parameters) and the outcomes (i.e. the output parameters) of these operation sequences are entirely derived from the semantics of the *Pipe* connector. The helper function *sig_bound* is utilised to remove the input parameter *cardId* in the resulting signatures, as it is produced internally in the composite. Thus for example to execute the operations *op*1 and *op*2, which abstract the execution sequences *getCardId-credit* and *getCardId-debit* respectively, it is only necessary to provide a value for the *amt* input parameter (see Fig. 4.1 (a)).

Functional Specification of the Basic Dispenser Composite

As introduced in Section 4.1.1, the Basic Dispenser composite is a composite component made of three dispenser components and one *Sequencer* connector. Let be f_1, f_2 and f_3 the functional specifications of the three instances of the Dispenser component to be composed:

```
-- Dispenser Component Instance D1, operations:
-- emptyDispenser ()
-- setTemperature (in int temp, out int errCode)
-- add (in int shots, out int errCode)
-- dispense (in int shots, out int errCode)
```

$f_1 = \{\langle\varnothing, \varnothing\rangle,$

$\langle\{temp\},\{errCode\}\rangle,$
$\langle\{shots\},\{errCode\}\rangle,$
$\langle\{shots\},\{errCode\}\rangle\}$

```
-- Dispenser Component Instance D2, operations:
-- emptyDispenser ()
-- setTemperature (in int temp, out int errCode)
-- add (in int shots, out int errCode)
-- dispense (in int shots, out int errCode)
```

$$\begin{aligned} f_2 = \{&\langle\varnothing,\varnothing\rangle,\\ &\langle\{temp\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle\}\end{aligned}$$

```
-- Dispenser Component Instance D3, operations:
-- emptyDispenser ()
-- setTemperature (in int temp, out int errCode)
-- add (in int shots, out int errCode)
-- dispense (in int shots, out int errCode)
```

$$\begin{aligned} f_3 = \{&\langle\varnothing,\varnothing\rangle,\\ &\langle\{temp\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle\}\end{aligned}$$

We will use the *seq_composite_fspec* function to generate the corresponding functional specification:

$$seq\_composite\_fspec : FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{aligned} &seq\_composite\_fspec =\\ &\qquad s_1, s_2, \ldots, s_n : Sig;\\ &\qquad f_1, f_2, \ldots, f_n : FSpec;\\ &\qquad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_concat(s_1, s_2, \ldots, s_n)\end{aligned}$$

As in the previous case, to resolve the *seq_composite_fspec* function we first compute the cartesian product of functional specifications f_1, f_2 and f_3, which results in:

$$\begin{aligned} f_1 \times f_2 \times f_3 = \{&\{\langle\varnothing,\varnothing\rangle,\langle\varnothing,\varnothing\rangle,\langle\varnothing,\varnothing\rangle\},\\ &\{\langle\varnothing,\varnothing\rangle,\langle\varnothing,\varnothing\rangle,\langle\{temp\},\{errCode\}\rangle\},\\ &\{\langle\varnothing,\varnothing\rangle,\langle\varnothing,\varnothing\rangle,\langle\{shots\},\{errCode\}\rangle\},\\ &\{\langle\varnothing,\varnothing\rangle,\langle\varnothing,\varnothing\rangle,\langle\{shots\},\{errCode\}\rangle\},\\ &\ldots\\ &\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle,\{\langle\varnothing,\varnothing\rangle\},\\ &\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{temp\},\{errCode\}\rangle\},\\ &\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle\},\end{aligned}$$

$$\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle,\{\langle\{shots\},\{errCode\}\rangle\}\}$$

After this cartesian product has been computed, the function *sig_concat* is applied to each one the tuples of the cartesian product. The union of the resulting tuples as required in $\bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_concat(s_1, s_2, \ldots, s_n)$ is then:

$$\begin{aligned}
seq\_composite\_fspec(f_1, f_2, f_3) = \{&\langle\varnothing,\varnothing\rangle,\\
&\langle\{temp\},\{errCode\}\rangle,\\
&\langle\{shots\},\{errCode\}\rangle,\\
&\langle\{shots\},\{errCode\}\rangle,\\
&\ldots\\
&\langle\{shots, shots\},\{errCode, errCode\}\rangle,\\
&\langle\{shots, shots, temp\},\{errCode, errCode, errCode\}\rangle,\\
&\langle\{shots, shots, shots\},\{errCode, errCode, errCode\}\rangle,\\
&\langle\{shots, shots, shots\},\{errCode, errCode, errCode\}\rangle\}
\end{aligned}$$

Using the notation introduced above, the functional functional specification of the Basic Dispenser composite component can be denoted as follows:

```
BasicDispenser
op1 ();
op2 (in int temp, out int errCode);
op3 (in int shots, out int errCode);
op4 (in int shots, out int errCode);
op5 (in int temp);
op6 (in int temp, in int temp, out int errCode, out int errCode);
op7 (in int temp, in int shots, out int errCode, out int errCode);
op8 (in int temp, in int shots, out int errCode, out int errCode);
op9 (in int shots);
op10 (in int shots, in int temp, out int errCode, out int errCode);
op11 (in int shots, in int shots, out int errCode, out int errCode);
op12 (in int shots, in int shots, out int errCode, out int errCode);
op13 (in int shots);
op14 (in int shots, in int temp, out int errCode, out int errCode);
op15 (in int shots, in int shots, out int errCode, out int errCode);
op16 (in int shots, in int shots, out int errCode, out int errCode);
op17 (in int temp, out int errCode);
op18 (in int temp, in int temp, out int errCode, out int errCode);
op19 (in int temp, in int shots, out int errCode, out int errCode);
op20 (in int temp, in int shots, out int errCode, out int errCode);
op21 (in int temp, in int temp, out int errCode, out int errCode);
op22 (in int temp, in int temp, in int temp, out int errCode, out int errCode, out int errCode);
op23 (in int temp, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op24 (in int temp, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op25 (in int temp, in int shots, out int errCode, out int errCode);
op26 (in int temp, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op27 (in int temp, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op28 (in int temp, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op29 (in int temp, in int shots, out int errCode, out int errCode);
op30 (in int temp, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op31 (in int temp, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op32 (in int temp, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op33 (in int shots, out int errCode);
op34 (in int shots, in int temp, out int errCode, out int errCode);
op35 (in int shots, in int shots, out int errCode, out int errCode);
op36 (in int shots, in int shots, out int errCode, out int errCode);
op37 (in int shots, in int temp, out int errCode, out int errCode);
op38 (in int shots, in int temp, in int temp, out int errCode, out int errCode, out int errCode);
op39 (in int shots, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op40 (in int shots, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op41 (in int shots, in int shots, out int errCode, out int errCode);
op42 (in int shots, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op43 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op44 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op45 (in int shots, in int shots, out int errCode, out int errCode);
op46 (in int shots, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op47 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op48 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op49 (in int shots, out int errCode);
op50 (in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op51 (in int shots, in int shots, out int errCode, out int errCode);
op52 (in int shots, in int shots, out int errCode, out int errCode);
op53 (in int shots, in int temp, out int errCode, out int errCode);
op54 (in int shots, in int temp, in int temp, out int errCode, out int errCode, out int errCode);
op55 (in int shots, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op56 (in int shots, in int temp, in int shots, out int errCode, out int errCode, out int errCode);
op57 (in int shots, in int shots, out int errCode, out int errCode);
op58 (in int shots, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op59 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op60 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op61 (in int shots, in int shots, out int errCode, out int errCode);
op62 (in int shots, in int shots, in int temp, out int errCode, out int errCode, out int errCode);
op63 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
op64 (in int shots, in int shots, in int shots, out int errCode, out int errCode, out int errCode);
```

As in the previous case, these signatures abstract the valid sequences of operation executions within the composite's constituents. For example, *op*1 abstracts the execution sequence *emptyDispenser-emptyDispenser-emptyDispenser* while *op*64 abstracts the execution sequence *dispense-dispense-dispense*. As before, we want to highlight that the requirements (i.e. the input parameters) and the outcomes (i.e. the output parameters) of these operation sequences are entirely derived from the semantics of the *Sequencer* connector. Note how each one of the signatures in the resulting functional specification is made up of a concatenation of the parameters of the composed components' signatures by applying the *sig_concat* helper function. The issue of having duplicated elements in the resulting *InParam* and *OutParam* sets of each one of the resulting signatures is because of composing several instances of the same component types. This allows, for example, invoking the sequence *dispense-dispense-dispense* with a different number of shots in each one of the dispensers, see Fig. 4.1 (b).

We want to highlight that, although parameter names duplications it is semantically valid in our approach, it is syntactically invalid at implementation-time, e.g. method signatures in Java cannot have duplicated parameter names. Thus, when duplicated parameters appear in an operation signature at implementation time it is necessary to rename them somehow. An approach for renaming them could be that of adding a suffix to the original ones to differentiate them, e.g. *shots1*, *shots2* and *shots3*.

4.2 Robotics System

The second case study consists of implementing a pair of systems to control a LEGO Mindstorms NTX educational robot [11] as the one depicted in Fig. 4.3. The LEGO Mindstorms NTX is a popular robotics kit that provides an immediate opportunity to develop systems to control robot interaction. The brain of a robot is the NTX, which is an intelligent computer-controlled brick that lets a robot come alive and perform different operations. The NTX has a set of output ports for attaching a variety of motors (e.g. wheels, harms, etc.) and a set of input ports for attaching a variety of sensors (e.g. distance, sound, light sensors, etc.).

Figure 4.3: A LEGO Mindstorms NTX educational robot.

4.2.1 The Proposed Architectures

The first system to construct is a Basic Robot Control System. In this system the sensed information from an ultrasonic sensor is utilised to drive the robot around searching to avoid obstacles.[3] The second system is an Advanced Robot Control System which is an extension of the former where a sound sensor is added in order to stop the robot when a sound within a specific the frequency is detected.

A widely utilised architectural style to develop control systems is the *closed control loop* [29]. This style includes three main subsystems: a *Controller Subsystem* –which continually receives information about the physical system and supplies continuous guidance about the changes to be performed to maintain its properties; a *Sensor Subsystem* –which is engaged in gathering information about the physical system and sending it to the Controller and a *Actuator Subsystem* –which is involved in performing the decisions taken by the *Controller Subsystem*.

Given the characteristics of the robot to control, we realised that the *Actuator Subsystem* can be defined as a composite component. This composite is depicted in Fig. 4.4. As can be seen, the Wheels Controller composite component is created out to two Wheel Motor atomic components (WM1 and WM2) together with a *Sequencer* connector (SEQ). Because the robot has only two wheels and they are moving up or down together at the same rate in both systems, we find convenient to compose them together in a reusable *Wheels Controller* composite component.

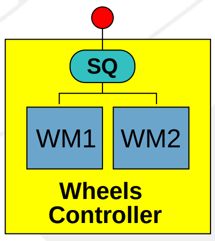

Figure 4.4: A useful composite for the Robot Control Systems.

Fig. 4.5 (a) and (b) show the proposed architectures for the Basic Robot Control System and the Advanced Robot Control System in terms of the *Sensor* and *Actuator Subsystems*. Note that, in the depicted architectures there is not a *Controller Subsystem* because in our approach to component composition the connectors encapsulate all the decision making.

In the Basic and the Advanced Robot Control Systems' architectures there is a *Sequencer* top-level connector (SQ2 and SQ3 in Fig. 4.5 (a) and (b) respectively), which connects the Bluetooth atomic component (B) and a *Condition Controlled Loop* connector (CCL). The *Condition Controlled Loop* in turn connects a *Pipe* connector (P). This particular arrangement of connectors allows activating the Bluetooth transceiver[4] –when the *Sequencer* invokes an operation in a Bluetooth atomic component B; and then launching the main loop of control of the systems –when the *Sequencer* invokes the *Condition*

[3]The ultrasonic sensor enables the robot to detect objects by measuring distance and detecting movement.

[4]By activating the Bluetooth transceiver the robot can receive data from a remote computer.

Controlled Loop. Note that the *Condition Controlled Loop* supports the cyclic execution of the *Pipe* connector which allows that the result of the execution in the *Sensor Subsystem* be the input of the *Actuator Subsystem*.

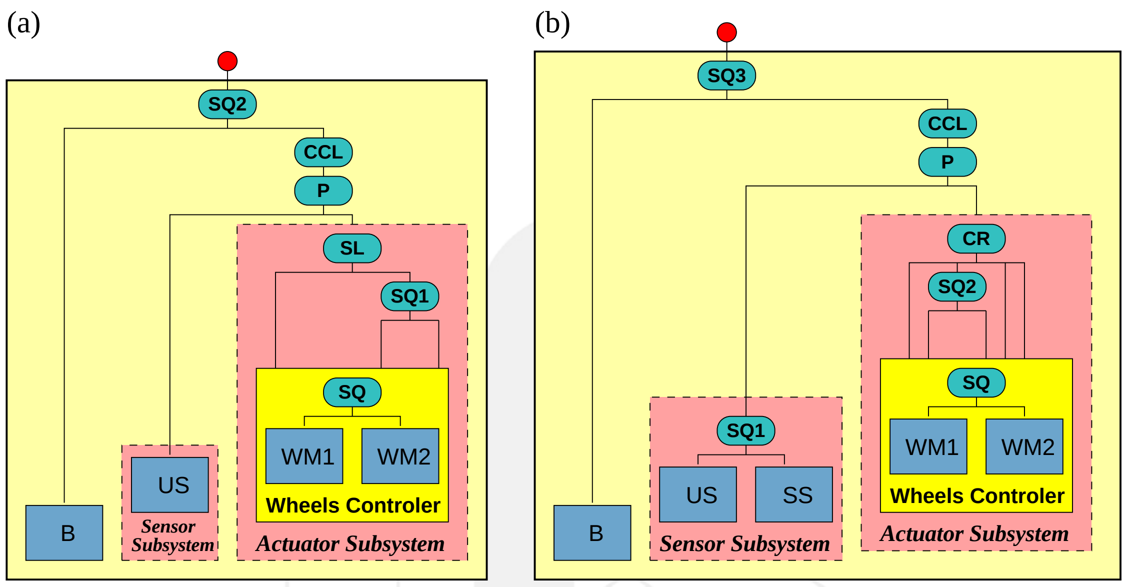

Figure 4.5: The proposed architectures for the (a) Basic Robot Control System and the (b) Advanced Robot Control System.

In the Basic Robot System, depicted in Fig. 4.5 (a), the *Sensor Subsystem* comprises only the Utrasonic Sensor component (US). On the other hand, the *Actuator Subsystem* includes three copies of the Wheels Controller composite component assembled via a *Selector* connector (SL) and a *Sequencer* connector (SQ1). Thus once launched the main loop of the system, the *Pipe* connector P will get first the distance reading via the execution of a particular operation in the *Sensor Subsystem* and then it will pass up this data to support the execution in the *Actuator Subsystem*. Based on the data received, the *Selector* connector determines which executions to perform to change the current state of the wheels of the robot. Only one of two executions are possible: (i) if the distance value is greater than 35 centimeters then the robot wheels are moved forward; (ii) otherwise the robot wheels are rotated and moved forward again –note how this behaviour is achieved by using a *Sequencer* connector which will call firstly the operation to rotate the wheels and secondly the operation to move them forward again.

The Advanced Robot System is depicted in Fig. 4.5 (b). In this version of the system the *Sensor Subsystem* is an assembly that includes one instance of the Utrasonic Sensor component (US), one instance of the Sound Sensor component (SS) and one *Sequencer* connector (SQ1). Internally in this subsystem the *Sequencer* connector retrieves the required data from the Utrasonic and Sound Sensors by sequentially executing the corresponding operations in each one of them. The *Actuator Subsystem* includes three copies of the Wheels Controller composite assembled via a *Chain of Responsibility* (CR). Once launched the main loop of the system the *Pipe* connector will get the distance and sound readings via an execution in the *Sensor Subsystem*, then it will pass up this data to support the execution in the *Actuator Subsystem*. Based on the data received, the

Chain of Responsibility connector determines which executions to perform from the following set: (i) if the distance value is greater than 35 centimeters then the robot wheels are moved forward; (ii) if the distance value is lower than 35 centimeters then the robot wheels are rotated and moved forward again –note how this behaviour is achieved by using a *Sequencer* connector (SQ2) (iii) if the decibels value is grater than 60 then the wheels are stopped and (iv) if the decibels value is lower than 60 then the wheels are moved forward.

Now that we have presented the details of the Wheels Controller composite and the manner in which it can be composed in the two Robot Systems, in the following section we describe how the functional specification of this composite can be achieved by using the proposed specification functions.

4.2.2 Composite Component Generation

As before, we will describe the functional specification of the Wheel component in a kind of ADL notation:

wheelMotor

$stop\ ();$
$forward\ ();$
$backward\ ();$
$setSpeed(\mathbf{in}\ int\ speed);$
$rotate(\mathbf{in}\ int\ count);$
$rotateNoWait(\mathbf{in}\ int\ count);$

Functional Specification of the Wheels Controller Composite

We have already mentioned that the Wheels Controller is a composite component created out to two Wheel Motor atomic components (WM1 and WM2) and a *Sequencer* connector (SQ). Then, Let be f_1 and f_2 the functional specifications of the two instances of the Wheel Motor component to be composed:

```
-- Wheel Motor Component instance WM1, operations:
-- stop ()
-- forward ()
-- backward ()
-- setSpeed (in int speed)
-- rotate (in int count)
-- rotateNoWait (in int count)
```

$$
\begin{aligned}
f_1 = \{&\langle\varnothing,\varnothing\rangle,\\
&\langle\varnothing,\varnothing\rangle,\\
&\langle\varnothing,\varnothing\rangle,\\
&\langle\{speed\},\varnothing\rangle,\\
&\langle\{count\},\varnothing\rangle,
\end{aligned}
$$

$\langle \{count\}, \varnothing\rangle\}$

```
-- Wheel Motor Component instance WM2, operations:
-- stop ()
-- forward ()
-- backward ()
-- setSpeed (in int speed)
-- rotate (in int count)
-- rotateNoWait (in int count)
```

$$\begin{aligned} f_2 = \{ & \langle \varnothing, \varnothing\rangle, \\ & \langle \varnothing, \varnothing\rangle, \\ & \langle \varnothing, \varnothing\rangle, \\ & \langle \{speed\}, \varnothing\rangle, \\ & \langle \{count\}, \varnothing\rangle, \\ & \langle \{count\}, \varnothing\rangle\} \end{aligned}$$

We will make use of the following function to generate the corresponding functional specification:

$$seq\_composite\_fspec : FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{aligned} & seq\_composite\_fspec = \\ & \qquad s_1, s_2, \ldots, s_n : Sig; \\ & \qquad f_1, f_2, \ldots, f_n : FSpec; \\ & \qquad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \bigcup_{(s_1,\ldots,s_n) \in \prod_{i=1}^{n} f_i} sig\_concat(s_1, s_2, \ldots, s_n) \end{aligned}$$

As in the previous case, to resolve the *seq_composite_fspec* function we first compute the cartesian product of functional specifications f_1 and f_2, which results in:

$$\begin{aligned} f_1 \times f_2 = \{ & \{\langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \varnothing, \varnothing\rangle, \langle \{speed\}, \varnothing\rangle\}, \\ & \{\langle \varnothing, \varnothing\rangle, \langle \{count\}, \varnothing\rangle\}, \\ & \{\langle \varnothing, \varnothing\rangle, \langle \{count\}, \varnothing\rangle\}, \\ & \ldots \\ & \{\langle \{count\}, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \{count\}, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \{count\}, \varnothing\rangle, \langle \varnothing, \varnothing\rangle\}, \\ & \{\langle \{count\}, \varnothing\rangle, \langle \{speed\}, \varnothing\rangle\}, \\ & \{\langle \{count\}, \varnothing\rangle, \langle \{count\}, \varnothing\rangle\}, \\ & \{\langle \{count\}, \varnothing\rangle, \langle \{count\}, \varnothing\rangle\}\} \end{aligned}$$

After this cartesian product has been computed, the function *sig_concat* is applied to each one the tuples of the cartesian product. The union of the resulting tuples as required in $\bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_concat(s_1, s_2, \ldots, s_n)$ is then:

$$\begin{aligned}
seq\_composite\_fspec(f_1, f_2, f_3) = \{ & \langle\{\varnothing,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{\varnothing,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{\varnothing,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{\varnothing, speed\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{\varnothing, count\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{\varnothing, count\},\{\varnothing,\varnothing\}\rangle,\\
& \ldots\\
& \langle\{count,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{count,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{count,\varnothing\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{count, speed\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{count, count\},\{\varnothing,\varnothing\}\rangle,\\
& \langle\{count, count\},\{\varnothing,\varnothing\}\rangle\}
\end{aligned}$$

Using the IDL syntax introduced before, the functional functional specification of the Basic Dispenser composite component can be denoted as follows:

```
BasicDispenser
op1 ();
op2 ();
op3 ();
op4 (in int speed);
op5 (in int count);
op6 (in int count);
op7 ();
op8 ();
op9 ();
op10 (in int speed);
op11 (in int count);
op12 (in int count);
op13 ();
op14 ();
op15 ();
op16 (in int speed);
op17 (in int count);
op18 (in int count);
op19 (in int speed);
op20 (in int speed);
op21 (in int speed);
op22 (in int speed, in int speed);
op23 (in int speed, in int count);
op24 (in int speed, in int count);
op25 (in int count);
op26 (in int count);
op27 (in int count);
op28 (in int count, in int speed);
op29 (in int count, in int count);
op30 (in int count, in int count);
op31 (in int count);
op32 (in int count);
op33 (in int count);
op34 (in int count, in int speed);
op35 (in int count, in int count);
op36 (in int count, in int count);
```

In this functional specification, the signatures abstract the valid sequences of operation executions in the composite's constituents. For example, *op*1 abstracts the execution sequence *stop*-*stop* while *op*36 abstracts the execution sequence *rotateNoWait*-*rotateNoWait*.

As before, we want to highlight that although parameter names duplications it is semantically valid in our approach, it is syntactically invalid at implementation-time. Therefore, when duplicated parameters appear in an operation signature it is necessary to rename them somehow at implementation time. An approach for renaming them could be that of adding a suffix to the original ones to differentiate them, e.g. *count1* and *count2*.

4.3 Summary

In this chapter we have illustrated the usefulness of the defined functions via the development of two case studies. The first case was about developing three different versions of a Drink Selling Machine System. For this system a Coffee Card Cashier and a Basic Dispenser were constructed. The defined composites were systematically and consistently specified by using the defined functions in Chapter 3. The Coffee Card Cashier was utilised two times, while the Basic Dispenser was utilised six times in the different versions of a Drink Selling Machine.

The second case study was about building two versions of a Robotics System. For this case study, a Wheels Controller composite component was created and utilised twice for the development of the Robotics Systems.

Chapter 5

Discussion, Conclusions and Future Work

In this chapter we discuss the benefit of our approach and we state the conclusions and future work.

5.1 Discussion

Management of both functional and non-functional properties is one of the main challenges in CBD community [26, 15]. Despite the former, the starting point in their management, that is their specification, is not addressed in a systematic and consistent manner in most CBD approaches. With few exceptions, current practice in CBD focuses on components and leaves their interactions specified as lines, which usually represent procedure calls or events [4, 20]. However, practical systems have quite sophisticated rules about component interaction. Therefore, and as it was illustrated in Section 1.2, such lines are not enough for defining a method for systematically and consistently deriving the specifications of functional and non-functional properties of component assemblies.

Although our approach to derive composites' functional specifications is tightly connected to the issue of having first-class connectors and passive components, we believe that the material presented in previous sections enables us to make some general observations as well as to discuss the potential usefulness, advantages and drawbacks of our approach with respect to related work.

Passive components such as speech and sound recognisers, image detectors, temperature sensors and so on have been widely utilised in the development of many software systems. When using this type of components, the separation of computation from control becomes more evident; control flow does not originate from components but from the architectural abstractions that manage them. On the other hand, the benefits of using first-class abstractions to represent connections among components have been already recognised [3, 21, 14, 9, 7, 28]. Thus, and despite the fact that first-class connectors and passive components are not common in most component models, a component model like ours is worth of considering for the development of component-based systems. We have developed some sophisticated components-based systems based on the semantics of our component model [31].

Although the availability of first-class class connectors facilitated the development of our approach, it seems difficult to develop a similar one for component models such as SOFA 2 [6] and ProCom [27] or for architecture description languages such as ACME[5] and UniCon[25]. All these works provide first-class connectors and support the generation of composite components. In all these approaches the specification of the functional interfaces of composite components is generated in an *ad hoc* manner (via delegation connectors). Thus, the exact nature of the composite's functional interfaces depends on whether and which of interfaces inside the composite have been forwarded. In contrast, in our approach the specification of the functional interfaces of composite component is *systematically* and *consistently* generated. We can achieve it because our approach is formalised in a set of simple connector-specific functions which take information from the functional specifications of the composed components. It makes our approach to have an algebraic basis which makes it more precise and objective.

As a corollary of the algebraic basis of our approach, the proposed functions can be composed. The ones defined for basic composition connectors can be reused in the definitions for the composite ones as shown in Chapter 3.

Although in this technical report the focus was given to how to use our functions to composite component specification, they can be also used to generate the operation signatures in the *FSpec*s of subsystems and final systems. In order to do that, a system developer can reduce the number of operation signatures to be consider as part of the *FSpec*s of the composed components and then apply the corresponding function.

To illustrate the former, consider the case of building the *Cashier Subsystem* in Fig. 4.2 (b). This subsystem requires the Product Manager atomic component PMgr, the Coffee Card Cashier composite component (which we have generated in Section 4.1.2) and the Coin Box component CB. These three units are composed via the *Exclusive Choice Pipe* composite connector. Consider then the following functional specifications of the components in this subsystem:

productManager

addProduct (**in** *int productId*, **out** *errCode*);
getPrice (**in** *int productId*, **out** *amount*);
deleteProduct (**in** *int productId*, **out** *errCode*);

CoffeeCardCashier

*op*1 (**in** *int amount*, **out** *int errCode*);
*op*2 (**in** *int amount*, **out** *int errCode*);
*op*3 (**out** *int amount*);

coinBox

collectCoins (**in** *int amount*, **out** *int errCode*);
giveChange (**in** *int change*, **out** *errCode*);
addCoins (**out** *int amount*);

As described in Section 4.1.1, this version of the *Cashier Subsystem* allows the customer to buy drinks by using either coffee cards or cash. This first option is supported by the Coffee Card Cashier composite, while the second one is supported by the Coin Box component CB. When the sell a drink function is required, internally in the *Cashier Subsystem* the *Exclusive Choice Pipe* connector ECP1 will first retrieve the drink's price by executing the corresponding operation in the Payment Manager Component PMgr and then, based on the payment method selected by the customer, it will direct the execution of the charge to either the Coffee Card Cashier or the Coin Box component CB. Thus, for generating the functional specification of this subsystem we do not require to consider all the operations in all the components but only a subset of them:

```
-- Product Manager Atomic Component. Operation getPrice.
```
$f_1 = \{\langle\{productId\}, \{amount\}\rangle\}$
```
-- CoffeeCardCashier Composite Component. Operation getCardId-debit.
```
$f_2 = \{\langle\{amount\}, \{errCode\}\rangle\}$
```
-- CoinBox Atomic Component. Operation collectCoins.
```
$f_3 = \{\langle\{amount\}, \{errCode\}\rangle\}$

We also require the value to be evaluated for the execution of the methods *getCardId* and *collectCoins* payment method:

```
-- The set of input parameters
```
$p = \{paymentMethod\}$

Next we show by using the *ecPipe_composite_fspec* function on these reduced functional specifications we can generate the functional specification of this *Cashier Subsystem.*

$$\begin{array}{|l}
ecPipe\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec \\
\hline
ecPipe\_composite\_fspec = \\
\qquad \lambda\, f_1, f_2, \ldots, f_n : FSpec; \\
\qquad \lambda\, p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n);\ \bullet \\
\qquad pipe\_composite\_fspec(f_1, sel\_composite\_fspec(p, f_2, \ldots, f_n))
\end{array}$$

As this function uses the *sel_composite_fspec* and *pipe_composite_fspec* functions, we will first to solve the first one

$$\begin{array}{|l}
sel\_composite\_fspec : InParam \times FSpec \times FSpec \times \ldots \times FSpec \rightarrow FSpec \\
\hline
sel\_composite\_fspec = \\
\qquad \lambda\, s_1, s_2, \ldots, s_n : Sig; \\
\qquad \lambda\, f_1, f_2, \ldots, f_n : FSpec; \\
\qquad \lambda\, p : InParam \mid \#p = \#(f_1 \times f_2 \times \ldots \times f_n); \\
\qquad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \\
\qquad \bigcup_{i=1}^{n} add\_in(ip_i, s_i) \mid ip_i \in p \land s_i \in \bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_union(s_1, s_2, \ldots, s_n)
\end{array}$$

As in the previous examples, to resolve the *sel_composite_fspec* function we first compute the cartesian product of functional specifications f_2 and f_3, which results in:

$f_2 \times f_3 = \{\langle\{amount\}, \{errCode\}\rangle, \langle\{amount\}, \{errCode\}\rangle\}$

After this cartesian product has been computed, the function *sig_union* is applied to each one the tuples of the cartesian product, which results in:

$sig\_union(\langle\{amount\}, \{errCode\}\rangle, \langle\{amount\}, \{errCode\}\rangle) = \langle\{amount\}, \{errCode\}\rangle$

After this function has been resolved, we resolve the last part of the function $\bigcup_{i=1}^{n} add\_in(ip_i, s_i)$, which results in:
$sel\_composite\_fspec = \langle\{paymentMethod, amount\}, \{errCode\}\rangle$

We will call f_{23} to this functional specification for clarity purposes. Once f_{23} has been calculated, we can proceed to calculate the *pipe_composite_fspec* function

$$pipe\_composite\_fspec : FSpec \times FSpec \times \ldots \times FSpec \to FSpec$$

$$\begin{aligned} &pipe\_composite\_fspec = \\ &\quad \lambda\, s_1, s_2, \ldots, s_n : Sig; \\ &\quad \lambda\, f_1, f_2, \ldots, f_n : FSpec; \\ &\quad \forall\, s_i \in (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \mid sig\_match(s_i, s_{i+1}) = true \bullet \\ &\quad\quad \bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_bound(s_1, s_2, \ldots, s_n) \end{aligned}$$

This function requires calculating the cartesian product of the following specifications:

```
-- Product Manager Atomic Component. Operation getPrice.
```
$f_1 = \{\langle\{productId\}, \{amount\}\rangle\}$
```
-- CoffeeCardCashier Composite Component. Operation getCardId-debit.
```
$f_{23} = \{\langle\{paymentMethod, amount\}, \{errCode\}\rangle\}$

which results in:

$$f_2 \times f_{23} = \{\langle\{productId\}, \{amount\}\rangle, \langle\{paymentMethod, amount\}, \{errCode\}\rangle\}$$

After this cartesian product has been computed, the function *sig_bound* is applied to each one the tuples of the cartesian product. The union of the resulting tuples as required in $\bigcup_{(s_1,\ldots,s_n)\in\prod_{i=1}^{n} f_i} sig\_bound(s_1, s_2, \ldots, s_n)$ is then:

$pipe\_composite\_fspec(f_1, f_{23}) = \{\langle\{productId, paymentMethod\}, \{errCode\}\rangle\}$
Using the ADL notation introduced before, the resulting functional specification of the *Cashier Subsystem* can be written as follows:

CoffeeCardCashier

$op1$ (**in** *int productId*, **in** *int paymentMethod*, **out** *int errCode*);

This functional specification contains only one operation as there is only one manner to use this subsystem. In contrast to a composite, this subsystems is not intended for reuse. Thus, by receiving the input parameters *productId* and the *paymentMethod* the system can then process the corresponding payment.

Our proposal may have some potential drawbacks though. For example, in the case study we observed that depending on the number of operations provided by the composed components, the application of some functions could explode potentially the number of operations in the resulting interfaces. In our case study the application of the *seq_composite_fspec* on the interfaces of the three dispenser components resulted in an interface containing 64 operation signatures. Taking into consideration the domain context for which the composite component is built, only 4 out of the 64 operations make sense. These operations are the ones abstracting the sequential execution of the same operation in each one of the dispenser components, i.e. *emptyDispenser-emptyDispenser-emptyDispenser*, *setTemperature-setTemperature-setTemperature*, *add-add-add* and *dispense-dispense-dispense*. Although by using our tool a composite developer can reduce the number of signatures of the resulting functional specification by keeping these execution sequences that are invalid or undesirable away, it could be desirable to automatically support this "filtering" by using some sort of enhanced behavioural information, e.g. feature models, behaviour protocols, etc.

An issue that we have not solved is the one of shared data. In some cases within a composite component, data needs to be defined to be shared between its constituents. So this data is important to interface generation. We are in the process of investigating solutions in this context.

5.2 Conclusions and Future Work

We presented our progress on developing an approach to systematically and consistently supporting interface specification of composite components. Specifically, we focused on the generation of composites' functional specifications. The composites are constructed via a connectors' catalogue, which was defined within the context of a new component model. The specification approach is based on a set of connector-specific functions, which allow deriving composites' functional specifications in a systematic and consistent manner. Although this piece of work is only at an initial stage it has an algebraic basis which makes it more precise and objective. We believe that it can be easily automated and can be very fast in terms of computations.

In the near future, we plan to extend our approach to deal with more sophisticated behavioural information, e.g. constraints on their temporal ordering of executions. Similarly, we have started to work on a similar approach to derive other elements of the composites' interfaces, i.e. the non-functional properties and the information about the deployment environment.

References

[1] C. Bock. UML 2 composition model. *Journal of Object Technology*, 3(10):47–73, 2004.

[2] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[3] T. Bures. *Generating Connectors for Homogeneous and Heterogeneous Deployment.* PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Prague, Czech Republic, 2006.

[4] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *IEEE Transaction of Software Engineering*, Submitted for publishing:1–25, October 2010.

[5] R. Monroe D. Garlan and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97, Toronto Ontario*, pages 169–183, November 1997.

[6] DSRG Charles University in Prague. SOFA 2. Component System that Makes a Change, 2009. `http://sofa.ow2.org/`.

[7] S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *Proceedings of the European Software Engineering Conference (ESEC/FSE)*, pages 483–500. Lectures Notes in Computer Science, Springer - Verlag, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1995.

[9] D. Garlan. Higher-order connectors. In *Proceedings of Workshop on Compositional Software Architectures*, January 1998.

[10] C.J.M. Geisterfer and S. Ghosh. Software component specification: A study in perspective of component selection and reuse. In *Proceedings of the 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS)*, Washington, DC, USA, 2006. IEEE Computer Society.

[11] The LEGO Group. LEGO.com MINDSTORM NTX Home. http://mindstorms.lego.com/.

[12] J. Han. A comprehensive interface definition framework for software components. In *Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC)*, page 110, Washington, DC, USA, 1998. IEEE Computer Society.

[13] G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[14] D. Hirsch, S. Uchitel, and D. Yankelevich. Towards a periodic table of connectors. In *Coordination Models and Languages*, page 418, 1999.

[15] S.A. Hissam, G.A. Moreno, J.A. Stafford, and K.C. Wallnau. Enabling predictable assembly. *Journal of Systems and Software*, 65(3):185–198, 2003.

[16] K.-K. Lau, L. Ling, and P. Velasco Elizondo. Towards composing software components in both design and deployment phases. In H.W. Schmidt *et al.*, editor, *Proceedings of the 10th International Symposium on Component-based Software Engineering, LNCS 4608*, pages 274–282. Springer, 2007.

[17] K.-K. Lau, L. Ling, P. Velasco Elizondo, and V. Ukis. Composite connectors for composing software components. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition, LNCS 4829*, pages 266–280. Springer-Verlag, 2007.

[18] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F.S. de Boer *et al.*, editor, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2006.

[19] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G.T. Heineman, I. Crnkovic, H. Schmidt, J. Stafford, C. Szyperski, and K. Wallnau, editors, *Proceedings of 8th International SIGSOFT Symposium on Component-based Software Engineering*, pages 90–106. Springer-Verlag Heidelberg, May 2005.

[20] K.-K. Lau and Z. Wang. A survey of software component models. Preprint CSPP-38, School of Computer Science, The University of Manchester, May 2006.

[21] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd Internationa Conference on Software Engineering*, pages 178–187. ACM Press, 2000.

[22] Sun Microsystems. JavaBeans web page. http://java.sun.com/products/javabeans/.

[23] R. V. Ommering, F. V. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.

[24] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPM Center, 2006.

[25] School of Computer Science Department of Carnegie-Mellon University. UniCon. `http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/`.

[26] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković. Integration of extra-functional properties in component models. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE)*, volume 5582, pages 173–190, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A component model for control-intensive distributed embedded systems. In M.R.V. Chaudron and C. Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE)*, pages 310–317. Springer Berlin, October 2008.

[28] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *International Conferene in Software Engineering Workshop on Studies of Software Design*, pages 17–32, 1993.

[29] M. Shaw. Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1):27–38, 1995.

[30] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[31] P. Velasco Elizondo. Systematic and automated development with reuse, 2009. `http://www.cimat.mx/~pvelasco/exo/exotool_en.html`.

[32] P. Velasco Elizondo and K.-K. Lau. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 83(7):1165–1178, 2010.

[33] M. Winter, T. Genssler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arvalo, P. Mller, C. Stich, and B. Schnhage. Components for embedded software – the PECOS approach. In *Proceedings of the 2nd International Workshop on Composition Languages*, pages 19 – 26, June 2002.