Deriving Functional Interface Specifications for Composite Components

Perla Velasco Elizondo[1] and Mbe Koua Christophe Ndjatchi[2]

[1] Centre for Mathematical Research (CIMAT), Zacatecas, Zac., 98060, Mexico
[2] Polytechnic University of Zacatecas (UPZ), Fresnillo, Zac., 99059, Mexico

Abstract. An interface specification serves as the sole medium for component understanding and use. Current practice of deriving these specifications for composite components does not give much weight to doing it systematically and unambiguously. This paper presents our progress on developing an approach to tackle this issue. We focus on deriving functional interface specifications for composite components, constructed via composition operators. In our approach, the composites' interfaces are not generated in an *ad hoc* manner via delegation mechanisms, but are derived systematically, consistently and largely automatically via a set of functions on the functional interfaces of the composed components. Via an example, we illustrate the aforementioned benefits as well as the fact that our approach provides a new view into the space of interface generation.

1 Introduction

A *component's interface specification*, or a component specification for short, defines the component and serves as the sole medium for its understanding and use by answering questions such as: What services are provided and required 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 about the information elements that a component specification should include [5,1]: (i) the *instantiation* mechanisms, (ii) the *functional* properties, i.e. the component's provided and required services, (iii) the *non-functional* properties, i.e. the component's quality attributes and (iv) the *context dependencies*, i.e. the information about the component's deployment environment. Independently from their form, all these elements are crucial for setting up and validating a component composition.

The idea of constructing *composite components* is recognised as a good practice in component-based development (CBD) because it is a means to maximise reuse [6]. By composite components we mean *reusable general-purpose* components made up of an assembly of two or more atomic components.[1] The issue of being a composite should be transparent for a user, as the composite should be utilised in the same manner as an atomic one. Thus, the availability of the specification of a composite component is crucial to allow its reuse, as it is the ability to consistently derive it to scale the development techniques of any CBD approach.

Ideally, the information of a composite's specification should be derived from the specifications of its constituents and the semantics of their composition [11,6]. Unfortunately, current practice of deriving specifications for composite components does not

[1] We consider an atomic component the most basic kind of component in a component model.

S. Apel and E. Jackson (Eds.): SC 2011, LNCS 6708, pp. 1–17, 2011.
© Springer-Verlag Berlin Heidelberg 2011

give much weight to doing so in a more systematic and unambiguous manner. This paper, besides extending our previous work, presents our progress on developing an approach to tackle this issue.

We have introduced an approach to CBD and demonstrated the construction of composite components in practice, e.g. [13]. In this approach composites are constructed via *composition operators*. Previously, the process of composites' interface generation was vaguely outlined, required a lot of human intervention and resulted in interfaces with only one service, which is not natural for the users of the composites. Thus, in this paper we present an approach where the composites' interfaces are derived considering both, the functional specifications of their constituents and the semantics of the operators utilised in their composition. The main contribution in this piece of work is a set of *operator-specific functions* to support an approach for consistently deriving functional interface specifications instead of generating them in an *ad hoc* manner via delegation mechanisms. Via an example we will show that our approach (i) provides a *new view* into the space of interface generation, which increases the number of services offered by a composite, (ii) has simple but formal *algebraic basis*, which makes interface derivation more precise, consistent and systematic and (iii) can be *largely automated*, which mitigates derivation effort.

This paper is organised as follows. In Section 2, we discuss composite components in current CBD approaches. In Section 3, we review the foundations of this work. Next, we present some of the defined functions to generate the functional specification of composite components. In Section 5, we demonstrate the use of these functions via an example. In Section 6, we discuss the benefits of our approach and briefly survey relevant related work. Finally, in Section 7, we state the conclusions and future work.

2 Composite Components in CBD Approaches

The idea of constructing reusable composite components has been recognised as a good practice in CBD. However, it is not a trivial task. We believe that a fundamental issue to enable their generation is the availability of an *algebraic approach to composition*. That is, an approach that when putting components together in some way, results in a new component that preserves some of the properties of its constituents. The lack of such an approach might be the reason why, although component composition is supported, only in some CBD methods it enables the construction of reusable composites [8].

To clarify the former, consider composition in JavaBeans [2,8]. In JavaBeans *beans* (i.e. atomic components) are Java classes which adhere to design and syntactic conventions. These conventions make their storage, retrieval and visual composition possible. When components are composed, the resulting "composition" takes the form of an *adaptor class*.[2] However, this class does not preserve the properties of a bean class; 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 that are closer to our notion of composite components. However, there are still some issues that make it difficult to formalise a method to consistently specify them. For example the Koala component model [2,8], which is used

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

in the consumer electronics domain, supports the construction of composites from pre-existing specifications. Specifications are written in some sort of definition language and can be stored in and retrieved from a repository to be reused for composite component definition. Koala supports the specification of various elements relevant to a component. In the example depicted in Fig. 1 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 shows (a) the `ITuner` interface specification, (b) the `CTunerDriver` component specification –in terms of a set of pre-existing interface specifications (e.g. `ITuner`), (c) the `CTVPlatform` composite component specification –in terms of a set of pre-existing component specifications (e.g. `CTunerDriver`) and (d) the `CTVPlatform` composite's graphical representation in Koala notation.

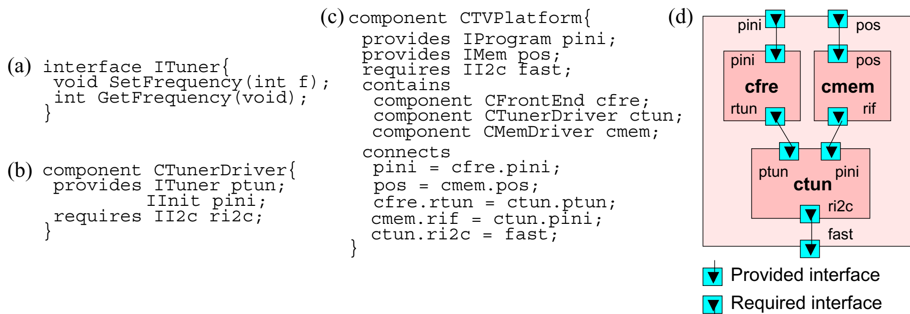


Fig. 1. (a) An interface, (b) an atomic and (c) a composite component specifications and (d) a composite component's graphical representation in Koala

We already mentioned the idea of 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 this is not achieved in Koala at all. The `CTVPlatform` composite is specified in the same manner in which its constituents are (i.e. it defines its functionality in terms of a set of pre-existing interfaces). However, the exposed interfaces `IProgram`, `IMem` and `II2c` result from manually forwarding them, via *delegation mechanisms*, from the inner components to the enclosing one according to the developer's needs.[3] This is in contrast to doing so based on the semantics of the components and their composition.

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. An alternative to generate a highly-reusable composite is that of providing a mean to invoke a number of valid sequences of services offered by its constituents [6]. By adopting a composition approach as the one depicted in Fig. 1, 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 maximising reuse. Note, however, that by

[3] This method follows the semantics of delegation operators in UML 2.0.

forwarding all the constituents' interfaces, which can be a remedy to fix the aforementioned situation, one could violate the composition semantics as it could lead to allow invoking invalid sequences of services.

3 The Foundations of This Work

The composites for which our approach is meant to work are constructed according to the semantics of a new component model [7]. In this model *components* are passive and general-purpose. They 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 *input parameters* and the types of *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. We distinguish between atomic and composite components. The latter are constructed from atomic (or even composite) components via *composition operators*. These operators encapsulate *control-* and *data-flow* schemes; many of them analogous to well-known *patterns*, e.g. [10,4]. The operators are *first-class units*, at both design and implementation time, that admit some sort of parametrisation to indicate the components they compose and the services that must be executed in the composed components.

Fig. 2 shows a system's structure in this component model. In this hierarchical structure, composites can be seen as atomic components and can in turn be a subject of further composition by using another operator (see the inner dotted boxes). As in Koala, the composite's interface elements (e.g. their provided services) are recreated from a lower level to an upper level. However, it is done via a *composition algebra* rather than via traditional delegation mechanisms.

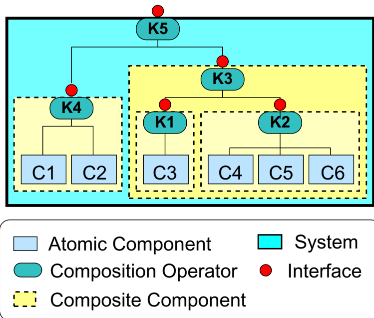
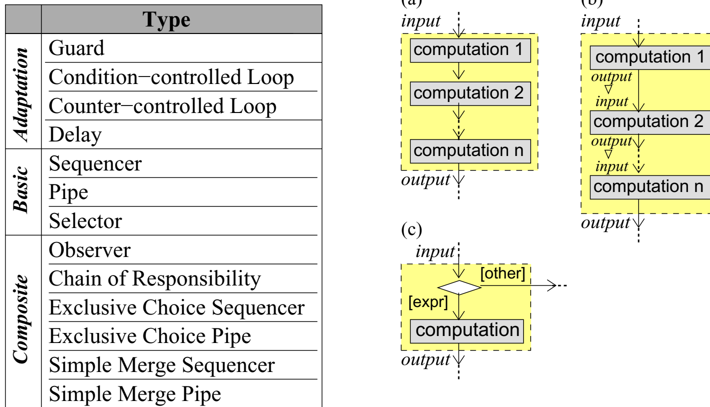


Fig. 2. A system's structure in the new component model

A catalogue of operators to allow component composition within this context has been presented [14]. The table on the left-hand side of Fig. 3 lists the operators in the catalogue. They are organised in (i) *adaptation*, (ii) *basic* and (iii) *composite operators*. Adaptation operators are *unary* operators which adapt the component in the sense that before any computation takes place inside the component, the execution of the control-flow scheme encapsulated by the operator is executed first. Basic composition operators are n-*ary* operators used to support component composition. While basic operators provide only one type of control- and data-flow scheme, composite operators combine many types. These operators have already been implemented and their usefulness demonstrated by constructing a variety of prototype systems, e.g. [13].

Due to lack of space, we do not explain all the operators in the catalogue. However, for clarification purposes, we describe the operators that we will use in the example

in Section 5. The descriptions are in terms of the notation on the right-hand side of Fig. 3. The dotted boxes represent the resulting assemblies. The computation boxes represent the computation in the composed/adapted component. Arrows represent the control-flow. Data required in the assembly to perform the corresponding computation is denoted as the *input* label, while the assembly computation result is denoted as the *output* label.



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

Fig. 3. The catalogue of operators and the behaviour of the assemblies resulting from the (a) *Sequencer*,(b) *Pipe* and (c) *Guard* operators

Both the *Sequencer* and *Pipe* composition operators can be used to compose two or more components, so that the execution of a service in each one of them is carried out in a *sequential* order, see Fig. 3 (a) and (b). The *Pipe* operator also models internal data communication among the composed units, so that the *output* generated by the execution of a component's service is the *input* to the next one in the chain. Fig. 3 (c) depicts the case of the *Guard* adaptation operator. Any computation in the adapted component is conditional upon the value of a Boolean expression (expr) being *true*.

Now that we have defined the generalities of our previous work, let us focus on our approach to derive functional interface specifications.

4 The Proposed Approach

We have outlined a new view of composition where operators are utilised to compose 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 service sequences in terms of the services offered by the composed components. The nature and number of all possible sequences (i.e. the composite's services) should be determined from both the functional specifications of the atomic components and the semantics of the operator utilised in their composition.

The resulting composite should provide a functional specification informing about these service 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 valid service sequences to invoke within the composite's constituents and their corresponding requirements and outcomes (i.e. their input and output parameters).

The composition algebra in our composition approach makes it easier to develop a set of operator-specific functions to specify how to generate the composites' functional specifications as outlined above. The semantics of these functions is based on algebra of sets and first-order predicate logic. Next, we introduce some of these such functions.[4]

4.1 Basic Formalism and Assumptions

In Section 1 we stated that 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 $\langle InParam, OutParam \rangle$ 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 a component's functional specification ($FSpec$), i.e. a set of operation signatures: $Sig == \langle InParam, OutParam \rangle$ and $FSpec == \mathbb{P}\, Sig$.

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 $\langle InParam, OutParam \rangle$ as long as these tuples are associated to different operation names.

In this basic formalism we will treat $Param$ and its partitions 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 for water, one for milk and one 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. When composing these specifications via certain type of operators, it could be required to have multiple occurrences of the same parameter. Note, however, that we assume that operation signatures are well formed meaning: $Sig == \langle InParam, OutParam \rangle$ such that $InParam \cap OutParam = \oslash$. We also assume that parameters with the same name and type are semantically equivalent.

Considering the former, we have defined a set of helper and operator-specific functions to derive the functional specifications of composite components. Helper functions perform part of the computation in operator-specific ones. In this paper we only present the functions utilised in the example in Section 5. However, detailed descriptions of all the defined functions can be found in [15].

[4] We assume basic knowledge of set theory, first-order predicate logic and the Z language syntax.

4.2 The Helper Functions

The function parameter complement ($param_comp$) maps a group of parameters to their complementary *role*, i.e. either input or output. On the other hand, the functions signature input parameter and signature output parameter (sig_in and sig_out respectively) get the input and output parameters of an operation signature respectively.

$$param_comp : Param \rightarrow Param$$

$$param_comp = p_1, p_2, \ldots, p_n \in Param \bullet \bigcup_{i=1}^{n} \sim p_i$$

$$sig_in, sig_out : Sig \rightarrow Param$$

$$s : Sig \bullet sig_in(s) = InParam \wedge sig_out(s) = OutParam$$

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 ones. To specify the issue of having duplicated elements in $InParam$ and $OutParam$, we use the $\uplus$ operator.

$$sig_concat : Sig \times \ldots \times Sig \rightarrow Sig$$

$$sig_concat = s_1, s_2, \ldots, s_n : Sig \bullet \langle \biguplus_{i=1}^{n} sig_in(s_i), \biguplus_{i=1}^{n} sig_out(s_i) \rangle$$

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

$$add_in, add_out : Param \times Sig \rightarrow Sig$$

$$\begin{aligned} & p : Param;\ s : Sig \bullet \\ & \quad add_in(p, s) = \langle \{p \cup sig_in(s)\}, sig_out(s) \rangle \wedge \\ & \quad add_out(p, s) = \langle sig_in(s), \{p \cup sig_out(s)\} \rangle \end{aligned}$$

$$sig_match : Sig \times Sig \rightarrow Boolean$$

$$sig_match = s_1, s_2 : Sig \bullet sig_out(s_1) \cap sig_in(s_2) \neq \emptyset$$

$$sig_bound : Sig \times \ldots \times Sig \rightarrow Sig$$

$$\begin{aligned} & sig_bound = s_1, s_2, \ldots, s_n : Sig \bullet \\ & \quad \langle \{ sig_in(s_1) \ \uplus \\ & \quad (sig_in(s_2) \setminus param_comp(sig_out(s_1))) \ \uplus \\ & \quad (sig_in(s_3) \setminus param_comp(sig_out(s_2))) \ \uplus \\ & \quad \ldots \ \uplus \\ & \quad (sig_in(s_n) \setminus param_comp(sig_out(s_{n-1}))) \}, sig_out(s_n) \rangle \end{aligned}$$

Now that we have presented the helper functions, next we present the operator-specific ones.

4.3 The Operator-Specific Functions

The $guard_composite_fspec$ function generates the functional specification of an assembly created via a *Guard* operator. Besides the functional specification of the adapted component, this function also takes one input parameter, which represents the value to be evaluated by the *Guard*'s Boolean expression. 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.

$$guard_composite_fspec : InParam \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} guard_composite_fspec = \\ \quad s_1, s_2, \ldots, s_n : Sig; \\ \quad f : FSpec; \\ \quad p : InParam \mid \#p = 1; \\ \quad (s_1, s_2, \ldots, s_n) \in f \bullet \bigcup_{i=1}^{n} add_in(p, s_i) \end{array}$$

The $seq_composite_fspec$ function generates the functional specification of a composite component created via the *Sequencer* operator. The helper function sig_concat, makes each operation signature contain the input and output parameters of the participating signatures. Finally, the $pipe_composite_fspec$ function generates the functional specification of a composite component created via the *Pipe* operator. The helper functions sig_match and sig_bound verify that the signatures in the participating specifications meet the requirements for internal data communication and remove the occurrences of the complementary parameters in the resulting signatures respectively.

$$seq_composite_fspec : FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} seq_composite_fspec = \\ \quad s_1, s_2, \ldots, s_n : Sig; \\ \quad f_1, f_2, \ldots, f_n : FSpec; \\ \quad (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \bullet \\ \qquad \bigcup_{(s_1,\ldots,s_n) \in \prod_{i=1}^{n} f_i} sig_concat(s_1, s_2, \ldots, s_n) \end{array}$$

$$pipe_composite_fspec : FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$\begin{array}{l} pipe_composite_fspec = \\ \quad 1 \leq i < j \leq n; \\ \quad s_1, s_2, \ldots, s_n : Sig; \\ \quad f_1, f_2, \ldots, f_n : FSpec; \\ \quad \forall\, s_i, s_j \in (s_1, s_2, \ldots, s_n) \in f_1 \times f_2 \times \ldots \times f_n \mid sig_match(s_i, s_j) \bullet \\ \qquad \bigcup_{(s_1,\ldots,s_n) \in \prod_{i=1}^{n} f_i} sig_bound(s_1, s_2, \ldots, s_n) \end{array}$$

Next we present the design of some composite components by using these functions.

5 Example

We will define some composites meant to be used to construct different versions of a Drink Vending Machine system (DVM). We chose this simple and small size example as it is enough to illustrate the use of our functions.[5] The DVM is limited to two general actions: (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.

Fig. 4 shows the proposed composites to use in the DVM as well as their behaviours. Fig. 4 (a) shows a Coffee Card Cashier composite, which is made of the Card Reader (CR) and Billing Component (BC). CR is responsible for getting coffee cards' identifiers and BC for debiting the cards. By composing these components with a *Pipe* P and a *Guard* G operators we can generate a composite that, once a coffee card has been inserted in the coffee machine's slot and the amount to debit to it has been specified (e.g. *amt*), it retrieves the card's identifier by executing a service in CR (e.g. getCardId). The obtained result can be passed up via P to G to check its value. If it has a valid value (e.g. *if cardId* $! =$ *null*), then the card can be debited by executing a service in BC (e.g. debit) and the result can be returned (e.g. *errorCode*).

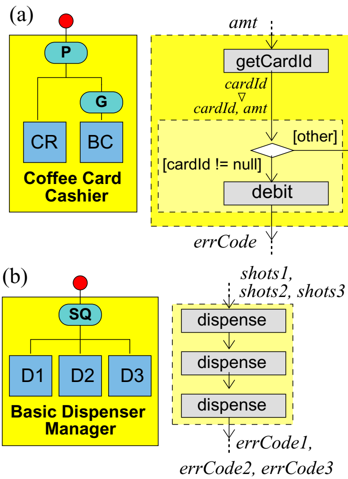


Fig. 4. Useful composites for the Drink Vending Machine systems and their behaviour

The Basic Dispenser composite, shown in Fig. 4 (b), is made up of three instances of the dispenser component and one *Sequencer* operator SQ. A Water (D1), a Coffee (D2) and a Milk Dispenser (D3) have been considered. The composite allows the sequential execution of one service in each one of these components, e.g. the dispense service. *Shots1-shots3* denote the number of shots to be dispensed by each dispenser, while *errorCode1-errorCode3* denote the resulting values of each one of these executions.

Now we describe how we use these composites in the DVM system. We have organised the DVM design into three subsystems: a *Cashier*, a *Drink Maker* and a *Maintenance*. The first two will deal with the function (1) and the last one will support the function (2). Fig. 5 (a) shows a version of the VDM in terms of the three subsystems. 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* SL will call the *Pipe* P3 and it in turn will call the *Cashier Subsystem* to deal with the drink payment. In this subsystem, the *Pipe* P1 will first retrieve the drink's price by executing a service in PMgr. Then, it will pass up the price when calling the required service in

[5] Composites providing more sophisticated services can be seen in [13,15].

the Coffee Card Cashier composite. Next, the *Pipe* P3 will pass up the result to the *Drink Maker Subsystem*. In this subsystem, the *Guard* G1 will allow any computation down on the hierarchy only if the drink payment has been processed. In such a case, this operator will call the *Pipe* P2. P2 will first retrieve the drink's recipe by executing a service in 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 the *Selector* SL will call the corresponding Basic Dispenser's service.

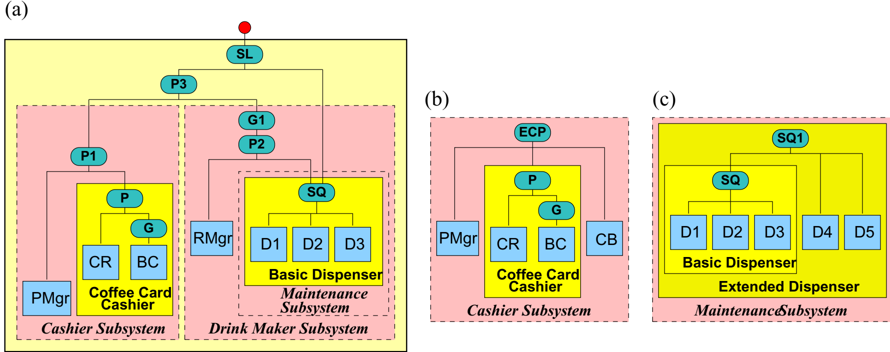


Fig. 5. The outlines of three alternative designs of the Drink Vending Machine system

Fig. 5 (b) shows a new version of the *Cashier Subsystem* to allow the customer to pay for drinks by using either coffee cards or cash. The first case is supported by the Coffee Card Cashier composite, while the second one is supported by the Coin Box component (CB). We use an *Exclusive Choice Pipe*[6] operator ECP to retrieve the drink's price from PMgr and then, based on the payment method selected by the customer, direct the execution of the charge to either the Coffee Card Cashier or CB. Finally, Fig. 5 (c) shows how the Basic Dispenser composite can be further composed to create an Extended Dispenser composite. This new composite includes the additional dispenser instances D3 and D4 for dealing with more ingredients. For space reasons, in Fig. 5 (b) and (c) we did not depict the complete DVM designs, but we want to highlight the fact that these new subsystems can replace the ones in Fig. 5 (a) to create new versions of the DVM.

5.1 Composite Component Generation

For clarity purposes, in Fig. 6 we describe the functional specifications of atomic components in some sort of IDL[7] syntax. The keywords `in` and `out` denote the *role* of a parameter in the operation. Next, we describe these specifications using the formalism described in Section 4.1.

Functional Specification of the Coffee Card Cashier Composite. Attending the bottom-up nature of our composition approach, we will start with the assembly involving the *Guard* operator (G) and the Billing Component (BC). Let $p = \{cardId\}$ be

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

[7] Interface Definition Language.

```
interface Dispenser{
  emptyDispenser ();
  setTemperature (in int temp, out errCode);
  add (in int shots, out int errCode);
  dispense (in int shots, out int errCode);
}

interface CardReader{
  getCardId(out int cardId);
}

interface BillingComponent{
  credit(in int cardId, in int amt, out errCode);
  debit(in int cardId, in int amt, out errCode);
  getBalance(in int cardId, out int amt);
}
```

Fig. 6. Functional specifications of atomic components

(a)

```
interface BasicDispenser{
 op1();
 op2(in int temp, out errCode);
 op3(in int shots, out int errCode);
 op4(in int shots, out int errCode);
 ...
 op63(in int shots, in int shots, in int temp,
      out int errCode, out int errCode, out int errCode);
 op64(in int shots, in int shots, in int temp,
      out int errCode, 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);
}
```

(b)

```
interface CoffeeCardCashier{
 op1(in int amt, out errCode);
 op2(in int amt, out errCode);
 op3(out int amt);
}
```

Fig. 7. The interface specifications of (a) the Basic Dispenser and (b) the Coffee Card Cashier

the input parameter to be evaluated by G's Boolean expression, i.e. "*in int cardId*". Let $f_1 = \{\langle\{cardId, amt\}, \{errCode\}\rangle, \langle\{cardId, amt\}, \{errCode\}\rangle, \langle\{cardId\}, \{amt\}\rangle\}$ be the functional specification of BC. By using the *guard_composite_fspec* function we can derive the functional specification
$f_2 = \{\langle\{cardId, amt\}, \{errCode\}\rangle, \langle\{cardId, amt\}, \{errCode\}\rangle, \langle\{cardId\}, \{amt\}\rangle\}$
The f_2's tuples denote the signatures of the "guarded" versions of the *debit*, *credit* and *getBalance* operations in the BC composite. As p's and f_1's $cardId$ are semantically equivalent, the *Guard*'s *add_in* helper function kept it in only one occurrence. However, within the assembly the parameter is utilised as the variable to be evaluated in G's Boolean expression (e.g. *if cardId* $! =$ *null*) and as the input parameter of the operation signatures.

Once f_2 has been obtained, it can be used together with the functional specification of the Card Reader component (CR) to generate the functional specification of the Coffee Card Cashier composite via the *pipe_composite_fspec* function. Thus, let [8]
$f_1 = \{\langle\varnothing, \{cardId\}\rangle\}$
and f_2 be the CR and the guarded BC specifications respectively, we can derive
$f_3 = \{\langle\{amt\}, \{errCode\}\rangle, \langle\{amt\}, \{errCode\}\rangle, \langle\varnothing, \{amt\}\rangle\}$
which represents the Coffee Card Cashier's functional specification. Using the IDL syntax introduced before, this functional specification can be rewritten as shown in Fig. 7 (b). In here, $op1$-$op3$ are signatures abstracting the three valid sequences of operations to invoke within the composite's constituents, i.e. $getCardId$-$credit$, $getCardId$-$debit$

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

and $getCardId$-$getBalance$. Note that, both the input and the output parameters of these operation sequences are entirely derived from the semantics of the *Pipe* operator. The helper function sig_bound is utilised to remove the input parameter $cardId$ in the resulting signatures, as it is produced internally within the composite, see Fig. 4 (a).

Functional Specification of the Basic Dispenser Composite. Let f_i, *i*=1,2,3, be the functional specifications of the three Dispenser's instances:

$$f_i = \{\langle\varnothing,\varnothing\rangle, \langle\{temp\},\{errCode\}\rangle, \langle\{shots\},\{errCode\}\rangle, \langle\{shots\},\{errCode\}\rangle\}$$

Applying the $seq_composite_fspec$ function, we can derive the one of the Basic Dispenser composite component:

$$\begin{aligned} f_4 = \{&\langle\varnothing,\varnothing\rangle,\\ &\langle\{temp\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle,\\ &\langle\{shots\},\{errCode\}\rangle,\\ &\ldots,\\ &\langle\{shots, shots, temp\},\{errCode, errCode, errCode\}\rangle,\\ &\langle\{shots, shots, shots\},\{errCode, errCode, errCode\}\rangle,\\ &\langle\{shots, shots, shots\},\{errCode, errCode, errCode\}\rangle\} \end{aligned}$$

The f_4's signatures abstract the valid sequences of operation executions within the composite's constituents. The IDL version of f_4 is shown in Fig. 7 (a). The $op1$ and $op64$ abstract the execution sequences $emptyDispenser$-$emptyDispenser$-$emptyDispenser$ and $dispense$-$dispense$-$dispense$ respectively. The input and output parameters of these signatures are entirely derived from the semantics of the *Sequencer* operator. Each one of the signatures in the resulting specification is made of a concatenation of the parameters of the composed components' signatures via the sig_concat helper function. The duplicated elements in the resulting $InParam$ and $OutParam$ sets are 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 (b).

5.2 Automation and Tool Support

We have generated a new version of our existing composition tool[9] that includes a set of algorithms that implement the defined functions. The tool operates on atomic components that are offered as *binary* files. The binaries correspond to a Java implementation and relate to a functional interface specification written as Java annotations. During component composition, the annotations (together with other component classes' metadata) are read via *reflection techniques*, to determine the content of components' functional specifications and derive the ones for the composites. Composites' specifications are attached to their implementations as Java annotations.

Fig. 8 shows a screenshot of the tool during the generation of the Basic Dispenser composite (described in Section 5.1). As can be seen, the names of the resulting signa-

[9] The tool enables component composition by dragging, dropping and composing pre-existing components and pre-existing operators into a visual assembler. The generation of Java code for the defined assemblies is also supported, see [13].

tures are generated by concatenating the names of the composed ones, e.g. *dispensedispensedispense*. For naming the input and output parameters, the tool uses the ones in the participating signatures. Although having duplicated elements in either the $InParam$ or the $OutParam$ sets is semantically valid in our approach, it is syntactically invalid at implementation-time, i.e. method signatures in Java cannot have duplicated parameter names. Thus, when this happens a suffix to the original ones is added to differentiate them, e.g. *shots1*, *shots2* and *shots3*.

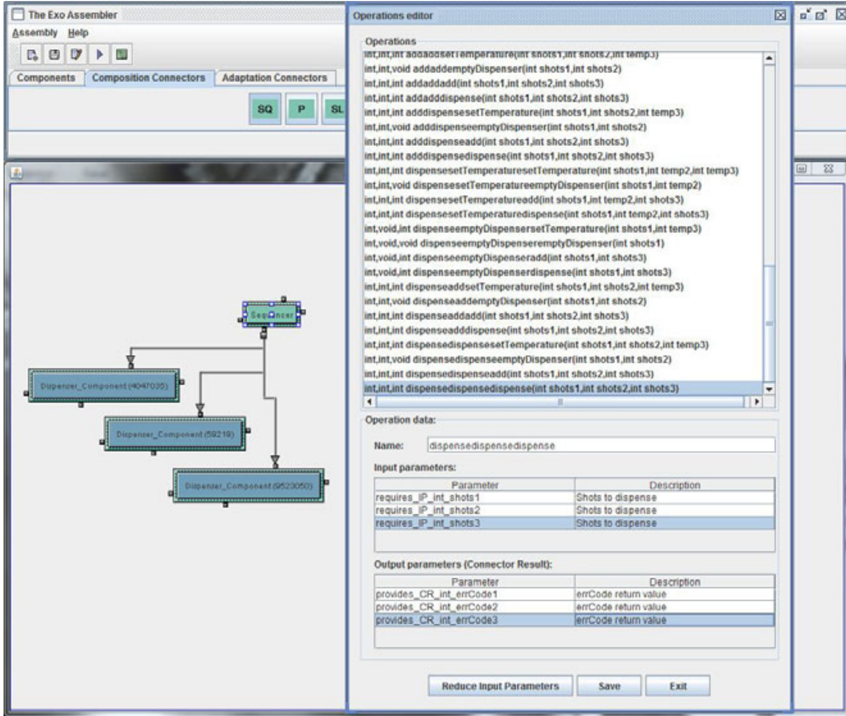


Fig. 8. A visual tool to support the generation of composite components.

Our tool largely automates the functional interface derivation. However, in some cases composite developer intervention could be needed to refine the content and syntax of the resulting specifications. For example, to eliminate signatures representing execution sequences that are invalid or undesirable, the composite developer has to provide the filtering criteria. Taking into consideration the domain context for which the Basic Dispenser composite 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, e.g. *dispense-dispense-dispense*. Similarly, to allow a better understanding to composite users, the composite developer might want to rename the resulting operations or parameters. The tool provides the means to allow this and some other refinements, e.g. see the lower part of Fig. 8 which allows the composite developer to rename operations and parameters.

6 Discussion and Related Work

Management of both functional and non-functional properties is one of the main challenges in CBD community. Despite the former, the starting point in their management, that is their specification, is not addressed completely in most CBD approaches [11]. With few exceptions, current practice in CBD focuses on components and leaves their interactions specified as lines representing method calls. As illustrated in Section 2, such lines are not enough for defining a method for systematically, consistently and automatically deriving the specifications of component assemblies. We have presented our progress on developing an alternative approach to tackle this issue. Even though we only focus on deriving functional properties, our approach presents differences with respect to related work. Specifically, our approach (i) provides a *new vision* for deriving the functional properties, which enables increasing the number of services offered by a composite, (ii) has simple but formal *algebraic basis*, which makes interface derivation more precise, consistent and systematic and (iii) can be *largely automated*, which mitigates interface derivation effort. Next we justify these claims.

During all these years several component models have been proposed. However, not all of them support the construction of composite components. About half of the models surveyed in [2,8] support them: AUTOSAR, BIP, BlueArX, Fractal, Koala, KobrA, MS COM, Open COM, PECOS, ProCom, SaveCCM, SOFA2.0.[10] In all these models, functional interfaces of composite components are derived by delegating "some" of the functionality from their subcomponents in an *ad hoc* manner rather than by consistently deriving them based on the semantics of the composition. Although this approach has demonstrated to be good enough for composite developers, its *ad hoc* nature make it require much intervention from the composite developer.

From the models listed above, only BIP, Fractal, ProCom and SOFA2.0 support explicit composition mechanisms. However, the nature of these composition mechanisms *is not algebraic* in the sense that when applied to units of composition of a given type, the resulting piece is not a unit of the same type that can be stored and further composed. We have demonstrated that having algebraic composition mechanisms facilitates the development of more precise, consistent, systematic and automated approaches to derivate composites and their functional interfaces. As a corollary of the algebraic basis of our approach, the proposed functions can be composed. That is, the ones defined for basic operators can be reused in the definitions for the composite ones. For example, the $Observer$ is a composite composition operator made up of one *Pipe* and one *Sequencer* operators. As can be seen, its corresponding function

$$obs_composite_fspec : FSpec \times \ldots \times FSpec \rightarrow FSpec$$

$$obs_composite_fspec = f_1, f_2, \ldots, f_n : FSpec \bullet$$
$$pipe_composite_fspec(f_1, seq_composite_fspec(f_2, \ldots, f_n))$$

uses both the $pipe_composite_fspec$ and the $seq_composite_fspec$ functions accordingly. Thus, we could say that we support some sort of interface composition. This observation leads us to the point of recognising that our work shares some principles and

[10] Note that not all the generated composites in these models are meant to be reusable –e.g. in BIP, Fractal and PECOS there is not a notion of repository where the generated composites can be stored to and retrieved from.

goals with algebraic composition mechanisms such as *parameterised modules* (a.k.a. functors) and *traits* [3].

In some functional languages (e.g. ML [12]), a module is characterised by a signature (i.e. an interface specification) and a structure (i.e. an implementation of the signature). A parameterised module denotes a function from structures to structures; that is, it accepts one or more structures of a given signature, and produces a new structure that implements a particular signature. In contrast to our approach, the signature of the new structure is chosen by the programmer in an *ad hoc* manner rather than derived from the parameterised module's arguments. On the other hand, traits are essentially groups of provided and required methods that serve as building blocks for classes by providing first-class representations of the behaviours of a class. Traits can be algebraically composed via the *sum* operator which, generally speaking, has the semantics of some sort of structural union. Thus the number of the behaviours (i.e. methods) in the composite trait is the "sum" of the behaviours in all the composed traits. In our approach, the number of the behaviours is derived differently as each behaviour abstracts a particular form of coordination between the behaviours of multiple components. Thus our approach supports a new notion for deriving the structure and behaviors of a composite.

Our proposal has some drawbacks though. In the example presented in Section 5, 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. Although by using our tool a composite developer can filter the number of signatures of the resulting specification by keeping some execution sequences away, it could be desirable to automatically support this filtering by using some sort of behavioural information, e.g. assertions or behaviour protocols.

Additionally, we are dealing with stateless components and we are not considering specifications containing assertions on operations. We are exploring the feasibility of using some formalism in algebraic specification languages to deal with state and assertion composition under the semantics of our composition operators. For example, in the Z specification language [9] schemas specify the state of a module as well as relevant operations on them. Z schemas have a name, a declaration part and a predicate part. The name part gives a name to the aspect of the the module being specified. The declaration part defines a number of variables of certain types. The predicate part defines invariants on these variables. The schema calculus of Z allows extending the schemas and combining them by using logical connectives such as conjunction, disjunction and implication. Conjunction and implication between schemas are defined by combining the declaration parts and taking in conjunction or implication the predicate parts. Schemas can be composed sequentially, which implies that the after-state of the first schema is to be matched with the before-state of the second.

7 Conclusions and Future Work

When the construction of reusable composite components is supported, the ability to derive their interface specifications is crucial to scale the development techniques of any CBD approach. We presented our progress on developing an approach to support

this issue. Specifically, we focused on the generation of composites' functional specifications. The composites are constructed via composition operators defined within the context of a new component model. The specification approach is based on a set of operator-specific functions, which allow deriving composites' functional specifications in a systematic, consistent and largely automatic manner. Via an example we have illustrated the aforementioned benefits as well as the fact that our approach provides a new view into the space of interface generation.

In the near future, we plan to extend our approach to deal with more sophisticated behavioural information. Similarly, we have started to work on a similar approach to derive other elements from the composites' interfaces, i.e. the non-functional properties and the information about the deployment environment. So far it seems feasible to derive directly composable non-functional properties as described in [2].

By doing this research we hope to gain more understanding on software composition and its automation, which is the ultimate goal not only for CBD, but also for some other component-based development paradigms such as service composition and software product lines.

References

1. Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C.: What characterizes a (software) component? Software - Concepts and Tools 19(1), 49–56 (1998)
2. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. IEEE Trans. on Software Engineering (2010) (pre-Prints)
3. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. on Prog. Languages and Systems 28, 331–388 (2006)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading (1995)
5. Geisterfer, C.J.M., Ghosh, S.: Software component specification: A study in perspective of component selection and reuse. In: Proc. of the 5th Int. Conf. on Commercial-off-the-Shelf (COTS)-Based Software Systems. IEEE Computer Society, Los Alamitos (2006)
6. Lau, K.-K., Ling, L., Velasco Elizondo, P.: Towards composing software components in both design and deployment phases. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 274–282. Springer, Heidelberg (2007)
7. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
8. Lau, K.-K., Wang, Z.: A survey of software component models. Preprint CSPP-38, School of Computer Science, The University of Manchester (May 2006)
9. Potter, B., Till, D., Sinclair, J.: An Introduction to Formal Specification and Z, 2nd edn. Prentice Hall PTR, Upper Saddle River (1996)
10. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPM Center (2006)
11. Sentilles, S., Štěpán, P., Carlson, J., Crnković, I.: Integration of extra-functional properties in component models. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 173–190. Springer, Heidelberg (2009)

12. Ullman, J.D.: Elements of ML programming. Prentice-Hall, Inc., Upper Saddle River (1998)
13. Velasco Elizondo, P.: Systematic and automated development with reuse (2009), http://www.cimat.mx/~pvelasco/exo/exotool_en.html
14. Velasco Elizondo, P., Lau, K.-K.: A catalogue of component connectors to support development with reuse. Journal of Systems and Software 83(7), 1165–1178 (2010)
15. Velasco Elizondo, P., Ndjatchi, M.K.C.: Functional Specification of Composite Components. Technical Report I-10-05/24-06-2010(CC/CIMAT), Centre for Mathematical Research (June 2010), http://www.cimat.mx/reportes/enlinea/I-10-05.pdf