A Taxonomy of Software Composition Mechanisms*

Kung-Kiu Lau and Tauseef Rana
School of Computer Science, The University of Manchester,
Manchester M13 9PL, UK
{kung-kiu,trana}@cs.man.ac.uk

Abstract—**Software composition aims to provide mechanisms for systematic construction based on well-defined software units. Various software composition mechanisms have been defined in the literature for different kinds of software units. In component-based development, it is desirable to have software units and composition mechanisms that support automated, systematic construction. In this paper, we first survey existing definitions of composition units and the corresponding composition mechanisms, and then use the survey to propose a taxonomy that identifies good candidates for composition units and composition mechanisms for component-based development.**

I. Introduction

Software composition [34] refers to the composition of software constructs into larger composite constructs. The primary motivation for software composition is reuse [43], but composition also provides a means for systematic software construction. For Component-based Software Development (CBD) [22], [49], composition is of the essence, since components, by definition, are units of composition [47], [48]. For CBD, software reuse is of course a fundamental objective, in order to reduce production cost; however, in addition, CBD also seeks to automate composition as much as possible, so as to reduce time-to-market as well.

In the most general terms, composition can be defined as any possible and meaningful interaction between the software constructs involved. A composition mechanism defines such an interaction. Clearly there are many different possible kinds of software constructs, with corresponding composition mechanisms [10], [35], [45], [43], [25], [48], [49], [4], [18], [7], [41], [37]. Simple type definitions can be composed into compound types by type composition [12]; arbitrary chunks of code can be joined together with glue and scripts [44]; typed constructs can be linked by message passing, e.g. direct method calls between objects, or port connections between architectural units [45], [8]; and so on.

In CBD it is desirable to have software constructs that make good composition units [40], together with suitable composition mechanisms that facilitate both reuse and systematic construction [2] and are also automatable. In this paper we first survey existing composition mechanisms for various kinds of software units, and then use the survey to propose a taxonomy that identifies good candidates for composition units and composition mechanisms, for defining software component models [22], [29], that support automated, systematic composition in CBD.

*This is a revised version of the paper that appeared in the proceedings of Euromicro SEAA 2010.

II. Different Views of Software Composition

In order for our survey to be as inclusive as possible, we consider all the views of software composition found in the literature, that is, the various perceptions (and definitions) of what composition means in all the relevant software communities. In any view of composition, composition is performed on software entities that are perceived as meaningful *units of composition*. We will focus on units of composition that define *behaviour*, rather than constructs that define primitive types or pure data structures. Composition mechanisms compose units of composition into larger pieces of software, i.e. they compose pieces of behaviour into larger pieces of behaviour. In this section, we outline the different views of composition and briefly discuss the generic nature of the associated units of composition and composition mechanisms.

A. The Programming View

One view of software composition is that it is simply what a programmer does when putting bits of code together into a program or an application. In this view, any legitimate programming language construct is a unit of composition; and composition is simply joining these constructs together using some other construct (e.g. sequencing) defined in the programming language. We call this the 'programming view' of composition.

Meaningful units of composition in the programming view include *functions* in functional languages, *procedures* in imperative languages, and *classes* [48] and *aspects* [25] in object-oriented and aspect-oriented languages.

Clearly the 'programming view' represents programming-in-the-small. To equate composition with this view, however, is to overlook many issues that are significant for software engineering, such as *reuse* and *systematic* or *automated construction*.

B. The Construction View

A higher-level view of composition is the view that software composition is "the process of constructing applications by interconnecting software components through their plugs" [33]. The primary motivation here is *systematic construction*.

We call this view the 'construction view' of composition. It is at a higher level of abstraction than the 'programming view': it typically uses *scripting languages* [39] to connect

pre-existing program units together. The 'construction view' thus represents programming-in-the-large [17], as opposed to programming-in-the-small.

In the 'construction view', the units of composition are referred to as components, but these are only loosely defined as software units with plugs, which are interaction or connection points. Consequently, components may be any software units that can be scripted together by glue. For example, components may be modules glued by module interaction languages [42], or Java Beans composed by Piccola [1], and so on.

System designs in the 'construction view' are represented by software architectures [45], [8]. A software architecture contains components and their inter-connections.

Although the 'construction view' hints at software reuse (via components) [32], [34], [43], it does not explicitly show how reuse occurs. In particular, it does not assume that components are supplied by third parties (and pre-exist in a repository). Software architectures similarly do not make any assumptions about component reuse.

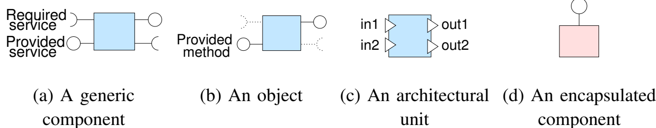
C. The CBD View

To define components precisely, we should define them in the context of a *component model* [22], [29]. A component model defines what components are (their syntax and semantics) and what composition operators can be used to compose them. Thus in [22] a software component is defined as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard".

The advent of CBD [11], [22], [49] brought about a sharper focus on not only component models (different kinds of components and composition mechanisms), but also repositories of (pre-existing) components and component reuse from such repositories. Thus CBD is motivated by systematic construction as well as *reuse* of *(pre-existing) third-party components*. We call this the 'CBD view'; it extends the 'construction view', by the additional emphasis on component models as well as reuse of third-party components.

Software architectures also subscribe to the 'CBD view', in addition to the 'construction view', in the sense that an architecture description language (ADL) [16], [30] could be considered to be a component model, with architectural units as components, and port connection as a composition mechanism for such components. However, in contrast to the 'CBD view', software architectures do not always assume or make use of third-party components or repositories of such components, as we remarked earlier.

In the 'CBD view', units of composition are components as defined in the chosen *component model*. A generic component is a unit with *provided* and *required* services (Fig. 1(a)).



(a) A generic component (b) An object (c) An architectural unit (d) An encapsulated component

Fig. 1. Components in current component models.

In current component models [29], components are objects (as in object-oriented programming), architectural units (as in ADLs), or encapsulated components [28] (with no required services). Objects have provided services (their methods) but do not specify their required services (Fig. 1(b)). Architectural units have their *input* and *output* ports as required and provided services respectively (Fig. 1(c)). Encapsulated components encapsulate computation (and control and data) and do not call other components, i.e. they have only provided services but no required services (Fig. 1(d)).

Generic components are composed by matching their required and provided services. Objects cannot be 'composed' this way, since they do not specify their required services; rather, they 'compose' by direct method calls. Architectural units compose by connecting their (compatible) ports. Encapsulated components cannot connect directly; rather they need to be coordinated by exogenous composition connectors [27], [50].

Finally, it is worth re-iterating that the boundaries between these views are not cut and dried. In particular, the construction view and the CBD view overlap, as already pointed out. This is mainly due to the generic nature of components defined in the construction view, which loosely covers components in all the current component models. We will see the overlap again in our survey (Fig. 8) in the next section.

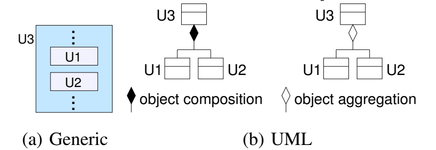
III. A Survey

Now we survey composition mechanisms that have been defined in all three views. As already mentioned, we view a unit of composition as a software unit that defines *behaviour*, and composition mechanisms as ways of building larger units of behaviour. Since it does not make much sense to consider composition mechanisms that are only unary in arity, our normal assumption is that composition mechanisms are (at least) *binary* in arity.

Composition mechanisms in all three views fall into four general categories: (i) containment; (ii) extension; (iii) connection; and (iv) coordination. We now briefly define and explain each category, using generic units of composition, and, for elucidation and illustration, we compare and contrast the category with corresponding UML mechanisms.

A. Containment

Containment refers to putting units of behaviour inside the definition of a larger unit. This is illustrated in Fig. 2(a), where `U3` contains `U1` and `U2`. Containment is thus *nested definition*. The behaviour of the container unit is defined in terms of that of the contained units, but the precise nature of the containment differs from mechanism to mechanism. Examples of containment are nested definitions of functions, procedures, modules and classes, as well as object composition and object aggregation.



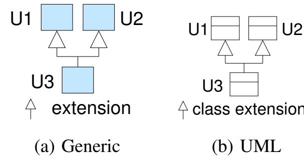
(a) Generic (b) UML

Fig. 2. Containment

Compared to (standard) UML, our notion of containment covers more composition mechanisms. In UML, containment is defined for classes only; there is no notation for nested class definition, and the only forms of containment are object aggregation and object composition (Fig. 2(b)).

B. Extension

Extension refers to defining the behaviour of a unit by extending that of at least two other units of composition. This is illustrated in Fig. 3(a). Examples of extension include multiple inheritance in object-oriented programming, aspect weaving [24] in aspect-oriented programming, subject composition [37] (or correspondence-combination, or superimposition [5]) in subject-oriented programming and feature composition [41] in feature-oriented programming (Fig. 8).



(a) Generic (b) UML

Fig. 3. Extension.

Multiple inheritance can be defined as a composition mechanism that extends multiple classes (e.g. `U1` and `U2` in Fig. 3(a)) into another class (`U3`) that inherits from these classes.

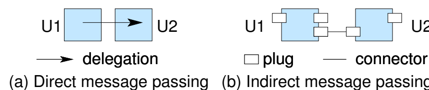
Aspect weaving can be defined as a (binary) composition mechanism that extends a class (say `U1` in Fig.3(a)) and an aspect (`U2`) into another class (`U3`) that is the result of weaving `U2` into `U1`. (Of course `U3` is just the new version of `U1`.)

Similarly, subject composition and feature composition can be defined as composition mechanisms that extend multiple subjects and features respectively (e.g. `U1` and `U2` in Fig.3(a)) into another subject or feature (`U3`) that is the result of superimposition between these subjects or features.

Compared to UML, our notion of extension covers more composition mechanisms. In UML, extension is used to define inheritance for classes only, and the only composition mechanism based on extension is multiple inheritance (Fig.3(b)). Other extension mechanisms, namely aspect weaving, subject composition and feature composition, can only be represented in UML as multiple inheritance if it is acceptable to represent an aspect, a subject or a feature as a class. However, if aspects, subjects and features are to be distinguished from classes, as they are intended to be, in aspect-oriented, subject-oriented and feature-oriented programming, then we cannot define aspect weaving, subject composition and feature composition as composition mechanisms in UML.

C. Connection

Connection refers to defining a behaviour that is an interaction between the behaviours of multiple units. This is illustrated in Fig. 4. This interaction is effected by the units either directly or indirectly invoking each other's behaviour. Connection is thus *message passing*, and as such it induces tight coupling between units that send messages to each other.

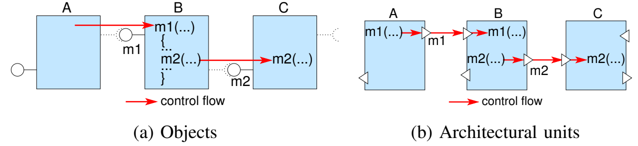


(a) Direct message passing (b) Indirect message passing

Fig. 4. Connection.

Examples of connection include object delegation [38] and port connection between architectural units (Fig. 8).

Direct message passing (Fig. 4(a)) is a form of delegation. An example is object delegation. Objects directly invoke each other's methods, i.e. they connect by direct method calls, or delegation. This is illustrated for three objects A, B, C in Fig. 5(a). In general, an object could call any number of methods in another object. This is true for an arbitrary assembly of connected objects.

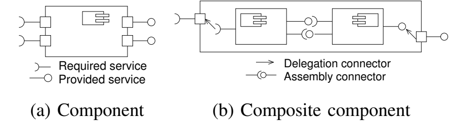


(a) Objects (b) Architectural units

Fig. 5. Connection by direct and indirect message passing.

Indirect message passing (Fig. 4(b)) is done via plugs in the units. Plugs provide input/output points via which units can communicate. An example of indirect message passing is architectural units connected via their ports. An architectural unit has ports, for input and output, which can be linked to the ports of other architectural units by connectors. Architectural units invoke each other's behaviour by messages passed via their ports. Fig. 5(b) shows three units linked via (some of) their ports. Connected ports have to be compatible of course.

Compared to UML, our notion of connection covers more composition mechanisms. In UML, connection is only defined for UML2.0 components, not for classes. In UML2.0, components are architectural units with input ports that are *required services* and output ports that are *provided services* (Fig. 6(a)). Port connection is done by using *assembly* connectors, and port forwarding or exporting is done by using *delegation* connectors, illustrated in Fig. 6(b).



(a) Component (b) Composite component

Fig. 6. Connection in UML.

Somewhat ironically, UML has no notation for object delegation. Association between classes can only express relationships between classes, but not method calls between objects.

D. Coordination

Coordination refers to defining a behaviour that results from coordinating the behaviours of multiple units. This is illustrated in Fig. 7. The coordination is performed by a coordinator which communicates with the units via a control and/or a data channel. The units themselves do not communicate directly with one another. Coordination thus removes all coupling between the units, in contrast to connection, which induces tight coupling through message passing. Examples of coordination are data coordination using tuple spaces [13], data coordination using data connectors [6] for parallel processes or active components, control coordination using orchestration [19] for (web) services, and control coordination

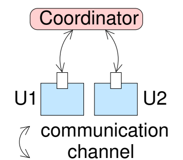


Fig. 7. Coordination.

using exogenous composition for encapsulated components (Fig. 8).

Tuple spaces are used in coordination languages to coordinate parallel processes, by storing and sharing typed data objects (tuples) between the processes. In contrast to connection mechanisms, these processes communicate only with the tuple space, but not directly or indirectly with each other.

Data connectors are data channels that coordinate the data flow between the ports of active components, thus separating the data flow from computation. The components execute their own threads, consuming data values on their input ports and putting data values on their output ports. The components do not communicate directly with each other. The flow of data values is defined by the data channels between them.

In control coordination, control connectors coordinate the control flow between passive components. The components do no have their own threads, and are executed only when control reaches them from the control connectors. Control coordination thus separates control flow from computation.

In UML there is no notion of coordination, and hence no notation for coordination.

E. The Survey

Our survey is structured according to the above four categories (and the three views) and is shown in Fig. 8.

View	Unit of Composition	Composition Mechanism				View
		Containment	Extension	Connection	Coordination	
Programming View	Function	Function nesting		Higher–order function Function call		
Programming View	Procedure	Procedure nesting		Procedure call		
Programming View	Class	Class nesting Object composition Object aggregation	Multiple inheritance	Object delegation		
Programming View	Mixin		Mixin inheritance			
Programming View	Mixin/Class		Mixin–class inheritance			
Programming View	Trait		Trait composition	Trait composition		
Programming View	Trait/Class		Trait–class composition	Trait–class composition		
Programming View	Subject		Subject composition			
Programming View	Feature		Feature composition			
Programming View	Aspect/Class		Weaving			
Programming View	Module	Module nesting		Module connection		Construction View
CBD View	Architectural unit			Port connection		Construction View
CBD View	Fragment box		Invasive composition	Invasive composition		Construction View
CBD View	Process			Channels	Data coordination	Construction View
CBD View	Web service				Orchestration (Control coordination)	Construction View
CBD View	Encapsulated component				Exogenous composition (Control coordination)	Construction View

Fig. 8. A survey of composition mechanisms.

We will explain representative mechanisms in the category, with examples. For lack of space we cannot explain all the mechanisms in detail.

The Containment category contains function nesting, procedure nesting, class nesting, object composition and object aggregation, and module nesting. We will explain object composition and object aggregation in object-oriented programming as representative examples. In object composition, the container object manages the life cycle of the contained objects, i.e. the latter get constructed and destroyed with the former. In contrast, in object aggregation, the life cycle of the contained objects is independent of that of the container object. This is illustrated by the C++ example in Fig. 9.

The `compose` class *composes* two objects of the `contained` class managing the life cycle of two instances (`first, second`). In contrast, the `aggregate` class only

```
class contained
{ ... }
class compose
{ public:
  ...
  private:
    contained first;
    contained second;
};
```

```
class aggregate
{ public:
  ...
  void setContained(contained *,contained *);
  private:
  contained *first;
  contained *second;
};
void aggregate::setContained(contained *c1,contained *c2)
{ first=c1; second=c2;  }
```

Fig. 9. Containment: Composing objects by object composition and object aggregation.

aggregates two objects of the `contained` class, because it only contains pointers to them. Class `aggregate` does not manage the life cycle of instances pointed by (`first, second`).

The Extension category contains multiple (class) inheritance, mixin-inheritance [10], mixin-class inheritance, trait composition [18], trait-class composition, subject composition, feature composition, (aspect) weaving and invasive composition. We choose aspect weaving as a representative mechanism to explain. An aspect [25] defines a crosscutting concern for some base code. It can be woven with the base code to change the latter's behaviour by adding behaviour (*advice*) at various points (*join points*) in the base code specified in a *pointcut* (that identifies matching join points). Weaving is done by an aspect weaving mechanism, which is a special language processor that weaves advices[1] into a class construct. Fig. 10 shows a simple aspect in AspectJ [24] to print out `Entering` before executing the `display` method of any class with any return type, and to print out `Exiting` after executing the method, that is woven with a Java class `application`. The

```
public class application{ ...
  public void display(){
  System.out.println("Mode");
  } ...
}
```

```
Output before weaving
Mode
```

```
public aspect trace{
  pointcut log():
    execution(public * *.display(..));
  before( ) : log( ){ //before advice
    System.out.println("Entering---");}
  after( ) returning : log( ){ //after advice
    System.out.println("Exiting---");}
}
```

```
Output after weaving
Entering---
Mode
Exiting---
```

Fig. 10. Extension: Composing an aspect with a class by aspect weaving.

pointcut `log` specifies the joinpoints as before and after the execution of any `display` method. The aspect `trace` thus extends the behaviour of the class `application` class.

The Connection category contains higher-order function, function call, procedure call, object delegation, trait composition, trait-class composition, module connection, port connection, invasive composition, and (process [23]) channels. We have already explained object delegation and port connection for architectural units (see Fig. 5). Here we give a more detailed example of port connection. Fig. 11 shows the composition of two architectural units A and B in ArchJava [3]. Port y of A is connected to port x of B. Port x of A and

```
component class A{
 port x{ requires int readNum();}
 port y{ provides int add();}
 port n{ requires char readTxt();}
 port m{ provides void printChar();}
 //implementation of provided methods
   ...
}
```

```
component class B{
 port x{ requires int add();}
 port y{ provides int sqr();}
 port n{ requires char readTxt();}
 port m{ provides void printChar();}
 //implementation of provided methods
   ...
}
```

```
component class AB{
 port x{ requires int readNum();}
 port y{ provides int sqr();}
 port n{ requires char readTxt();}
 port m{ provides void printChar();}
 private final A a = new A();
 private final B b = new B();
 connect a.y,b.x;
 glue n to b.n;
 glue x to a.x;
 glue m to b.m;
 glue y to b.y;
}
```

Fig. 11. Connection: Composing architectural units by port connection.

ports n, m, y of B are forwarded (by delegation connectors) as ports n, m, y of the composite AB by gluing the former to the latter. Forwarding different ports would result in a different

[1]As well as inter-type declarations.

composite with different ports. In general, a port may have multiple services, which may be either required or provided services; in this example we have only used ports with a single service.

The Coordination category includes data coordination, (web service) orchestration and exogenous composition (of encapsulated components). We explain orchestration for web services as a representative example. A web service [19] provides a set of operations that can be invoked by users via its WSDL (web service description language) [15] interface (with web enabled protocols). A sequence of invocations can be defined as a workflow, in a workflow language like BPEL (business process execution language) [36], and when the workflow is executed on a workflow engine, the invocations take place. Such a workflow is called an *orchestration*. Thus orchestration is a composition mechanism for web services. This is illustrated in Fig. 12 for two web services $WS1$ and $WS2$. The workflow,

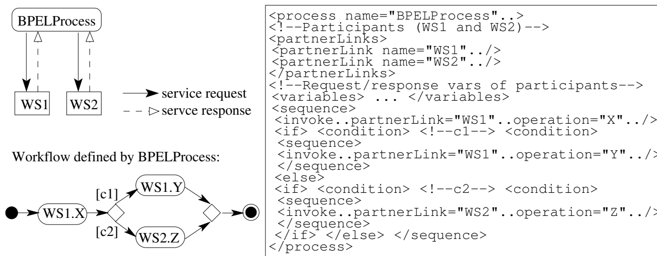


Fig. 12. Coordination: Composing web services by orchestration.

depicted by an activity diagram, is defined as a BPEL process: it invokes operation X in $WS1$, and then invokes either operation Y in $WS1$ or operation Z in $WS2$ depending on whether condition $c1$ or $c2$ holds, and then terminates. Thus, orchestration coordinates the invocation of operations in $WS1$ and $WS2$.

Our survey shows some interesting characteristics of the three views, and the composition mechanisms therein. Each view is based on a particular kind of unit of composition. In the programming view, units of composition do not have plugs. In the construction view, units of composition have plugs: modules have interaction points as plugs; architectural units have ports as plugs; fragment boxes [7] have hooks as plugs; processes have channels as plugs. In the CBD view, units of composition have proper interfaces for composition: web services have WSDL interfaces; encapsulated components have interfaces for exogenous composition.

The boundaries between views are of course not clear cut. As we pointed out in Section II, the construction and the CBD views overlap. This is evident in Fig. 8. The construction view also overlaps slightly with the programming view. A module could be a unit of composition in the programming view. However, modules with interfaces do have plugs for interacting with other modules; so a module is also a unit of composition in the construction view. Another example is a feature. A feature in feature-oriented programming does not have plugs, but a feature in the Genvoca model [9] does have plugs; such a feature would be a unit of composition in the construction view.

In each view, there is a predominant kind of composition mechanism, except the programming view, where all composition mechanisms except coordination are used. In the construction view, without the assumption of third-party components, the predominant composition mechanism is connection. This reflects the primary concern of constructing larger pieces of software from smaller pieces. The fact that modules use nesting betrays its programming view roots. In the CBD view, with the presumption of (pre-existing) third-party components, the predominant composition mechanism is coordination. This is due to the assumption of third-party components: web services are assumed to be available on web servers, while encapsulated components are assumed to be in repositories provided by third parties.

IV. Algebraic Composition Mechanisms

Our survey is not based on any desiderata for composition mechanisms, but it does provide a comprehensive source of information for further analysis of the mechanisms in terms of desirability criteria. In this section we propose a taxonomy based on a desideratum for CBD, namely systematic construction. We will show that mechanisms that are *algebraic* meet this desideratum, and identify such mechanisms.

When a composition mechanism is applied to units of composition of a given type, the resulting piece of software may or may not be another unit of composition of the same type. If it is, then it can be used in further composition; composition mechanisms that produce units of composition of the same type as the composed units of composition are *algebraic*. Algebraic composition mechanisms are good for hierarchical composition (and therefore systematic construction), since each composition is carried out in the same manner regardless of the level of the construction hierarchy. Indeed in the 'construction view', such mechanisms are deemed the most desirable since they can constitute a component algebra [2].

We only consider one-sorted algebra, not many-sorted algebras, where 'algebraic' would mean the resulting unit is of the same type as at least one of the composed units. In practice, in any programming paradigm, there is usually only one pre-dominant, paradigm-defining sort, e.g. object-oriented programming, service-oriented programming, etc.

	Composition Mechanism			
	Containment	Extension	Connection	Coordination
Algebraic	Function nesting Procedure nesting Module nesting Class nesting Object composition Object aggregation	Multiple inheritance Mixin inheritance Trait composition Subject composition Feature composition Invasive composition	Higher−order function Trait composition Port connection Invasive composition Channels	Exogenous composition
Non−Algebraic		Mixin−class inheritance Trait−class composition Weaving	Function call Procedure call Module connection Object delegation Trait−class composition	Data coordination Orchestration

Fig. 13. Algebraic vs. non-algebraic composition mechanisms.

Analysing the mechanisms in our survey in Fig 8, we arrive at the taxonomy of algebraic versus non-algebraic mechanisms in Fig. 13.

In the Containment category, all the mechanisms are algebraic, since the composite is always the same type as the composed units.

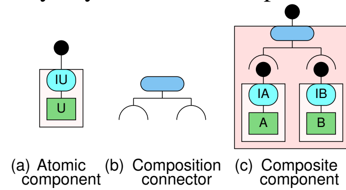
In the Extension category, some mechanisms are algebraic, while some are not. Multiple inheritance yields a class from two classes and is therefore algebraic. Similarly, mixin inheritance, subject composition and feature composition are algebraic. Trait composition can be done by either extension or connection, but it is always algebraic since it always yields another trait. Invasive composition performs both extension (by overwriting) and connection (via hooks), but it is always algebraic because it always yields another fragment box.

Mixin-class inheritance and weaving yield, respectively a class from a mixin and a class, and a class from an aspect and a class, and are therefore not algebraic. Trait-class composition falls into both the Extension and Connection categories, depending on whether the trait composition involved is done by extension or connection, but trait-class composition is always non-algebraic since it yields a class from a trait and a class.[2]

Like the Extension category, in the Connection category, some mechanisms are algebraic and some are not. A higher-order function composes functions and yields a function, and is therefore algebraic. So is port connection, which composes architectural units and yields an architectural unit. Channels connecting processes create new processes, and are therefore algebraic.

A function call returns a pair of functions rather than a single function, and is therefore non-algebraic. Similarly, procedure call, module connection, and object delegation are non-algebraic.

Finally, in the Coordination category, only exogenous composition is algebraic, since the composition of two encapsulated components always yields an encapsulated component. In exogenous composition (Fig. 14), the components are encapsulated (Fig. 1(d)). There are two basic types of components: (i) *atomic* and (ii) *composite*. An atomic component (Fig. 14(a)) consists of a *computation* unit (U) and an *invocation connector* (IU). A computation unit contains a set of methods which do not invoke methods in the computation units of other components; it therefore encapsulates computation. An invocation connector passes control (and input parameters) received from outside the component to the computation unit to invoke a chosen method, and after the execution of method passes control (and results) back to whence it came, outside the component. It therefore encapsulates control. A composite component (Fig. 14(c)) is built from atomic components by using a *composition connector* (Fig. 14(b)). Such a connector encapsulates a control structure, e.g. sequencing, branching, or looping, that connects the subcomponents to the interface of the composite component. Since the atomic components encapsulate computation and control, so does the composite component. Encapsulated components therefore encapsulate control (and computation) at every level of composition. Fig. 14 clearly shows that exogenous composition is algebraic: exogenous composition of encapsulated components always yields another encapsulated component. The composition connector provides the interface of the composite, which is derived directly from the interfaces of the composed components.



(a) Atomic component (b) Composition connector (c) Composite component

Fig. 14. Exogenous Composition.

[2]Our classification of subject composition as algebraic, and aspect weaving as non-algebraic, mirrors the dichotomy between symmetric and asymmetric aspect mechanisms [21], [26] in aspect-oriented software development.

Data coordination is not algebraic since it does not yield a single process; rather it yields the same set of processes (either sharing a tuple space or connected by data connectors). Orchestration of web services is not algebraic since the result of an orchestration is a workflow, rather than a web service, as we showed in Section III-E. Of course the workflow could be turned into web service, by creating a WSDL interface for it, but this would require an extra step after orchestration. Indeed, some BPEL editors force the user to take this extra step in order to make the orchestration executable as a web service.

V. Composition Operators

Another desideratum for CBD is that composition mechanisms should be automatable. A composition is automatable if it can be explicitly defined as a composition operator, i.e. like a mathematical function, that can be defined and then applied to arbitrary arguments, i.e. units of composition, of specified types. For example, a higher-order function $h : X \to Z$ that composes two functions $f : X \to Y$ and $g : Y \to Z$ (where X, Y, Z are types) can be defined explicitly in terms of f and g as $h(x) = g(f(x))$. The operator h can be used to compose any two functions with type signatures $X \to Y$ and $Y \to Z$.

Applying a composition operator does not require any glue that has to be constructed manually. With composition operators defined from algebraic composition mechanisms, we can automate hierarchical composition. In this section, we propose a taxonomy of algebraic composition mechanisms that can be defined as operators versus those that cannot. This taxonomy is shown in Fig. 15.

Composition Operator?	Algebraic Composition Mechanism: Containment	Extension	Connection	Coordination
No	Function nesting Procedure nesting Module nesting Class nesting Object composition Object aggregation	Multiple inheritance Trait composition Feature composition Invasive composition	Trait composition Port connection Invasive composition Channels	
Yes		Mixin inheritance Subject composition	Higher–order function	Exogenous composition

Fig. 15. Algebraic composition mechanisms as operators.

In the Containment category, no mechanism can be defined as an operator, since nesting can be done in arbitrary ways.

In the Extension category, multiple inheritance, trait composition, and feature composition, all perform extension that may require glue for conflict resolution and overriding in general, and therefore cannot be defined as operators. Invasive composition requires glue for both extension and connection, and therefore cannot be defined as operators.

By contrast, mixin inheritance never requires glue, since it performs extension in a fixed manner. A mixin M is a set of methods, and can be defined as a record $\{f_1 \mapsto m_1, \ldots, f_n \mapsto$

$m_n\}$ with fields $f_1, \ldots, f_n$ whose values are the signatures $m_1, \ldots, m_n$ of M's methods. Mixin inheritance can be defined as record combination, which is a binary operation $\oplus$ [10] such that $M_1 \oplus M_2$, for any M_1 and M_2 yields a new mixin M_3 which is a new record with the fields from M_1 and M_2, where the value for each field is the value from the left argument M_1 (or the right argument M_2) in case the same field is present in both records.

An example (Fig. 16) in MixedJava [20] shows mixin A with methods $m1, m2, m5$; $m5$ prints the message 'Alpha'. Mixin B has methods $m3, m4, m5$; $m5$ prints the message 'Beta'. The first composition expression generates a composite mixin AB, in which A's $m5$ overrides B's $m5$. Similarly, the second composition expression generates a composite mixin BA, in which B's $m5$ overrides A's $m5$.

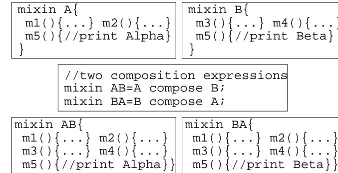


Fig. 16. Mixin inheritance.

Similarly, it is possible to define simple correspondence-combination operators for composing arbitrary subjects, e.g. a simple 'merge-and-overwrite' operator. However, it is difficult to define an operator for complex correspondence-combination mechanisms that can compose arbitrary subjects.

In the Connection category, for any two given traits and two architectural units, respectively, there are in principle many different possible pairs of matching services and compatible ports, and each permutation of possible pairs gives rise to an operator. Thus composing traits and architectural units by connection is necessarily done in an *ad hoc* manner, and cannot be defined as operators. Similarly, for any two given processes, there are many different possible channels for connecting them. Thus composing processes by channels connection cannot be defined as operators.

In the Coordination category, exogenous composition can be defined as composition operators (connectors), as we have already seen in the previous section.

Finally, this taxonomy is a sub-taxonomy of the taxonomy (Fig. 13) presented in the previous section. Together they form the taxonomy that identifies desirable composition mechanisms for CBD. Fig. 15 shows that these mechanisms are mixin-inheritance, subject composition and higher-order function (from the programming view) and exogenous composition (from the CBD view). Of these, only exogenous composition is being used in CBD. Apart from exogenous composition, current component models predominantly use object delegation and port connection (for architectural units).

VI. Discussion and Conclusion

Various categories for software composition mechanisms have been proposed before. Nierstrasz and Dami [33] suggest three different types of compositional paradigms for components (static abstractions with plugs): (i) functional composition, (ii) blackboard composition and (iii) extensibility. Components are seen as (mathematical) functions from input values to output values. In functional composition, components are composed like (mathematical) functions. This corresponds to the higher-order function mechanism in our Connection category. Blackboard composition is data sharing by components, and is therefore data coordination in our Coordination category. Extensibility is not a separate mechanism, but part of functional composition; it allows individual components to be extended (by single inheritance), and requires any such extension to be preserved in any functional composition involving extended components. Nierstrasz and Dami do not have our Containment and Extension categories.

Sametinger [43] categorises software composition mechanisms into two basic forms: (i) internal and (ii) external. In internal composition mechanisms, composed units become inherent parts of the composite, e.g. when source code is compiled and linked to an executable file. This corresponds to our Containment category in a coarse-grained way; it is not clear whether object aggregation is internal. In external composition mechanisms, composed units execute independently and communicate with other composed units by interprocess communication techniques. This covers our Connection and Coordination categories, but again in a very coarse-grained manner. It is not clear which of these forms our Extension category belongs to.

Sommerville [46] defines three types of composition mechanisms for architectural units: (i) sequential, (ii) hierarchical and (iii) additive. In sequential composition, the 'provided' interfaces of the units are linked by glue code that executes their services in sequence; what happens to the 'required' ports is not defined. Without 'required' ports, this mechanism seems to be a control coordination mechanism, and seems to be non-algebraic. Hierarchical composition is the same as port connection in our Connection category. Additive composition simply yields a composite whose interface is the set of the interfaces of the components. This is a degenerate form of port connection in which only delegation connectors are used (to forward ports to the composite). Sommerville does not have the Containment or Extension categories since he only addresses architectural units. He also seems not to have the Coordination category.

Szyperski [48] classifies software composition approaches into two categories: (i) symmetric and (ii) asymmetric. Symmetric means the definition of composition is located in (one of) the composed components, e.g. object delegation, while asymmetric means the location of composition definition is outside in a neutral place, e.g. container-based composition like in EJB. These are coarse-grained categories, with symmetric covering our Containment, Extension and Connection categories, while asymmetric corresponds to our Coordination category.

Mehta *et al* [31] define composition mechanisms for components as connectors, and categorise them into connectors for: (i) communication (ii) coordination (iii) conversion and (iv) facilitation. Communication connectors transfer data, whilst coordination connectors transfer control, between components. These connectors belong to our Connection category, since

they compose components by message passing. Conversion connectors convert the interaction required by one component to that provided by another, e.g. conversion of data format; thus they are adaptors. Our categories do not include adaptors; we do not consider them to be composition mechanisms since they are unary operators. Facilitation connectors provide mechanisms for facilitating and optimizing component interactions. They do not feature in our categories.

The only work related to our taxonomy for CBD is that of Chaudron [14]. He does not propose any taxonomy, but he does define desiderata for composition mechanisms for CBD. Interestingly, Chaudron's desiderata support our taxonomy for CBD. Three of his criteria which are relevant here state that: (i) composition mechanisms should be exogenous to components, i.e. not built into the components themselves; (ii) composition mechanisms should provide separate mechanisms for dealing with control flow and data flow; (iii) composition languages should provide means for building higher level, larger-granularity composition abstractions. (i) and (ii) support our classification of exogenous composition (of encapsulated components) as desirable for CBD (Fig. 15), while (iii) supports our choice of algebraic mechanisms as desirable for CBD (Fig. 13).

For practical development, we will always need to use a combination of different kinds of components and composition mechanisms. Non-algebraic mechanisms or mechanisms that cannot be defined as operators may be better for top-level system design. On the other hand, given a top-level architectural design, it may be better to provide all its required services by designing the desired composites using composition operators that can be applied automatically.

We have not addressed run-time or dynamic composition, e.g. *proximity-based* composition (objects in a context may be automatically connected) [47], and *data-driven* composition [47].

Finally, we agree with Szyperski [47], [48] that for CBD the 'universe of composition' is as yet largely unexplored. Our work here is a response to his 'call-to-arms' [47].

ACKNOWLEDGEMENTS

We thank Uwe Assman, Don Batory, David Lorenz, Oscar Nierstrasz, Johannes Sametinger, Clemens Szyperski and Steffen Zschaler for factual information, helpful discussions and insightful comments. We also thank Michel Chaudron for pointing out a mistake in an earlier version of the paper.

REFERENCES

[1] F. Achermann *et al*. Piccola – a small composition language. In *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[2] F. Achermann and O. Nierstrasz. A calculus for reasoning about software composition. *Theoretical Computer Science*, 331(2-3):367–396, 2005.

[3] J. Aldrich, C. Chambers, and D. Notkin. Component-oriented programming in ArchJava. In *1st OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 1–8, 2001.

[4] G. Alonso *et al*. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[5] S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Software Composition*, *LNCS* 4954, pages 20–35. Springer, 2008.

[6] F. Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. in Comp. Sci.*, 14(3):329–366, 2004.

[7] U. Assman. *Invasive Software Composition*. Springer Verlag, 2003.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[9] D. Batory *et al*. The GenVoca model of software-system generators. *IEEE Software* 11(5):89–94, 1994.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP 90*, pages 303–311. ACM Press, 1990.

[11] M. Broy *et al*. What characterizes a software component? *Software – Concepts and Tools*, 19(1):49–56, 1998.

[12] M. Buchi and W. Weck. Compound types for Java. In *Proc. OOPSLA 98*, pages 362–373. ACM Press, 1998.

[13] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, 1989.

[14] M. Chaudron. Reflections on the anatomy of software composition languages and mechanism. In *Proc. Workshop on Comp. Lang.*, 2001.

[15] E. Christensen *et al*. Web services description language (WSDL) 1.1. Technical report, W3C, 2001.

[16] P.C. Clements. A survey of architecture description languages. In *8th Int. Workshop on Soft. Spec.and Design*, pages 16–25. ACM, 1996.

[17] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Soft. Eng.*, 2(2):80–86, 1976.

[18] S. Ducasse *et al*. Traits: A mechanism for fine-grained reuse. *ACM Trans. Prog. Lang. Syst.*, 28(2):331–388, 2006.

[19] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[20] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, 1999. Springer-Verlag.

[21] W. H. Harrison, H.L. Ossher, and P.L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Research Report RC22685, IBM Thomas J. Watson Research, 2002.

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

[23] C.A.R. Hoare. Communicating sequential processes, 2004.

[24] G. Kiczales *et al*. An overview of AspectJ. In *Proc ECOOP 01*, pages 327–353. Springer-Verlag, 2001.

[25] G. Kiczales *et al*. Aspect-oriented programming. In *Proc. ECOOP 97*, pages 220–242. Springer-Verlag, 1997.

[26] S. Kojarski and D.H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proc. 28th ICSE*, pages 212–221. ACM, 2006.

[27] K.-K. Lau and M. Ornaghi. Control encapsulation: A calculus for exogenous composition. In *Proc. 12th CBSE*, *LNCS* 5582, pages 121–139. Springer-Verlag, 2009.

[28] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th CBSE*, *LNCS* 3489, pages 90–106. Springer-Verlag, 2005.

[29] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. on Soft. Eng.*, 33(10):709–724, 2007.

[30] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans on Soft. Eng.*, 26(1):70–93, 2000.

[31] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd ICSE*, pages 178–187. ACM, 2000.

[32] O. Nierstrasz. Research topics in software composition. In *Proc. Langages et Modèles à Objets*, pages 193–204, 1995.

[33] O. Nierstrasz and L. Dami. Component-oriented software technology. In [35], pages 3–28. Prentice-Hall, 1995.

[34] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Comput. Surveys*, 27(2):262–264, 1995.

[35] O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall International, 1995.

[36] OASIS. Web services business process execution language, April 2007.

[37] H. Ossher *et al*. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3):179–202, 1996.

[38] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proc. OOPSLA '01*, pages 283–299. ACM, 2001.

[39] J.K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[40] C. Pfister and C. Szyperski. Why objects are not enough. In *Proc. 1st Int. Component Users Conf.* SIGS Publishers, 1996.

[41] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. ECOOP'97*, pages 419–443. Springer-Verlag, 2002.

[42] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *Journal of System and Software*, 6(4):307–334, 1987.

[43] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[44] J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[45] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[46] I. Sommerville. *Software Engineering*. Addison-Wesley, eighth edition, 2007.

[47] C. Szyperski. Back to universe. *Software Development*, September 2002.

[48] C. Szyperski. Universe of composition. *Software Development*, August 2002.

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

[50] P. Velasco Elizondo and K.-K. Lau. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 2010. http://dx.doi.org/10.1016/j.jss.2010.01.008.