Data Encapsulation in Software Components

Kung-Kiu Lau and Faris M. Taweel

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{Kung-Kiu,Faris.Taweel}@cs.manchester.ac.uk

Abstract. Data encapsulation is a familiar property in object-oriented programming. It is not only useful for modelling things in the real world, but it also facilitates reuse by enabling the creation of multiple instances of the same class, each with its own identity and private data. For CBSE, this kind of reuse is clearly also one of the key desiderata. However, it must be achieved in conjunction with composition, which is central to CBSE. In this paper we show how data encapsulation can be combined with composition, by extending a component model we have defined previously.

1 Introduction

Data encapsulation is a familiar property of objects, as in object-oriented programming. It is not only useful for modelling things in the real world, but it also facilitates reuse by enabling the creation of multiple instances of the same class, each with its own identity and private data. For CBSE, this kind of reuse is clearly also one of the key desiderata, since components are considered to be reusable templates for multiple component instances. However, since composition is central to CBSE, the question is how to design composition mechanisms or operators that make data encapsulation possible at every level of composition, that is, how to make sure that every *composite* component created by composition encapsulates its own data. In this paper, we argue that this combination of data encapsulation and composition is not possible in current component models; and then show that it can be achieved by extending a component model that we have defined previously.

Current component models can largely be divided into two categories [9,5]: (i) models where components are *objects*, as in object-oriented programming; (ii) models where components are *architectural units*, as in software architectures [14,1]. Exemplars of these categories are Enterprise JavaBeans (EJB) [3,12] and architecture description languages (ADLs) [2,10] respectively. In models where components are objects, components are assembled by method calls. However, this is not (algebraic) composition, since an object O_1 assembled with an object O_2 by calling a method in O_2 will result in two objects, not one (composite) object. Therefore, even though data encapsulation is possible in O_1 and O_2 separately, there is no composition mechanism that can compose O_1 and O_2 properly, let alone preserve data encapsulation.

H.W. Schmidt et al. (Eds.): CBSE 2007, LNCS 4608, pp. 1–16, 2007.

In component models where components are architectural units, port connections provide a composition mechanism, and composites can be defined. However, data encapsulation is not always defined or possible. In fact, the role of data is very unclear in architectural units. These units can represent both computation and data, or just data, and data encapsulation is not considered as part of composition in general. Where architectural units have data ports, it could be argued that these ports represent data encapsulation. Even in this case, however, it is not clear whether data encapsulation is possible at every level of composition.

In this paper, we describe an approach to composition that allows data encapsulation at every level of composition. Our approach is based on a component model [6] where composition operators are first-class citizens, and they also enable every component instance, in particular a composite component instance, to encapsulate its own data.

2 Composition with Data Encapsulation

Components are intended for composition, and so they should be compositional, i. e. if C_1 and C_2 are components, then a composition C_3 of C_1 and C_2 must be a component too. Furthermore, the composition should be defined as a composition operator that composes components into new (composite) components. In other words, in a component model, components and composition operators should be first-class citizens. Any component model should have this property.

A good component model should also allow components to encapsulate data, but to be really useful it should do so at every level of composition. We have proposed a component model in [6] and in this paper we describe how we can extend this model and use it to achieve this kind of data encapsulation.

For illustrative purposes, we shall consider a simple banking example.

Example 1. Consider a banking system with two bank consortia BC1 and BC2, consisting of the sets of banks {B11, B12} and {B21, B22} respectively. Each bank, in turn, consists of a set of branches, e.g. bank B21 has branches {BB211, BB212}, and so on.

In our component model we would build up the system by composing components using composition operators (see Fig. 2).

In our model, we have two kinds of components: *atomic* and *composite* components. Composite components are built from atomic (and other composite) components, but *all* components are templates, with no data except constants,

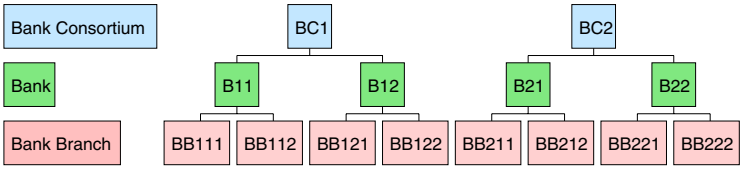

Fig. 1. A banking example

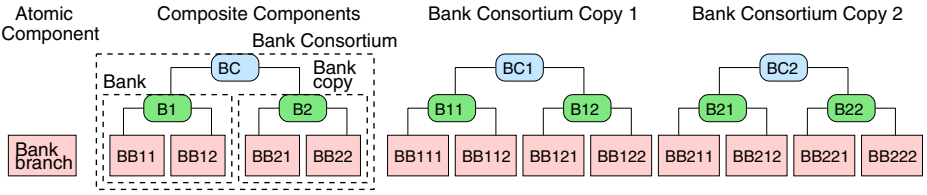

Fig. 2. A component-based implementation of the banking system

but with code for the services they provide. Since components are templates, it is meaningful and possible to make *copies* of components. For the bank system, for a particular bank consortium, a bank branch can be an atomic component with code for the usual operations like *withdrawal*, *deposit* and *check balance*. This provides a template for all bank branches, and so we can construct many bank branches (all the BB's) as copies of this component.

Furthermore, in our model, it is possible to create *instances* from different component copies. For example, bank branches BB111 and BB211, which are different copies of BB11 (which is in turn a copy of the bank branch atomic component), and which belong to different consortia, can be each instantiated with their own address, sort code and customer accounts.

In our model, composite components, just like atomic components, can also be copied (and the copies instantiated later). For example, two bank branches, say BB11 and BB12, can be composed by a suitable composition operator to produce a bank composite component B11. The latter is a template that contains all the operations its sub-components provide. Therefore, it would make sense to construct other bank components from this component by copying. The original component as well as its copies contain, in addition to operations, some place holders for private data that can be initialised when the complete (composite) components are instantiated.

Similarly, a bank consortium component can be constructed by composing bank components. In Fig.2, using a suitable composition operator, a bank component composed with a copy of a bank component yields a bank consortium component. This new component can be further composed with a copy of itself to build the bank system.

It is worth noting that in the bank example, only one atomic component (bank branch) and one composition operator are necessary to build the entire bank system. Each composition in our implementation results in a properly defined composite. Clearly our model provides proper composition mechanisms. The question is whether we can also make it encapsulate data.

3 Our Component Model

Before we discuss how we extend our component model to enable data encapsulation, in this section we briefly outline the model that we presented in [6].

In our model, we have two kinds of basic entities: (i) *computation units*, and (ii) *connectors*. A computation unit U encapsulates computation. It provides a set of methods (or services). *Encapsulation* means that U's methods do not call methods in other computation units; rather, when invoked, all their computation occurs in U. Thus U could be thought of as a class that does not call methods in other classes.

There are two kinds of connectors: (i) *invocation*, and (ii) *composition*. An invocation connector is connected to a computation unit U so as to provide access to the methods of U.

A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components (atomic or composite). For example, a *sequencer* connector that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$ in that order. Another example is a *selector* connector, which selects (according to some specified condition) one of the components it composes, and calls its methods.

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite* (see Fig. 3). An

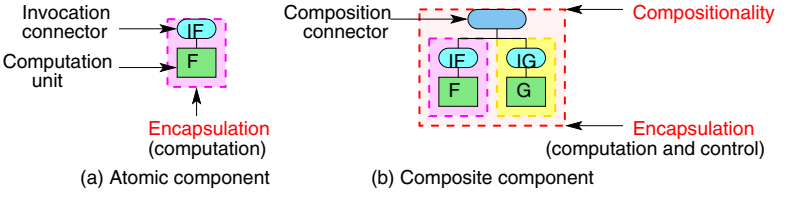

Fig. 3. Atomic and composite components: encapsulation and compositionality

atomic component consists of a computation unit with an invocation connector that provides an interface to the component. A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite.

For example, in the bank system (Fig. 2) in Example 1, the atomic component BB11, a bank branch, may be defined as shown in Fig. 4(a), with an invocation connector IBB11, and a computation unit with the methods *deposit, withdraw, balance*. The composite component B1, a bank, may be defined as shown in

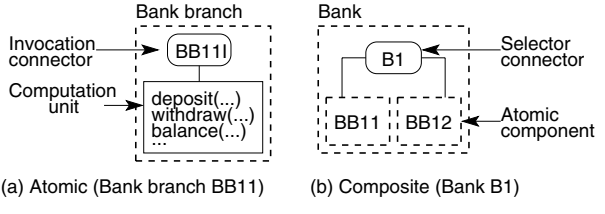

Fig. 4. Sample atomic and composite components in the bank example

Fig. 4(b), as a composition of the atomic components BB11 and BB12 using a selector connector (denoted here by B1 too, for convenience). The bank consortium composite component in Fig. 2 may also be composed (from banks) using a selector connector, since the consortium has to choose the bank with the branch to which the customer's account belongs.

In our model, invocation and composition connectors form a hierarchy [8]. This means that composition is done in a hierarchical manner. Furthermore, each composition *preserves* encapsulation. This kind of compositionality is the distinguishing feature of our component model. An atomic component encapsulates *computation* (Fig. 3(a)), namely the computation encapsulated by its computation unit. A composite component encapsulates *computation* and *control* (Fig. 3(b)). The computation it encapsulates is that encapsulated in its sub-components; the control it encapsulates is that encapsulated by its composition connector. In a composite, the encapsulation in the sub-components is preserved. Indeed, the hierarchical nature of the connectors means that composite components are *self-similar* to their sub-components; this property provides a basis for hierarchical composition.

In the next section, we will show how to extend our model to include data encapsulation.

4 Data Encapsulation

Our approach to data encapsulation is illustrated by Fig. 5. Basically, we want to extend our model (Fig. 3) to allow each component (atomic or composite) to define place-holders for its own data at *design* time. These place-holders are indicated by patterned squares in Fig. 5. Thus, whereas in our current model, a composite encapsulates computation and control (Fig. 3(b)), in the extended model, a composite additionally encapsulates data (Fig. 5(b)).

Our extension is centred on the constructor of a component. We want to be able to make copies of a component at design time, so that they all have the same types of data place-holders. Copies of a component will also have the same constructor as the of the original component. At run-time, we want to be able to create an instance of a component or a copy by using the component's constructor, and we want to be able to initialise its data place-holders with

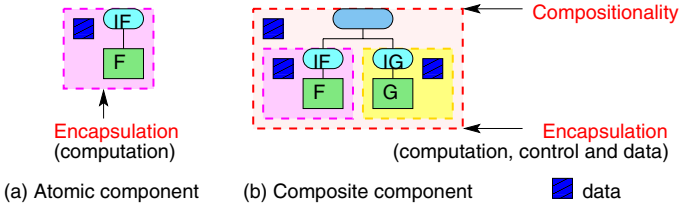

Fig. 5. Data encapsulation

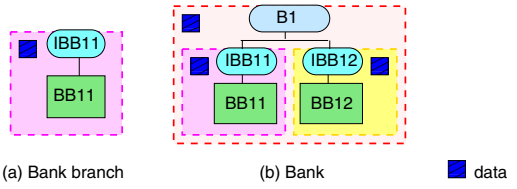

Fig. 6. Data encapsulation in the bank example

actual data. This way, instances of different copies (atomic or composite) can encapsulate their own private data.

For the bank example, this is illustrated by Fig. 6. In Fig. 6(a), a branch component BB11 encapsulates its customers' data. Using a suitable composition connector, a bank component can be constructed from branch components BB11 and BB12, where both components are actually copies of the bank branch component. As shown in Fig. 6(b), the bank (composite) component encapsulates its own data which is separate from its sub-components' data. The encapsulation of the latter in each branch sub-component is preserved in the bank composite.

The process of instantiating a component (or a copy) often requires initialisation of encapsulated data. Such data can either be constants defined in the component's design phase; or data created at instantiation time. A component therefore must have a constructor which enables data initialisation to be performed. In our model, we use *data constructors* in the component constructor for this purpose. Data constructors may require to read data from external resources during the data initialisation process. Therefore, connectors must have data I/O semantics to carry out their tasks. In our model, connectors are capable of performing data I/O operations.

Initialisation of encapsulated data in the bank example is illustrated in Fig. 7. A bank branch component must be initialised with the branch name which is a constant string. It must also read and persist data about the process that owns the branch component instance. The latter data may include date, process account and network information, etc. For simplicity, we assume that the component only logs its instantiation date. The invocation connector of a branch component accesses these data values during component construction at runtime. A bank component composed from two branches also has its own separate constructor which performs its data initialisation operations. These operations include, for example, setting the bank name as well as logging a record on the instantiation date and other system data. As far as data is concerned, the bank component and its sub-components set their initial data independently, each using its own constructor. In Fig. 7, bank component B1 reads bank name (*B1*) and date (*sysdate*). Its sub-components perform similar data initialisation operations when their instances are created. Initialisation performed by each connector of B1 is indicated by arrows bearing data names.

Data encapsulation is a valuable notion for reuse by copying. A component that encapsulates its data (in addition to computation and control) and has

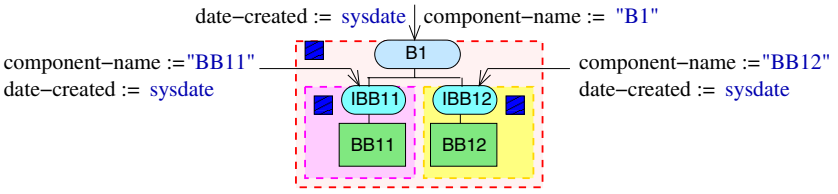

Fig. 7. Initialisation of encapsulated data

its own data constructor is a suitable unit for copying. A component in design phase specifies its local data as place holders. Copying a component creates a new component that encapsulates its own data (specifications). Data constructors of copies perform data initialisation of each copy. Therefore, data encapsulation enables copying of our components.

Furthermore, in a component-based system, it is often desirable to create multiple instances of a component or to create instances of different copies of a component. In the bank system, all branches are copies of a branch component. Instantiating all these branches must be possible. In fact, without each branch encapsulating its data together with a data constructor, it would be impossible to have different instances from different copies. In our model, it is even possible to make many copies of the same component, since instances maintain their own data.

5 Implementation

In this section we show how we implement data encapsulation in our extended component model, by using the bank example to show how a bank system can be constructed from two connectors and one computation unit. First, we outline our implementation of the extended component model using Oracle Database 10g Enterprise Edition, release 10.1.0.4.0. The choice of a database language is natural, since we are concerned with data here.

5.1 Connectors and Components

We have implemented our extended component model as a repository that stores computation units as well as templates for connectors and components (both atomic and composite), at both design time and run-time. While the repository depends on metadata that Oracle maintains on computation units, connectors and component templates are stored as records in database tables, e.g. CONNECTORS, COMPONENTS, ENC_DATA, ENC_DATA_INST, etc. The repository provides services such as creating and copying components, searching, browsing, and component instantiation. Components, connectors and computational units are coded in the *PL/SQL* programming language [13]. PL/SQL is a 4GL programming language that is used by Oracle to specify its programs' interfaces, so it is the obvious choice for us. However, PL/SQL lacks support

```
--- Specification (interface)
PACKAGE "BB" AS
      FUNCTION balance (p_accnt_no CHAR) RETURN INTEGER;
      FUNCTION withdraw (p_accnt_no CHAR, amnt NUMBER) RETURN CHAR;
      FUNCTION deposit (p_accnt_no CHAR, amnt NUMBER) RETURN CHAR;
END BB;
--- Implementation
PACKAGE BODY "BB" AS
      ...
      FUNCTION withdraw (p_accnt_no CHAR, amnt NUMBER) RETURN CHAR;
         IS BEGIN
         ...
         RETURN bal;
      END balance;
      ...
 END BB;
```

Fig. 8. PL/SQL specification and implementation of bank branch package

for reflection, which is necessary for implementing our component model. So our implementation of the repository has to compensate for this.

In our implementation, computation units are Oracle *packages* and connectors are Oracle *object types*. A package is a database construct that groups logically related PL/SQL types, variables, and subprograms (functions and procedures). It can implement a computation unit in our component model provided its subprograms do not call subprograms in other packages. A package has two parts, a specification and a body. The specification is the interface to the package. It publishes the types, variables, constants, exceptions, cursors (record sets) and subprograms. The body implements cursors and the subprograms. Fig. 8 shows an example of a PL/SQL package. It is an outline of a package for a bank branch in the bank example.

An object type is a user-defined composite data type representing a data structure and subprograms to manipulate the data. It is like an object in an object-oriented programming language. Like packages, object types are specified in two parts, a specification (interface) and a body (implementation). Fig. 9 shows an example of an object type. It is an outline of the invocation connector type, Invoker (see below).

We use object types to implement our connectors hierarchically. The root of the hierarchy is the supertype CConnector, with three sub-types Invoker, Selector and Sequencer, for invocation connectors, selector connectors and sequencer connectors respectively. For example, in Fig. 9, the Invoker type inherits from CConnector (indicated by the keyword *UNDER*).

The super-type CConnector provides the implementation of the most important procedure for data, the *data_constructor*. This procedure creates and initialises data instances from data specifications in components. Constructors of all sub-types of CConnector invoke the *data_constructor* procedure to create and initialise their components' data instances. In a sub-type, the constructor starts by creating and initialising its internal data. Then, it calls the *data_constructor* procedure which reads its component data specifications from the repository; and creates and initialises the required data instances. This is illustrated in Fig. 9 for the constructor of Invoker. The call to *data_constructor* is highlighted.

```
--- Specification (interface)
CREATE OR REPLACE TYPE "INVOKER" UNDER   CCONNECTOR   (
  CONSTRUCTOR FUNCTION invoker(p_cname VARCHAR2, p_cuname VARCHAR2)
     RETURN SELF AS RESULT,
END; -- Invoker specifications.
--- Implementation
CREATE OR REPLACE TYPE BODY "INVOKER" IS
  CONSTRUCTOR FUNCTION invoker(p_cname VARCHAR2, p_cuname VARCHAR2)
    RETURN SELF AS RESULT IS
    l_data_value raw(16);
  BEGIN
  ... -- other initialization operations.
  SELF.cuname := p_cuname;
  ...
     self.data_constructor(p_cuname); -- CREATE COMPONENT LOCAL DATA
  END;
END; --- Invoker.
```

Fig. 9. PL/SQL specification of an invocation connector

An atomic component is constructed from a package for a computational unit and an object type for an invocation connector. The invocation connector keeps a reference to the computation unit's name. For example, in Fig. 9, the variable *cuname* is assigned the user-provided computation unit name *p_cuname*. The component is created in the repository by executing a procedure which generates a specification and stores it as records in the relevant repository database tables. These records contain data on the component, its invocation connector, interface, services and their return types and parameters, etc. At this point, the component is in the design phase. Copying the component is equivalent to retrieving all the component specification (records) and storing it back under a new component name. The new name can be a unique user- or system-provided string. The computation unit must exist in the repository for this operation to complete successfully.

A composite component is constructed from existing components. The procedure concerned takes three parameters: a string name for the new component, a list containing the sub-components' names and a connector type. The number of components that can be composed depends on the semantics of the composition connector. This procedure generates and stores the specification of the constructed component in a similar manner to the the procedure for constructing atomic components. In our repository, a composite component is always constructed from copies of other components. That is, if F and G are components in the repository, then in a composite H of F and G, the sub-components must be copies of F and G, each renamed by the repository. Thus the repository is implemented in such a way that any composition operation leaves unchanged the original components involved in the composition process.

5.2 Data Encapsulation

Once a component is constructed and stored in the repository, a set of operations are available to support the specification of data intended for encapsulation in

the component. Data specification for a component include information on the properties of component's encapsulated data elements such as their names, types, state (persistent or transient), initial values, initial actions, etc. The specification is used by the repository to create and manage data instances at run-time.

At design time, repository operations generate, update and store components' data specifications in a repository database table named *ENC_DATA*. The operation to encapsulate a data element in a component takes a parameter (a positive integer value) for specifying the order for data instantiation. Names for data elements must be unique in the scope of the component that owns them. Data properties of a data element entry are set using either a generic procedure that is used to define arbitrary data elements, or a compatible type-specific procedure, e.g. for integer and string data elements. Copying a component in the repository results in copying its data specification too. Accordingly, for a component to be reusable, data initialisation must be delayed to the run-time phase, except when data is not application specific.

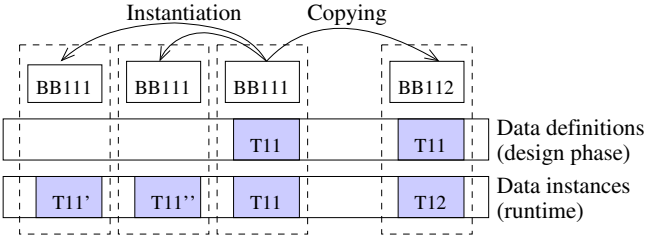

Fig. 10. Encapsulated data in component copies and instances

At run-time, a component constructor is used to create an instance of the component. This instance is uniquely identified in our run-time system by a CID (component ID). During the construction process, the *data_constructor* procedure is invoked. It retrieves the component's data specification stored in the *ENC_DATA* table; creates and initialises data instances; and stores these instances in a global temporary data space. The initialisation of a data element can either be achieved by a simple assignment of a constant or a computed value. Computed values are specified as scripts (anonymous PL/SQL blocks) and are executed by the run-time system during component construction. Data instances stored in the data space are made available to their components by reference. The repository identifies them via their CIDs. The global data space is implemented as an Oracle global temporary table. Entries in this kind of tables pertaining to a particular component are automatically garbage collected when the parent process of the component terminates. Further instantiations of a component create new, different and independent data instances for each component instance. Different instances of a component maintain their own data instances at run-time. Fig. 10 shows encapsulated data in component copies and instances in the bank example. Branch BB111 owns its data definition T11. A copy BB112 has its own copy T11 of the data definition. Two instances of BB111

own two different instances of data: T11' and T11". The two instances start with the same data, but their data becomes different over time.

Component constructors must be capable of performing data I/O operations required for data initialisation, among others. These operations (read, write) are implemented as *data connectors* used to input and output data from various data sources including the global temporary data space. The repository automatically creates a data connector for each encapsulated data element, method parameter and return value. For standard data sources such as relational databases, a complete set of data connectors is available and ready for use. For non-standard or unknown data sources, data connectors are created as stubs that must be manually replaced before running the system. Data connectors for non-standard data sources can be added to the repository and reused in building new systems. Few of the current component models support relational data sources, and only .NET supports additionally XML data sources [11].

5.3 The Bank System

Now, we can work out the implementation of a bank system based on the bank example in Example 1 (Fig. 1) in detail. In particular, we demonstrate data encapsulation and how our component model enables copying and multiple instantiation of its components. We also show that the composition scheme in our model preserves and propagates data encapsulation at every level of composition.

Consider a bank system consisting of the 2 consortia BC1 and BC2, as shown in Fig. 1, with a simplified entity-relationship (ER) diagram as shown in Fig. 11.

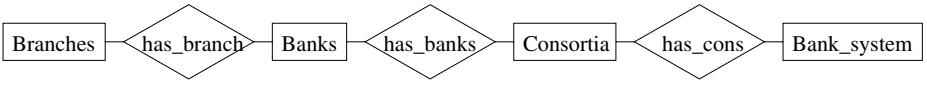

Fig. 11. A simple ER diagram for the bank system

For simplicity, we choose to encapsulate only three data elements in each component, namely, a component name, date of instantiation and data details. A bank branch encapsulates separately its name (BRANCH_NAME), date and customers' data (CUSTOMERS). A bank component encapsulates data on its branches (BRANCH) as well as the bank's name (BANK_NAME) and date. A consortium holds local data on its member banks (BANK), the name of the consortium (CONST_NAME) and a date. The bank system holds local data on all the consortia it has (CONST), its name (BNET) and a date. The date (named SYS_DATE in each) is a place holder that stores the component's instantiation date. The date holds an initialisation script (PL/SQL anonymous block) that can be executed by the run-time system to return the instantiation date. In Fig. 11, the relationships *has-cons*, *has-banks* and *has-branch* are data encapsulated in the components bank system, consortia and banks respectively.

To build the system, we start with one computational unit BB (an Oracle package), one invocation connector and one selector composition connector.

```
EXEC REPOSITORY.CREATE_ATOMIC_COMPONENT('BB111', 'BB');
EXEC REPOSITORY.ENCAPSULATE('BB111', 'BRANCH_NAME', 0);
EXEC REPOSITORY.ENCAPSULATE('BB111', 'SYS_DATE', 0);
...
EXEC REPOSITORY.SET_ENC_DATA_PROPERTIES(132, 'VARCHAR2', 'T');
EXEC REPOSITORY.SET_ENC_DATA_VALUES
 (133, NULL, 'BEGIN :A:=ANYDATA.CONVERTDATE(SYSDATE);END;',NULL);
```

Fig. 12. The construction of atomic component BB111

```
L_LIST(1) := 'BB111';
L_LIST(2) := 'BB111';
L_REF_CONN := REPOSITORY.CREATE_COMPOSITE_COMPONENT('B11', 'SELECTOR', L_LIST);
...
EXEC REPOSITORY.ENCAPSULATE('B11', 'BANK_NAME', 0);
.....
EXEC REPOSITORY.SET_ENC_DATA_VALUES
        (145, NULL, 'BEGIN :A:=ANYDATA.CONVERTDATE(SYSDATE);END;',NULL);
```

Fig. 13. The construction of composite component B11

These three elements are sufficient to build the entire bank system outlined above. Our first component is an atomic component for a bank branch (BB111). It is constructed from BB and an invocation connector (Fig. 12). In Fig. 12, the first command creates the component, and the third defines one of its encapsulated data elements, SYS_DATE. This variable is stored in the repository with a unique ID (integer) which is used at run-time in initialising the data element. The last command assigns SYS_DATE a script ('BEGIN...END;') that returns the system date when executed. This script is required in every branch component, therefore it has been assigned to SYS_DATE in the design phase. The rest of BB111's encapsulated data is not initialised until the final system has been constructed.

We assume that the business logic for all bank branches is the same. Therefore, bank components are constructed from copies of BB111. In Fig. 13 we construct a bank component B11 using a selector connector to compose two copies of BB111. The resulting composite component encapsulates its own data. Its SYS_DATE is also initialised with the same script used for BB111. Data encapsulated in B11 is independent of its sub-components' data. Fig. 14 shows a listing of B11 specification where data encapsulated in its sub-components has not been influenced by the composition. The composition process has preserved the sub-components' encapsulated data and propagated data encapsulation to the next level.

Similarly, a bank consortium component BC1 can be created in the same way. We also define its encapsulated data. Finally, we create the bank system component (BS) from two copies of BC1 and a definition of its local date. With this step, the system is complete and it is possible to proceed with data initialisation. Data initialisation is based on knowledge provided by the repository on the components' encapsulated data. Many steps similar to those for initialising SYS_DATE are performed to make BS ready to run.

```
SQL> EXEC GET_DATA_SPECS('BS');      -- Listing of B11 component architecture & its data
.............
___Data Name_____Q_S_Type_____Initial Value_____________________________________
– B11 renamed by the repository
.    BANK_NAME      1 T VARCHAR2   B11
.    SYS_DATE       2 T DATE       BEGIN :A := ANYDATA.CONVERTDATE(SYSDATE)....
.    BRANCH         3 P TABLE
– BB112 renamed by the repository
.    BRANCH_NAME    1 T VARCHAR2 BB112
.    SYS_DATE       2 T DATE       BEGIN :A := ANYDATA.CONVERTDATE(SYSDATE)...
.    CUSTOMERS      3 P TABLE
– BB111 renamed by the repository
.    BRANCH_NAME    1 T VARCHAR2 BB111
.    SYS_DATE       2 T DATE       BEGIN :A := ANYDATA.CONVERTDATE(SYSDATE)...
.    CUSTOMERS      3 P TABLE
```

Fig. 14. Data encapsulation in composite component B11

It is clear from the design phase process outlined above, how data encapsulation is supported by the composition scheme in our model. Composition preserves data encapsulation and propagates it. Furthermore, reuse by copying has been demonstrated in the creation of BS; all branches are copies of BB111, banks are copies of B11 and consortia are copies of BC1.

At run-time, BS and its sub-components must first be initialised with data, before BS can be instantiated. After this, creating a BS component results in creating instances for all its sub-components. Each component constructor creates its independent data instances and stores them in a data space identified by a CID. A data trace extracted from the run-time system for two instances of B11 and B11' shows different data instances for each component (Fig. 15).

The system can now receive client requests such as withdraw, deposit, balance, etc. This is illustrated by Fig. 16. By getting its account information via an ATM, the top-level connector reads the client's consortium code (BCC) to decide which consortium to direct control to. The consortium's top-level connector (BC2) reads the bank code (BC) to choose the bank. The bank then reads the client sort code (SC) to determine the client's bank branch. The bank branch reads the service requested, account number and amount, processes the client's request and returns a report. In this process, each component's top-level connector performs the necessary I/O operations it needs to coordinate control flow to the right bank branch. Fig. 16 shows a client request for (*withdraw)* to branch BB212.

DATA_NAME	COMP. NAME	CID	References to data in data space
BANK_NAME	B11	229	29AF4BB598B860ABE0440003BA3A89CB
SYS_DATE		229	29AF4BB598B960ABE0440003BA3A89CB
BRANCH_NAME	BB111	230	29AF4BB598BB60ABE0440003BA3A89CB
SYS_DATE		230	29AF4BB598BC60ABE0440003BA3A89CB
BRANCH_NAME	BB112	231	29AF4BB598BF60ABE0440003BA3A89CB
SYS_DATE		231	29AF4BB598C060ABE0440003BA3A89CB
BANK_NAME	B11'	232	29C6EA036AF801A1E0440003BA3A89CB
SYS_DATE		232	29C6EA036AF901A1E0440003BA3A89CB

Fig. 15. Data trace at run-time for two instances of B11

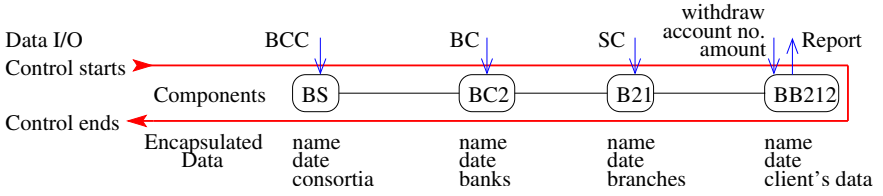

Fig. 16. Processing a withdraw request for a BB212 client

6 Discussion and Related Work

In current component models, a component is either an object or an architectural unit [9,5]. Components that are objects in these models are not compositional and so data encapsulation in composites is not meaningful. However, data encapsulation does occur at the level of atomic components. In component models where components are architectural units, composites are defined and can be (new) entities in their own right. Therefore, in these models, data encapsulation is potentially meaningful for both atomic and composite components. Of these models, only Koala [15] and PECOS [4] address data encapsulation.

In Koala, data is specified as attributes and data components. The latter are modules (non-interface components). Initialisation of attributes is expressed as either provides interfaces or requires interfaces. In a composition, both kinds of interfaces must either be exposed via the interface of the composite, or satisfied internally by a data component. Such a data component is encapsulated inside the composite, which compromises reuse if the data encapsulated is application specific. For example, to implement the bank system in Koala, encapsulated data must be initialised at the composite level, thus breaching information privacy of customers of each branch. An alternative is to initialise data at the level of each branch component. However, this compromises reuse. Therefore, in Koala, composition and reuse are conflicting concerns. Furthermore, copying and multiple instantiation are not supported because their components are C modules.

PECOS models data as *ports*, *attributes* (properties) and *connectors*. A port is a type containing a data type, range and direction. Attributes are constant data that can be specified for any PECOS component. A connector is a data type shared between two or more ports. Connections between components in a composite can not cross the composite's boundary. A connector between two composites represents an independent variable which must be synchronised with data variables held at the connected ports [4].

With respect to data, PECOS components can be classified as controllers (with their own thread of control) or passive components. The first category includes active and event components. Composites in PECOS are hierarchical. In a composite, the root must be a controller component. A controller holds data and shares it with all the passive components it controls (its sub-components). This

leads us to conclude that the general notion of data encapsulation is achieved only at the level of composite components. Components' ports are points for passing, but not holding, data. Because of the need for data synchronisation among components, data initialisation in PECOS is not recommended to be performed in constructors, but rather in a special method provided by the model. Copying is not possible in PECOS, but multiple instantiation is possible because a component defines its encapsulated data and has its own data constructor method to create its data instances.

In contrast to these models, our model provides a constructor which is the only method needed for instantiating the component and its data. Data initialisation occurs at the level of each component's top-level connector, and not recursively, as in Koala. Copying and multiple instantiation is supported because of our approach to data in the model.

Our notion of data encapsulation is defined at the level of component models, not at the level of programming languages. In particular, it should not be confused with encapsulation in object-oriented languages, where objects can encapsulate private data. Our notion of data encapsulation comes with composition, whereas data encapsulation in objects does not. No object-oriented language provides a single programmatic operator for composition that preserves data encapsulation in the way that our composition connectors do.

Furthermore, our notion of data encapsulation with composition leads to more reuse, via copies and instances. In the bank example, we only need one atomic component and one connector to build the whole system. This kind of increased reusability is not found in other component models. For instance, in Koala, copies and instances are not possible, and in PECOS, copies are not possible.

Finally, in [7] we defined data encapsulation in a different way. We defined it as a way to handle data operations separately from all other operations in component-based systems. This is unrelated to our definition here. Previously, separating dataflow and control flow was our focus. In that context, our goal was achieved by storing data in a global space, but only at the expense of encapsulation. In the present work, we have achieved separation and encapsulation at the same time.

7 Conclusion

In this work, we have encapsulated data in components. Our goals have been achieved by extending the semantics of the composition connectors and, accordingly, the composition scheme, in the component model we proposed in [6]. As a result, data encapsulation has not only become an invariant property of the scheme but it is also propagated in composition to newly constructed components (components are self-similar). Furthermore, data encapsulation has enabled component copying at design time, and multiple instantiation at runtime. Data encapsulation and reuse are not conflicting concerns in our model, in contrast to other models. Therefore, our model provides truly reusable software building blocks.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
2. Clements, P.C.: A survey of architecture description languages. In: 8th Int. Workshop on Software Specification and Design, pp. 16–25. ACM Press, New York (1996)
3. DeMichiel, L., Keith, M.: Enterprise JavaBeans, Version 3.0. Sun Microsystems (2006)
4. Genssler, T., Christoph, A., Schulz, B., Winter, M., Stich, C.M., Zeidler, C., Müller, P., Stelter, A., Nierstrasz, O., Ducasse, S., Arévalo, G., Wuyts, R., Liang, P., Schönhage, B., van den Born, R.: PECOS in a Nutshell (September 2002), http://www.pecos-project.org/
5. Lau, K.-K.: Software component models. In: Proc. ICSE06, pp. 1081–1082. ACM Press, New York (2006)
6. 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)
7. Lau, K.-K., Taweel, F.: Towards encapsulating data in component-based software systems. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 376–384. Springer, Heidelberg (2006)
8. Lau, K.-K., Velasco Elizondo, P., Wang, Z.: Exogenous connectors for software components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
9. Lau, K.-K., Wang, Z.: A survey of software component models. 2nd edn., Pre-print CSPP-38, School of Computer Science, The University of Manchester (May 2006), http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp38.pdf
10. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
11. Microsoft. Data access development overview: within the Microsoft Enterprise Development Platform. Microsoft Enterprise Development Strategy Series. Microsoft (March 2005), http://msdn.microsoft.com/netframework/technologyinfo/entstrategy/default.as px
12. Monson-Haefel, R.: Enterprise JavaBeans 3.0, 5th edn. O'Reilly & Associates (2006)
13. Russell, J.: PL/SQL User's Guide and Reference, 10g Release 1 (10.1). Oracle (2003)
14. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
15. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Computer 33(3), 78–85 (2000)