The University of Manchester

September, 2006

Computer Science
University of Manchester

Data Encapsulation in Component-based Software Systems

Kung-Kiu Lau and Faris M. Taweel

School of Computer Science
The University of Manchester
Preprint Series
CSPP-39

Data Encapsulation in Component-based Software Systems

Kung-Kiu Lau Faris M. Taweel

September, 2006

Abstract

A component-based system consists of components linked by connectors. Data can reside in components and/or in external data stores. Operations on data, such as access, update and transfer are carried out during computations performed by components. Typically, in current component models, control, computation and data are mixed up in the components, while control and data are both communicated by the connectors. As a result, such systems are tightly coupled, making reasoning difficult. In this report we demonstrate an approach for encapsulating data.

1 Introduction

A software system consists of three elements: *control*, *computation*, and *data*. The system's behaviour is the result of the interaction between these elements. Therefore, the latter determines whether the system has desirable properties such as loose coupling and ease of analysis and reasoning. It is reasonable to expect that it is advantageous to encapsulate these elements, and separate them from one another, since this should make reasoning more tractable. For example, some recent research in stimulus reactive systems has focused on separating control flow from data flow [16, 11]. There is even a component model for software agents that separates dataflow and control flow; and encapsulates control [22].

Paradoxically perhaps, for component-based software systems, it is not any easier to achieve such separation of concerns. A component-based system consists of components linked by connectors, as exemplified by software architectures [27]. In such a system, data can reside either in components or in external databases (which are often also regarded as components). Operations on data, such as access, update and transfer are carried out during computations performed by components. Typically, in current component models, control, computation and data are mixed up in the components, while control and data are both communicated by the connectors. As a result, such systems are tightly coupled, making reasoning difficult. More seriously, this impedes component reuse, which is a key objective for component-based development.

In this report, we present an approach for encapsulating data in components for a component model we are developing. Currently, our component model is based on encapsulating control and computation [19]. We therefore introduce into our component model the notion of data encapsulation. We explain the underlying principles and demonstrate the feasibility of this development.

In the sequel, we outline the component model in Section 2 . Then, in Section 3, data encapsulation is introduced. In Section 4, we apply our approach to a bank system. In Section 5, we evaluate our data encapsulation approach and conclude by Section 6.

2 Encapsulation in our Component Model

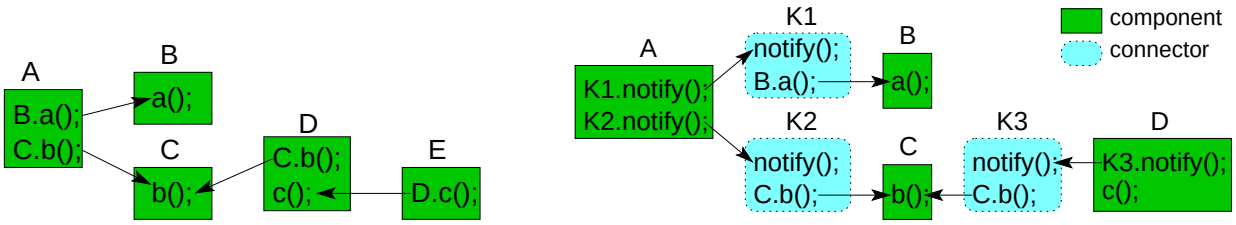
The notion of encapsulation is an essential concept to our component model. *Components* encapsulate computation and data, the latter being the focus of this report. Composition operators, realised as connectors, encapsulate control. In our model, components are constructed from *computational units* and connectors. A computational unit can be a class, module, Oracle package, etc. constrained to perform computations within its boundaries. It should not invoke other units' methods or access data external to its boundaries. Connectors are used to construct atomic components from computational units, and compose them to form composite components. As compositional operators, connectors operate on two or more components whether atomic, composite or a mix of both kinds.

2.1 Encapsulating Control in Connectors

Connectors in our component model are exogenous [19]. Their distinctive feature is that they encapsulate control. In current component models, connection schemes use message passing, and fall into two main categories:[1] (*i*) connection by direct message passing; and (*ii*) connection by indirect message passing. Direct message passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Fig. 1 (a)), using method or event delegation, or remote procedure call (RPC).

[1] For a survey, see [20].

Component models that adopt direct message passing schemes as composition operators are EJB, CCM, COM [4] etc.

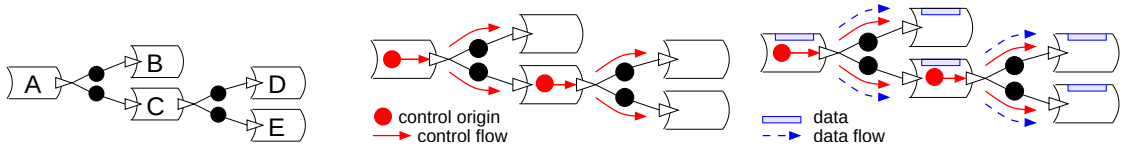


(a) Direct message passing (b) Indirect message passing

Figure 1: Connection by message passing.

Indirect message passing corresponds to coordination (e.g. RPC) via connectors, as exemplified by ADLs. Here, connectors are typically glue code or scripts that pass messages between components indirectly. A connector, when notified by a component invokes a method in another component (Fig. 1 (b)). Besides ADLs, other component models that adopt indirect message passing schemes are JavaBeans [14], Koala [28], SOFA [3] etc.

In connection schemes by message passing, direct or indirect, control originates in and flows from components (Fig. 2(b)). This is clearly the case in both Fig. 1(a) and (b). Furthermore, computation, control and data are intermixed (Fig. 2). Components initiate control and perform computation (Fig. 2(b)). Connectors provide communication between components for both control and data where both flow in tandem between components (Fig. 2(c)). Clearly in current component models, neither control nor data is encapsulated.



(a) Components and connectors (b) Control flow (c) Data flow

Figure 2: Current component models.

In our model, components do not call methods in other components, and control originates in and flows from exogenous connectors, leaving components to encapsulate computation and data. This is illustrated by Fig. 3. Fig. 3(a) shows an example of exogenous connection. Here components do not call methods in other components. Instead, all method calls are initiated and coordinated by exogenous connectors. The latter thus encapsulates control, as is clearly illustrated by Fig. 3(b), in contrast to Fig. 2(b). Exogenous connectors thus truly encapsulate control, i.e. they *initiate* and *coordinate* control.

2.2 Encapsulating Computation in Components

As is clearly evident from the previous section, components are not artifacts of a programming language, such as classes, modules, packages, etc., nor are they simply parts of a whole. In our model, a component is a new concept. It is a software unit intrinsic to which are the encapsulation and compositionality properties [17].

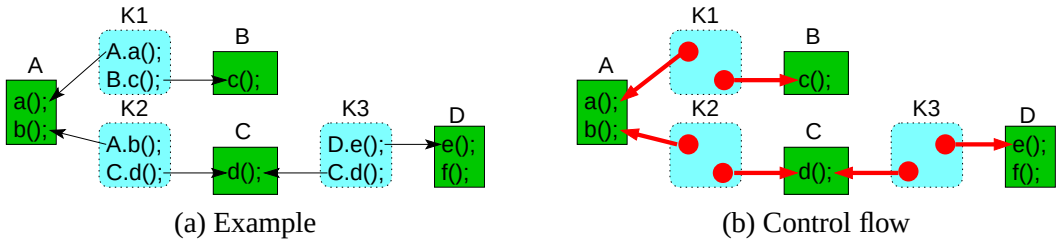


Figure 3: Connection by exogenous connectors.

A component encapsulates both computation and data. Data encapsulation is equivalent to making the component's data private. Encapsulating computation is restricting all computational tasks performed by the component not to cross its boundaries.

The notion of encapsulation in computing is not new. It has acquired different meanings depending on the paradigm or discipline. In the object-oriented paradigm, encapsulation is localising data together with the methods (computation) that operate on this data, in objects, and restricting access to data exclusively to these methods. While data is encapsulated, computation is not. Objects (instances of classes) perform method call on other objects (4(a)).

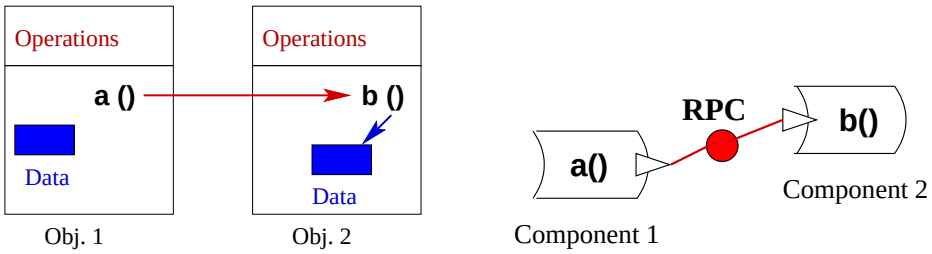


Figure 4: Objects and architectural units.

Component models supporting data components such as EJB [10], CCM [25] and .NET [21] can encapsulate data, but not computation. The same applies to port-connector type components in ADLs [27], UML2.0 [9, 24] and Koala [1]. Components in the above models can call methods in other components directly or via remote procedure calls (RPC) over connectors (and ports) (Fig. 4(b)).

The other property of a component, compositionality, represents the ability to construct new components from existing ones. The composition of two components A and B must lead to a new component C defining the same characteristics of A or B, i. e., encapsulation and compositionality.

In contrast to our components, objects and classes are not compositional. Method calls are the only means used for objects and classes. However, as these objects and classes are 'composed', the result is not a new class or object nor does this composition preserve computation. Port-connector type components can compose but because they do not encapsulate computation, they also not compositional.

Alone, encapsulation is not beneficial. To utilise this concept, a means for accessing hidden computation, control and data must be made available: It is the *interface*. Objects and classes provide no real interfaces. While they have 'interfaces', these are no more than advertising rather than access to encapsulated properties. Methods of objects and classes are directly accessible, not via interfaces. Port-connector components use their ports as their interfaces.

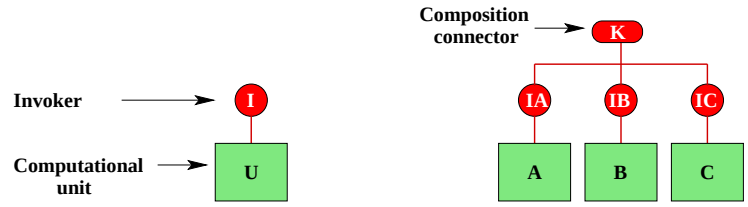
In our component model, components are built from computational units and exogenous connec-

tors. As stated previously, a computational unit only performs computational task that do not cross its boundaries. That is, method calls by one computational unit against others is not permitted. It also has no direct access to external data. Consequently, computational units encapsulate computation and data.

Exogenous connectors encapsulate control (Section 2.1). They are hierarchical in nature, with a type hierarchy [19]. At the bottom of the hierarchy, and because components are not allowed to call methods in other components, we have an exogenous *method invocation connector*. This is a *unary* operator that takes a component, invokes one of its methods, and receives the result of the invocation. To structure the control flow in a set of components or a system, at the next level of the type hierarchy, we have other connectors for sequencing exogenous method calls to different components. So we have n*-ary* connectors for connecting invocation connectors, and n*-ary* connectors for connecting these connectors, and so on. As well as invocation connectors, we have defined and implemented *pipe* connectors, for sequencing, and *selector* connectors, for branching.

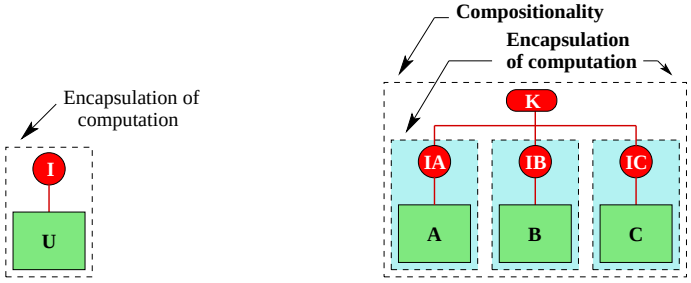
There are two different kinds of components: *atomic components* and *composite components*. An atomic component is constructed from a computational unit satisfying our requirements above and an invocation connector (invoker for short). The invoker exposes an interface for the atomic component. A composite component on the other hand is one or more components composed together by a connector of arity equal to the number of components where the connector exists at level ≥ 2. The connector is called the *composition connector* and exposes the composite interface interface.

Diagrammatically, a computational unit U connected to an invoker I forms an atomic component (Fig. 5). The component encapsulates computation, since the computational unit does so, and the invoker invokes methods provided by the computational unit (Fig. 6(a)). Clearly, a composite component encapsulates computation as depicted in Figure 6(b).



(a) An atomic component (b) A composite component

Figure 5: Atomic and composite components.



(a) An atomic component (b) A composite component

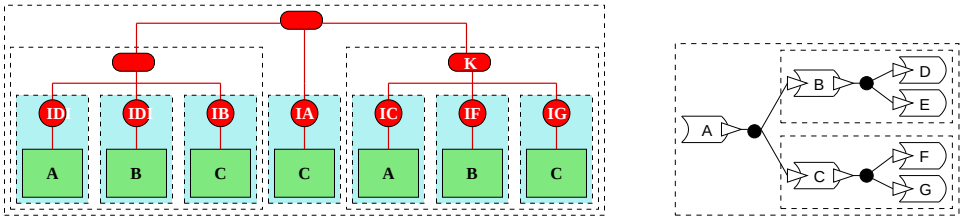
Figure 6: Encapsulation and compositionality.

2.3 Composition that Preserves Encapsulation

To construct composite components (and systems in some cases), we need composition operators that preserve encapsulation and compositionality. These operators must work on components' interfaces, which is a requirement imposed by encapsulation. The operators can not be glue code. Methods calls of objects and ADL connectors, used in current component models, are too not suitable. Component models lack proper compositional operators, in our view, because of the absence of encapsulation and compositionality.

Accordingly, we use exogenous connectors at levels ≥ 2 as compositional operators. These operators are compositional and therefore preserve and propagate encapsulation. Figure 7(a) shows a system which is a composite components. It has a top-level connector that exposes its interface. The system (composite component) is composed from other components, each of which has a top-level connector too. These components, whether atomic or composite, are all compositional. They encapsulate control and computation.

Figure 7(a) illustrate an important property, viz. *self-similarity*. Each dotted box indicates a composite component. Starting from atomic components, dotted boxes encapsulates one another until the outer most box is achieved which is in fact our system. It is clear from the diagram that the structure of every composite component is similar to that of each of its subcomponents.



(a) Exogenous connection (b) Acme

Figure 7: Self-similarity of a composite component.

The significance of self-similarity is that it provides the basis for a compositional way to constructing systems from components. Component models do not provide such a basis. For example, Figure 7(b) represents an Acme architecture of the system depicted in Figure 7(a). While (E,D) is a composite component (7(a)), it is not in Figure 7(b). In Figure 7(b), (B,D,E) is not similar to the top-level component, since the later has no interface.

2.4 A Temperature Controller example

Let's build a simple component-based system for a temperature controller using three *computational units*: (i) $ControlPanel$ that has two buttons to increment and decrement temperature one unit at a time; (ii) $T+$ that processes requests for incrementing temperature; and (iii) $T-$ that decrements temperature. The computational units are not components yet. $T+$ and $T-$ are designed with high reusability in mind. It is possible to use them with other hardware that increases or decreases one step at a time, say, ventilation levels, or any other similar task.

In the system requirements, it is stated that temperature must be controlled within the range $[17, 27]$ via a panel that has two buttons: one to increment and the other to decrement temperature, one unit at a time. Therefore, we construct the atomic components T_UP from $T+$, T_DOWN from $T-$ and $Panel$ from $ControlPanel$. As functionality of each of these atomic components is well understood,

the next step is to compose T_UP and T_DOWN using a selector. The new composite UP_DOWN is composed with $Panel$ using a pipe connector. The resulting composite is in fact the required temperature controller.

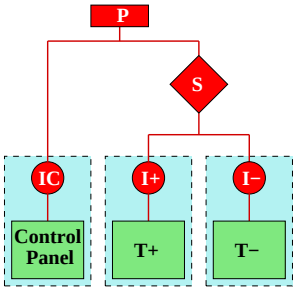


Figure 8: Temperature controller system.

All components in Figure 8 are well encapsulated. Each atomic components is composed from an invoker and a computational unit. The later implements provided services such as $increase$, $decrease$, etc; and invokers expose the functionality necessary to invoke these methods. That is, invokers provide interfaces for the respective atomic components. For example, T_UP encapsulates computation that implements the $increase$ method (as well as encapsulating data) in such a way that processing a request is completely performed within itself.

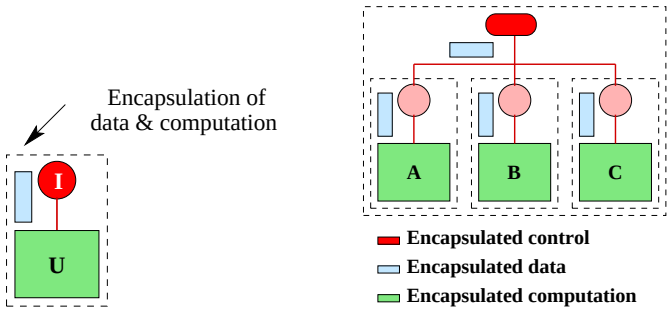
With regard to compositionality, all components above are compositional. The temperature controller itself is a component with a pipe composition connector as its interface. The composite (system) encapsulates computations, control and data in its subcomponents, T_UP and T_DOWN.

In this simple example, we have demonstrated encapsulation and compositionality with regard to computation and control. However, data related issues have been ignored in this example, though mentioned in the specifications. For example, data input/output as well as how the temperature range is enforced are not clear. We direct our attention to data in the next section.

3 Encapsulating Data

We have presented, in Section 2, our component model and its underlying concepts of encapsulation and compositionality. In that context, only two elements of software systems have been addressed: computation and control. The third element however, *data*, is yet to be addressed. In our model, connectors encapsulate control and atomic components encapsulate computation. Since composite components are constructed from composition operators and components, they must encapsulate control and computation. The model is unique in its approach of encapsulation and compositionality. We believe that extending these concepts to encompass data in a way that preserves self-similarity will maximise the benefits of encapsulation and compositionality. More importantly, it will allow for constructing efficient systems with regard to performance and space. Therefore, the question we want to address in this report is how we can encapsulate data in our component model.

Initially, computation and data are introduced into our components by computational units. Constructing an atomic component from a computational unit and an invoker definitely results in encapsulating computation. We also assume that the construction process introduces data into the component which is also encapsulated. In this case, the data originates from the computational unit, since the invoker functionality is only to invoke methods implemented by the computational unit. In Figure 9



(a) An atomic component (b) A composite component

Figure 9: Encapsulation of computation, control and data.

(a), encapsulated data indicates that local data of the computational unit has become the component's encapsulated data.

Data encapsulation must have the same meaning for composite and atomic components. To illustrate, and without loss of generality, we compose two atomic components. Each atomic component has its own encapsulated data as in Figure 9 (a). The resulting composite component can also have its own encapsulated data. Looking at it from a design point of view, the composite is always the result of a design decision that usually involves data [13, 2]. The data, rooted in the system requirement, is encapsulated as data of the composite (Fig. 9(b)).

It is important here to point out that other encapsulated data also exists in a component as a result of the component construction. But this data is constant and only accessible by the top-level connector of the component. Thus, we are not considering it in our treatment because it is not influential and it is therefore ignored.

Example 3.1 : In this example we focus on data aspects of the temperature controller system presented in Section 2.4. In particular, we trace the origin of each component's data and address data input and output.

From a reuse point of view, $T+$ and $T-$ are actually generic software modules which provides incrementing and decrementing functionality, one unit at a time. These units can be used to control any hardware whose behaviour changes by increasing or decreasing a property one step at a time. In addition to temperature, the units can be used, for example, to change ventilation levels of a climate control system. The difference between these units is in the data used to set their operating minima or maxima. Therefore, creating atomic components from them results in creating data local to each component. T_UP constructed from $T+$ and T_DOWN constructed from $T-$ create data that sets their maximum (Max_T) and minimum (Min_T) values respectively. We observe that each data requirement originally local to $T+$ or $T-$ has become encapsulated data for each atomic component.

Composing T_UP and T_DOWN to construct UP_DOWN also creates additional data local to UP_DOWN. This data is not necessarily related to data local of T_UP and T_DOWN. To illustrate this aspect, we state more requirements of the temperature controller. In particular, the system must remember two temperature settings, a factory and a user temperature settings. The UP_DOWN component must encapsulate data introduced by the new requirements, namely $User\_T$ and $Factory\_T$ (Fig. 10). In this case, data encapsulated by UP_DOWN does not originate from its sub-components. □

To access encapsulated data in components, we need to decide on whether data is accessed from

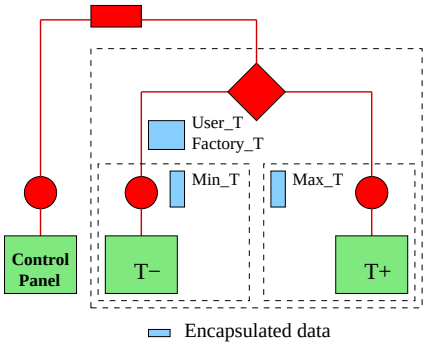


Figure 10: Data encapsulation in the temperature controller system.

control, computation or both. The later option is immediately excluded because a computational unit is required to have no access to other computation and data external to its boundaries. Thus, control, the only other part of any system, must allow for data access operations. The new requirements entails semantical changes to the existing component model. In particular, since the control part of any system is specified using connectors where the top-level connector of a component might arbitrarily be of any type, each connector must be capable of data input and output.

Related to I/O operations are data *initialisation* and *finalisation*. When a component is created, it often needs to initialise data. Similarly, the same component may also need to finalise (persist, garbage-collect, etc.) part (or all) of its data before its destruction. Lacking the capability for data access by the encapsulated computation, entails more modifications to the semantics of control connectors. Therefore, we equip every connector with a data initialisation and finalisation sections. These sections contain data access operations necessary for the component to perform upon its construction and destruction. Each section is executed once throughout the lifetime of a component. We use an up-ward triangle atop the connector to indicate a non-empty initialisation section and a down-ward triangle situated at the bottom of the connector to to indicate a finalisation section.

Both modifications introduced into the semantics of control connectors preserve data encapsulation. A connector is now responsible for data I/O access operations. Direct access to encapsulated data is forbidden and can only be achieved indirectly via the component's interface. Similarly, the initialisation/finalisation roles of a connector screen this data from external access. Furthermore, since each top-level connector of a component specifies data I/O, initialisation and finalisation data access operations, the component is actually encapsulating the required and provided data. We illustrate these modifications by elaborating on Example 3.1.

Example 3.2 In Figure 10, Min_T is data encapsulated by T_DOWN; Max_T is encapsulated by T_UP; and $User\_T$ & $Factory\_T$ are encapsulated by UP_DOWN. The forth component, $Panel$, encapsulates no data. It is just a dummy input terminal of the temperature controller system. Values T input via $Panel$ are either $+1$ or -1 indicating requests for incrementing or decrementing temperature. No variable holding T data is shown in Figure 10 because T is just a parameter of $increase$ and $decrease$ methods provided by the UP_DOWN component.

When the system is started, all data above must be initialised. Min_T and Max_T are set to $17C^0$ and $27C^0$ respectively. $Factory\_T$ and $User\_T$ are set to $19C^0$ and the previous user setting respectively. The system is always initialised to operate at $User\_T$. A new temperature controller system has $Factory\_T$ and $User\_T$ set to $19C^0$. At shutdown, the system writes $User\_T$ value to a persistent data store so that it is remembered the next the system is started.

Data	Component	Initialisation	Finalisation	Value
Min_T	T_DOWN	$\surd$		$17C^0$
Max_T	T_UP	$\surd$		$27C^0$
$Factory\_T$	UP_DOWN	$\surd$		$19C^0$
$User\_T$	UP_DOWN	$\surd$	$\surd$	$19C^0 \mid t$

Table 1: Encapsulated data in the temperature controller and data access operations.

The above operations are specified in initialisation and finalisation sections of the relevant control connectors. Table 1 shows encapsulated data, initialisation and finalisation.

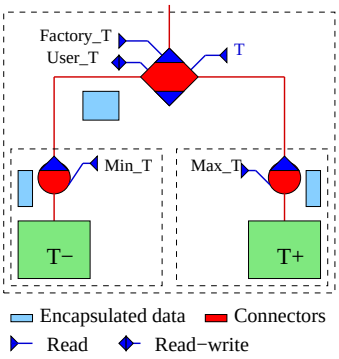


Figure 11: Data encapsulation in the temperature controller system.

Finally, we specify data I/O of the system. Based on Table 1, the $Panel$ component represents a data input operation and it encapsulates no computation. Therefore, and according to our new semantics, it can be replaced by a $read$ data access operation that inputs T value. As a result, the top-level connector, which is a pipe, is no longer needed. Data access operations of $Factory\_T$ and $User\_T$ are $read$ and $read - write$ operations respectively, provided by the selector connector. Min_T and Max_T are $read$ operations performed by invokers of T_DOWN and T_UP respectively. Figure 11 presents the new architecture of the temperature control system. It is clear from the figure that each component encapsulates its data. Furthermore, as each connector specifies its provided and required data, components additionally encapsulates these data access operations. So, each level in the control hierarchy encapsulates data and access operations upon the next higher level. Moreover, the new data semantics of connectors introduce a significant simplification to the system architecture (Fig. 11) when compared to the previous architecture in Figure 10. □

Compositionality has not be affected by upgrading the semantics of the composition connectors. Components are still compositional in just the same way as before. In fact, the new semantics has enforced compositionality because (*i*) data encapsulation in components results from composition and (*ii*) some connectors such as pipes establish data relationships among sub-components. Furthermore, the encapsulation of data together with a component's required and provided data requirements does not impede compositionality. However, the new semantics imposes an additional requirement on composition: persistent data requirement of a sub-component in a composite must be propagated to the top-level interface of the composite, where it is explicitly expressed. Otherwise, it will not be possible to correctly deploy the component. For example, in Figure 11, all encapsulated data (except for T) in the system is persistent. Accordingly, each component interface must explicitly express its data persistence require-

ments: (i) T_UP requires Max_T, (ii) T_DOWN requires Min_T and (iii) UP_DOWN requires $Factory\_T$ and $User\_T$. At design time, (i) and (ii) are propagated to the interface of UP_DOWN. The three requirements may show up as separate entries in the UP_DOWN or merged in one entry. The later represents a requirement for one persistent data store.

Finally, despite the modifications introduced into the semantics of composition connectors, self-similarity is still preserved. Every component encapsulates data together with its I/O data requirements. Persistent data (provided or required) for a composite is propagated to the top-level connector of a composite (or a system). The new semantics do not break self similarity with regard to encapsulating computation nor does it impede compositionality.

4 Example: The Bank System

Using a simple application, we now demonstrate encapsulation of data, control and computation as well as compositionality and self-similarity. The example we have chosen is a bank consortium system, whose architecture was described in terms of our component model without data encapsulation in [19]. The system has just one ATM that serves two bank consortia ($BC1$ and $BC2$), each with two bank

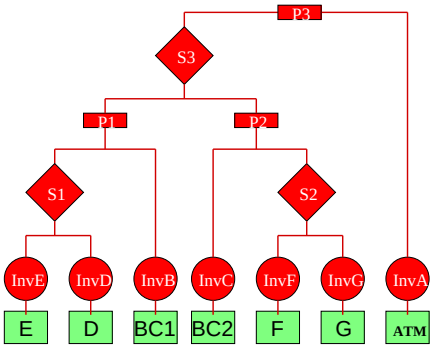


Figure 12: Architecture of the bank consortium application (no data encapsulation).

branches (E and D, F and G respectively). The ATM passes customer requests together with customer details to the customers bank consortium, which in turn passes them on to the customers bank branch. The bank branches provide the usual services of withdrawal, deposit, balance check, etc. (Fig. 12).

We have re-designed the system making use of data encapsulation introduced into the semantics of our component model. Atomic components are constructed from bank branches' systems. Each atomic component encapsulates the branch's data and computation. Having atomic components created, a number of composition operations are performed, using composition connectors, to yield the bank system. With each operation performed, data and control is encapsulated at another level in a new composite component. Composition is a bottom-up design process which is performed in piece-wise manner.

At level one, an invocation connector is connected to every computational unit (branch system). This enables all the services of a component to be invoked. Data pertinent to that specific bank branch is also encapsulated in the component. Usually, this data is comprised from persistent data in many forms: files, databases, etc. Invocation connectors that are used to create an atomic components, must furnish data I/O operations required by the component's provided services (parameters and return values). For example, the *withdraw* method requires two input parameters: account number and amount. It also outputs a report. The Invoker of each bank branch must provide all these I/O operations for the withdrawal method

to run correctly. Composition starts at level two where atomic components are connected by level-two connectors of type selector to effect appropriate behaviour among these components. Two components representing bank consortia result from the later composition: $BC1$ and $BC2$. Each bank consortium must encapsulate data on on its branches and PINs that belongs to each one of them. BCs' selector provide two I/O operations: the first to read the PIN and the second to read $BCode$ based on the input PIN. At level three, a selector is used to compose $BC1$ and $BC2$ to form the bank consortium system BS. The selector provide one I/O operation to read the bank consortium code ($CCode$). It is interesting to notice that, in the same way a bank branch encapsulates its data, each bank consortia encapsulates its data, too. The later is persistent data on bank branches and PIN codes that belongs to each one of them.

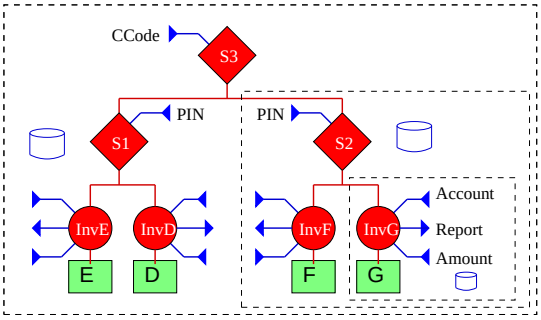


Figure 13: Architecture of the bank consortium application with data data encapsulation.

Execution of the system starts at the top-level connector of BC; this connector is the one to initiate control flow. In the previous implementation [19], the bank example was implemented using Java objects (connectors and computational units are classes) (Fig. 12). There, only control and computation were encapsulated. Data was left to follow control starting from the ATM component to bank branch for processing and results (statement, say) flow back to the ATM. In this simple example, data size and its route length may not be significant. In general, these two issue raise performance and space considerations. Data encapsulation provide an efficient solution the problem. In our design (13), data required by a components is encapsulated in the component itself. For example, the BC component encapsulates the data relating bank branches and PIN numbers; and each branch encapsulates its own data on its customers' accounts. For each component, be it a branch or the bank consortium, the top-level connector is responsible for data I/O operations.

Consequently, data encapsulation serves not only the reduction of problems related to performance and space, but also encapsulates data where it is needed. Furthermore, self-similarity is preserved. BS is similar to any of its sub-components $BC1$ and $BC2$ as well as to bank branches E, D, G and F. Every component encapsulates data and computation, and all composites additionally encapsulate control.

5 Related Work and Evaluation

In our component model, data is introduced into components when they are first constructed from computational units and invocation connectors. At this level, data is only accessible via invocation connectors; i.e., it is encapsulated in each atomic component. When atomic components are composed to build new composite components, the composition operation introduces additional data. The resulting data is also encapsulated since it is only accessible by the top-level connector of the component. It is clear

that, in both cases, the introduction of encapsulated data into components is a direct consequence of the composition operation performed using composition connectors that specify data semantics. Composition connectors defines semantics for data I/O as well as for data initialisation and finalisation. With our scheme of composition, data encapsulation has become systematic and at the same time a direct product of composition. In this sense, we argue that our approach to data encapsulation is more comprehensive. In fact, the term data encapsulation designates a new notion that current component models do not address, though encapsulation is a major objective of the component-based approach.

In the sequel of this section, we justify our claim based on a survey on data approaches in current component models [18]. The survey included JavaBeans [14], Enterprise JavaBeans (EJB) [10], Component Object Model (COM) [4], .NET [21], CORBA Component Model (CCM) [25], Koala [28], SOFA [26, 8], KobrA [1], Architectural Definition Languages (ADLs) [27], UML 2.0 [9, 24], PECOS [12], Pin [15, 29], Fractal [7] and MALEVA [5]. Our approach is to investigate whether composition schemes in current component models involve data semantics and assess these semantics with respect to our notion of data encapsulation. Some component models distinguish elementary from composite components, while others do not recognise the notion of a composite. Therefore, we treat each case independently before addressing data in the model as a whole. However, we start by outlining explicit approaches to data in current component models. Component models address data explicitly using three main abstractions: (i) *properties* or *attributes* that are mainly used, but not limited to, component configuration; (ii) *data component*s for modelling data sources particularity persistent data; and (iii) *data port*s as a means for data I/O.

Attributes are local variables used to store data in components. The data can be of any type supported by the specification environment. Furthermore, these attributes can be single-valued or collection variables. Access to an attribute value is restricted via *setter* and *getter* methods, called data accessor methods: a setter method sets the value of an attribute and a getter method returns its value. These methods are means for specifying accessibility of attributes: an attribute with both methods specified is a read-write attribute, while an attribute with one method indicates a read or write access depending on which accessor method is used. Depending on the component model, accessor methods can be provided services (part of the provides interface) or required services (part of the requires interface). Components supporting attributes might establish relationships based on the values the attributes can have. A component may place a constraint on the value of an attribute of another component. The attributes are called constrained attributes. Also, an attribute of one component may trigger a behaviour in another component. These attributes are called bound attributes.

Few component models provide data components as a vehicle for modelling various differing sources of data such as relational databases, XML files and even data local to applications. These models are: EJB, CCM, .NET, UML 2.0 and KobrA. However, the way data components are defined vary from one component model to the other. Variations include the services they provide, the way they model data sources and their scope of applicability. Basically speaking, a data component specifies and uniquely identifies a persistent state together with possibly a behaviour (transaction). Data components are managed by their implementation or by the component model infrastructure (container). A data component must be bridged to the data store it models via a data connector (except for UML 2.0 and KobrA whose components are modular units of a system). These connectors are either provided as part of the infrastructure implementation or provided in the model semantics.

Ports are points of interaction among entities. The interaction can be a method invocation that triggers an internal behaviour of the entity owning the port, or a pure data access operation. The later may also incur behaviour. We distinguish these ports by calling them data ports. Data access imposes many characteristics such as direction of data flow, type information and constraints. For data ports, the method of interaction can be by method calls or events in which case they exchange typed messages. In

this context, it is data ports that we are concerned with. Functional ports comprise functional interfaces. However, it is important to notice that data ports might be the only ports an entity can have as their interaction points with their environments.

Factors leading to data encapsulation at the atomic level are different from those at the composite level, though they may interact to enhance or degrade encapsulation at the composite level. Henceforth, we address encapsulation at the atomic and composite levels separately.

At the level of atomic components, all models support data encapsulation because access to data and computation is only permitted via interfaces. Component models with explicit approaches to data guarantee encapsulation via attributes, data components and data ports. But there are exceptions. PECOS, which is a dataflow-oriented component model, makes data publicly available on data ports. Therefore, atomic components do not encapsulate data. Another interesting case can be found in CCM's data components. Transparent persistence results in data components with a public state. This kind of persistence occurs when CCM data components are directly specified using a programming language such as Java. However, the persistent state is encapsulated for CCM data components that are specified using Persistent State Description Language [23]. In this case, state members are only accessible via accessor methods. Encapsulation at this level is also improved by explicit and constructive data semantics. For example, Koala forbids any data declarations in its interfaces. The requirements forces all data to be internally represented in the component and accessed via accessors. Other models allow data declarations for constant data (EJB) or variable data (COM) in their interfaces. However, excepts for the above two cases, atomic components encapsulate data as part of their construction process.

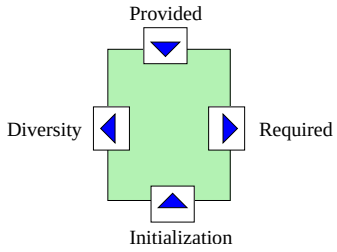
On the second level, composition schemes are more influential on encapsulation than data approaches. In fact, many component models do not support explicit data approaches. These models include COM, SOFA, ADLs and Pin. We even argue that data approaches counter data encapsulation at the composite level since they factor out the data concern from the composition semantics. This is clearly the case for data components. The encapsulation of a data component in a composite does not result from the composition operation. The composition operation has no semantics relevant to data components, since data component types are treated as any other component type in the model. Accordingly, component models supporting data component types do not encapsulate data. EJB, CCM, .NET, UML 2.0 and KobrA are members of this category. Fractal is to follow the approach of this category, but data components in Fractal are still a draft proposal [6].

Semantics that most affects data encapsulation is the definition of the composite entity. JavaBeans, EJB, COM, .NET, and CCM do not define the notion of a composite entity. Therefore, the most encapsulation they support is that at the atomic level. Other composition semantics are also important. These semantics are best presented in Koala and PECOS. Therefore, we discuss them in the context of their component models.

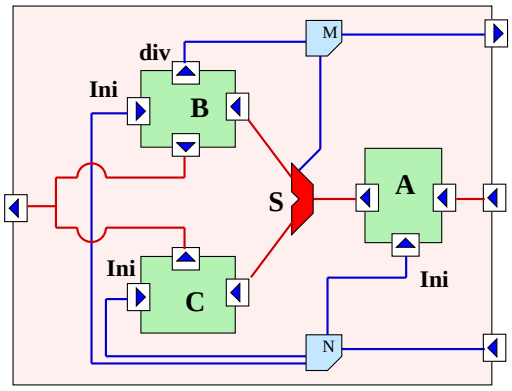
In Koala, components do not contain configuration specific information. A component must parameterise all of its configuration requirements, and specify function calls that set them via interfaces called *diversity interfaces*. For Koala atomic components, diversity interfaces must be satisfied in exactly the same way as any other requires interface. However, in composite components and system templates, diversity interfaces must be implemented in a module external to the sub-components. The module acts as target for all function calls originating from sub-components and *connectors*. In Koala, modules are treated as interface-less components and allow for connecting them alongside normal components. The role of modules in this case is a store for configuration data of a composite or a system. Modules are necessary requirements for composites that result from the composition operation.

There is also one more case where modules are used to store data in Koala composites and systems. Initialisation requirements for components are expressed via accessor methods in provides interfaces. In composites, initialisation must also be performed using modules. The need for these modules is

necessitated by the connection semantics of interfaces: a requires interface can receive many provides interfaces, but a provides interface can only connect to one requires interface. That is, a function has only one implementation that can be called by many interfaces. Figure 14 depicts a Koala component that has one interface of each kind and how two modules are connected in a composite to satisfy configuration and initialisation requirements. A component can have provides, requires, diversity and initialisation interfaces (Fig. 14 (a)). Component B has a initialisation (Ini) and a diversity (div) interface. Components A and C have initialisation interfaces. All initialisation interfaces for A, B and C are connected to the module N. Component B diversity interface is implemented by module M, which also implements configuration requirements of the connector that connects all the sub-components in the composite (Fig. 14 (b)).



(a) An atomic Koala component



(b) A composite Koala component

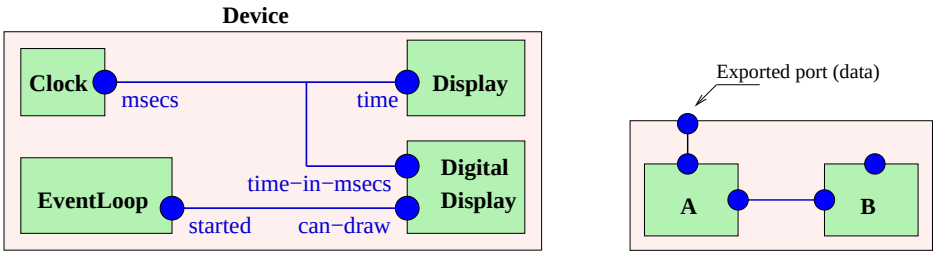
Figure 14: Data in Koala components.

It is clear that Koala supports data encapsulation at the level of composite components and system templates. For atomic components, Koala hides attributes by enforcing their access via accessor methods. No data declaration is allowed in Koala interfaces.

In PECOS, components are units that contain computation and data. The behaviour of a component is represented by the activation of its computation part which consumes data available on the component's ports (or internal component data) and produces some data on its ports. A port is a reference to data that can be read and written by a component and enables a component to be connected to another component (through a connector). Clearly, ports act as public providers and receivers of data. Thus, the data they hold is public at the level of atomic components. In the design phase, connectors are used to build composite components where connectors describe data-sharing relationships between the ports of sub-components. The composition operation results therefore in encapsulation of data on connected ports. In Figure 15 (a), a digital clock system is presented. Any individual component encapsulates no data, even in the sense of objects; data is public by virtue of the ports' semantics. But, the composite (a system in this case) encapsulates three data sets through connections: (Clock.msecs, DigitaDisplay.time-in-msecs), (Clock.msecs, Display.time) and (EventLoop.started, DigitaDisplay.can-draw). However, the encapsulation of a set breaks if any of its ports is exported to the environment (Fig. 15 (b)).

Furthermore, access to a sub-component (data port) from other components external to the composite is forbidden. In fact, access can only be establish via connectors determined at design time.

The above discussion leads to the conclusion that all models (in general) support data encapsulation at the atomic level (except for PECOS and CCM transparent persistence). Only two component models support data encapsulation at the composite level: Koala and PECOS. In our discussion, we have ex-



(a) Internal data for a system (b) Public data for a composite

Figure 15: Data in PECOS.

posed important factors that influence encapsulation: these factors include (i) explicit data approaches that are not a requirement for encapsulation in atomic components (few models do not have them), (ii) the necessity for the definition of the composite entity in the model so that encapsulation becomes relevant, and (iii) most importantly, the need to embed data semantics into the composition operators rather than factoring it out in independent types such as data components.

Our component model provides one component type and adopts no explicit data approaches. It embeds the semantics of data I/O together with data initialisation and data finalisation into the semantics of composition connectors. At the atomic level, components encapsulate data as part of their construction process from computational units and invocation connectors. Composites encapsulate data as part of the composition operation effected by the composition connectors equipped with data semantics. As more composition is performed that may involve other composites at the design and deployment phase, data continues to be encapsulated. We believe that our model goes beyond any other model in this regard, particularly PECOS and Koala. The first does not encapsulate data at the atomic level, while the second allows both components and connectors to access data modules. In our model, only composition connectors can access data, setting a clear separation of concerns. PECOS and Koala do not define composition in the deployment phase which restricts data encapsulation to the design phase only. Our model places no restrictions on composition, and data encapsulation is a characteristic that is maintained consistently with composition in design and deployment phases.

6 Conclusion

In this report we have presented a way to encapsulate data in a component model that already encapsulates control and computation. Our approach to encapsulating data is effected through changing semantics of the composition connectors. The changes include the following: (i) allowing composition connectors and invokers to perform I/O operations and (ii) allowing each connector to have an initialisation and a finalisation sections for initialising and finalising data. With the new semantics, atomic components encapsulate computation and data, and composite components additionally encapsulate control. Our approached has resulted in encapsulating data at both levels: atomic components and composite components. Further composition operations involving composites also maintain data encapsulation.

Not all current component models encapsulate data in both kinds of components. Two main categories have been observed in Section 5. One category provides encapsulation at the level of atomic components (with two exceptions) which includes all current component models. The other, encapsulate data at the composite level and includes PECOS and Koala. Both models embed into composition operations data semantics that results in data encapsulation. However, PECOS does not encapsulate

data at the atomic level, and Koala allows connectors and components to access encapsulated data. Both models do not support further data encapsulation in the deployment phase because composition is not allowed.

The immediate benefits of our approach is encapsulation all of the elements of a software system: computation, data and control; which leads to improved reusability, verifiability and amenability to reasoning. Furthermore, the examples used in this report demonstrate modelling simplifications by eliminating data I/O devices and data sources.

The resulting model is, to the best of our knowledge, the first model with encapsulation of control, computation and *data*. Compared to related work, viz. component models, our work seems unique in encapsulating the three elements: computation, control and data.

References

[1] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wust, and Jorg Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, 2002.

[2] Carliss Y. Baldwin and Kim B. Clark. *Design rules: The Power of Modularity*, volume 1. MIT Press, London ; Cambridge, Mass., 200.

[3] Dusan Balek. Connectors in software architectures, 2002.

[4] Don Box. *Essential COM*. Addison-Wesley, Harlow, 1998.

[5] Jean-Pierre Briot, Thomas Meurisse, and Frédéric Peschanski. Une expérience de conception et de composition de comportements d'agents à l'aide de composants. *L'Objet*, 11(3), 2006.

[6] E. Bruneton, T. Coupaye, and J.B. Stefani, editors. *Perseus: The Persistence Framework Specification, version 1.3 (Draft)*. The ObjectWeb Consortium, France, January 2004.

[7] E. Bruneton, T. Coupie, and J. B. Stefani. The fractal component model. Technical report specification, v2.0-3, The ObjectWeb Consortium, Feb. 2004.

[8] Lubomir Bulej. Generator of connectors for sofa/dcup. Master thesis, Technical University in Prague, Prague, May 2002.

[9] John Cheesman and John Daniels. *UML components : a simple process for specifying component-based software*. Component software series. Addison-Wesley, London ; Boston, Mass., 2001.

[10] Linda G. DeMichiel, editor. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 12 2003.

[11] Berndt Farwer and Mauricio Varea. Object-based control/data-flow analysis. Technical Report DSSE-TR-2005-1, University of Southampton, Department of Electronics and Computer Science, Highfield, Southampton SO17 1BJ, United Kingdom, March 2005.

[12] Thomas Genssler, Alexander Christoph, Benedikt Schulz, Michael Winter, Chris M. Stich, Christian Zeidler, Peter Mller, Andreas Stelter, Oscar Nierstrasz, Stephane Ducasse, Gabriela Arevalo, and Roel. Pecos in a nutshell, 2002.

[13] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM Press.

[14] Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A, August 8 1997.

[15] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt C. Wallnau. Pin component technology (v1.0) and its c interface. Technical Report CMU/SEI-2005-TN-001, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA 15213-3890, April 2005.

[16] Ouassila Labbani1, Jean-Luc Dekeyser1, and Pierre Boulet1. Mode-automata based methodology for scade. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, volume LNCS 3414, pages 386–401, Berlin Heidelberg, March 9-11 2005. Springer-Verlag GmbH.

[17] Kung-Kiu Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F.S. de Boer *et al*., editor, *Proc. 4th International Symposium on Formal Methods for Components and Objects, LNCS 4111*, pages 1–21. Springer-Verlag, 2006.

[18] Kung-Kiu Lau, Faris M. Taweel, and Perla Velasco. A survey on data approaches in component models. Survey CSPP-XX, The University of Manchester, Manchester, UK, October 2005.

[19] Kung-Kiu Lau, Perla I. Velasco, and Zheng Wang. Exogenous connectors for components. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE'05)*, May 2005.

[20] Kung-Kiu Lau and Zheng Wang. A survey of software component models (second edition). Preprint CSPP-38, The University of Manchester, Manchester, UK, May 2006.

[21] Juval Lowy. *Programming .NET components*. O'Reilly, Sebastopol, Calif., 2003.

[22] Thomas Meurisse and Jean-Pierre Briot. Une approche base de composants pour la conception dagents. *Technique et science informatiques*, 20(4):583 – 602, 2001.

[23] OMG. *Persistent State Service Specification, version 2.0 - formal/02-09-06*. Object Management Group, http://www.omg.org/docs/formal/02-09-06.pdf, Sept. 2002.

[24] OMG. *UML 2.0 Superstructure Specification*. Object Management Group, http://www.omg.org/docs/ptc/03-08-02.pdf, 2002.

[25] OMG. *CORBA Component Model Specification, V4.0*. Object Management Group, http://www.omg.org/docs/formal/02-06-69.pdf, 2006.

[26] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: architecture for component trading and dynamic updating. In *The Fourth International Conference on Configurable Distributed Systems*, pages 43 – 51. IEEE Press, May 1998.

[27] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[28] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[29] Kurt C. Wallnau and James Ivers. Snapshot of ccl: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA 15213-3890, November 2001.