Incremental Construction of Component-based Systems

Kung-Kiu Lau, Keng-Yap Ng, Tauseef Rana and Cuong M. Tran
School of Computer Science
The University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu,ngk,trana,tranc@cs.man.ac.uk

ABSTRACT

Building large and complex systems in one step (the 'big bang' approach) is a very challenging task, given that humans can only deal with a limited measure of complexity at a time. A more practical approach would be to build such systems incrementally, i.e. iteratively increment an incomplete version of the system under construction until the system is completed. In software engineering, there are such approaches, but they are generally top-down, and not component-based. In this paper we present a component-based approach, which is bottom-up, and demonstrate its feasibility by applying it to the CoCoME example.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Software component model, incremental construction, component composition

1. INTRODUCTION

The general context of this paper is incremental development, i.e. "feedback-driven refinement with customer involvement and clearly delineated iterations" [21]. However, we do not address the whole process of incremental development. Rather we focus only on growing the system in increments, i.e. incremental *system* construction, or just *incremental construction*, for short.

As the name suggests, incremental construction builds a system by iteratively incrementing an incomplete version of the system under construction until the system is completed. This can be expressed as a sequence of systems S_i, starting from an initial one S_0, i.e. $S_0 \subseteq S_1 \subseteq S_2 \subseteq \dots$, where $S_i \subseteq S_{i+1}$ means S_{i+1} contains S_i plus an increment *inc*, i.e. $S_{i+1} = S_i + inc$.

For large and complex systems, this kind of iterative development process can provide a practical solution for managing scale and complexity, since it is widely accepted that humans can only deal with a limited measure of complexity at a time, such as that embodied in one increment.

In software engineering, many approaches to incremental construction have been proposed, all based on the notion of stepwise refinement [32, 12], i.e. building a complex program from a simple program by adding features incrementally. Some of these approaches focus on specification refinement, e.g. [1, 4]. In these approaches, both specifications and (their) implementations are theories, and a formal specification S_i (starting with an initial specification S_0) of the program is incrementally refined to a specification S_{i+1}. The refinement relation between S_i and S_{i+1} is defined by $S_i \sqsubseteq S_{i+1}$, which means that S_i is (logically) satisfied by S_{i+1}. S_{i+1} can therefore be regarded as an implementation of S_i. The aim of the refinement step is to produce an S_{i+1} that is increasingly executable in a chosen programming formalism.

Some approaches based on stepwise refinement focus on code refinement. In these approaches, an initial skeleton program is constructed, and then one feature is added to it at a time, until the program contains all the desired features. For example, in [6, 5], a feature-oriented programming language is used. Such a language is object-oriented: features are objects and are added together by means of object-oriented inheritance mechanisms, e.g. mixin inheritance [8].

However, all these approaches to incremental construction are not truly component-based. In fact they are all top-down: they start with either a specification of the whole program or a top-level (skeleton) program. This is in contrast to truly CBD approaches, which are bottom-up, i.e. starting from pre-built system-independent components [10, 11].

In CBD, system development is based on a *component model* [26]. Such a model defines components and their composition mechanisms; thus it fixes the ways in which a system can be constructed from components. Therefore it follows that a component model also defines ways in which a system can be incremented. More importantly, if composition is hierarchical, then scalability should be achievable. In this paper we propose an approach to incremental construction of component-based systems, and demonstrate its feasibility.

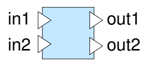
2. COMPONENT AND SYSTEM BEHAVIOUR

In incremental construction, $S_i \subseteq S_{i+1}$ means the *behaviour* of S_{i+1} contains the *behaviour* of S_i. Therefore, before we discuss incremental construction in CBD, we have to first define the behaviour of a system and the behaviour of an individual component.

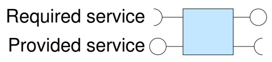
A system consists of three basic elements: (i) *control*; (ii) *computation*; and (iii) *data*. Control triggers computations, which are function or expression evaluations, assignments, etc. Computations are performed on data in the system.

The *behaviour* of a system is the result of the system executing its set of computations (according to its control flow) on its data.

We will consider systems made up of generic components. A generic component is an architectural unit [29, 27] with ports for inputs and outputs. This is illustrated in Fig. 1a, where `in1` and `in2` are input ports, whilst `out1` and `out2`



(a) Generic architectural unit.



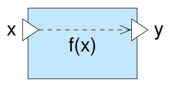
(b) UML2.0 component.

Figure 1: A generic component.

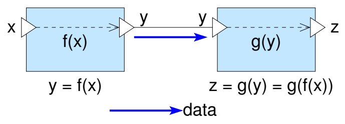
are output ports. A generic component is a unit of (control and) computation and/or data. Ports provide interaction points between components: specifically, components can pass control and/or data to one another via their ports.

A generic component can also be depicted as a UML2.0 component with services (Fig. 1b). Services are either methods or just data. In Fig. 1b, a socket denotes a required service, whilst a lollipop denotes a provided service. For services that are only data, each lollipop and each socket can be implemented simply as one data port. On the other hand, for services that are methods, each socket needs to be implemented as a pair of control/data ports: one for sending control (to make a method call) and data (associated with the call, i.e. method name and parameters), and one for receiving control (after a method call) and data (any return values for the call). Similarly, each lollipop needs to be implemented as a pair of control/data ports: one for receiving control (for a method call) and data (associated with the call), and one for sending control (after a method call) and data (any return values for the call). (An example of this can be seen in Fig. 4.)

Architectural units are composed by linking their ports. The two main styles of connectors[1] are 'pipe-and-filter' and 'rpc' (remote procedure call). In 'pipe-and-filter' style connections, components act as filters; only data values are passed between components, and the computation defined in a component is a mathematical function that maps the (data values on) input ports to (data values on) the output ports. This is illustrated in Fig. 2a.[2]



(a) A filter component.



(b) A composite filter component.

Figure 2: Pipe-and-filter connection style.

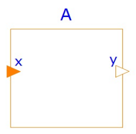
The behaviour of a filter component is fixed, with respect to control and computation; it is simply a function (`f` in Fig. 2a) of the component's input data (`x` in Fig. 2a).

[1]Others are 'event broadcast', etc., see [29].

[2]For simplicity, here we assume each component has only one input and one output port.

Furthermore, the behaviour of a composite filter component is also fixed (with respect to control and computation). This is illustrated in Fig. 2b, where the behaviour of the composite is the function `g(f(x))`.

Modelica [14] components are examples of filter components. Fig. 3 shows a Modelica component, together with its code. The behaviour of the component is defined by an equation (`y=2*x`) that expresses the data value on the out-



```
block A
 extends Modelica.Blocks.
  Interfaces.IntegerBlockIcon;
 ...
 equation
  y= 2 * x;
end A;
```

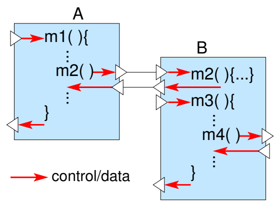
Figure 3: A Modelica component.

put port `y` as a function of the data value on the input port `x`, as in Fig. 2a.

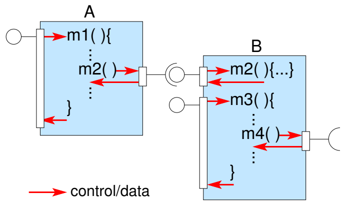
In Modelica, composite components are produced by 'pipe-and-filter' connection, as in Fig. 2b. The behaviour of a composite containing a component `A` is defined by a set of equations containing the equation for `A`, i.e. it contains the behaviour of `A`, as in Fig. 2b.

In 'rpc' style connections, the computation defined in a component can be called remotely by another component, and therefore messages containing rpc's and their results, if any, are passed between components. These messages pass control (for method calls) as well as data (for parameters and return values of method calls) between components.

This is illustrated in Fig. 4. The method `m1` of `A` calls the method `m2` provided by `B`. The method `m3` of `B` calls a



(a) Using ports



(b) Using services

Figure 4: RPC connection style.

method `m4` provided by some other component.

Clearly in 'rpc' connection style, the control and computation in a component is not fixed; they depend on control and computation defined in other components. Consequently, in contrast to filter components, the behaviour of an 'rpc' component is *not* fixed, as a far as control and computation are concerned; it is not simply a function of the component's input data (which now can include data for parameters of in-coming method calls as well as return values for out-going method calls), but is a function also of computation defined externally in other components. For example, in Fig. 4, the behaviour of `A` depends on (how `m2` is defined in) `B`; while the behaviour of `B` depends on (how `m3` gets defined by the method `m4` it calls in) whatever component `B` calls in order to provide `m3`.

It follows that any composite built by a 'rpc' connection also does not have fixed behaviour, with respect to control and computation. Indeed, in general, the behaviour of any component (composite) in a system is potentially only fixed when the whole system has been completely constructed.

ArchJava [2] components are examples of 'rpc' components. Fig. 5 shows ArchJava components that correspond to those in Fig. 4 and their composition.

```
component class A{
 public port a1{provides void m1();}
 public port a2{requires void m2();}
 public void m1(){
 . . .
 a2.m2();
 . . .
 }
}
```

```
component class B{
 public port b1{provides void m2();}
 public port b2{provides void m3();}
 public port b3{requires void m4();}
 public void m2(){
 . . .
 }
 public void m3(){
 . . .
 b3.m4();
 . . .
 }
}
```

```
component class AB{
 ...
 private final cA=new A();
 private final cB=new B();
 connect cA.a2, cB.b1;
 ...
 }
}
```

Figure 5: ArchJava components.

3. INCREMENTAL CONSTRUCTION IN CBD

Having defined the behaviour of generic components and systems of such components, we are now in a position to discuss incremental construction in the context of CBD.

A component-based approach to software development is based on a component model [26]. The component model defines what components are as well as composition mechanisms for composing components. How a component-based approach builds systems is thus determined by its component model. Clearly if the composition mechanisms defined in the component model are incremental, then so is the approach.

For example, in a component model in which components are filters, and the composition mechanism is 'pipe-and-filter' style connection, composition is incremental. This can be seen in Fig. 2b, where the behaviour of the composite is `g(f(x))`. This clearly contains the behaviour `f(x)` of the first component. Thus we can regard the second component as the current system and the composition of the two components as the new system, after incremental composition.

An example of a pipe-and-filter system is one that filters out letters 'a', 'b', 'c' . . . , from a piece of input text. Such a system can be built incrementally as follows. Start with a (current) system S_0 consisting of only a component A that filters out 'a'. Compose S_0 with a component B that filters out 'b'. This gives a new system S_1; clearly S_1 filters out both 'a' and 'b', and thus the behaviour of S_1 contains the behaviour of S_0, and so on.

A counter-example of incremental composition is *invasive composition* [3] when it is used to destroy component behaviour. In the component model underlying invasive composition, components are architectural units with hooks that provide write access to component code during composition. Thus as the name 'invasive composition' suggests, the behaviour of the components in the current system S_i can be altered by the composition. When invasive composition destroys component behaviour in S_i, the behaviour of S_{i+1} may not contain that of S_i. Of course, invasive composition could also achieve incremental composition by adding new behaviour to S_{i+1} without destroying the behaviour in S_i.

The most widely used component models in CBD are ADLs (architecture description languages) [29, 27]. In these models, components are generic architectural units and the composition mechanism is 'rpc' style connection. In such a model, composition is incremental in general, since 'rpc' style connection does not destroy any existing behaviour, unlike destructive invasive composition. However, incremental construction is difficult to achieve in practice. This is because the behaviour of any architectural unit, and therefore the behaviour of any current system S_i, may not be fixed (as shown in Fig. 4). As a result, it may be very difficult, if not downright impossible, to reason about behaviour containment by the next system S_{i+1}.

So which component model should we choose? We could choose a component model in which components are filters and composition is by 'pipe-and-filter' style connection. However, in such systems, components only generate and pass data among themselves, whilst control for the whole system is implicitly fixed, as a pipe, i.e. control flow between connected components is simply sequencing. In other words, two connected components are executed in sequence, with the data produced by the first component sent as input to the second component, which then executes. Such a component model obviously has limited expressiveness.

What about a component model in which components are 'rpc' components and composition is by 'rpc' style connection? Such a component model would have more expressiveness than a 'pipe-and-filter' model. However, as we explained above, it would be difficult to achieve incremental construction in practice with such a component model.

We will use a component model that is Turing complete in terms of expressiveness for defining systems. The model is called X-MAN, and its composition mechanisms are incremental. Furthermore, X-MAN also defines other kinds of increments, apart from components, that can be added to a system. Next we will briefly describe X-MAN and show how it can be used for incremental construction.

4. THE X-MAN COMPONENT MODEL

X-MAN has been described in other papers, e.g. [25, 22, 30, 18]. Here we give a brief summary, and show that composition in X-MAN is incremental.

In X-MAN there are two kinds of basic entities: *computation units* and *connectors*; computation units encapsulate *computation* whereas connectors encapsulate *control*.

Components are built from these basic entities. There are two kinds of components: atomic and composite (Fig. 6). An atomic component contains a computation unit and an invocation connector (Fig. 6a). The computation unit de-

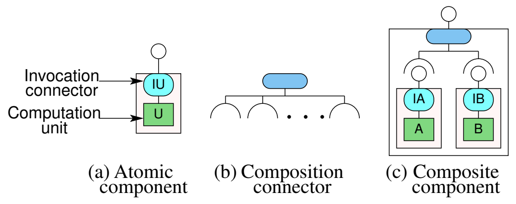


Figure 6: X-MAN component model.

fines *computation* in the form of methods or functions which can be invoked via the invocation connector. The computation defined by a computation unit `U` (when invoked) is performed entirely within the scope of `U`, i.e. `U` does not call other computation units. Thus an atomic component *encapsulates*[3] computation and has only provided services but no required services. In other words, the behaviour of an atomic component is fixed, with respect to control (coming from the invocation connector) and computation (defined in the computation unit).

[3]In the sense of 'enclosure in a capsule'.

A composite component (Fig. 6c) is built from atomic components by *composition connectors* (Fig. 6b). Fig. 6c shows a composite built from two atomic components. A composition connector defines *control* for coordinating components (as well as data flow between them). A composite component is also encapsulated, and it also has only provided services; this is a direct consequence of the encapsulation of an atomic component. The behaviour of a composite is also fixed with respect to control defined by the composition connector, and control and computation defined in the sub-components. Furthermore, the behaviour of a composite clearly contains the behaviour of the sub-components. Therefore composition in X-MAN is incremental.

Apart from the invocation connector in an atomic component, X-MAN connectors [30] include the Turing-complete set of control structures: sequencing, branching and looping. Sequencing and branching are *composition connectors* for multiple components, whilst looping is an adaptor for a single component. For sequencing, we have the *sequencer* and *pipe* composition connectors; for branching, we have the *selector* composition connector (Fig. 7a). A sequencer

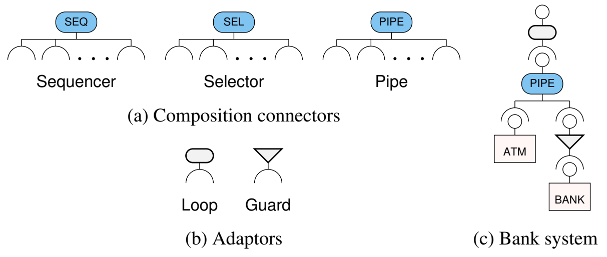


Figure 7: X-MAN connectors.

passes only control to the next component, whereas a pipe passes control and results from the first component to the next component, as shown in the bank system example in Fig. 7c, where customer details and requests are entered into an ATM and then passed, along with control, on to the bank.

For looping, X-MAN has a *loop* connector (Fig. 7b). The loop connector is an adaptor for a single component, adding a loop to the control defined by the top-level connector of the component. A loop connector must define a finite loop if used inside a composite; otherwise the composite will not be well defined. However, if used at the top level of a system, a loop connector can define an infinite loop, as in the Bank example (Fig. 7c).

Another adaptor in X-MAN is the *guard* connector (Fig. 7b). This defines a condition, and when applied to component, will only let control reach the component if the condition evaluates to true. For example, in the Bank example (Fig. 7c), a guard is applied to the bank to allow access only if the customer details entered into the ATM are valid.

5. INCREMENTAL CONSTRUCTION USING X-MAN

We have shown that composition in X-MAN is incremental. This makes it possible to use X-MAN for incremental construction. X-MAN's composition mechanisms can increment a system with additional behaviour: control (by adding composition connectors) and computation (by adding components). Furthermore, X-MAN's adaptors can also increment a system's behaviour: a loop adds repeated behaviour, whilst a guard adds alternative behaviour.

5.1 Adding Increments

Now we identify all the possible increments in X-MAN and explain how they can be added.

5.1.1 *By Composition*

We can add increments in control and computation to the current system by using the composition mechanisms, i.e. we can add a composition connector K_i and a (composite) component C_i to the current system S_i, and thus increment the behaviour of S_i by adding extra control (K_i plus composition connectors in C_i) and computation (computation units in C_i). This can be done in two ways:

(i) Adding the increments K_i and C_i to S_i itself, i.e. composing S_i with C_i using K_i.

This is illustrated in Fig. 8a, which is part of the bank system in Fig. 7c. It consists of an increment added to the ATM component by composing ATM with the Bank component. Here S_i is the ATM component, K_i is the pipe connector and C_i is the Bank component. The total increment added to S_i is circled in dotted line.

(ii) Adding the increments K_i and C_i to a sub-component of S_i.

This is illustrated in Fig. 8b, which shows a system constructed from the system in Fig. 8a by adding an increment to the sub-component Bank by composing Bank with the Bank2 component (to add another bank also linked to ATM). Here S_i is the system in Fig. 8a, K_i is the selector connector and C_i is the Bank2 component. The total increment added to Bank is circled in dotted line.

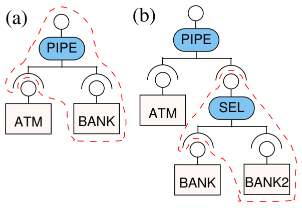


Figure 8: Composition.

These two alternatives allow incremental construction to be carried out on both atomic components and composite components alike, and therefore can be applied to an initial system S_0 that is just an atomic component, and to any subsequent S_i that is a composite component. Thus they define incremental construction in a recursive manner for any S_i. This is possible because of the *hierarchical* nature of composition in X-MAN.

5.1.2 *By Increasing Composition Connector Arity*

We can add increments in control and computation in the current system S_i by simply increasing the arity of any composition connector within S_i. Our composition connectors have variable arities (as can be seen in Figs. 6 and 7), and so it is possible to add components to an existing composite. In other words, we can add a (composite) component C_i to the current system S_i by increasing the arity of any composition connector in S_i by 1.

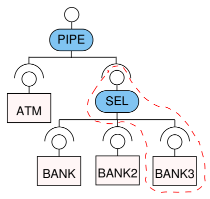


Figure 9: Increasing arity.

This is illustrated in Fig. 9, where an increment is added to the {Bank,Bank2} sub-component in the system in Fig. 8b, by increasing the arity of the selector connector by 1, and adding the Bank3 component (to add yet another bank to the network).

5.1.3 *By Adding Adaptors*

We can add increments in control and computation to the current system by adding adaptors to individual components within the system. We can adapt any sub-component C_i of the current system S_i with a finite loop; this will add a finite repetition of the behaviour of C_i. We can adapt S_i itself with an infinite loop to add an infinite repetition of the behaviour of S_i.

An example of an infinite loop can be seen in the Bank system in Fig. 7c, where the loop connector adds an increment to the system, which is the composition of ATM and Bank by a pipe connector.

We can also adapt any sub-component C_i of the current system S_i with a guard; this will add a piece of alternative behaviour: if the condition in the guard is true, then the behaviour of C_i is invoked in S_i; otherwise, it is not.

An example of a guard can be seen in the Bank system in Fig. 7c, where the guard connector adds an increment to the sub-component Bank.

5.2 The Construction Process

Having a component model that can be used for incremental construction is only half the story. The other 'half' concerns the iterative construction process itself. Clearly this process is guided by the system designer. Human guidance is needed since the initial system S_0 has to be chosen, as has every increment inc_i to be added at every i-th iteration. Whether the target system will be achieved depends crucially on these choices at each step.

Our basic tactic in each iteration of the construction process is to identify a piece of behaviour specified in the functional requirements, and then identify an increment that defines that behaviour, and add this increment to the current system. By repeating this, we hope to eventually arrive at a final system that satisfies all the requirements, i.e. it has all the behaviour specified by the requirements.

We assume that functional requirements for the system under construction are defined by *use cases*. For each use case, we apply our aforementioned tactic and incrementally construct a system that satisfies that use case.

We also assume that all the necessary components can be either (identified and) constructed from scratch or retrieved from an existing repository. That is, of the standard CBD life cycle [10, 11, 24] (which contains a component development process and a system development process) we focus only on the system development process.

The iterative construction process that we adopt is defined in pseudo code as follows:

```
1. Start with use case 1:
   (i)    from the use case, identify the initial system $S_0$;
   (ii)   from the use case, identify an increment $inc_0$;
   (iii)  construct $S_1$ by adding the increment $inc_0$ to $S_0$;
   (iv)   i:=1;
   (v)    from the use case, identify an increment $inc_i$;
   (vi)   construct $S_{i+1}$ by adding the increment $inc_i$ to $S_i$;
   (vii)  i:=i+1;
   (viii) repeat steps (v-vii) until all the requirements in the
          use case have been met.
2. Repeat step (viii) in 1 for each remaining use case.
```

Now we illustrate our incremental construction process using X-MAN on an example, the CoCoME example [28], which is a benchmark in CBD.

6. THE COCOME EXAMPLE

The CoCoME system is a trading system or point of sale (POS) system used in retail business.

6.1 System Overview

The CoCoME system comprises four sub-systems (Fig. 10): (a) cash desks (used by cashiers at the checkout), (b) store servers (used by the store manager), (c) enterprise server (used by the enterprise manager) and (d) stock exchanger (triggered by the enterprise server to re-allocate stocks between stores). In each store, there are many cash desks with a PC each. Each PC is connected to a store server. A store server contains a centralised repository that shares an inventory system among PCs in the store. All products are registered in the inventory on the store server serving queries from the PCs, as well as keeping transaction records from them.

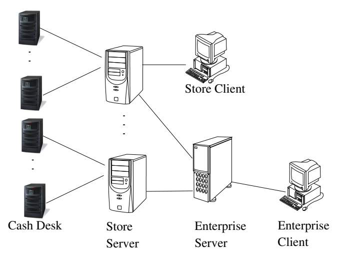


Figure 10: CoCoME system architecture.

Each store server is connected to the enterprise server at the headquarters. There are many stores under an enterprise and each store hosts a store server reporting its transactions and inventory status to the enterprise server. The enterprise server consolidates reports from the stores, and facilitates coordination of re-stocking in each store.

The cash desks subsystem to be deployed on PCs at the cash desks is the key part of the CoCoME system. Each cash desk consists of a cashbox, a barcode reader, a card reader, a display light (Fig. 11). The PC for the cash desk controls these peripherals. Besides keeping cash in the drawer, the cashbox also acts as a keyboard to receive input from cashiers as well as displaying output. The barcode reader is used to scan product IDs, the card reader is used to read credit/debit cards during payment, whereas the display light is used to indicate the operating mode of the cash desk (normal/express checkout).

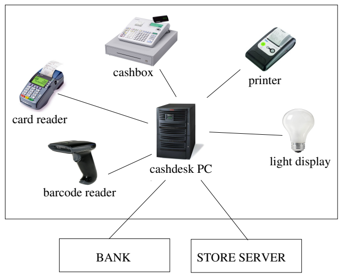


Figure 11: Cash desk components.

6.2 Use Cases

The functional requirements for CoCoME contain 8 use cases.

In Use Case 1 and 2, the actor is a cashier, and the subsystem is a cash desk. The use cases describe how a cashier uses a cash desk to execute the sale process in normal and express checkout modes respectively. In Use Cases 3, 4, 5 and 7, the actor is the store/stock manager, and the subsystem is a store server. The use cases describe how the store manager uses the store servers to order products, check-in products, retrieve stock reports and change product prices respectively. In Use Case 6, the actor is the enterprise manager, and the sub-system is the enterprise server. The use case describes how the enterprise manager uses the enterprise server to generate reports of 'mean time to delivery' for suppliers. Use Case 8 is an automated mechanism to re-allocate stocks between stores. The sub-system is the stock exchanger.

For lack of space, we will only discuss the cash desks sub-system described in Use Cases 1 and 2. In fact, we have implemented all 8 use cases.

Before we show our incremental construction for Use Cases 1 and 2, we summarise the use case here (for details see [28]). Use Case 1 is about the sale process conducted at the cash desk to checkout products in the customer shopping cart. The cashier initiates the new sale process by hitting the 'New Sale' button upon customer arrival at the cash desk with the products, and then starts entering product IDs to the system by using either a barcode reader or the keyboard on the cashbox. Information on each product will be retrieved from the store server and displayed on the display panel of the cashbox. Once all products are entered, the cashier will hit the 'Sale finished' button to calculate the amount due. Then, the cashier will need to select either cash or card payment mode. In cash mode, the cashier will enter the amount received and the system will display the change due as well as pop open the cashbox drawer. The cashier then keeps the money in the drawer and returns the change due if any. The transaction will be recorded and a receipt will be printed when the cashbox is closed by the cashier. Whilst in card payment mode, the customer will be required to insert a credit card into the card reader, and enter the pin to allow transaction to happen. The card ID and PIN are sent to the bank for validation. This repeats until a valid PIN is entered unless the cashier switches the system to cash payment mode. The transaction is complete upon successful PIN validation. The transaction will be recorded and a receipt will be printed for the customer.

Use Case 2 extends the *normal* checkout mode specified in Use Case 1 with *express* checkout mode. The system checks for the transaction history and evaluates if the cash desk shall switch to *express* checkout mode automatically. In *express* checkout mode, the display light will be switched on (from black to green), customers are restricted to checking out at most 8 products, and credit card payment is prohibited.

6.3 Construction of Cash Desks Sub-system

Now we outline our incremental construction for the cash desks sub-system specified in Use Cases 1 and 2, highlighting the kinds of increments that are added in various steps of the process.

When a customer arrives at the cash desk, the cashier presses the 'New Sale' button to create a new session. A new session resets the system and clears all the buffers from the previous transaction. The software component that has this behaviour is the session component (`SS` in Fig. 12) that provides the `newSale()` service. This component is therefore chosen as the initial system S_0.

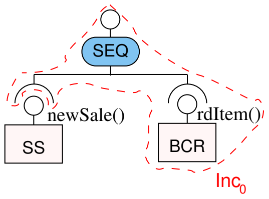


Figure 12: S_1

Next, the cashier is required to capture all the product IDs one by one using either the barcode reader or the keyboard on the cashbox. The behaviour of capturing a single product ID is provided by the **`rdItem()`** service that can be provided by two alternative software components, namely the **`barcode reader`** (BCR) and the **`keyboard`** (KB). We choose the BCR component first, which, together with a **`sequencer`** (SEQ) we identify as the increment Inc_0. Inc_0 is then added to S_0 to yield S_1 (Fig.12). This is an example of adding an increment to the current system by composition, as described in Section 5.1.1(i).

To add the alternative behaviour of capturing product IDs by the **`rdItem()`** service of the KB component, next we compose KB with the BCR sub-component using a **`selector`** (SEL) to yield S_2 (Fig. 13). This increment, Inc_1, consists of the **`selector`** SEL and the KB component. This is an example of adding increments to a sub-component of the current system by composition, as described in Section 5.1.1(ii).

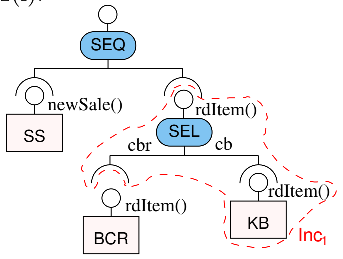


Figure 13: S_2

Then, the product ID captured is used to retrieve product price and description from the store server. This behaviour is provided by the **`getDetails()`** service of the **`server`** component (SVR), which is incorporated together with a **`pipe`** (PIPE) as the next increment Inc_2 to yield S_3 (Fig. 14). This is another example of adding increments to a sub-component of the current system by composition.

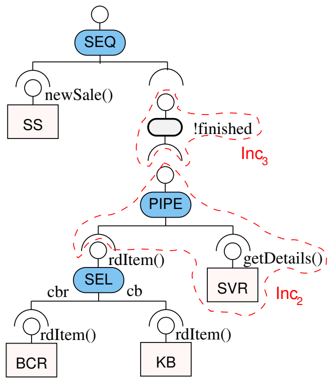


Figure 14: S_3 and S_4

The same behaviour has to be repeated for all products until all product IDs are captured, with the price and description of each product retrieved from the store server. The behaviour that requires repetition already exists in S_3, namely the sub-component of S_3 that is composed by PIPE. A finite `loop` (Inc_3) is therefore added atop PIPE to iterate this sub-component. S_4 shown in Fig. 14 is produced after Inc_3 is added. This is an example of adding an adaptor as an increment, as described in Section 5.1.3.

After all product IDs are captured, the customer can pay by either cash or card. These two alternative behaviours are addressed one by one incrementally. The cash payment is chosen first. In cash payment mode, the total amount due will be calculated by a calculator (or accumulator). The component that provides this behaviour is the `calculator`

(CAL) component that provides the `getPayment()` service. Since reading all the product IDs and calculating the amount due should happen sequentially, the CAL component can be added as teh incremetn Inc_4 to S_4 as the last component of the `sequencer` SEQ by increasing its arity, as described in Section 5.1.2. S_5 shown in Fig. 15 is produced as the result.

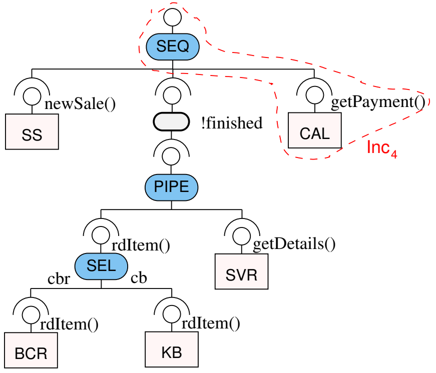


Figure 15: S_5

Next, a `guard` is added to ensure that the `getPayment()` service will only be invoked in cash payment mode. The guard is thus the increment Inc_5 and the system shown in Fig. 16 is the resulting system S_6. This is another example of adding adaptors as increments, as described in Section 5.1.3. The adapted sub-component is the CAL component.

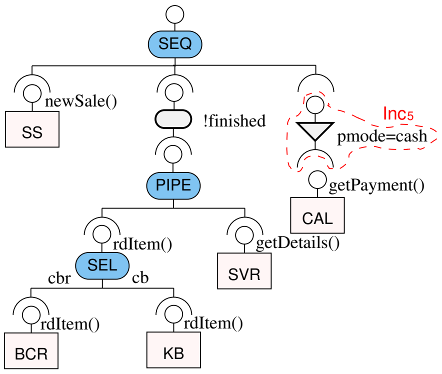


Figure 16: S_6

The above-mentioned process recurs until all behaviours specified in Use Case 1 are incremented to the system under construction. Since Use Case 2 extends Use Case 1, the same process also recurs to Use Case 2 to increment the system constructed from Use Case 1.

By carrying out the construction process for all other behaviours specified in Use Cases 1 and 2, we produced the cash desks sub-system in 23 incremental steps, and the mechanisms of adding increments used in each step is summarised in Table 1.

6.4 Refactoring

An important element of our incremental construction process, which for lack of space we can only discuss very briefly here, is *refactoring*. Refactoring [31, 13] changes a system's architecture without changing its behaviour. Refactoring is necessary for our incremental construction process because sometimes it is only possible to achieve the desired increment by first refactoring the current system architecture.

Table 1: Mechanisms used in cash desks sub-system

Mechanism	**Step**
Adding increments to current system by composition	1
Adding increments to a sub-component by composition	2, 3, 7, 9, 11, 12, 14, 18, 20, 23
Adding increments by increasing composition connector arity	5, 8, 15, 16, 17, 19, 22
Adding guards as increments	6, 10, 21
Adding loops as increments	4, 13

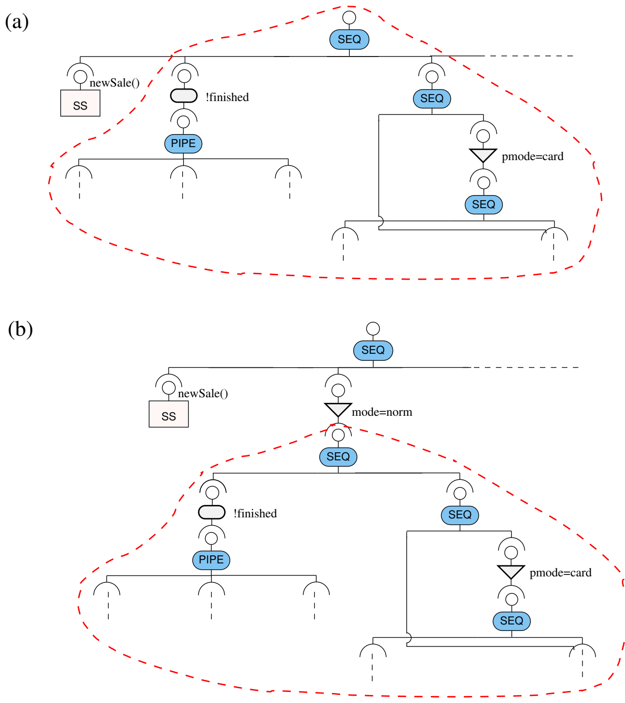


Figure 17: Refactoring sequencer: (a) before and (b) after

For instance, in the cash desks sub-system (Use Case 2) shown in Fig.17(a), the sub-component enclosed by the dotted line will only be invoked in `normal` checkout mode, hence a guard expressing `mode=norm` should be added atop the sub-component. Refactoring allows us to replace the top sequencer by two sequencers, thus creating two hierarchy levels so that a guard can be added between the two levels (Fig.17(b)). Obviously, without refactoring, incremental construction is unable to achieve this intended system behaviour.

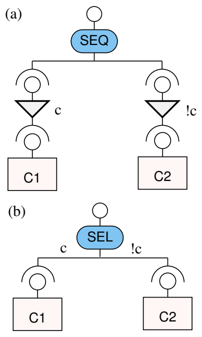


Figure 18: Refactoring connectors.

Refactoring also allows a set of connectors to be replaced with a single connector. For example, as shown in Fig. 18, a sequencer that composes two guards which are logical complements can be replaced with a selector. This happens in the cash desks sub-system too (Use Case 1): the guard for cash and card payment modes `(pmode==cash OR pmode==card)` can be replaced with a selector. This simplifies the system architecture.

Conversely, some basic connectors can be replaced with composite connectors which correspond to design patterns [23].

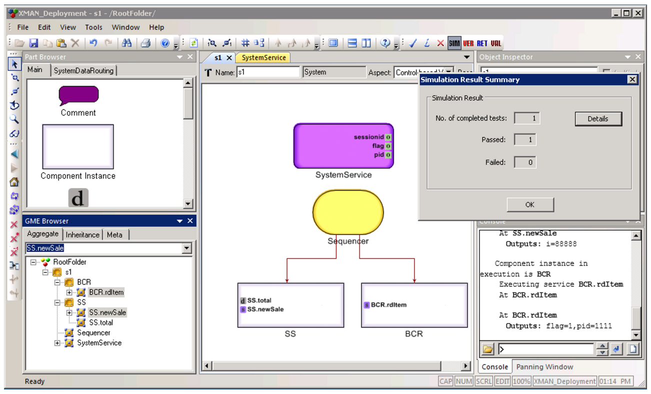
6.5 The Complete Implementation

The complete CoCoME system has been implemented in the X-MAN Tool. The X-MAN Tool is developed using Model-driven Engineering in GME [15], a generic modelling environment that generates a graphical editor tool from a metamodel of the elements of a system (in this case the elements of the X-MAN component model). In the X-MAN Tool, components are designed and developed in the C programming language [20] and stored in a repository. During the incremental construction process, these components are selected, deployed and composed with other components.

The complete CoCoME system is constructed by implementing its four sub-systems, each one constructed incrementally.

While constructing each sub-system, in every step of the construction process, it is essential to produce the correct behaviour for the newly incremented system. To ensure this, each newly incremented system was tested dynamically. The X-MAN Tool supports this kind of dynamic testing, since every sub-system in X-MAN is executable, and X-MAN provides a simulator for execution results. So we used X-MAN Tool to automate system simulation with test cases defined in XML [17].

Every test case defines inputs and asserts simulation outputs based on boundary value analysis. A test case passes simulation if the real simulation outputs match the asserted outputs. The X-MAN Tool will show the simulation result summary with Pass-Fail ratios and a detailed result showing the simulation traces (as shown in the super-imposed dialogue boxes in Figs. 19 and 21 respectively.



Simulation Result Summary

Simulation Result

No. of completed tests: 1

Passed: 1

Failed: 0

```
    At SS.newSale
      Outputs: i=88888

   Component instance in
execution is BCR
    Executing service BCR.rdItem
    At BCR.rdItem

    At BCR.rdItem
      Outputs: flag=1,pid=1111
```

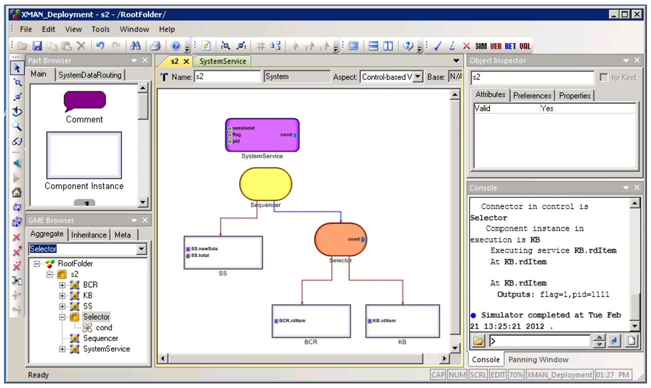
Figure 19: S1 in X-MAN Tool

Increments that did not behave correctly were duly replaced. We also built extra components if the required behaviours could not be fulfilled by components in the repository.

Figs. 19 and 20 show the implementations of S1 and S2 described in Figs. 12 and 13 respectively. The dialogue box super-imposed on the screen-shot in Fig. 19 (top right-hand corner) shows its simulation results summary.

Fig. 21 shows the implemented cash desks sub-system and its simulation results.

The top left-hand panel is the X-MAN Part Browser showing the X-MAN architecture elements. The GME Browser is on the bottom left, showing all deployed components. The



```
  Connector in control is
Selector
   Component instance in
execution is KB
    Executing service KB.rdItem
    At KB.rdItem

    At KB.rdItem
      Outputs: flag=1,pid=1111

● Simulator completed at Tue Feb
21 13:25:21 2012 .
```

Figure 20: S2 in X-MAN Tool

super-imposed dialogue box on the top right shows the simulation results of the system, and on the bottom right is the console, showing the simulation inputs and real outputs for each component. The super-imposed dialogue box on the bottom shows 15 completed simulation cases and the execution traces of the cash desks sub-system for Test Case 1 (normal checkout mode, keyboard input and cash payment mode). The test case passed because the real simulation outputs matched the asserted outputs (change==121.38, total==78.62). The middle pane shows the X-MAN architecture of the cash desk sub-system.

Database connections, e.g. between the cash desks and the store server (database), are not visible because they are internal to components.

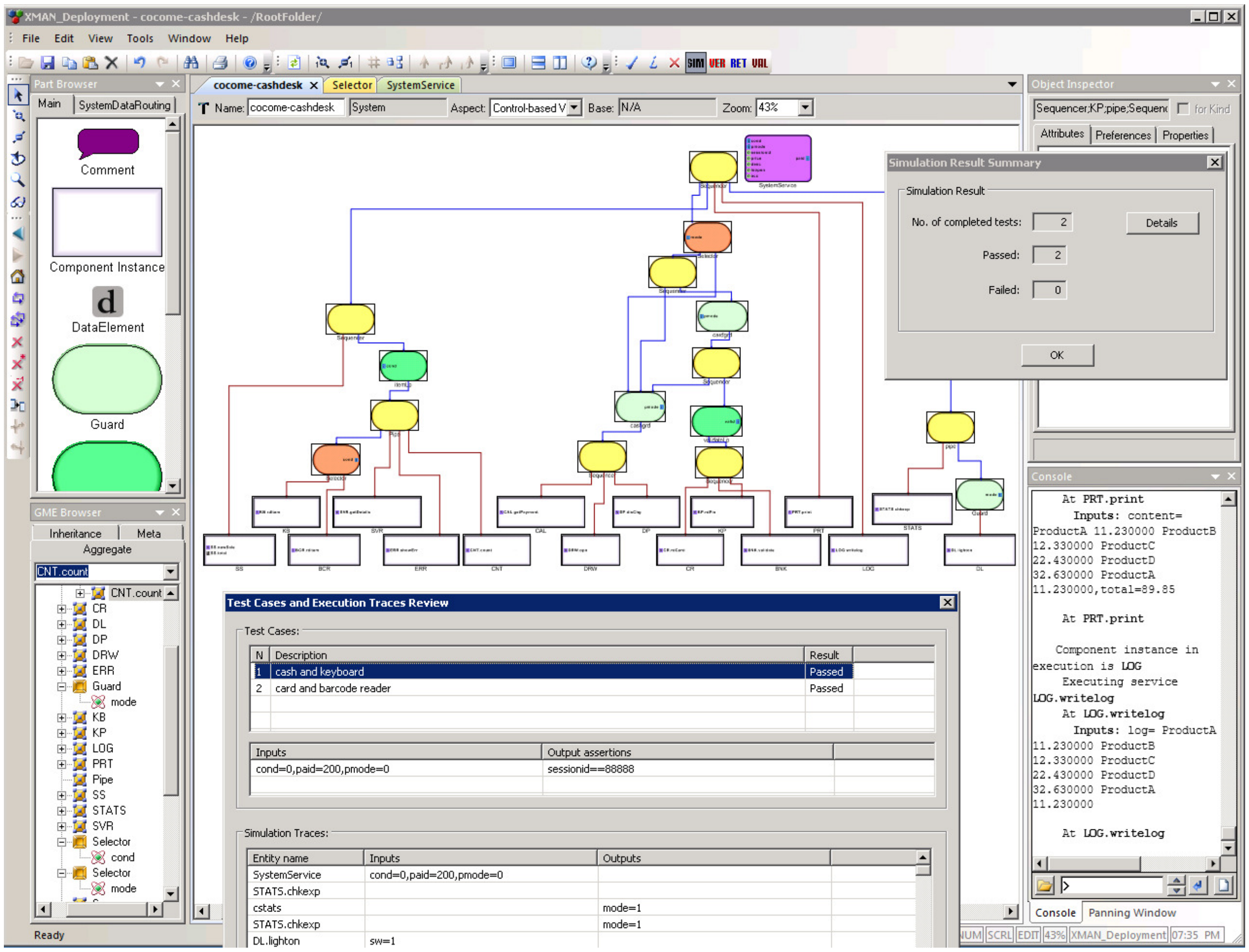
7. DISCUSSION AND CONCLUSION

In this paper we have proposed a component-based approach to incremental construction. In contrast to existing approaches in software engineering, which are top-down and not component-based, our approach is bottom-up and truly component-based. Our approach has a well-defined component model, X-MAN, and a tool for component and system development. We have also demonstrated the feasibility of our approach by applying it to the CoCoME example, which is a kind of benchmark in CBD.

One advantage of incremental construction is that it tackles complexity, in the sense that it is easier to build a system bit by bit, rather than building the whole system in one 'big bang'. Our implementation of CoCoME has demonstrated this. Similar work in incremental construction using web services [16] shares this motivation. However, the composition mechanism used there, namely orchestration, is strictly speaking, not hierarchical (unlike composition in X-MAN). This is because orchestration results in a workflow rather than a web service. Therefore it is not clear precisely how incremental composition of web services is defined.

We believe that the ability to construct a system bit by bit has an interesting implication for software engineering: by using incremental construction, it is possible to develop a system use case by use case. As far as we are aware, this is not possible in current software engineering practice. In the Unified Process [19], for instance, UML models for system design may be defined use case by use case, but it is not possible to build the system use case by use case.

Our tool for incremental construction also allows us to do incremental testing. We can check that each increment we add does indeed result in a new sub-system with the


XMAN_Deployment - cocome-cashdesk - /RootFolder/
File Edit View Tools Window Help
SIM VER RET VAL
Part Browser
Main SystemDataRouting
Comment
Component Instance
DataElement
Guard
GME Browser
Inheritance Meta
Aggregate
CNT.count
CR
DL
DP
DRW
ERR
Guard
mode
KB
KP
LOG
PRT
Pipe
SS
STATS
SVR
Selector
cond
Selector
mode
Ready
cocome-cashdesk Selector SystemService
Name: cocome-cashdesk System Aspect: Control-based V Base: N/A Zoom: 43%
Object Inspector
Attributes Preferences Properties
NUM SCRL EDIT 43% XMAN_Deployment 07:35 PM

Simulation Result Summary

Simulation Result

No. of completed tests: 2

Passed: 2

Failed: 0

Details OK

Test Cases and Execution Traces Review

Test Cases:

N	Description	Result
1	cash and keyboard	Passed
2	card and barcode reader	Passed

Inputs	Output assertions
cond=0,paid=200,pmode=0	sessionid==88888

Simulation Traces:

Entity name	Inputs	Outputs
SystemService	cond=0,paid=200,pmode=0	
STATS.chkexp		
cstats		mode=1
STATS.chkexp		mode=1
DL.lighton	sw=1	

Console

```
     At PRT.print
       Inputs: content=
ProductA 11.230000 ProductB
12.330000 ProductC
22.430000 ProductD
32.630000 ProductA
11.230000,total=89.85

     At PRT.print

    Component instance in
execution is LOG
     Executing service
LOG.writelog
     At LOG.writelog
       Inputs: log= ProductA
11.230000 ProductB
12.330000 ProductC
22.430000 ProductD
32.630000 ProductA
11.230000

     At LOG.writelog
```

Console Panning Window

Figure 21: The cash desks sub-system in X-MAN Tool

required behaviour identified in the current iteration. This is important since any wrong increment will propagate to all subsequent iterations. Incremental testing is possible in X-MAN because every X-MAN architecture is executable, and the X-MAN Tool provides a simulator for execution results.

In CBD, the most widely used component models are ADLs (architecture description languages), e.g. UML2.0. As descriptions of software architectures, ADLs are meant to describe the whole system, and are therefore not intended for incremental system construction. In particular, ADL descriptions are usually used to identify components in the system, rather than to define the system in terms of pre-existing components (in a repository for a domain) which are not system-specific. In other words, an ADL description contains system-specific components, rather than domain-specific but non-system-specific components from a repository.

However, newer ADLs, e.g. SOFA [9], do have component repositories, so they should be more suitable for incremental construction. Although, as we said earlier, architectural units tend not to have fixed behaviour (until the whole system has been constructed), we believe it is possible to customise newer ADLs such as SOFA for the purpose of incremental construction. This is future work that we intend to explore.

We have briefly touched on refactoring and its significance for incremental construction. This is another important topic for future research. Our experience so far tells us clearly that without suitable refactoring techniques, it is inevitable that sooner or later our incremental construction process comes unstuck.

Another benefit of refactoring is that it can be used to simplify architectures. Our use of composition connectors for constructing systems inevitably leads to big hierarchies of connectors. It is very useful to be able to reduce the complexity in such hierarchies wherever possible. Refactoring techniques for connectors can provide just such a facility. This is a topic that we are currently actively investigating.

One issue that also needs further investigation is the granularity of components. Our experience suggests that the smaller the components, i.e. the smaller the pieces of behaviour encapsulated in components, the easier the incremental construction process is. However, although this seems to be obviously the case, we have yet to gain sufficient empirical evidence for it. The crucial issue is whether it is possible in general to decide what level of granularity would be optimal, if any. This is an open question at present.

Finally, we are encouraged by our experience of applying our incremental approach to the CoCoME example. This example is non-trivial, and we have successfully implemented and tested the whole system, using incremental construction. So we have some confidence that our approach at least shows the promise of scaling up.

8. ACKNOWLEDGEMENTS

Keng-Yap Ng would like to thank the Ministry of Higher Education Malaysia and Universiti Putra Malaysia for sponsoring his PhD study at the University of Manchester. So too, Tauseef Rana would like to thank the National University of Sciences and Technology (NUST), Pakistan.

9. REFERENCES

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.

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

[4] R.-J. Back. Incremental software construction with refinement diagrams. Technical Report 660, TUCS - Turku Centre for Computer Science, Turku, Finland, Jan 2005.

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Soft. Eng.*, 30(6):355–371, 2004.

[6] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, 1994.

[7] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM*, 9(5):366–371, 1966.

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

[9] T. Bures, P. Hnetynka, and F. Plášil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. *Fourth Int. Conf. on Soft. Eng. Research, Management and Applications 2006*, pages 40–48, 2006.

[10] B. Christiansson, L. Jakobsson, and I. Crnkovic. CBD process. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Soft. Syst.*, pages 89–113. Artech House, 2002.

[11] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *Proc. Int. Conf. on Soft. Eng. Advances*, pages 44–53, 2006.

[12] E.W. Dijkstra. Stepwise program construction. In *Selected Writings on Computing: A Personal Perspective*, pages 1–14. Springer-Verlag, 1982.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1 edition, July 1999.

[14] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Computer Society, 2003.

[15] GME: The Generic Modeling Environment. http://w3.isis.vanderbilt.edu/projects/gme/.

[16] C. Granell, J. Poveda, and M. Gould. Incremental composition of geographic web services: an emergency management context. *University of Crete*, pages 343—348, 2004.

[17] E. R. Harold and W. S. Means. *XML in a Nutshell: A Desktop Quick Reference (In a Nutshell), 3rd ed.* O'Reilly Media, 2004.

[18] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer, and S. Sharma. Component-based design and verification in X-MAN. In *Proc. of Embedded Realtime Soft. and Syst.*, 2012.

[19] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[20] B. W. Kernighan and D. Ritchie. *The C Programming Language, 2nd ed.*. Prentice Hall, 1988.

[21] C. Larman and V.R. Basili. Iterative and incremental development: A brief history. *Computer*, 36:47–56, June 2003.

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

[23] K.-K. Lau, I. Ntalamagkas, C. Tran and T. Rana. Design Patterns as Composition Operators. In L. Grunske, R. Reussner and F. Plášil, editors, *Proc. 13th Int. Symp. on Component-based Software Engineering, LCNS 6092*, pages 232–251. Springer-Verlag, 2010.

[24] K.-K. Lau, F. Taweel, and C. Tran. The W Model for component-based software development. In *Proc. 37th EUROMICRO Conf. on Soft. Eng. and Advanced App.*, pages 47–50. IEEE, 2011.

[25] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G.T. Heineman *et al.*, editor, *Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106. Springer-Verlag, 2005.

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

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

[28] A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, editors. *The Common Component Modeling Example: Comparing Software Component Models. LNCS*, 5153. Springer Berlin Heidelberg, 2008.

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

[30] P. Velasco Elizondo and K.-K. Lau. A catalogue of component connectors to support development with reuse. *The Journal of Syst. and Soft.*, 83:1165–1178, 2010.

[31] W. Opdyke Refactoring Object-Oriented frameworks. University of Illinois, Urbana-Champaign, 1992.

[32] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.