Extracting Elements of Component-based Systems from Natural Language Requirements

Kung-Kiu Lau, Azlin Nordin and Keng-Yap Ng
School of Computer Science, The University of Manchester,
Manchester M13 9PL, UK
Email: {kung-kiu,anordin,kng}@cs.man.ac.uk

***Abstract*—Extracting keywords from requirements has been done for various modelling purposes, e.g. for defining object-oriented analysis and design models, but it has not been done for mapping requirements directly to (executable) component-based systems. In this paper we argue that the latter is possible if the underlying component model provides suitable encapsulation and hence separation of key elements of component-based systems. We show how we can extract keywords that correspond to elements of a component model that we have defined.**

I. Introduction

In software engineering, the development process starts with requirements gathering (from the customer). The next step of the process is the analysis of the requirements (in natural language). This is followed by a design of the system, based on the analysis. The design is then implemented and tested (and maintained and so on). The traditional example of this process is the Waterfall Model [1], where the process is sequential, and a system design consists of modules that interact with one another. A modern example is the Unified Development Process [2], where the process is iterative, and a system design consists of classes that interact with one another (represented by UML class and interaction diagrams).

For component-based systems, however, we believe that we do not have to use the same development process of going from requirements to their analysis to the final system. The main reason is that for component-based development, we should use a component model [23] for defining systems, and a good component model should allow us to define component-based systems with minimal or no coupling between the components, but maximal cohesion within individual components – definitely less coupling and more cohesion than modular or object-oriented systems. This means that it should be easier to identify elements of a component-based system individually and separately from the requirements, so much so that we can directly map requirements to elements of a component-based system; that is we can go from requirements directly to system design.

In [3], we demonstrated how this process can be carried out, but we only briefly outlined the main elements of the process. A key step of this process is the extraction of keywords (from natural language requirements) that correspond to elements of the chosen component model. Once such elements have been identified, creating a component-based system only requires piecing them together according to the component model.

This paper focuses on the process of extracting elements of component-based systems from natural language requirements. This process is made feasible by a key property of our chosen component model, namely *encapsulation*, which means components have no mutual dependencies.

II. Related Work

Extracting keywords from natural language requirements is not a new idea. Indeed it has been practised for a long time. However, none of the existing techniques has been designed or used for extracting keywords that correspond to elements of component-based systems, i.e. systems defined using component models. Rather, existing techniques have been used to extract keywords that map to abstract concepts, intermediate requirements models, object-oriented analysis models, and even to skeleton programming language constructs.

[4], [5], [6], [7], [8], [9], [10], [11] extract keywords from requirements and use them to construct object-oriented analysis and design models (e.g. UML class diagrams and object diagrams). [12] uses extracted keywords to define structured models (e.g. data flow diagrams). [13], [14], [15], [16] use extracted keywords to construct database models (e.g. Entity Relationship Diagram, Extended Entity Relationship). All these models are intermediate models because they are used as input to a process of refinement into more detailed designs which eventually lead to a final system design.

[17], [33] extracts keywords as candidates for object-oriented concepts, and uses these concepts in the requirements elicitation process, rather than for building the final system. [18] extracts keywords which correspond to data types, operators, and control structures; and uses these to derive pseudo codes.

In short, as far as we know, there is no related work that directly maps natural language requirements to component-based systems. The work that is the most closely related to our work is that of Behaviour Trees [19], [20], [21], [22], but they map requirements to behaviour trees, not directly to component-based systems.

III. Component Models

A component model [23] defines components and their composition. A good component model should enable us to define component-based systems with minimal coupling between components and maximal cohesion within individual components. Coupling results from external dependencies, direct or indirect, between components, and is induced by the composition mechanisms of a component model. For example, in architecture description languages [24], components are architectural units with ports, and are composed by port connection. Such connections induce external dependencies between components, albeit indirectly.

High cohesion in a component means altogether the sub-components of the component do not have many external dependencies. It is therefore also a consequence of the composition mechanisms of a component model. A composition mechanism that allows the definition of a composite component from sub-components will increase cohesion if it allows the composite to have fewer external dependencies than its sub-components.

We have defined a component model, called X-MAN, that defines systems with no coupling at all. The complete absence of coupling also means the components have maximum cohesion.

A. The X-MAN Component Model

In our component model [26], [27], [28], we define: (i) *computation* and (ii) *control* separately.[1] Computation is defined and encapsulated in *components* whilst control is defined and encapsulated in *composition connectors*. Components do not call one another; instead, composition connectors define and coordinate all the control among components.

Fig. 1 shows the basic elements of our component model. An atomic component (Fig. 1(a)) contains a computation

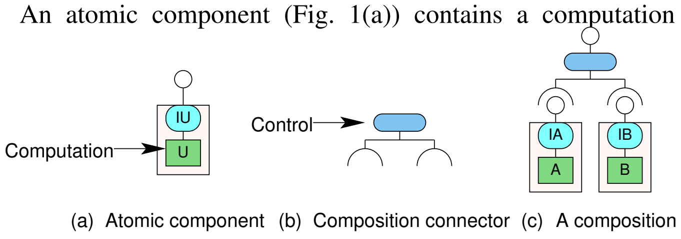


Figure 1: X-MAN component model.

unit (U), and an invocation connector (IU). A computation unit provides methods or functions that can be invoked via the invocation connector. When invoked, the computation unit performs the computation entirely within itself, and is thus encapsulated (i.e. 'enclosed in a capsule'). As a result, an atomic component encapsulates computation and has only a provided interface (denoted by a lollipop) and no required interface. Parameters for invocation are passed from composition connectors via the invocation connector.

Components are composed by composition connectors (Fig. 1(b)). A composition connector receives control and returns control; it also defines a control structure that determines the control flow between receiving and returning control. A composition connector thus encapsulates control. Our composition connectors define the usual control structures: sequencing and branching. For sequencing we have the Sequencer and Pipe connectors;[2] for branching we have the Selector connector (Fig. 2). In a composition (Fig. 1(c)), the

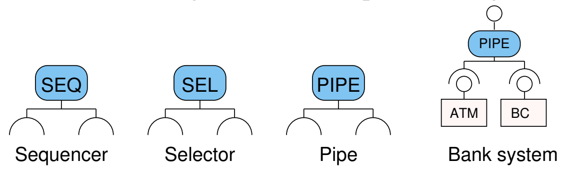


Figure 2: Composition connectors.

composition connector coordinates control flow between the sub-components. This is illustrated in Fig. 3 for the Sequencer composition connector.

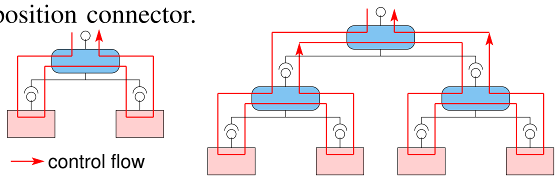


Figure 3: Hierarchical composition by Sequencer.

A simple example of a composition is the bank system (Fig. 2) that composes an ATM with a bank consortium (BC) by using a Pipe connector. Customer details and requests are passed to ATM, which validates them and then pass them on to BC.

Looping is not a composition connector since it only applies to a single component. It is therefore an adaptor. A loop at the top-most level of a system can be infinite, but elsewhere it must be finite, in order that compositionality is preserved throughout.

The result of a composition is another component with a provided interface (Fig. 1(c) and the bank system in Fig. 2). This means that composition in our model is hierarchical. This is illustrated in Fig. 3 for the Sequencer. In each composition, encapsulation of computation is preserved since components do not call one another.

Components encapsulate their own data [29], but for lack of space we will not discuss this.

To summarise, the key elements of our component model are: (i) computation (computation units) (ii) control (composition connectors); and (iii) data. Computation units encapsulate computation; composition connectors encapsulate control; components encapsulate their own data. Encapsulation of control and computation (and data) allows us to separate these elements. This makes it possible to identify these elements individually and separately from raw requirements.

IV. Extracting Keywords from Requirements

Now we explain how we take requirements in natural language and extract keywords that correspond to the basic elements of X-MAN as presented in the previous section. We will use the requirements for a Trading System used in The Common Component Modelling Example (COCOME)[30], a reference example for component-based development, to illustrate our approach.The Trading System is used for handling point-of-sales terminal (POST) transactions in a supermarket. The system comprises nine main functions including: (1) Process Sale, which handles Cash Desk operations; (2) Manage Express Checkout, which deals with transaction modes, i.e. normal and express; (3) Order Products, which allows the

[1]For simplicity, we assume data flows with control.
[2]Pipe passes results on, whereas Sequencer does not.

Store Manager to order products from suppliers; (4) Receive Ordered Products, which allows the Store Manager to update received orders; (5) Show Stock Reports, which permits the Store Manager to view all available stock; (6) Show Delivery Reports, which allows the Enterprise Manager to generate reports; (7) Change Price, which permits the Store Manager to update a product's price; (8) Product Exchange (on low stock) Among Stores, which handles product order between stores and (9) Remove Incoming Status, which allows the Store Manager to update the received product. Altogether, there are 47 requirements.

We will show all the keywords extracted for COCOME and what elements of X-MAN they correspond to, as well as the entire component-based system for COCOME. However, how the extracted keywords are used to construct a component-based system is beyond the scope of this paper. It is briefly explained in [3].

The keyword extraction process is carried out for one requirement at a time. This is possible because of encapsulation in our component model. It is desirable because analysing one requirement is more manageable than the usual practice of analysing all requirements together. It is also desirable because it scales to any number of requirements, and because it is always a finitely terminating process.

For each requirement, the extraction process consists of the following steps:

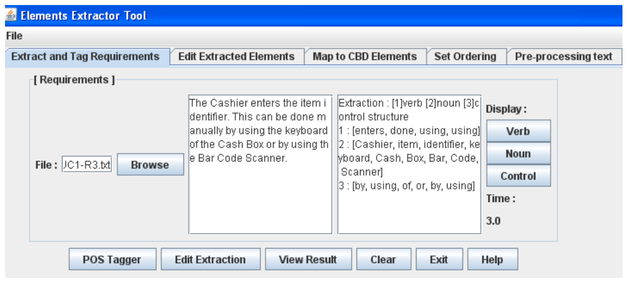
1) Run a POS Tagger on the requirement to extract verbs, nouns, prepositions and conjunctions. A POS Tagger is able to parse a piece of text and extract words corresponding to parts of speech (POS) specified by rules defined by the user.
2) Analyse the results of the POS Tagger category by category; for each category, keywords are further analysed and filtered according to pre-defined heuristics (see below).
3) Identify implicit computations and control. We will discuss how we deal with such issue in the following subsection.

We have implemented a simple tool for editing and analysing the keywords extracted from requirements. Fig. 4 shows the screen shot of the Extractor tool. Initially, a requirement will be input to the tool. Each word will be syntactically tagged using a pre-defined selected POS tagger set. In addition, the tool includes built-in heuristics which allow filtering of irrelevant words. For instance, *articles* (e.g. 'the', 'a', 'an') will not be extracted. Moreover, a user can highlight words according to verb, noun or control features (Fig. 4). The tool thus helps by suggesting keywords that may denote control, computation or data based on the POS tagger and some predefined rules (see below). Nonetheless, the analyst needs to manually filter and finalise all the selected keywords.

A. Identifying Keywords that Denote Computations

Based on the result from the POS tagging process, the next step is to identify computations.

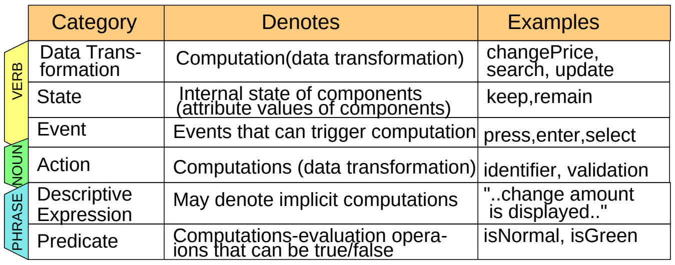
A verb may express computation. However, not all verbs can be considered as computations. By referring to the definition of



The Cashier enters the item i
dentifier. This can be done m
anually by using the keyboard
of the Cash Box or by using th
e Bar Code Scanner.

```
Extraction : [1]verb [2]noun [3]c
ontrol structure
1 : [enters, done, using, using]
2 : [Cashier, item, identifier, ke
yboard, Cash, Box, Bar, Code,
 Scanner]
3 : [by, using, of, or, by, using]
```

Figure 4: The Extractor tool.



	Category	Denotes	Examples
VERB	Data Trans-formation	Computation(data transformation)	changePrice, search, update
	State	Internal state of components (attribute values of components)	keep,remain
	Event	Events that can trigger computation	press,enter,select
NOUN	Action	Computations (data transformation)	identifier, validation
PHRASE	Descriptive Expression	May denote implicit computations	"..change amount is displayed.."
	Predicate	Computations-evaluation operations that can be true/false	isNormal, isGreen

Figure 5: Extracting computations.

computations, we selectively identify candidates of computations based on: (1) verbs that belong to one of these categories: *Data Transformation*, *State* or *Event*; (2) nouns that are *Action* nouns; or (3) phrases that are *Descriptive Expressions* or *Predicates*. The computation extraction category can be summarised as in Fig. 5. We further discuss each element of the computation category and demonstrate examples to motivate and explain the identification process.

From the literature, we adopt existing categories of keyword extraction that are relevant to computations and adapt them according to what we want to extract. The first category is *Data Transformation* verbs. A *Data Transformation* verb denotes function evaluation, which takes data as input, performs some processes and outputs data, in order to achieve a specific objective. A data transformation in general excludes manual operations, i.e. physical human activities, for example arrive, hand over, press, leave. Any database transaction can also be considered as data transformation, e.g. search, update. We adopt this category from Action [4], [31]. Let us look at some examples.

Example 1: Consider the following COCOME requirement:

```
[UC7-R2] The Store Manager selects a product item
and changes its sale price.
```

Analysis. For this requirement, although we may find two verbs i.e. *select* and *change*, we need to consider what kind of function evaluation each of the verbs will correspond to. If there is no processing involved, then we ignore the candidate computation. In this case, we select *change* computation, which belongs to the *Data Transformation* verb category. We anticipate this computation should involve some processing of the item's price.

Example 2: Consider the following COCOME requirement:

```
[UC1-R7] The Printer writes the receipt and the
Cashier hands it out to the Customer.
```

Analysis. In the second example, although we identify *writes* and *hands out* verbs, we are not concerned with verbs that are physically performed by human and which do not involve any data transformation. Thus, we choose *write* computation, which belongs to *Data Transformation* verb category.

The second category in Fig. 5 is the *State* verb, which is adopted from *State* [4], [31]. A *State* verb denotes computations that realise states, i.e. change the data that belongs to components. Candidates of states can be identified from verbs (that can be extracted by the POS tagger, i.e. past simple, past participles, present participle) and adjectives [4] that may imply system states.

Example 3: Consider the following COCOME requirement: `[UC2-1] The considered Cash Desk is in normal mode and just finished a sale which matches the condition of an express checkout sale.`

Analysis. Here, *normal* is an adjective of Cash Desk that means, *normal* is a data item that belongs to the Cash Desk. We imply there must be computation that handle any data transformation on this data.

The third category in Fig. 5 is the *Event* verb, which is adopted from *Emergence* [31]; it denotes an event that can trigger computations. Any triggering events must be associated with the corresponding notifications, i.e. the respective computations. We can identify computations based on *event* verb because any interaction between user and system or hardware devices must be handled. Hence, whenever we identify an *event* verb consisting of interactions between user and the system, the system shall provide the corresponding computations to handle these interactions.

Example 4: Consider the following COCOME requirement: `[UC2-2A] The Cashier presses the button Disable Express Mode. The color of the Light Display is changed from green into black color.`

Analysis. The term *presses* is an event that triggers the change of the Light Display from green into black. Here, we address this *Event* verb as to denote computation to be dealt i.e. change the light color.

The fourth category in Fig. 5 is the *action* noun, which is adopted from Action [4]. An *action* noun can denote data transformation provided by a component, e.g. authentication, registration, initialisation.

Example 5: Consider the following COCOME requirement: `[UC1-R5b] In order to initiate card payment the Cashier presses the button Card Payment at the Cash Box.`
`i. The Cashier receives the credit card from the Customer and pulls it through the Card Reader.`
`ii. The Customer enters his PIN using the keyboard of the card reader and waits for validation.`

Analysis. In this case, clearly pressing a button (i.e. an *Event* verb) indicates interaction between user and the system interface. Hence, in this requirement, when the system receives notification that the *card payment* button is pressed, we need to assign the corresponding computation that deals with such interaction. For this requirement the relevant verbs are (1) *payment* and (2) *validation*, which both come from *Action* noun category; (3) *enters*,which is a *Data transformation* verb (can be renamed into *readPIN* so that the computation is modelled from the system's perspective).

So far, we have used the first four categories in the table in Fig. 5 to identify explicit computations, i.e. computations that are explicitly identifiable from the requirements. These computations correspond to keywords extracted by the POS tagger. However, POS tagging alone cannot uncover all the computations that we need. The main reason is that the requirements may not specify explicitly some of the intended computations. Furthermore, the functional requirements are written from the user's point-of-view and not from the developer's point-of-view. Thus, apart from identifying explicit computations, we adopt *Descriptive Expression* and *Predicate* [18] to guide us in identifying implicit computations. These form the last two categories in Fig. 5.

A *descriptive expression* phrase, e.g. "...the change amount...", may denote a computation to calculate the change amount. Abbott [18] specifies that a *descriptive expression* describes a possible object whose identity (and possibly even whose existence) must be determined by some computation. Thus we use a *descriptive expression* to identify computations.

Example 6: Consider the following COCOME requirement: `[UC6-R2] A report which informs about the delivery mean times is generated.`

Analysis. The expression "...informs about the delivery mean times ..." must somehow be determined by a computation to calculate the mean times of a delivery, hence we say that the expression is associated with a calculate mean time computation.

A *predicate* phrase denotes operations that can return true or false, such as checking status or state (e.g., isInNormalMode, isBlack).

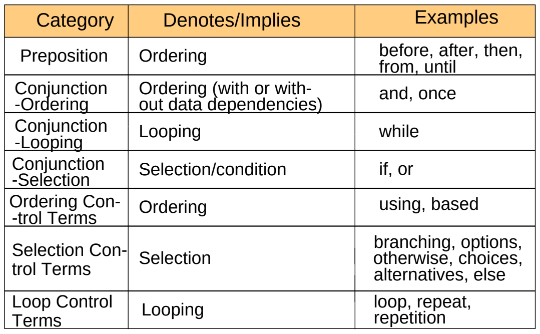
Example 7: Consider the following COCOME requirement: `[UC1-11] If the Inventory is not available, the system caches sale.`

Analysis. In this requirement, we identify "*is not available*" phrase that may denote computation to check the availability of the Inventory. Hence, we provide *checkAvailability* computation.

To conclude, based on the guidance from this computation category, we identify candidates for computation units. By using the extraction tool (Fig. 4), the user can choose to highlight the extracted verbs, and start identifying the verb categories in order to identify computations. Next, he can also highlight nouns, and identify any relevant action noun for candidates of computations.

B. Identifying Keywords that Denote Control

After we identify computations in each requirement, the next step is to look for control. A *control structure* such as *if…then…else, while, iterate, loop, selection* denotes execution flow, i.e. sequential, branching, looping [18]. In identifying control, we also follow the same strategy as in identifying computation, in which we identify control from: (1) explicit control from the extraction of the POS tagging process (2) pre-defined control terms (Fig. 6) and (3) implicit control that may imply execution flow. We will now introduce and show examples for each of the categories. The first category is *Preposition*. We look for prepositions of time that imply ordering (e.g. after, then, before), and prepositions that may imply origin of movement or direction[3] (i.e. from, of). Prepositions that are not based on time, for instance prepositions of place (position and direction) (e.g. in, at, under, on, below) are of no interest to us.



Category	Denotes/Implies	Examples
Preposition	Ordering	before, after, then, from, until
Conjunction -Ordering	Ordering (with or with-out data dependencies)	and, once
Conjunction -Looping	Looping	while
Conjunction -Selection	Selection/condition	if, or
Ordering Con--trol Terms	Ordering	using, based
Selection Con-trol Terms	Selection	branching, options, otherwise, choices, alternatives, else
Loop Control Terms	Looping	loop, repeat, repetition

Figure 6: Extracting control.

Example 8: Consider the following COCOME requirement: `[UC8-9] The Enterprise Server is not available: The request is queued until the Enterprise Server is available and then is send again.`

Analysis. The preposition *then* here denotes an execution order while the preposition *until* implies repetition.

The *Conjunction* category can be detailed into three sub-categories, i.e. conjunctions that may imply ordering, selection (branching), and repetition (looping). According to Berry and Kamsties [32], 'and' denotes: (1) concurrency of events or actions (2) conditions to be met (3) temporal order of events or actions or (4) enumerations which may not imply any ordering. In our work, we do not deal with 'and' that denotes concurrency.

Example 9: Consider the following COCOME requirement: `[UC1-R11] The System caches each sale and writes them into the inventory.`

Analysis. Here we identify an 'and' conjunction that explicitly shows an ordering execution from caching sale transaction to update the inventory. The conjunction *and* does not always imply sequential ordering, it may also suggest concurrency processes instead.

However, not all occurrences of 'and' denote ordering.

Example 10: Consider this COCOME requirement: `[UC2-R2C] Cash and also card payment is allowed and the Costumer is allowed to buy as much goods as he likes.`

[3]http://www.eslcafe.com/grammar/prepositions09.html.

Analysis. The 'and' conjunction here does not indicate sequential ordering, but indicates both payment methods, i.e. cash and card payment instead. Thus, we exclude this kind of 'and' conjunction from our identification process.

Control terms denote or imply predefined execution flow which is not derived from conjunctions and prepositions, e.g. using, based, branching, selection, loop, repeat, otherwise, alternatives etc. We selectively identify these terms and set them as control terms.

Example 11: Consider this COCOME requirement: `[UC1-R9] The Cashier enters the item identifier. The system displays the description and price. Otherwise, the product item is rejected.`

Analysis. We identify the word 'otherwise' that explicitly shows a branching execution from the identify item computation.

Another example to demonstrate a looping control term is the following.

Example 12: Consider this COCOME requirement: `[UC1-R4] Using the item identifier the System presents the corresponding product description, price, and running total. The steps are repeated until all items are registered.`

Analysis. The keyword 'repeated' explicitly shows a looping execution for the identify item computation.

From an English language structure, we can identify and infer control based on explicit prepositions from the text. So far, we perform the identification of control based on explicit control either from the POS tagging extraction (of conjunctions or prepositions) or from the predefined control terms. However, an *implicit* control may be recognised from the requirements ordering, i.e. the way the requirements are written.

In Example 11, there is no explicit control that we can identify between *identify item* computation and *display* the information. However, implicitly, we infer such an ordering is required from identify item to display the information. This is sensible and can be justified from the way the requirements are written.

Apart from this, implicit control may also be identified from descriptive expression.

Example 13: Consider this COCOME requirement: `[UC2-R1D] The maximum of items per sale is reduced to 8 and only paying by cash is allowed.`

Analysis. Besides the *and* conjunction that denotes ordering execution, this statement provides constraints that are useful for a looping structure, i.e. to repeat 8 times while identifying item transactions. Here, there are no explicit keywords that lead us to extract a looping control structure. Nonetheless, we can see such a loop.

C. Identify Keywords that Denote Data

In general, we extract keywords that denote data by noun extraction. However, not all nouns are relevant or meaningful

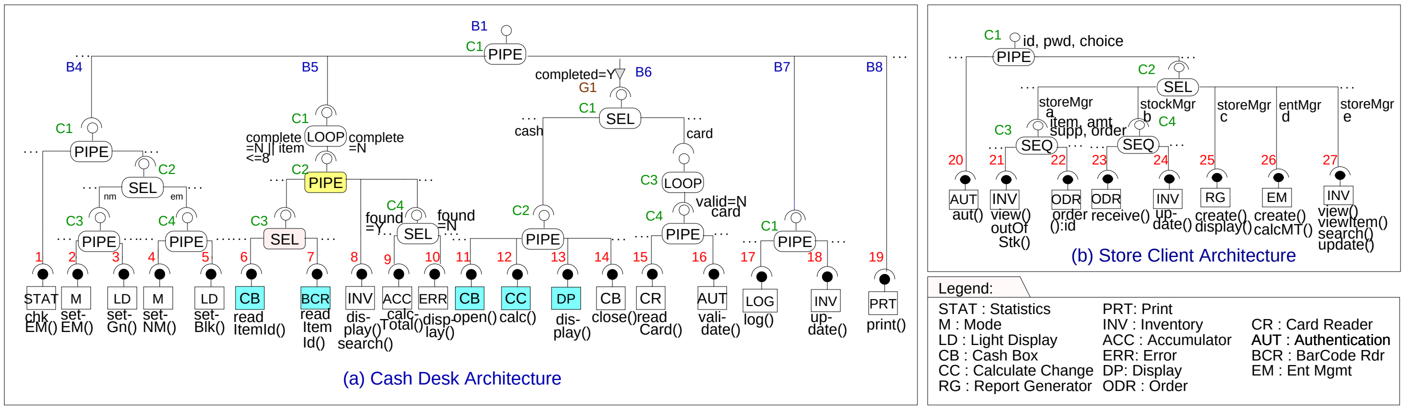


Figure 7: The complete COCOME system in X-MAN.

to be identified. We use the POS tagger to extract all the nouns. Based on the result of the extraction, we further filter only the relevant data based on values that are useful for computations, constraints for branching mechanism, or handling data dependencies between computations. This includes storing constant and initialisation values [29]. We demonstrate the data extraction in the following examples.

Example 14: Consider the following COCOME requirement that shows useful data for the identified computation:

`[UC1-R9] The system displays the description and price.`

Analysis. Here, we extract *description* and *price* as relevant data to be used for the display computation.

Example 15: Consider the following COCOME requirement that shows data useful for a branching mechanism:

`[UC7-R3] The Store Manager selects a product item and changes its sale price.`

Analysis. In this requirement, apart from *product item* and *price* as relevant data to be used for the change price computation, the *Store Manager* must also be verified prior to the invocation of the computation. Hence, we need to provide data for *Store Manager* account as well. This data will be used as validation data in order to verify its authorisation level.

Example 16: Consider the following COCOME requirement that demonstrates identification of data as a constraint:

`[UC8-R5E] If the entered amount of an incoming product is larger than the amount accounted in the Inventory, the input is rejected.`

Analysis. In this statement, we identify *amount* as a constraint on the *amount accounted* (useful for computation rather than for constraint) in the Inventory.

V. The Complete COCOME System

Now we briefly discuss the complete COCOME system derived from the keywords that we extracted from all the COCOME requirements. As mentioned earlier, how we derive the complete system from the extracted keywords is beyond the scope of this paper. Nonetheless, we wish to demonstrate two important things: firstly that the encapsulation in our component model does indeed enable us to identify elements of component-based systems individually and separately from the requirements (as we claimed in Section III-A); and secondly that the system does indeed satisfy the requirements.

Fig. 8 shows the overall architecture of COCOME. It is a client-server architecture. Fig. 7 shows the system in X-MAN derived from keywords extracted from the requirements. It models only the client side in Fig. 8. The system comprises two sub-architectures: Cash Desk and Store Client. Computations provided by the servers in Fig. 8 can be deemed as 'remote' computation units of components in the X-MAN system. For example, in Fig. 7, computations of the Inventory(INV) component reside in the server. Interactions between servers in Fig. 8 are not modelled because they are not deterministic, e.g. interaction between Store Servers and Enterprise Server. Nonetheless, we have covered this in our implementation as a separate system with its own execution thread and scheduler.

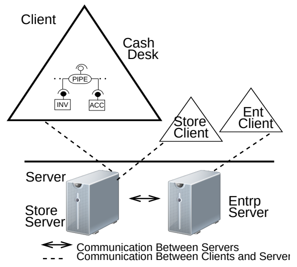


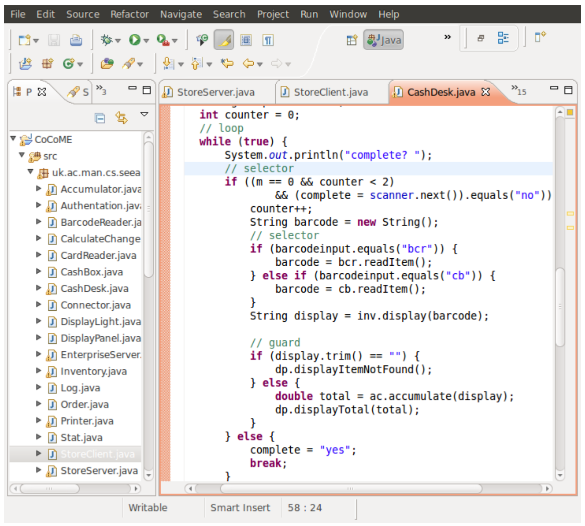
Figure 8: COCOME architecture.

Fig. 9 shows the composition in X-MAN for the COCOME system in Fig. 7.

A. Effects of Encapsulation in Component Model

Fig. 10 shows a sample of the keywords extracted from the requirements. They are keywords extracted from the requirements for the Sale Transaction process. The table shows clearly each extracted keyword, the requirement from which it was extracted from, the component or connector in the X-MAN system (Fig.7) it was mapped to, and the label of this component or connector. For example, the keyword 'enterItemID' was extracted from requirement [UC1-R3], mapped to the component BCR (BarCodeReader), which is labelled 7 in the X-MAN system; the keyword 'or' was extracted from requirement [UC1-R3], mapped to the connector SEL, labelled B5C3 in the X-MAN system.

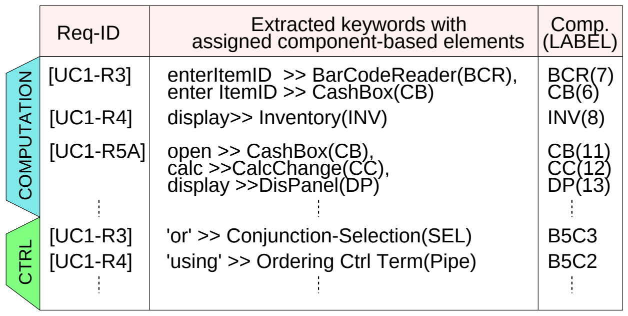
This table shows clearly that each keyword is extracted from just one requirement. Moreover, it is mapped only once, and mapped to only one component or connector. This



```
int counter = 0;
// loop
while (true) {
    System.out.println("complete? ");
    // selector
    if ((m == 0 && counter < 2)
            && (complete = scanner.next()).equals("no"))
        counter++;
        String barcode = new String();
        // selector
        if (barcodeinput.equals("bcr")) {
            barcode = bcr.readItem();
        } else if (barcodeinput.equals("cb")) {
            barcode = cb.readItem();
        }
        String display = inv.display(barcode);

        // guard
        if (display.trim() == "") {
            dp.displayItemNotFound();
        } else {
            double total = ac.accumulate(display);
            dp.displayTotal(total);
        }
    } else {
        complete = "yes";
        break;
    }
```

Figure 9: X-MAN composition for COCOME.



	Req-ID	Extracted keywords with assigned component-based elements	Comp. (LABEL)
COMPUTATION	[UC1-R3]	enterItemID >> BarCodeReader(BCR), enter ItemID >> CashBox(CB)	BCR(7) CB(6)
	[UC1-R4]	display>> Inventory(INV)	INV(8)
	[UC1-R5A]	open >> CashBox(CB), calc >>CalcChange(CC), display >>DisPanel(DP)	CB(11) CC(12) DP(13)
	⋮	⋮	⋮
CTRL	[UC1-R3]	'or' >> Conjunction-Selection(SEL)	B5C3
	[UC1-R4]	'using' >> Ordering Ctrl Term(Pipe)	B5C2
	⋮	⋮	⋮

LABEL column shows the architectural elements in the system architecture (Fig. 7)

Figure 10: Keywords extraction for Sale Transaction.

gives evidence that keywords can be extracted and mapped individually and separately. This is important because it makes the whole extraction and mapping process much easier to manage and it make it scalable to any number of requirements.

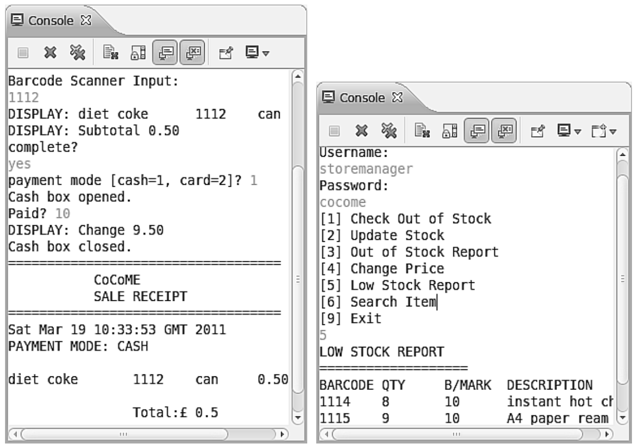
B. Validating the COCOME System

To validate the COCOME system we created, we executed a prescribed set of test cases presented in [30]. Using the final system, which is constructed based on the derived architecture (see Fig. 7), we managed to successfully execute all of the provided test cases (see Fig. 11).

TC-UC1A	Purchase of goods with cash payment.
TC-UC1B	Concurrent purchase of goods at more than one cash desks.
TC-UC1C	Purchase of goods with card payment.
TC-UC1D	Invalid item id read, manual entry of item id.
TC-UC1E	Wrong PIN entry for credit card, card validation fails.
TC-UC2A	Manage express checkout (switch to express mode, credit card payment is not possible.
TC-UC3A	Generate report of low stock product.
TC-UC4A	Order low stock products, correct delivery is recorded.
TC-UC5A	Generate report of available stock in a store (Store Manager).
TC-UC5B	Generate report of cumulated available product in the enterprise (Enterprise Manager).
TC-UC6A	Provide report containing mean time to delivery for each supplier (Enterprise Manager).
TC-UC7A	Change price of a product (Store Manager).
TC-UC8A	Product exchange among stores.

Figure 11: List of COCOME test cases.

Fig. 12a shows the execution of Test Case-UC1A, which tests for the purchase of goods with cash payment. The test case is considered passed once item is entered and the payment is made. The system shall display the amount paid and the change amount to the Store Client. Fig. 12b shows the execution of Test Case-UC3A, which tests for generation of low stock products by the Store Manager. For this test case, the Store Manager also needs to be authenticated prior to the report generation. The test case passes if the report is generated.



```
Barcode Scanner Input:
1112
DISPLAY: diet coke      1112    can
DISPLAY: Subtotal 0.50
complete?
yes
payment mode [cash=1, card=2]? 1
Cash box opened.
Paid? 10
DISPLAY: Change 9.50
Cash box closed.
===================================
           CoCoME
           SALE RECEIPT
===================================
Sat Mar 19 10:33:53 GMT 2011
PAYMENT MODE: CASH

diet coke       1112    can      0.50

                Total:£ 0.5
```

(a) Test Case-Use Case 1A

```
Username:
storemanager
Password:
cocome
[1] Check Out of Stock
[2] Update Stock
[3] Out of Stock Report
[4] Change Price
[5] Low Stock Report
[6] Search Item
[9] Exit
5
LOW STOCK REPORT
===================
BARCODE QTY     B/MARK  DESCRIPTION
1114    8       10      instant hot ch
1115    9       10      A4 paper ream
```

(b) Test Case-Use Case 3A

Figure 12: Test cases execution.

As we have mentioned in Section V, we included the servers in the component's implementation. Therefore, the inter-server communication between Store Server and Enterprise Server is not modelled in the architecture shown in Fig. 8. That is why Test Case UC-8A, the product exchange among stores is not tested on this architecture.

VI. Discussion and Conclusion

We have presented an approach for extracting keywords from natural language requirements. Our primary concern is to identify elements of component-based systems as defined in our component model, namely computations, control and data. We use the help of POS tagging process to tag each requirement statement, and based on the result of the extraction, we filter and identify the component-based elements as presented in Section IV-A, Section IV-A, and Section IV-C.

Following this premise, we demonstrated examples of the identification process for each category of the component-based elements. We have also elucidated that we can actually use the identified computations, control and data to guide us in constructing the complete COCOME system. Although our approach is basically heuristic, and requires human guidance and decision making, we believe this is possible because the underlying component model provides a way to realise such an approach.

The main feature of our component model that enables us to realise this approach is encapsulation, i.e. components encapsulate computation and have no external dependencies on other components. As a result, while we map the extracted keywords to component-based elements, we do not waste any effort on resolving issues with component dependencies. We have demonstrated this contention by showing the link between the extracted keywords from requirements, the component-based elements they map to, and where they appear in the architecture.

One possible drawback of our approach is that the granularity of the components maybe too fine, as they correspond to individual keywords. In this regard, we are investigating refactoring techniques for transforming sub-architectures into composite components and thereby raising the level of granularity of components. Indeed the architecture in Fig. 7 is already refactored, but not for the purpose of increasing component granularity. In general, refactoring is an integral part of building any architecture.

Natural language has no precedence and associativity as in mathematics or programming concepts [32]. We provide no rules for such precedence. However, it is important to emphasise that our concerns are how to identify control which encapsulates computation execution, and how this can be used to meaningfully and correctly represent the behaviours that satisfy a specific requirement. With this restriction, our current work only covers functional requirements. Dealing with non-functional requirements is another interesting and challenging research area to be considered in the future.

To sum up, we have demonstrated the feasibility of our approach. In future, we intend to increase the capabilities of the extraction tool, as well as to develop tools that support the process to derive systems from extracted keywords.

References

[1] W. Royce, "Managing the Devt of Large Software Systs," in *ICSE'87*. IEEE Computer Society Press, 1987, pp. 328–338.

[2] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Dev. Process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] K. Lau, A. Nordin, T. Rana, and F. Taweel, "Constructing component-based systems directly from requirements using incremental composition," in *36th Euromicro Conference*, Lille, France, 2010, pp. 85–93.

[4] M. Saeki, H. Horai, and H. Enomoto, "Soft. devt. process from natural lang. spec.," in *ICSE'89*. 1989, pp. 64–73.

[5] G. Booch, "Object-oriented devt.," *IEEE Trans. Software Eng.*, vol. 12, no. 2, pp. 211–221, 1986.

[6] N. Juristo, A. M. Moreno, and M. Lopez, "How to use linguistic instruments for object-oriented analysis," *IEEE Software*, vol. 17, pp. 80–89, 2000.

[7] L. Mich, "NL-OOPS: from natural lang. to object oriented req. using the nat. lang. processing system LOLITA," *Natural Language Eng.*, vol. 2, no. 2, pp. 161–187, 1996.

[8] D. W. Brown, *An Intro. to Object-Oriented Analysis: Objects and UML in Plain English*, 2nd ed. Wiley, Jul. 2001.

[9] S. Delisle, K. Barker, and I. Biskri, "Object-oriented analysis: Getting help from robust comp. linguistic tools," in *in G. Friedl, H.C. Mayr (eds) Appl. of Natural Lang. to Info. Systems, Oest. Comp. Ges.*, 1999, pp. 167–172.

[10] H. Harmain and R. Gaizauskas, "CM-builder: A nat. lang.-based case tool for object-oriented analysis," *Automated Soft. Eng.*, vol. 10, pp. 157–181, 2003.

[11] S. Overmyer, L. Benoit, and R. Owen, "Conceptual modelling through linguistic analysis using LIDA," in *the 23rd ICSE*, May 2001, pp. 401–410.

[12] V. Ambriola and V. Gervasi, "Processing natural lang. req.," in *Proc. of the 12th Int. Con. on Auto. Soft. Eng.*, ser. ASE '97. IEEE Computer Society, 1997, pp. 36–45.

[13] P. P. Chen, "Entity-Relationship diagrams and English sentence structure," in *Proc. of the 1st Int. Conf. on the Entity-Relationship Approach to Syst. Analysis and Design.* , 1980, p. 1314.

[14] S. Hartmann and S. Link, "English sentence structures and EER modeling," in *Proc. of the 4th Asia-Pacific Conf. on Conceptual Modelling - Volume 67*. 2007, pp. 27–35.

[15] A. M. Tjoa and L. Berger, "Transf. of req. spec. expressed in natural lang. into an EER model," in *Proc. of the 12th Int. Conf. on the Entity-Relationship(ER) Approach: ER Approach*. Springer-Verlag, 1994, pp. 206–217.

[16] S. Du and D. Metzler, "An automated multi-component approach to extracting entity relationships from database req. spec. doc.," in *Natural Lang. Proc. and Info. Sys.*, ser. Lecture Notes in Comp. Sc., C. Kop, G. Fliedl, H. Mayr, and E. Mtais, Eds. 2006, vol. 3999, pp. 1–11.

[17] L. Goldin and D. M. Berry, "AbstFinder, a prototype natural lang. text abstraction finder for use in requirements elicitation," *Automated Soft. Eng.*, vol. 4, no. 4, pp. 375–412, 1997.

[18] R. J. Abbott, "Program design by informal English descriptions," *Commun. ACM*, vol. 26, no. 11, pp. 882–894, 1983.

[19] R. Dromey, "From req. to design: Formalizing the key steps," in *Proc. of the 1st ICSE. and Formal Methods*, 2003, pp. 2–11.

[20] R. G. Dromey, "Arch. as an emergent property of requirements integration," in *STRAW' 03 : Proc. of the 2nd Int. Software Req. to Architectures Workshop*, 2003, pp. 77–84.

[21] C. Gonzalez-Perez, C. Gonzalez-Perez, B. Henderson-Sellers, and G. Dromey, "A metamodel for the Behavior Trees(BT) modelling technique a metamodel for the BT modelling tech.," in *Info. Tech. and App., 2005. ICITA 2005. Third Int. Conf. on*, B. Henderson-Sellers, Ed., vol. 1, 2005, pp. 35–39 vol.1.

[22] R. G. Dromey, "Eng. large-scale software-intensive systems," in *Soft. Eng. Conf., 2007. ASWEC 2007. 18th Australian*, 2007, pp. 4–6.

[23] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. on Software Engineering*, vol. 33, no. 10, pp. 709–724, October 2007.

[24] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software arch. desc. lang.," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[25] C. Bock, "UML 2 composition model," *Journal of Object Technology*, vol. 3, no. 10, pp. 47–73, 2004.

[26] K.-K. Lau, P. Velasco Elizondo, and Z. Wang, "Exogenous connectors for software components," in *Proc. 8th Int. Symp. on Component-based Software Eng., LNCS 3489*, G. Heineman *et al.*, Ed. Springer-Verlag, 2005, pp. 90–106.

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

[28] K.-K. Lau, L. Ling, and Z. Wang, "Composing components in design phase using exogenous connectors," in *the 32nd Euromicro Conference*, vol. 0. 2006, pp.12–19.

[29] K. K. Lau and F. Taweel, "Data encapsulation in software components," *Component-Based Software Eng.*, p. 116, 2007.

[30] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, *The Common Component Modelling Example: Comparing Software Component Models*, 1st ed. Springer Pub. Co., Inc., 2008.

[31] C. Rolland and C. Proix, "A natural lang. approach for requirements eng.," in *Adv. Info. Sys. Eng.*, ser. Lecture Notes in Computer Science, P. Loucopoulos, Ed. 1992, vol. 593, pp. 257–277.

[32] D. M. Berry, P. D. C. Science, M. M. Krieger, and P. D. Mathematics, "From contract drafting to software spec.: Linguistic sources of ambiguity - a handbook version 1.0," 2000.

[33] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall, Oct. 2004.