Constructing Component-based Systems Directly from Requirements using Incremental Composition

Kung-Kiu Lau, Azlin Nordin, Tauseef Rana and Faris Taweel
School of Computer Science, The University of Manchester, UK
Email: {kung-kiu,anordin,trana,fmt}@cs.man.ac.uk

***Abstract*—In software engineering, system construction normally starts from a requirements specification that has been engineered from raw requirements in a natural language. Such a specification is derived from intermediate requirements models such as use case models. These models at best only approximate the raw requirements. In this paper we propose a component-based approach that maps raw requirements directly into architectures, with a view to maximising the match between the final system and the raw requirements. Our approach is based on a component model that supports incremental composition.**

I. Introduction

In software engineering, system construction normally does not start from raw requirements (in a natural language). Rather, raw requirements are usually 'engineered' into a requirements specification, which then provides the starting point for the system construction process. The requirements engineering process typically defines intermediate requirements models such as use case models. Intermediate requirements models have to be constructed manually, using human knowledge, experience and ingenuity. Such models at best only approximate the raw requirements. Therefore the same is true of the requirements specification that results from the requirements engineering process.

To maximise the chance of achieving a better match between the final system and the raw requirements, it would seem sensible to map raw requirements directly into elements of the desired system, and thereby construct the system. The question is whether such an approach is feasible and practicable. We believe it is, if it is based on a suitable component model. In this paper we propose an approach based on a component model that is an extension of a model that we defined previously.

Our approach is founded on the premise that an individual raw requirement can be mapped to a partial (component-based) architecture containing appropriate components and (composition) connectors. Furthermore, a partial architecture can be extended in an incremental manner such that the extended (partial) architecture satisfies additional requirements, as well as those already satisfied in the initial partial architecture. Thus our approach can process requirements one by one and incrementally construct partial architectures that satisfy the requirements cumulatively; and when all the requirements have been processed, we have the complete architecture that satisfies all the requirements.

II. Related Work

Since our starting point is requirements in a natural language, there is a lot of existing work in linguistic analysis that is relevant, e.g. [4], [24], [18], [1], [29], [7], [15], [31], [12], [28]; in particular, techniques for extracting information that pertains to system or architecture design. For lack of space we will not survey these techniques, but instead will briefly summarise how we adopt and adapt them for our own use in Section III-C.

Having extracted relevant information from the requirements, the next step is to map them to architectures. There is another class of related work here. Our approach is incremental, and deals with one requirement at a time. This is in contrast to work that takes into account all the requirements at once (e.g. [32]), including incremental architecture design, which incrementally adds behaviour or properties to an architectural skeleton (e.g. [3]). Our work is also different from work that incrementally develops requirements hand-in-hand with architectures (e.g. [27]).

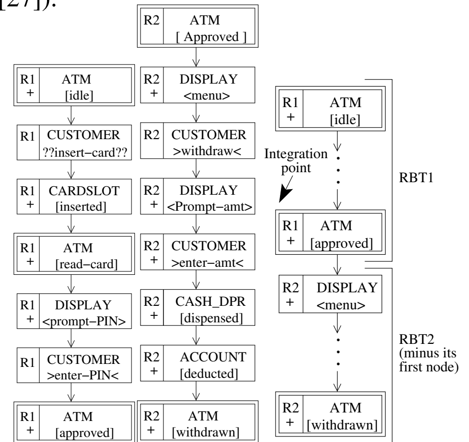


Fig. 1: Behaviour Trees.

The work that is most closely related to our work is the Behaviour Tree approach [10], [9], [13], [11]. This translates individual (raw) requirements into *behaviour trees*. A behaviour tree is a graph that represents the behaviour of a set of entities (called components) which realise or change states, make decisions, respond to or cause events, and interact by exchanging information and/or passing control . Behaviour trees for all the individual requirements (called *requirements* behaviour trees (RBTs)) are merged into a behaviour tree (called the *design* behaviour tree (DBT)) that describes the required behaviour of the whole system. Fig. 1 shows the RBTs for requirements R1 and R2 of Example 1 (Section III-E) and their merged RBT. From the DBT of the whole system, a component interaction network is extracted,

together with the behaviour of individual components.

III. Our Approach

It is better to use a component model as the basis of an approach to constructing systems directly from requirements. The behaviour tree approach does not use a component model: it builds the behaviour tree of the desired system, extracts a system structure from the behaviour tree, and then generates code for the system. It is therefore not a component-based approach. In particular, it does not have the notion of pre-existing components, but instead generates all the code for every system from scratch.

A component-based approach is better not only because it can re-use pre-existing components, but also because it can actually construct the system's architecture incrementally. The behaviour tree approach builds the behaviour tree for the system incrementally, but not the system itself. In this section we discuss the elements of our approach.

A. Incremental Composition

To use a component-based approach for building systems directly from requirements, we need to choose a suitable component model, in particular one that supports *incremental composition*. In a component model, a composition corresponds to an architecture, and by incremental composition we mean composition that (i) allows the addition of more components, as well as the addition of further compositions, to an existing architecture; and (ii) preserves the behaviour (and hence properties) of the existing architecture within the incremented architecture.

By preserving the behaviour of the existing architecture, incremental composition supports an incremental approach to mapping requirements directly to systems. Requirements can be successively mapped one at a time into a partial architecture by adding further components and/or compositions. Initially, we start with an empty architecture, and increment it with a partial architecture such that it satisfies one requirement. Then for each of the other requirements, we successively increment the current partial architecture (by adding more components and compositions) such that, each time, the new architecture satisfies the new requirement, as well as all the previous ones, by virtue of behaviour preservation. The complete architecture is the final architecture when all requirements have been mapped in this way.

$$\begin{array}{rcl} \{R1\} & \sqsubseteq & S1 \\ & & \rotatebox{90}{$\sqsubseteq$} \\ \{R1,R2\} & \sqsubseteq & S2 \\ & & \rotatebox{90}{$\sqsubseteq$} \\ \vdots & \vdots & \vdots \\ & & \rotatebox{90}{$\sqsubseteq$} \\ \{R1,R2,\ldots,Rn\} & \sqsubseteq & Sn \end{array}$$

Fig. 2: Incremental composition.

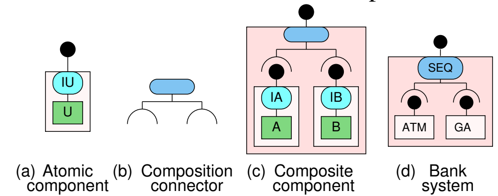
The semantics of incremental composition with respect to requirement mapping can be expressed as the relations in Fig. 2, where Rs are requirements, Ss are partial architectures, and $\sqsubseteq$ denotes the 'subset of' or 'is contained by' relation. We use $\sqsubseteq$ loosely: $\{R1, \ldots, Rn\} \sqsubseteq S$ means partial architecture S satisfies the set of requirements $R1, \ldots, Rn$; $S1 \sqsubseteq S2$ means partial architecture $S2$ contains partial architecture $S1$.

B. A Component Model with Incremental Composition

We have formulated a component model that supports incremental composition. We will first describe the basic elements of our model, and then we will explain how the model supports incremental composition.

In our component model [20], [19], [21] computation and control are encapsulated separately. This separation and encapsulation enables us to map requirements to partial architectures in our model, by identifying computation and control specified in requirements and mapping them to corresponding elements in our model.

In our model, components are encapsulated: they encapsulate control, data as well as computation. Our components have no external dependencies, and can therefore be depicted as in Fig. 3(a) and (c), with just a lollipop (provided service), and no socket (required service).



(a) Atomic component (b) Composition connector (c) Composite component (d) Bank system

Fig. 3: Our component model.

There are two basic types of components: (i) *atomic* and (ii) *composite*. Fig 3(a) shows an atomic component. This consists of a *computation* unit (U) and an *invocation connector* (IU). A computation unit contains a set of methods which do not invoke methods in the computation units of other components; it therefore encapsulates *computation*. An invocation connector passes control (and input parameters) received from outside the component to the computation unit to invoke a chosen method, and after the execution of method passes control (and results) back to whence it came, outside the component. It therefore encapsulates *control*. A composite component is built from atomic components by using a *composition connector*. Fig. 3(b) shows a composition connector. This encapsulates a control structure, e.g. sequencing, branching, or looping, that connects the sub-components to the interface of the composite component (Fig. 3(c)). Since the atomic components encapsulate computation and control, so does the composite component. Our components therefore encapsulate control (and computation) at every level of composition.[1] Clearly, composition in our model is hierarchical, and it preserves encapsulation at every level.

Fig. 3(d) shows a simplified bank system with two components *ATM* and *GA*, composed by a *sequencer* composition connector *SEQ*. Control starts when the customer keys in his PIN (and maybe also the operation he wishes to carry out). The connector *SEQ* passes control to *ATM*, which checks the customer's PIN; then it passes control to *GA* (get account), which *g*ets hold of the customer *a*ccount details (and possibly perform the requested operation). Control then passes back to the customer.

Other composition connectors in our model include *pipe* for

[1]They also encapsulate data at every level of computation [22]. For simplicity we omit this here.

sequencing (it is the same as *sequencer* except a *pipe* passes the results from one component as input to the next), and *selector* for branching (it selects one component). We also have unary connectors which act as adaptors for composition connectors: *loop* for looping,[2] and *guard* for passing or inhibiting control flow to a composition connector.

In order to support incremental composition, (i) we allow composition connectors to be *open* in arity, thus allowing any number of components to be added to an existing composition connector; (ii) we allow composites to be *open*, i.e. to have open or incomplete interfaces, as shown in Fig. 4. A composition connector is open by default; the '. . . ' adjacent to an open composition connector in Fig. 4 denotes available *composition points*, i.e. points where more components or compositions can be added. A closed composition connector (e.g. the one in Fig. 3) by contrast does not have any available composition points. An open composition connector can be *closed* (i.e. can become a closed connector) by simply removing its available composition points; this is a change in property, and can be introduced manually. By contrast a closed connector cannot become open.

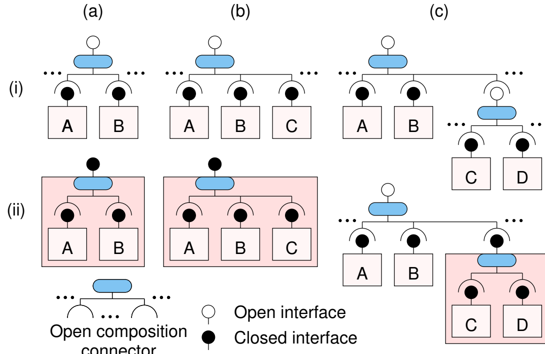


Fig. 4: Incremental composition in our model.

An open composition connector creates an open composite with an open interface ((a)(i), (b)(i), (c)(i) and (c)(ii) in Fig. 4). whereas a closed composition connector yields a closed composite with a closed interface ((a)(ii), (b)(ii) and the composite of C and D in (c)(ii) in Fig. 4).

An open composite can be closed by closing its composition connector, but only if all its sub-components are closed. In Fig. 4, (a)(ii) and (b)(ii) are closed when the composition connectors in (a)(i) and (b)(i), respectively, are closed (all their sub-components are already closed). Thus closing a composition is done hierarchically, from the bottom up. In (c)(i) in Fig. 4, the top open interface can only be closed after the open interface for the composite containing C and D has been closed (as in (c)(ii)).

Due to encapsulation in components, and hierarchical composition that preserves encapsulation, the composition in Fig. 4 satisfies the relations in Fig. 2 (considering open composites as partial architectures), i.e. it is indeed incremental composition. Encapsulation ensures that newly added components do not alter the behaviour of existing components, and hierarchical composition preserves requirements that have already been satisfied by the current partial architecture.

Finally, in current component models [23], only Koala [26] supports a form of incremental composition by preserving properties defined in diversity interfaces of sub-components. In other component models, the entire system is designed and/or constructed in one step, either in design phase (as in Enterprise JavaBeans [8] and UML2.0 [25]) or in deployment phase (as in JavaBeans [14]).

C. Extracting Information from Natural Language Requirements

Since we intend to map requirements to architectures in our component model, we wish to identify elements in requirements that correspond to the key semantic concepts in our model, and identify words (in requirements) that represent these concepts. Most object-based mapping approaches rely largely on identifying nouns and verbs because the object-oriented computational model supports only operands and operators [6]; object-oriented software development [5] uses the rule that nouns correspond to objects or classes whereas verbs correspond to messages between objects.

As we have seen, the key semantic concepts in our component model are *computation* and *control*. Computation means data transformation or function evaluation, whereby values or functions are computed and variables may be updated. Control means the flow of execution of pieces of computation. Thus the result of a piece of control invoking a computation is a piece of behaviour.[3]

We summarise what we extract from natural language requirements in the following tables. The main elements that we identify are verbs. Verbs generally refer to actions, events and processes [17]. We adopt and adapt Saeki's [29] and Roland's [28] rules for identifying verbs and mapping them into our component model elements.

Figure 5 shows what we can extract from verbs. In Figure 5, Computation is adopted from Action [29], [28], State is adopted from State [29], [28] and Event is adopted from Emergence [28].

Category of verbs	Denotes	Examples
Computation	Computation (data transformation)	withdraw,deposite, cooking
State	Internal state of components (attribute values of components)	keep,remain
Event	Events (that can trigger computation)	press,cancel,push

Fig. 5: Elements that can be extracted from verbs.

A *computation* verb, e.g. *withdraw*, denotes a data transformation, which takes data as input, performs some function evaluation and outputs data, in order to achieve a specific objective. In general, data transformations can involve data access operations i.e. input/output operations; however, we do not use computation verbs to denote such data transformations.

A *state* verb, e.g. *keep*, *maintain*, *cooking*, denotes computations that realise states, i.e. change the attribute values of components.

[2] All loops must be finite, except for a loop at the top level of a system.

[3] For simplicity we assume that data follows control.

An *event* verb, e.g.*press*, denotes an event that can trigger computations.

Figure 6 shows what we can extract from nouns. In Figure 6, Conceptual component is adopted from Class [29], Data is adopted from Value [29], State is adopted from Attribute [29] and Computation is adopted from Action [29].

Category of nouns	Denotes	Examples
Conceptual component	Conceptual hooks for components	power tube, authentication
Data	Value or set of values	1,c,integer
State	Attribute name and state	closed,open
Computation	Computation (data transformation)	registration,tranmission, movement

Fig. 6: Elements that can be extracted from nouns.

A *conceptual component* noun, e.g. *power tube*, denotes an abstraction of a candidate component that can be identified from nouns such as devices (e.g. power tube, auto-teller machine, etc.).

A *state* noun, e.g. *closed*, denotes an attribute name of a state.

A *data* noun, e.g. the number *1*, denotes a value that may need to be stored and retrieved. These values may represent status, grade, result of computation, etc.

A *computation* noun, e.g. *registration*, denotes data transformation provided by a component such as processes(authentication, initialisation).

Table 7 shows what we can extract from phrases. In Figure 7, Descriptive expression is adopted from Descriptive expression [1], Predicate is adopted from Predicate [1] and Control structure is adopted from English control structure [1].

Category of phrases	Denotes	Examples
Descriptive expression	May denote components or computations	"the earlier date" may denote date or compareDate( )
Predicate	Computations - operations that can be true/false	is enabled,is invalid
Control structure	Control structure	if,then,else,while, iterate,loop,after

Fig. 7: Elements that can be extracted from phrases.

A *descriptive expression* phrase, e.g. *the earlier date*, may denote computations. Abott [1] specifies that a descriptive expression describes a possible object whose identity (and possibly even whose existence) must be determined by some computation. In our work, we are more concerned with computations rather than identifying objects. For example, "If the PIN is incorrect, the card is ejected ". The expression "..the PIN is incorrect.. "must some how be determined by a computation to verify the PIN, hence we say that the expression is associated with a verification computation.

A *control structure* phrase denotes flow of control such as if..then..else, while, iterate, loop, after, selection. Some constraints identified from requirements can also determine control structures.

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

D. Mapping Individual Requirements to Partial Architectures

Starting with raw requirements, we construct systems directly from them. Our approach iterates over the individual requirements, one at a time. For the first requirement we deal with, we analyse it using the tables in the previous section and map it into a partial architecture in our component model. This partial architecture is the *initial* system architecture. Then for each subsequent requirement we deal with, we analyse and map it into a partial architecture and use incremental composition to compose this partial architecture with the current system architecture. When the last requirement has been processed, the *final* system architecture is obtained. In this section, we explain the steps of this process.

1) Step 1: Identify computations and choose components. For the current requirement, we analyse it and look for verbs, nouns or phrases that denote computation.

For example, in the following requirement R1 (from Example 1, Section III-E):

> **R1** A customer will be required to insert an ATM card and enter a personal identification number (PIN).

we can identify candidate computations 'insert card' and 'enter PIN'. So we know we need to provide components with 'read card' and 'read PIN' operations .

For each computation, we have to choose a candidate component, which is either an existing component in a repository or a new conceptual component identified from a noun in the requirement. Of course we can group many computations into one component. Conversely, we can split a single computation between several components.

For R1, we decide to separate the computation into two candidate components, CardReader and PinReader. For lack of space, we do not discuss how to identify these components in a repository; rather we assume that if they cannot be found in a repository, then we will develop them.

2) Step 2: Identify control flow and choose composition connectors. Whenever we identify more than one component, the next step is to identify the flow of execution between those components.

In R1, we map the word 'and' to a sequential control flow between the two computations.

The results of Steps 1 and 2 so far for R1 are summarised as follows: We add an interface to each component that shows the operations it provides. The interface also signifies encapsulation.

Step1	Component	CardReader
	Computation(s)	readCard( )
	Interface	readCard( )
Step2	Component	PinReader
	Computation(s)	readPin( )
	Interface	readPin( )
	Control flow	readCard( ) → readPin( )

Fig. 8: Results of Steps 1 and 2 so far for R1

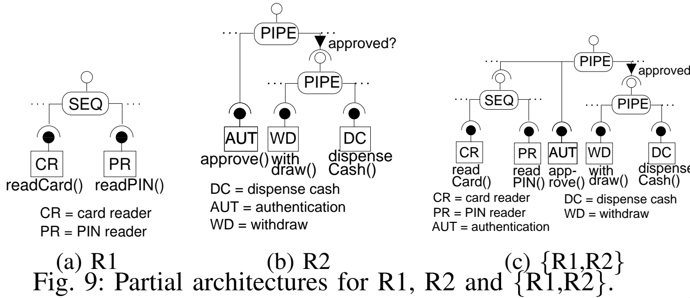
Based on the control flow identified, we can decide the type of composition connector that

is appropriate: use a *sequencer* to sequence between computations; use a *pipe* if data from one component is needed to be piped to the other components; use a *selector* for branching. In addition, we can use *guard* or *loop* for adapting a composition connector. Of course, we can use a combination (i.e. composition) of composition connectors (and adaptors).

For R1, the appropriate composition connector is clearly a sequencer.

3) Step 3: Create partial architecture. The results of Steps 1 and 2 should be sufficient to enable us to construct a partial architecture for the requirement.

For R1, the partial architecture is shown in Fig. 9(a).



(a) R1 (b) R2 (c) {R1,R2}

Fig. 9: Partial architectures for R1, R2 and {R1,R2}.

4) Step 4: Compose this partial architecture with the current system architecture. This is done using incremental composition.

Suppose in addition to R1, we also have R2 (also from Example 1, Section III-E):

> **R2** A customer must be able to make a cash withdrawal from the linked account. Approval must be obtained from the bank before cash is dispensed.

Then using Steps 2 to 3 we can map R2 to the partial architecture shown in Fig. 9(b). Step 4 composes this with the current system architecture (the partial architecture for R1 (Fig. 9(a)), to yield the new system architecture shown in Fig. 9(c). The details for Fig. 9(b) and (c) will be explained later in Example 1 (Section III-E).

5) Step 5: Finalise the system architecture. When all the requirements have been mapped, we have an architecture for the whole system. This architecture still has available composition points, and can therefore be refined, adapted or optimised. Components could be combined into larger composites; a set of connectors could be optimised to a single connector; connectors could be adapted by adaptors to add behaviour that is implicit in the requirements. The last step of the finalisation process is to remove any remaining available composition points, thus closing the whole (final) architecture.

The incremental composition step, Step 4, may not be possible if the partial architecture of the current requirement cannot be related to the current system architecture. This can easily happen since the requirements document is unstructured. When this happens, we have to postpone the incremental composition for this requirement until it becomes possible. If it never becomes possible, then as with all requirements, there may be problems with the requirements themselves, and we may need to consult the client to resolve any ambiguities, inconsistencies or incompleteness in the requirements.

In each incremental composition, there may be many possible composition points (denoted by '...' in an open composition connector) associated with many open composition connectors. A correct composition point must be chosen in order for the composition to achieve the behaviour that is stated in the requirement. In particular, once we have chosen the composition connector, we need to decide whether a new component can be added at any composition point in the connector, or whether it must be added before or after any of the other existing components composed by the connector. This decision of course depends on the expected behaviour stated in the requirement. We adopt the following rule:

[Rule 1] *Choosing a correct composition point.* If the requirement states that the new component does not alter the existing control flow, then the new component can be added anywhere. For example, if the composition connector in question is a *selector*, then new components can be added anywhere. We will see examples of correct composition points in Example 1 (Section III-E).

E. A Complete Example

We demonstrate our approach on a complete example, a simplified Automated Teller Machine (ATM) system.[4]

Example 1.: The requirements for ATM are as follows:

- R1 The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).
- R2 A customer must be able to make a cash withdrawal from the linked account. Approval must be obtained from the bank before cash is dispensed.
- R3 A customer must be able to deposit cash to the linked account that can be inserted to the cash slot. Approval must be obtained from the bank before physically accepting the cash.
- R4 A customer must be able to make a transfer of money between any two accounts originated from the linked account.
- R5 A customer must be able to make a balance enquiry of the linked account.
- R6 If the customer fails to be authenticated, the card will be rejected.
- R7 After each transaction, the ATM will display and print a receipt containing the transaction information.

We have already shown how to map R1 to a partial architecture (Fig. 9(a)). This is our initial system architecture.

Consider R2. We identify the computation verbs (Table 5) *approve*, and *dispense cash*, and the computation noun (Table 6) *cash withdrawal* as computations. So we identify Authentication (for *approve*), Withdraw and DispenseCash components. The result of the *approve* computation has to be checked before we allow the *cash withdrawal* and *dispense cash* computations to be performed. So we use a *pipe* connector to compose Authentication with a composite of Withdrawal and DispenseCash. The latter composite is the result of another *pipe* connector. The partial architecture for R2 is as shown in Fig. 9(b). We need a guard adaptor for the composite

[4]Taken from http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Requirements.ht

of the Withdraw and DispenseCash components, because we only allow control to reach it if the result of invoking the Authentication component is positive.

Now we try to use incremental composition to compose the partial architecture for R2 with the current system architecture (partial architecture for R1). The results of the computations in R1 are needed for those in R2. Therefore we need to use a *pipe* to compose the two partial architectures. The top-level pipe in the partial architecture for R2 provides a suitable composition point for the partial architecture for R1. This is in accordance with Rule 1. The partial architecture that results from this incremental composition is shown in Fig. 9(c).

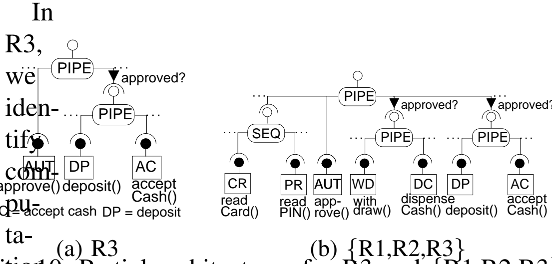


Fig. 10: Partial architectures for R3 and {R1,R2,R3}.

In R3, we identify computation verbs *deposit* and *accept cash* as computations; and a pipe to pass data between them. The prior authentication that is required is provided by the same Authentication component in the partial architecture for {R1,R2}. Thus the partial architecture for R3 is as shown in Fig. 10(a).

To compose the partial architecture for R3 with the current system architecture (partial architecture for {R1,R2}), we need not duplicate the Authentication component in the latter, which means that the top-level pipe in the latter provides a suitable composition point for the partial architecture for R3. This is in accordance with Rule 1. The result of this incremental composition is the partial architecture in Fig. 10(b).

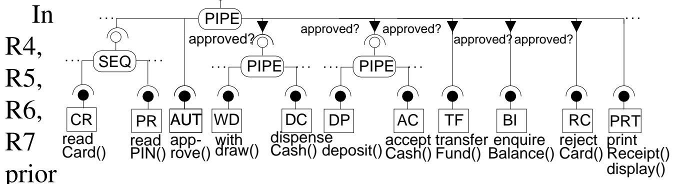


Fig. 11: Partial architecture for {R1,R2,R3,R4,R5,R6,R7}.

In R4, R5, R6, R7 prior authentication is also required, and is provided by the existing Authentication component. In each case, the top-level pipe connector in the current system architecture provides the correct composition point. Therefore, incremental composition for R4,R5 and R4 is carried out in the same manner as for R3. Omitting the details, the partial architecture for {R1,R2,R3,R4,R5,R6,R7} is as shown in Fig. 11. In R7, we identify computation verbs *display* and *print receipt* as computations. We decide to put these two computations into a single component PrintReceipt.

The remaining step is to finalise the system architecture. In Fig. 11, we can see that the guards used are checking the same values, i.e. the approval status and the transaction type. These constraints can be encapsulated in a *selector* connector whereby the constraints can be checked only once and the result determines which component receives the control flow. Another refinement we can make is to add a *loop* connector to the top-level connector. This makes the ATM system always ready for the next customer. This is not explicitly stated in the requirements, but is always required for on-line systems. Finally we close all the open compositions, and thus closing all the open interfaces. The final system is shown in Fig. 12.

Fig. 12: Final system architecture.

F. Implementation

At present we do not have a tool for editing and analysing requirements, and we map requirements to partial architectures manually. However, we do have a tool that implements incremental composition. The value of such a tool is obviously that it enables us to construct systems and then execute them. Thus it allows us to validate a final system architecture with respect to the system's requirements. By running suitable test cases, we can show that the system satisfies its requirements. The success of the tool in general also experimentally validates our approach of system construction directly from requirements in a natural language.

We have used the tool to build the ATM system from Example 1. Fig. 13(a) shows the partial architecture for R1. It corresponds to Fig 9(a). Fig. 13(b) shows the result of composing the partial architectures for R1 and R2. It corresponds to Fig. 9(c).

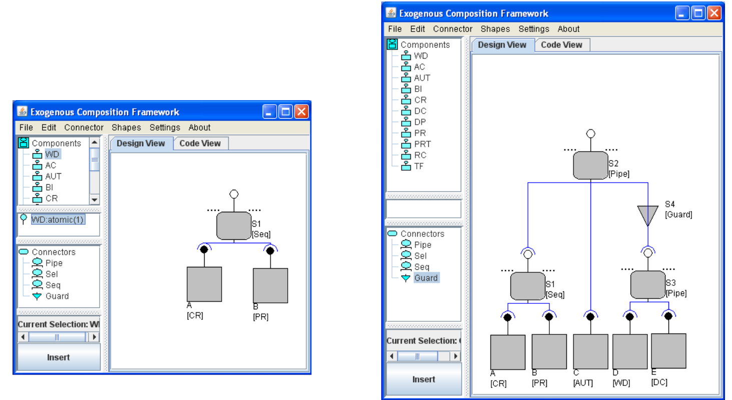
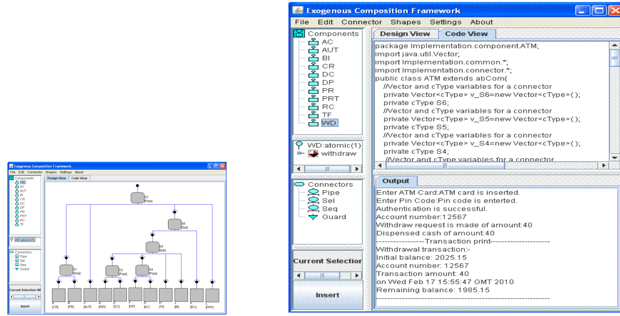


Fig. 13: Partial architectures for R1 and {R1,R2}.

The final system architecture for Example 1 after the finalisation step is depicted in Fig. 14(a). It corresponds to Fig. 12.

We have also used the tool to validate the ATM system with respect to its requirements. Fig. 14(b) shows sample results produced by executing the ATM system. We do not have the



(a) Final system architecture. (b) Sample runs.

Fig. 14: Final system architecture for ATM and its execution.

space to show the details of the test cases, but the results do indeed validate the ATM system.

Our tool facilitates experimenting with case studies, and thus allows us to test our approach on case studies of arbitrary size and complexity. In addition to the ATM example, we have experimented with a number of case studies from the literature. One of the non-trivial examples is the Steam Boiler case study [2]. This contains thirty requirements specifying the control of the water level in a steam-boiler. The system comprises the steam-boiler, a water level measurement device (WLMD), four water pumps, four pump controller devices (PCD), a steam measurement device (SMD), and a message transmission system (MTS). The program operates in several modes of operation: initialisation, normal, degraded, rescue and emergency stop. We have used our tool to compose a steam boiler system from its requirements. The final architecture is shown in Fig. 15, with 24 components and 11 composition connectors (with 6 guards).

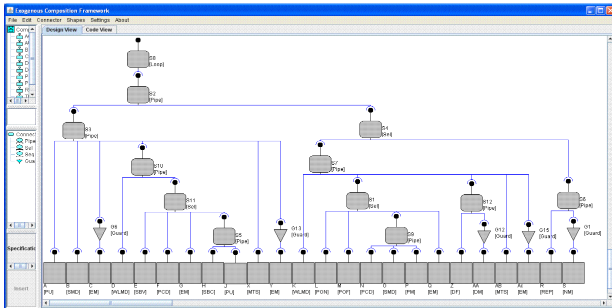


Fig. 15: Final system architecture for the Steam Boiler system.

IV. Discussion and Conclusion

We have presented an approach for constructing component-based systems directly from raw requirements. Our primary concern is to have an architecture that satisfies all the requirements. Clearly the architecture that our method produces may not be the best possible design according to various criteria. For example, the corresponding system may not be efficient in terms of execution speed.

Our approach is based on a component model, and in this regard, is different from the behaviour trees approach, which is the approach that is most closely related to our work. Furthermore, our component model supports incremental composition, which again distinguishes our work from existing related approaches. We have defined incremental composition, and demonstrated how it works for incremental system construction using individual requirements. Such an incremental approach allows us to deal with any number of requirements, and therefore it should scale up to arbitrarily large requirements documents. To demonstrate that this is the case, we need to implement a tool for editing and automatically analysing requirements and mapping them to partial architectures. We intend to do so in future.

Our approach is basically heuristic, and requires human guidance and decision making. Nonetheless we have defined the steps and rules we follow, for the sake of consistency. The biggest challenge, especially when dealing with a large number of requirements, arises when we fail to find a suitable composition point in the current system architecture for composing it with the partial architecture for the current requirement. Our current strategy is to put the current partial architecture 'on hold' and try and compose it with the (current) system architecture when it becomes possible (which is what the behaviour trees approach does). In the examples we have experimented with, this strategy has worked. For example, in the Steam Boiler example, this problem arises, probably due to the fact that the set of requirements for the case study is ordered according to the operation modes of the program. Eventually, however, we manage to compose the partial architectures of all the requirements (Fig. 15). In this regard, it is interesting to note that conceptual components (Table 6) can be used for as yet unknown computations, which can be determined from subsequent requirements.

On the other hand, we have non-determinism when there are more than one possible composition point. This is bound to arise when dealing with a large number of requirements. There cannot be hard and fast rules here, and human guidance is the only practical solution.

In our analysis of natural language requirements, many other elements could be identified than at present. We have only focused on computation and control so far. Our component model can be refined to incorporate these elements, e.g. active components, data flow, etc., and indeed different versions of our model are being constructed to accommodate them.

Another important issue to be investigated in future is architecture refactoring, and when and how it is possible. Our present approach clearly may not work for generic software systems, since it is predicated on the belief that all key concepts derived from the requirements can be encapsulated. The latter will only be true for systems with restricted behaviours or for highly compositional domains. In order to overcome this handicap, we need to be able to refactor an architecture, specifically the connector hierarchy, such that the behaviour demanded by a new requirement can be correctly added to the architecture. We plan to investigate techniques like [16].

Finally, clearly the main aim of our future work has to be to experiment with our approach on large requirements

documents. For this we will need to implement an editor and analyser for requirements. Such a tool will also allow us to map individual requirements to partial architectures, and to carry out incremental composition, automatically as far as possible. It will also provide support for recording and managing partial architectures that are 'on hold', as well as matching them with possible composition points in the current system architecture. When this tool is ready, we will be able to compare our approach with the behaviour trees approach, which already has such a tool for mapping requirements to behaviour trees but not directly to architectures.

REFERENCES

[1] R.J. Abbott. Program design by informal English descriptions. *Comm. ACM*, 26(11):882–894, 1983.

[2] J.-R. Abrial, E. Börger, and H. Langmaack. The Steam Boiler Case Study. In *Formal Methods for Industrial Applications*. Springer-Verlag, 1996.

[3] O. Barais, L. Duchien, and A.-F. Le Meur. A framework to specify incremental software architecture transformations. In *Proc. EUROMICRO Conference*, pages 62-69. IEEE, 2005.

[4] C. Ben Achour. Linguistic instruments for the integration of scenarios in requirements engineering. In *Proc. 3rd Requirements Engineering: Foundation for Software Quality (REFSQ'97)*, pages 93–106, 1997.

[5] G. Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2):211–221, 1986.

[6] N. Boyd. Using natural language in software development. *JOOP*, 11(9):45–55, 1999.

[7] P.P. Chen. Entity-Relationship diagrams and English sentence structure. In *Proc. 1st Int. Conf. on ER Approach to Systems Analysis and Design*, page 1314. North-Holland, 1980.

[8] L. DeMichiel and M. Keith. Enterprise JavaBeans, Version 3.0. Sun Microsystems, 2006.

[9] R.G. Dromey. Architecture as an emergent property of requirements integration. In *Proc. 2nd Int. Software Requirements to Architectures Workshop*, pages 77–84, 2003.

[10] R.G. Dromey. From requirements to design: Formalizing the key steps. In *Proc. of the 1st Int. Conf. on Software Engineering and Formal Methods*, pages 2–11, 2003.

[11] R.G. Dromey. Engineering large-scale software-intensive systems. In *Proc. 18th Australian Software Engineering Conf.*, pages 4–6, 2007.

[12] N.E. Fuchs, U. Schwertel, and R.Schwitter. Attempto Controlled English language manual, version 3.0. Tech. Rep. 99.03, Dept. of Computer Science, University of Zurich, 1999.

[13] B. Henderson-Sellers, and R.G. Dromey. A metamodel for the behavior trees modelling technique. In *Proc. 3rd Int. Conf. on Inf. Tech. and App.*, vol. 1, pages 35–39, 2005.

[14] G. Hamilton, editor. JavaBeans Specification. Sun Microsystems, 1997.

[15] S. Hartmann and S. Link. English sentence structures and EER modeling. In *Proc. 4th Asia-Pacific Conf. on Conceptual Modelling*, pages 27–35. Australian Computer Society, 2007.

[16] I. Ivkovic and K. Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *Proc. Software Maintenance and Reengineering*, pages 135–144. IEEE 2006.

[17] H. Jackson. *Analysing English.* 2nd Ed. Pergman Press, 1982.

[18] L. Kof. An application of natural language processing to domain modelling two case studies. *International Journal On Computer Systems Science Engineering*, 20:37–52.

[19] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In *Proc. 4th FMCO*, pages 1–21. Springer-Verlag, 2006.

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

[21] K.-K. Lau, L. Ling, and Z. Wang. Composing components in design phase using exogenous connectors. In *Proc 32nd EUROMICRO Conf. on SEAA*, pages 12–19. IEEE, 2006.

[22] K.-K. Lau and F. Taweel. Data encapsulation in software components. In *Proc. 10th CBSE*, pages 1–16. Springer-Verlag, 2007.

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

[24] K. Li, R.G. Dewar, and R.J. Pooley. Object-oriented analysis using natural language processing. Tech. Rep., Heriot-Watt University, Edinburgh, Scotland, 2005

[25] OMG. OMG Unified Modeling Language Specification, Version 2.1.2. 2007.

[26] R. van Ommering, F. van der Linden, J. Kramer and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer* 33(3):78-85, 2000.

[27] L. Rapanotti, J.G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *Proc. Int. Conf. on Requirements Engineering*, pages 80–89, 2004.

[28] C. Rolland and C. Proix. A natural language approach for requirements engineering. In *Advanced Information Systems Engineering*, pages 257–277. 1992.

[29] M. Saeki, H. Horai, and H. Enomoto. Software development process from natural language specification. In *Proc. 11th ICSE*, pages 64–73. ACM, 1989. ACM.

[30] R.N. Taylor *et al*. A component- and message-based architectural style for GUI software. In *Proc. 17th ICSE*, pages 295–304. ACM, 1995.

[31] A.M. Tjoa and L. Berger. Transformation of requirement specifications expressed in natural language into an EER model. In *Proc. 12th Int. Conf. on ER Approach*, pages 206–217. Springer-Verlag, 1994.

[32] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, LNCS 2804, pages 25–43. Springer-Verlag, 2003.