Synchronization Level Specification and Matching of Software Components

Anjali Kumari, Ketaki A. Pradhan, Lahiru S. Gallage and Rajeev R. Raje

Department of Computer and Information Science, Indiana University Purdue University Indianapolis

Indianapolis, IN 46202, U. S. A.

{ankumari, ketaki.again}@gmail.com, {lspileth, rraje}@cs.iupui.edu

Abstract - **Generating distributed systems from independently developed and deployed components is a promising alternative for today's dynamic and interconnected world. If such components have to collaborate with each other effectively, they must indicate their contracts explicitly. Traditionally, only the syntactical interfaces are depicted for such components. Recent attempts have argued for a need for multiple levels of contracts in addition to the syntactical level. One of these additional levels is the synchronization level, which is critical in the presence of several concurrent requests a component may service. This paper provides an approach for describing the synchronization contract and also indicates rules for matching concurrency specifications of different components in order to facilitate replaceability between them. A case study is provided to indicate the significance of the proposed method.**

Keywords-**Distributed Systems, Software Components, Synchronization Contracts, Matching Operators.**

1. Introduction

A promising approach for developing a distributed computing system (DCS) is using the principles of the component-based software development (CBSD). Although, building a DCS using the CBSD approach offers advantages such as reusability and scalability, it also introduces a few challenges. These challenges arise because software components are independently developed and deployed; they are designed and implemented without a prior knowledge about the system that they may be a part of and other components with which they may interact. A few of the major challenges with CBSD are:

- Component Contracts – A contract of a component indicates what it expects of its clients and what the clients can expect of it [1]. The challenge here is to provide a formal method for describing a comprehensive contract of a component.
- Component Models – There exists different development models and technologies that can be used to develop a software component. This results in heterogeneity, which needs to be bridged while developing a composed system out of components.
- Composition Predictability – The components used to develop a system have a set of attributes. The challenge here is to provide a mechanism for deriving the system attributes from individual components attributes.
- Component Certification – A component specification (along with the attributes) should also provide parameters against which the component can be verified and validated [2].
- Non-functional Attributes – Apart from the functional attributes, some of the safety-critical domains and real-time systems require assurances about the non-functional (also called as QoS – Quality of Service) attributes of the components as well. Thus, the QoS attributes have to be also formally specified in a contract.

One of the approaches suggested to tackle these challenges is to specify comprehensive, more than the basic syntactical level, contracts and use these comprehensive contracts while integrating a DCS from independent components. In [2], Beugnard et al. have suggested four levels of contracts. These four levels are: a) Syntactic/Signature level, b) Semantic level, c) Synchronization level, and d) Quality of Service (QoS) level. The design by contract methodology [3] is extended to design by multi-level contract [4] to incorporate these additional levels of a contract. Such an approach highlights aspects, apart from the traditional syntactical aspect, of components and forces the developer to consider all these levels during the development of components and a DCS made out of such components, thereby, achieving a higher confidence in them. Component contracts are, thus, an important part of developing a high-confidence DCS.

The specifications of contracts of the components are matched and combined to predict the behavior of the resulting system. This helps in verifying the behaviors of the resulting system against the required system specifications and detecting problems before actually integrating the components. Once the contracts of the components have been created, they are evaluated with the contracts of the other components which are brought together to form a system. It helps to determine whether the underlying semantics allows components to interoperate with each other. Although much of the focus until now has been on the signature/syntactic and semantic level [5, 6, 7], there is a need for specifying and matching components at the synchronization as well as the QoS levels. The focus of the paper is to specify the synchronization contract of software components and to define associated matching operators.

2. Related work

Zaremski et al. [5, 6, 7] present a way to efficiently retrieve software entities (i.e., service and components) from a library (i.e., public registries which collect, index, and categorize these services based on their meta-data). Their approach is based on the assumption that software entities are stored in a library and each entity in a library has a signature (type information) and a specification (behavioral information). It presents a solution for the retrieval, indexing, navigation, substitution, sub-typing, and modification of the components. It also claims that as a component's behavior plays an important role in describing its functionality, matching of its specifications becomes an essential step in order to determine whether a particular relation (e.g., compatibility) holds among the components. It also defines a comprehensive set of matches for the functions and the modules of components and for the signature and specification of the component's operations. Thus, it addresses the specification and matching of the first two levels (i.e., syntax and semantics) of the multi-level specifications proposed in [2]. This work presented in this paper builds upon the foundation provide in [5, 6, 7].

Li et al. [8] provide a formal multi-level contract matching for embedded systems, at levels of signature, functionality, QoS and Synchronization. This is similar to the approach proposed in this paper, where we perform component matching at levels of syntax, semantics, synchronization and QoS. However, in [8], the synchronization contract is made up of interaction and glue contracts and is specified using CSP-based techniques. This view is at a different level of abstraction than proposed in this paper and the matching operators defined in [8] are not as comprehensive as proposed in this paper.

Schmidt et al. [9] describe matching at the QoS level of contracts by QoS contract negotiation as a Constraint Satisfaction Problem. They classify the QoS properties into coarse-grain and fine-grain to provide an effective matching when selecting integrated components in distributing nodes. This classification of the QoS properties into coarse-grain and fine-grain is similar to the generalized and specialized matching approaches that this paper proposes at the synchronization level. However, the matching semantics for the QoS level (typically using arithmetic comparisons) is different than the semantics at the synchronization level (as discussed in section 4).

Hatfield et al. [10] describe the matching of contracts from a point of bilateral substitution. They propose a model in which more than one relevant concept of substitution exists and different properties hold under different conditions in matching with contracts, unlike more traditional matching markets without terms of contract. This concept can be used as a future work in the multi-level matching process for selecting relevant components for a specific query.

Formal methods are also used to represent synchronization contract details, for example Klaus et al. [11] describe a formal mechanism for many-to-many matching markets with contracts and provide an analysis of the relations between the resulting sets. Their method describes how the contract should be presented (in general) as required for their set pair-wise matching operators. However, it does not provide ways to convert publically available contract information (e.g., synchronization contract) at different levels. Collet et al. [12] propose a contract for representing component interfaces in terms of both functional and non-functional contracts which are suitable for component oriented programming. They also investigate how the different levels of contract may relate to each other and what properties must be provided on each form of contract. They have identified Synchronization contracts to deal with concurrency issues, but they fail to provide details of its contents, which this papers tries to provide.

There have been various attempts at defining synchronization contracts in different programming languages. For example, Behrends [13] presents an implementation of a synchronization mechanism for object-oriented programs, called the universe model, which is based on declaratively specified contractual relationships between client and supplier components at the design level. These contracts specify basic mutual exclusion and semantic predicates. The explicit logical compositions of those basic contracts are done by boolean operators. [13] also presents a classification scheme for synchronization contracts, and describes how they can be added to existing object-oriented languages by means of a simple declarative language that expresses these contracts. Stirewalt et al. [14] specify a synchronization aspect for object-oriented languages, by following the concepts of separation of concerns. Their proposal of a Synchronization Units Model unifies new features for expressing synchronization in multi-threaded programs. They suggest that the semantics of a contract should be a self-contained partial specification (called a view). Similarly, Petrascu et al. [15] propose a contract aware component modeling language (ContractCML) to provide software components with a four level contract specification - syntax, semantics, synchronization, quality of service. Their method introduces Contract Component Modeling Language which is domain specific and their synchronization contract is also based on domain specific details related to component synchronization, which is different from the approach provided in this paper. All these approaches are focused on specific languages, while the work presented in this paper is general and independent of a specific language or a paradigm.

The problem of synchronization could be thought at many levels, not only in designing and developing at the single component level but also at higher levels. For example, Service Oriented Architecture (SOA) promotes reuse of components to integrate systems and while integrating components there is a requirement to synchronize them. One accepted way of achieving this task is by following Business Process Execution (BPEL) [16] standards. For example, the BPEL execution semantics are used to synchronize components at relational database level by Oracle SOA Suite [17]. The suite achieves business to business integration and business activity monitoring by using Web services as their synchronization middleware. The BPEL also provides necessary synchronization for SOA composite applications [18], where set of dynamic services are working towards achieving a common business task. Using the BPEL and related standards at the component integration level provide similar information which could be used as a synchronization level contract at the component integration level. However, this is different from providing a detailed synchronization level contract at the component level, which is the focus of this paper.

There exist few other generic efforts to describe synchronization contracts. For example, Molina-Jimenez et al. [19] describe automatically managing partnerships within virtual organizations by using contracts between interacting entities. They also describe a method of representing permissions, obligations, prohibitions, actors (agents), time constraints, and message type checking. Synchronization contract details are viewed in the forms of correct order of acknowledgements of synchronized messages within specific virtual organizations. Also, Bartoletti et al. [20] investigate how contracts are used to organize and regulate the interactions between inter-communicating processes. The basis for their proposal is a variant of the concurrent constraints calculus, featuring primitives for multiparty synchronization via contracts. The contract can use these primitives for representing their synchronization mode and also use it to model some contract-based interactions. A part of their work focuses on encoding the p -calculus and graph rewriting for synchronization primitive operations. Finally, Huang et al. [21] describe a case study on contract-based synchronization for IP-based telecommunication services. The contract presented in their study is mainly based on messaging and security information and their synchronization. Basically, the synchronization information is used by a middleware which manages lifecycle concerns of threads, to automatically synchronize concurrent service executions based on declarative synchronization contracts. None of these methods formalize a comprehensive approach to specify synchronization contracts of components, which is the intention of this research.

The primary shortcoming of above mentioned methods is that most of the synchronization contracts proposed in these efforts are specific to either a particular application or a programming language. For example, as indicated above, Li et al. [8] provide a formal multi-level contract matching for embedded systems and Behrends et al. [13] present an implementation of a synchronization mechanism for object-oriented programs. Also, there are very few attempts which use the synchronization contracts during searching for the best suitable components for a particular requirement. As the main contribution of this paper is the formal and comprehensive specification of the synchronization contract, it addresses the limitations, which are described above, of the other prevalent approaches. The paper also briefly indicates associated matching rules that use this contract.

3. THE SYNCHRONIZATION LEVEL CONTRACTS

In a component-based DCS, components run concurrently and communicate typically through synchronous remote procedure calls. In this scenario, synchronization helps in guarding the methods of the server component that are accessed by multiple client components at the same time and the resources provided by the server component. Thus, a synchronization contract of a server component needs to provide appropriate concurrency-related information to its clients to ensure a correct usage in the presence of concurrent requests. Two main questions that any synchronization contract needs to address are: what should the synchronization contract contain?; and how is the synchronization contract formally represented? This section of the paper provides an approach to answer these questions. The proposed synchronization level contract for a software component contains the following information:

- Synchronization policy – this part of the synchronization contract indicates the policy used by the component's developer to handle multiple client accesses/requests to a particular method. Examples of various policies include the mutual exclusion, the readers-writers policy, and the producer-consumer policy.
- Behavior of the Synchronization policy – This describes the behavior of the synchronization policy utilized by the components to protect their methods from concurrent accesses. It is provided by indicating the change in the state of the variables of the software components, before and after the execution of the methods.
- Implementation of the Synchronization policy –This indicates the method used to implement the synchronization policy in a component. For instance, the policy mutual exclusion can be implemented using semaphores or mutexes.
- Synchronization pre-condition – The synchronization precondition associated with a method specifies the synchronization conditions/constraints that the caller must satisfy in order to start the execution of that method. It specifies the obligations that a client component must meet before it may invoke the server method.
- Synchronization action – The synchronization action specifies the action that will be taken by the component if the associated pre-condition is satisfied.
- Synchronization invariants – The synchronization invariants specify the conditions that must hold throughout the process of a client making an invocation on a server component's method, the server component executing the method and the termination of that method.
- Synchronization post-condition – The synchronization post-condition associated with a method specifies the conditions that must hold after the execution of that method is completed. It specifies the guarantees that the server component makes to its clients.

3.1. The Synchronization Policy Catalog

There exist a number of basic synchronization policies that could be used to protect the shared resources. A component developer can use these basic synchronization policies or their variations to protect their components during concurrent accesses in a DCS. Hence, there is a need for a catalog that provides a list of basic synchronization policies and their associated formal representation. The proposed synchronization policy catalog provides a list of the synchronization policies that could be utilized by the software components to protect their methods during concurrent accesses. It contains detailed descriptions of the various synchronization policies. It acts as a reference manual for the component developers to incorporate the synchronization policies while implementing components. This catalog provides the following information about the synchronization policies:

- Name: It indicates the name of the synchronization policy.
- Processes: It indicates the minimum number of processes involved in the policy.
- Region: It indicates the number of critical regions that the processes are going to be in while trying to access the shared resource. It also indicates the number of the shared resources that are being protected from the concurrent accesses by the policy.
- Behavior: It indicates the behavior of the synchronization policy in terms of the processes and the regions.

The following is the list of basic synchronization policies identified by this work:

- Mutual Exclusion Policy
- Conditional Synchronization
    a. Barrier
    b. Bounded Buffer
    c. Readers Writers
- Priority Synchronization
    a. FCFS Priority
    b. SJF Priority
- Other Synchronization Policies
    a. Bound
    b. Relay

Below, as an example, the description of the behavior of one of the synchronization policies is indicated:

Mutual Exclusion Policy:

- Name: MutualExclusion
- Processes: *n* Processes
- Region: *n* Critical Regions; *one* Shared Resource
- Behavior:
    a. If a process P_i is executing in its critical region then no other processes P_j can be executing in its critical region.
    b. If no process is executing in its critical region and there exist some

processes that wish to enter their critical regions, then the selection of the process that will enter the critical region next cannot be postponed indefinitely.

It can be implemented using the following methods:

a. Request (P_i) – It provides a method for the process Pi to request the shared resource.
   i. Pre-condition: state of the process P_i is ready.
   ii. Invariant:
      - If the state of a process P_i is executing in its critical section and the state of another process P_j is also executing in its critical section then $P_i = P_j$.
      - State of any other process P_j is ready, waiting, or released.
   iii. Action: Change the state of the process P_i to waiting.
   iv. Post-condition: The state of the process P_i becomes waiting.

b. Acquire (P_i) – It provides a method for the process Pi to acquire the shared resource.
   v. Pre-condition: state of the process P_i is waiting.
   vi. Invariant:
      - If the state of a process P_i is executing in its critical section and the state of another process P_j is also executing in its critical section then $P_i = P_j$.
      - State of any other process P_j is ready, waiting, or released.
   vii. Action: Change the state of the process P_i to executing.
   viii. Post-condition: The state of the process P_i becomes executing.

c. Release (P_i) – It provides a method for the process Pi to release an already acquired resource.
   ix. Pre-condition: state of the process P_i is executing.
   x. Invariant:
      - If the state of a process P_i is executing in its critical section and the state of another process P_j is also executing in its critical section then $P_i = P_j$.
      - State of any other process P_j is ready, waiting, or released.
   xi. Action: Change the state of the process P_i to – "released".
   xii. Post-condition: The state of the process P_i becomes released.

The behavior for other synchronization policies can be described in a similar way. The TLA+ (Temporal Logic of Action plus) [22] is used to formally represent the behavior of synchronization policies.

3.2. *The Synchronization Contract*

As indicated earlier, the synchronization contract of a component should provide information about the synchronization policy utilized by each of the methods of the component, the method used to implement the synchronization policy, and the synchronization behavior's pre- and post- conditions, and invariants. Thus, the synchronization contract of a component consists of the following attribute-value pairs:

<u>Name of the Component</u> – It indicates the name of the component and is represented as:
Component: <ComponentName> … where *ComponentName* is the name of the component.

<u>Name of the Interface</u> – It indicates the name of the interface of the component. It is represented in the following form:
Interface: <InterfaceName> … where *InterfaceName* is the name of the interface supported by the component.

<u>Name of the Synchronization Policy</u> – It indicates the name of the synchronization policy from the synchronization policy catalog. It is represented as:
SynchronizationPolicy: <SynchronizationPolicyName> … where *synchronizationPolicyName* is the name of the synchronization policy from the synchronization policy catalog.

<u>Synchronization Policy Implementation</u> – It indicates the name of the method used to implement the synchronization policy. It is represented as:
SynchPolicyImplementation:
<SynchronizationPolicyImplementation>
… where *synchroniztionPolicyImplementation* represents the name of the implementation method.

Synchronization Precondition – It states the pre-condition with respect to the synchronization behavior of the method of the component. It is indicated by the following form:
SynchronizationPrecondition: <precondition> … where *precondition* represents the pre-condition associated with the synchronization behavior of the method.

Synchronization Invariant – It states the invariant with respect to the synchronization behavior of the method of the component. It is represented using the following form:
SynchronizationInvariant (<MethodName>, <Invariant>) … where *MethodName* is the name of the method of the component and *Invariant* represents the synchronization invariant associated with that method.

Synchronization Action – It states the action, with respect to the synchronization behavior, that is taken by the method of the component. It is represented using the following form:
SynchronizationAction: <Action> … where *Action* represents the synchronization action to be taken by the method.

Synchronization Postcondition – It states the post-condition with respect to the synchronization behavior of the method of the component. It is represented using the following form:
SynchronizationPrecondition: <postcondition> … where *postcondition* represents the post-condition of the synchronization behavior associated with the method.

Hence, the synchronization contract of a component has the following general form:

Component: <ComponentName>
Interface: <InterfaceName>
MethodName: <MethodName>
SynchronizationPolicy: <SynchronizationPolicyName>
SynchPolicyImplementation: <SynchronizationPolicyImplementation>
SynchronizationPrecondition: <precondition>
SynchronizationInvariant: <Invariant>

SynchronizationAction: <Action>
SynchronizationPostcondition: <postcondition>

It should be noted that although the above example contact indicates only one method, a component can have many methods that need to be synchronized and in that case, the associated attributes (e.g., policy, implementation, and action) are repeated in the contract for all these methods.

4. Matching of contracts

Once a multi-level contract is created for a component, it is useful in two ways: a) to develop the component using the design-by-multi-level contracts technique, and b) to select a relevant choice out of available ones while creating a DCS. In order to achieve the latter objective, the process of contract matching is a necessary step in selecting relevant choices out of the available ones. As there are four levels in the contract and each level has different purpose and representation, the matching operators at different levels also have different semantics. As the focus of this paper is the synchronization level, a detailed discussion about the matching policies for that level is presented below.

4.1. Syntactic and Semantic Level Contract Matching

Zaremski et al. [5, 6, 7] argue that the syntactic/signature matching of software components boils down to type matching of the method's parameters and the return values. It defines a set of exact and relaxed matches for matching the signature of the methods provided by the components. The relaxed matches allow reordering of elements in a tuple, uncurrying of arguments to a method, renaming of type constructors and instantiation of type variables. [5, 6] define a set of matching criteria for the syntactic and the semantic level. The matching at this level is done assuming that the component's specifications have been matched at the syntactic level. The matches are defined in terms of their functions pre- and post-conditions. Hence, it is assumed that matching of at the syntactical and semantic levels, using the operators defined in [5, 6, 7], is carried out before the matching at the synchronization level.

4.2. Synchronization Level Contract Matching

The matching at the synchronization level is based on the details available in the synchronization contract which are described in the previous section. The matching process helps in determining if the synchronization behavior of a DCS remains unchanged if a component, constituting that system, is replaced by another component. Hence, the matching process requires comparing the synchronization contract of the replaced component with that of the replacing component for substitutability and also comparing the contract of the replacing component with that of the other components that form the system for compatibility. The matched component should be such that replacing with the component does not change the behavior of the system and there is no deadlock or starvation. The two types of checks with respect to synchronization are defined as follows:

- Substitutability – With respect to synchronization, substitutability is defined as the ability of two components to replace each other without changing the synchronization behavior of the system that is formed using the original component.

- Compatibility – With respect to synchronization, compatibility is defined as the ability of two components to interoperate and synchronize properly when brought together to form a system.

1.1.1 Matching of the Synchronization Policy

Matching of software components at the synchronization level is achieved by matching the behavior of their synchronization policies. It helps in determining the relationships between the synchronization policy utilized by a target software component and that indicated by a query (referred to as a "query component"). The synchronization policies of the query component and a target component may be related to each other through one of the following relationships:

- Equivalence – This relationship indicates that the two synchronization policies are behaviorally equivalent, i.e., replacing one synchronization policy with another in a component's implementation does not change the synchronization behavior of the component.
- Implication – This relationship indicates that a synchronization policy, $SP_{1,}$ implies the other policy SP_2, and hence, the synchronization behavior of the component does not change if SP_2 is replaced with SP_1, but the synchronization behavior of the component may change if the synchronization policy SP_1 is replaced with SP_2.
- Reverse Implication – This relationship indicates that a synchronization policy, $SP_{2,}$ implies the other policy SP_1, and hence, the synchronization behavior of the component does not change if SP_1 is replaced with SP_2, but the synchronization behavior of the component may change if the synchronization policy SP_2 is replaced with SP_1.

The synchronization policies of the target and query components can be matched for equivalence, implication, and reverse implication. This is achieved by matching the associated preconditions, invariants, and post-conditions of the methods supported by the synchronization policies of these components. For instance, assume that there are two synchronization policy specifications, SP_1 and SP_2, and these policies support the methods MSP_{1i} and MSP_{2i}, respectively. The preconditions of MSP_{1i} and MSP_{2i} are given by MSP_{1iPre} and MSP_{2iPre} respectively. The post-conditions of MSP_{1i} and MSP_{2i} are given by MSP_{1iPost} and MSP_{2iPost} respectively. The invariants of MSP_{1i} and MSP_{2i} are given by MSP_{1iInv} and MSP_{2iInv}, respectively. Then the matching of the specifications of the two synchronization policies is defined as follows:

- Equivalence: $SP_1 \equiv SP_2$, where ≡ represents the equivalence relationship, iff:
  - $MSP_{1iPre} \Leftrightarrow MSP_{2iPre}$ and
  - $MSP_{1iInv} \Leftrightarrow MSP_{2iInv}$ and
  - $MSP_{1iPost} \Leftrightarrow MSP_{2iPost}$… where ⇔ indicates the logical equivalence relation.

- Implication: SP_1 $SP_{2,}$ where represents implication relationship, iff:
  - $MSP_{1iPre} \rightarrow MSP_{2iPre}$ and
  - $MSP_{1iInv} \rightarrow MSP_{2iInv}$ and
  - $MSP_{1iPost} \rightarrow MSP_{2iPost}$ … where → indicates the logical implication relation.

1.1.2 Generalized and Specialized Matches

The matching of the synchronization behavior of a target component with that of a query component can be further divided into two categories, similar to the operators defined in [5, 6, 7]:

i. Generalized Match – A generalized match consists of comparing the synchronization policies utilized by the components but not their implementations. Hence, a generalized match between a query component, QC, and the target component, C, that supports methods MQC_i and M_i, respectively, is defined as:

   matchgen (C, QC) = For each method M_i and MQC_i ($SP\ R_1\ SPQC$)…(Eq. 4.1)

   where SP and SPQC are the synchronization policies used by the target component C and the query component QC, respectively. R_1 is the relationship that exists between the synchronization polices SP and SPQC. It can take the values '≡' or depending on if there is an equivalence or implication relationship between them.

ii. Specialized Match – A specialized match consists of comparing the synchronization policies utilized by the components and their associated implementations. Hence, a specialized match is obtained by further constraining the generalized match and is defined as:

matchspec (C, QC) = matchgen (C, QC) && (SP_{impl} R_2 $SPQC_{impl}$)…(Eq. 4.2)

where SP_{impl} and $SPQC_{impl}$ are the techniques used to implement the synchronization policies SP and SPQC respectively. R_2 is '≡' for the specialized match. && indicates logical "AND" operation.

The generalized and specialized matches described above can further be divided into exact and relaxed matches (similar to the definitions in [5, 6, 7]) as defined below.

1.1.2.1 Generalized Match

Exact Match – In exact match, the relation R_1 (indicated in Equation 4.1) between the two synchronization policies is the equivalence relation. Hence, exact match is defined as:

exactMatchgen (C, QC) = For each method Mi and MQCi (SP ≡ SPQC)…(Eq. 4.3)

Relaxed Match – In relaxed match, the relation R_1 (indicated in Equation 4.1) between the two synchronization policies is the implication relation. Hence, relaxed match is defined as:

relaxedMatchgen (C, QC) = For each method Mi and MQCi (SP SPQC)…(Eq. 4.4)

1.1.2.2 Specialized Match

Exact Match – The exact match for the specialized case is obtained by putting an additional constraint, in the form of checking the implementation techniques, on Equation 4.3. Hence, the exact match is defined as:

exactMatchspec (C, QC) = exactMatchgen (C, QC) && (SP_{impl} ≡ $SPQC_{impl}$)…(Eq. 4.5)

Relaxed Match – The relaxed match for the specialized case is obtained by putting an additional constraint, in the form of checking the implementation techniques, on Equation 4.4. Hence, the relaxed match is defined as:

relaxedMatchspec (C, QC) = relaxedMatchgen (C, QC) && (SPimpl SPQCimpl)…(Eq. 4.6)

4 CASE STUDY

1.1 Document Management System

The proposed synchronization contract and the matching operators are described in this section using a case study from the domain of a Document Management System. This system provides the following basic document management services:

1. Validating a user – The service allows the users to validate their username and password to check if he/she is allowed to perform the following functions on the documents: create a new document, delete an existing document, read an existing document, and write an existing document.
2. Create a new document – This service enables the users to create a new document.

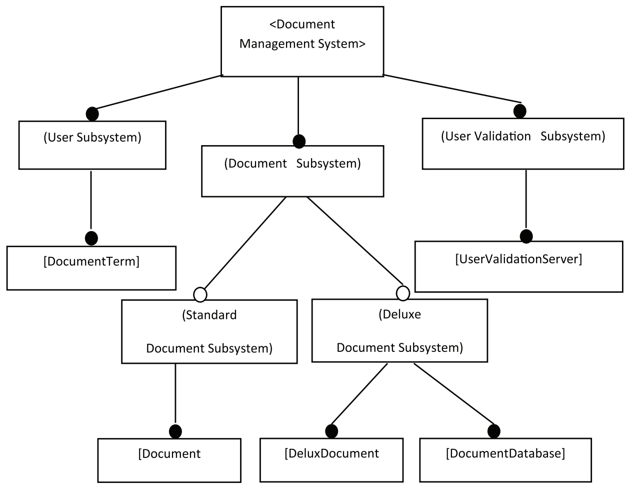

FIGURE 5.1 FEATURE DIAGRAM OF THE DOCUMENT MANAGEMENT SYSTEM

3. Deleting an existing document – The services enables the users to delete an existing document.
4. Read an existing document – It allows the users to read an existing document, but restricts them to change the document.
5. Write a document – It allows the users to read as well as write/update the document.
6. List the documents – It allows the user to list the names of the documents in the folder.

The Document Management System consists of the following subsystems: 1) User Subsystem, 2) User Validation Subsystem, and 3) Document Subsystem. The user subsystem acts as the interface to the users and provides all the functionalities needed by the users of the

Document Management System. It provides the user with the functionalities and requires some of the functionalities from the other two subsystems. The user validation subsystem provides the service of validating a user to the User subsystem. The document subsystem provides the functionalities needed to manage the documents. The functionalities provided by the user validation subsystem and the document subsystem are required by the user subsystem to perform its tasks. Figure 5.1 shows the feature diagram (using the notations in [23]) of the document management system. It indicates the family of systems that can be formed from its subsystems. The leaf nodes in the above diagram represent the abstract components. Abstract components are guidelines for developing concrete components. The abstract components, connected to a subsystem with a line and a solid circle, represent required components; whereas those with hollow circles represent a choice between the abstract components. For instance, in Figure 5.1, the Standard Document Subsystem and the Deluxe Document Subsystem are the choices for the Document Subsystem of the Document Management System.

Two types of systems can be formed from the subsystems shown in the above feature diagram (Figure 5.1):

1. A Simple Document Management System consisting of a *DocumentTerminal*, a *DocumentServer* and a *UserValidationServer.*
2. A Deluxe Document Management System consisting of a *DocumentTerminal*, a *UserValidationServer*, a *DeluxeDocumentServer*, and a *DocumentDatabase*.

The paper uses the Simple Document Management System for illustrating the key concepts discussed in the paper. The next section gives the description of the functionalities provided by the components of the Simple Document Management System.

1.2. A Simple Document Management System

The Simple Document Management System consists of three components as mentioned in the previous section. A component diagram of the Simple Document Management System is shown in the Figure 5.2. The following describes the functionalities of these three components along with the description of their interfaces and collaborator components. Collaborator components are other components that a component interacts with. There are two types of collaborators, preprocessing collaborators – other components on which this component depends, and post-collaborators – other components that may depend on this component.

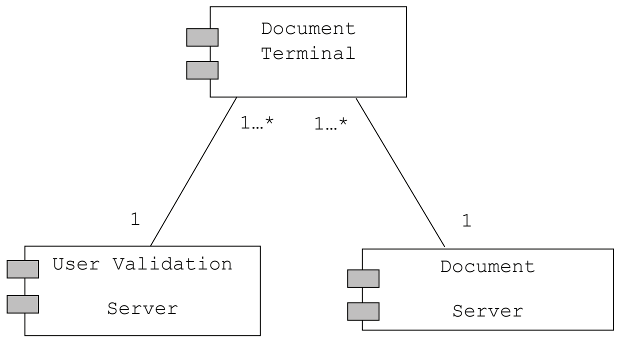

FIGURE 5.2 COMPONENT DIAGRAM OF SIMPLE DOCUMENT MANAGEMENT SYSTEM

The three components of the system are being described here:

1. Document Server – It provides basic document services to the document terminal such as *create document, get document, write document* and *delete document*.
   Interfaces:
   Provided: *IDocumentManagement*
   Required: None
   Collaborator components:
   Preprocessing: *DocumentTerminal*
   Postprocessing: None

2. User Validation Server – It provides user validation service for the users of the system such as *validate user*.
   Interfaces:
   Provided: *IValidation*
   Required: None
   Collaborator components:
   Preprocessing: *DocumentTerminal*
   Postprocessing: None

3. Document Terminal – It provides a Graphical User Interface for the users of the system. The component gets the services from the other two components (User Validation Server, Document Server) and provides those services to the user.

   Interfaces:
   Provided: *IDocumentTerminal*
   Required: *IDocumentTerminal*
   Collaborator components:
   Preprocessing: None
   Postprocessing:
   DocumentValidationServer, DocumentServer

1.2.1. Interface Model

The section describes the methods of the interfaces used by the components of the Simple Document Management System. The interfaces are:

1. *IDocumentManagement* – It consists of the following methods –
   a. *void createDocument (String documentName)* – A method that allows users to create a new document with the name specified by the argument '*documentName*'.
   b. *void deleteDocument (String documentName)* – A method that allows users to delete a document with the name specified by the argument '*documentName*'.
   c. *void readDocument (String documentName*) – A method that allows users to open a document, with the name specified by the argument '*documentName*'. The contents of this document cannot be modified or updated.
   d. *void writeDocument (String documentName)* – A method that allows users to open a document, with the name specified by the argument '*documentName*'. The contents can be modified and stored back into the database.
   e. *String[] listDocuments ( )* – A method that returns a list of all the documents in the folder to the users.

2. *IValidationServer* – It consists of the following method –
   a. *boolean validateUser (String username, String password)* – A method that validates a user by validating the username and the password entered by the user.

3. *IDocumentTerminal* – It consists of the following methods –
   a. *void createDocument (String documentName)* – A method that allows users to create a new document with the name specified by the argument '*documentName*'.
   b. *void deleteDocument (String documentName)* – A method that allows the users to delete a document with the name specified by the argument '*documentName*'.
   c. *void readDocument (String documentName)* – A method that allows the users to open a document, with the name specified by the argument '*documentName*'. The contents of the document cannot be modified or updated.
   d. *void writeDocument (String documentName)* – A method that allows the users to opens the document, with the name specified by the argument '*documentName*'. The contents can be modified and stored back into the database.
   e. *String[] listDocuments ( )* – A method that returns a list of all the documents in the folder to the users.
   f. *Boolean validateUser (String username, String password)* – A method that validates a user by validating the username and the password entered by the user.

1.2.2. Component Interactions

The user interacts with the Document Terminal, by providing the username and the password. The Document Terminal uses the services of User Validation Server to validate the user. Once validated, the user is allowed to send requests to the Document Terminal. The Document Terminal uses the services of the Document Server to fulfill the requests of the users and returns back the results to the users.

1.2.3. Synchronization Level Contract

Once the component interfaces are defined, the next step is to specify the synchronization level contracts of all the three components, *DocumentTerminal, UserValidationServer* and the *DocumentServer*. For the sake of brevity, only the synchronization level contract of the *DocumentTerminal* is shown below.

- Component: *DocumentTerminal*
  Interface: *IDocumentTerminal*
  a. *createDocument (documentName, C)*
     - Signature :- *void createDocument(String documentName, Client C)*

- Synch Policy :- *SP("createDocument", "Mutual Exclusion")*
- Pre-condition :- *((nonExistent(documentName) == True) && (validateUser(C)))*
- Invariant :- None
- Action :- *execute("createDocument")*
- Post-condition:- *(exist(documentName) == True)*

b. *deleteDocument (documentName, C_i)*
- Signature :- *void deleteDocument(String documentName, Client C_i)*
- Synch Policy :- *SP ("deleteDocument", "Mutual Exclusion")*
- Precondition:- *forall C_i in clients[]: (~execute("readDocument", C_i) && ~execute("writeDocument", C_i) && ~execute("deleteDocument", C_i) && ~execute("listDocuments", C_i) && (exist(documentName) == True)) && (validateUser(C_i)))*
- Invariant :- *forall C_i in clients[]: (~execute("readDocument", C_i) && ~execute("writeDocument", C_i) && ~execute("deleteDocument, C_i) && ~execute("createDocument", C_i) && ~execute("listDocuments", C_i))*
- Action :- *execute("deleteDocument", documentName)*
- Post-condition :- *((nonExistent(documentName) == True)*

c. *readDocument (documentName, C_i)*
- Signature :- *DocumentType readDocument(String documentName, Client C_i)*
- Synch Policy :- *SP("readDocument", "ReadersWriters")*
- Precondition:- *forall C_i in clients[]: (~execute("writeDocument", C_i) && ~execute("deleteDocument", C_i) && ~execute("createDocument", C_i) && (exist(documentName) == True)) && (validatedUser(C_i)))*
- Invariant :- *forall C_i in clients[]: (~execute("writeDocument", C_i) && ~execute("deleteDocument", C_i) && ~execute("createDocument", C_i) && ~execute("listDocuments", C_i))*
- Action :- *execute("readDocument", documentName)*
- Post-condition:- *(nonModified(documentName) == True)*

d. *writeDocument (documentName, C_i)*
- Signature :- *void writeDocument(String documentName, Client C_i)*
- Synch Policy :- *SP("readDocument", "ReadersWriters")*
- Precondition :- :- *forall C_i in clients[]: (~execute("writeDocument", C_i) && ~execute("deleteDocument", C_i) && ~execute("createDocument", C_i) && ~execute("readDocument", C_i) && (exist(documentName) == True)) && (validateUser(C_i)))*
- Invariant :- *forall C_i in clients[]: (~execute("writeDocument", C_i) && ~execute("deleteDocument", C_i) && ~execute("createDocument", C_i) && ~execute("readDocument", C_i) && ~execute("listDocuments", C_i))*
- Action :- *execute("writeDocument", documentName)*
- Post-condition :- *(nonModified(documentName) == False)*
-

e. *listDocuments ( )*
- Signature :- *String [] listDocuments( )*
- Synch Policy :- *None*
- Precondition: *True*
- Invariant :- *None*
- Action :- *execute("listDocuments")*
- Post-condition :- *True*

f. *validateUser(C)*
- Signature :- *boolean validateUser(String username, String password)*
- Synch Policy :- *None*
- Precondition :- *True*
- Invariant :- *None*
- Action :- *execute("validateUser")*
- Post-condition :- *True*

Notations:

execute (Method M, Client C): returns true if client *C* is executing Method *M*.

~execute (Method M, Client C): returns true if client *C* is

not executing method *M*.
SP (Method M, Synchronization Policy P): returns true if the method *M* follows the synchronization policy *P* for multiple accesses by the clients.

The synchronization contracts for the remaining two components, *UserValidationServer* and the *DocumentServer,* will contain similar details as indicated above and are described in [24].

1.1.4 Matching of Contracts

The component interaction in simple document management system as explained above indicates that the user interacts with *DocumentTerminal.* The components *DocumentServer* and *UserValidationServer* provide services to *DocumentTerminal* which in turn provides results to the user. In order to determine whether these three components are compatible to each other or not from the perspective of synchronization, the contracts of *DocumentServer* and *DocumentTerminal* and that of *UserValidationServer* and *DocumentTerminal* have to be matched. The component *DocumentTerminal* requires the functionalities of the component *DocumentServer* and hence, the specification of *DocumentServer* should satisfy the properties of the *DocumentTerminal*. This is achieved using the TLA+ specifications as indicated in [24].

With respect to substitutability in the context of the case study, various matching operators defined above have to be applied to select whether one instance of a particular type (e.g., *DocumentTerminal)* can be replaced by another instance. An empirical validation of the above mentioned multi-level matching operators is described in [25] and is omitted here for the sake of brevity. It uses two metrics to assess the outcomes of the matching process: a) quality of the components (as indicated by precision and recall) returned by the multi-level matching, and b) the individual level and overall time taken to perform the matching operation. The results of these experiments indicate that multi-level matching does indeed return more relevant components for a particular query at the cost of additional time.

5. Future work

Several future extensions to this work are possible and a few of them are:

- The formal representation of the contracts at the synchronization levels is done manually. For complex system creating such specifications might be time consuming. Moreover, the user has to learn a new language to represent the contracts. Hence, a system to automatically generate these formal representations would make this task easier.
- The catalog of the basic synchronization policies can be extended to include other synchronization policies as well and provide specifications for them.
- A runtime monitoring system can be designed to verify the actual system against the specification of the system derived from composing the component's specification.

6. Summary

The paper illustrates a mechanism to define synchronization contracts for software components. Such a contract forms one level of the multi-level specification and aids in creating software components that follow the design by multi-level contract technique. It also defines various matching operators that are applicable at the synchronization level. These operators assist in identifying relevant components from available choices for a particular query.

References

[1] R. Heckel and M. Lohmann, “Towards Contract-based testing of Web Services”, *Electronic Notes in Theoretical Computer Science*, Vol. 82, No. 6; pp. 1-12, 2004.

[2] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins, “Making Components Contract Aware”, pp. 38-44, *IEEE Computer* July 1999.

[3] B. Meyer, “Applying Design by Contract”, *IEEE Computer,* Vol. 25, No. 10; pp. 40-51, 1992.

[4] R. Raje, M. Auguston, B. Bryant, A. Olson, and C. Burt, “A Unified Approach for the Integration of Distributed Heterogeneous Software Components”, *Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration*; pp. 109-119, 2001.

[5] A. Zaremski, and J. Wing, “Signature Matching: A Tool for Using Software Libraries”, *ACM Transactions on Software Engineering and Methodology,* ; Vol. 4, No. 2; pp. 146-170, 1995.

[6] A. Zaremski, “Signature and Specification Matching”, *Technical Report CMU-CS-96-103, Carnegie Mellon Computer Science Department, PhD Thesis*, 1996.

[7] A. Zaremski, and J. Wing, “Specification Matching of Software Components”. *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 4, pp. 333-369*,* 1997.

[8] C. Li, X. Zhou, Y. Dong, and Z. Yu, “A Formal Model for Component-Based Embedded Software Development”, *International Conferences on Embedded Software and Systems*, pp. 19-23, DOI 10.1109/ICESS.2009.51, 2009.

[9] H. Schmidt, I. Crnkovic, G. Heineman, and J. Stafford, " An approach for QoS contract negotiation in distributed component-based software", *In Proceedings of the 10th international conference on Component-based Software Engineering* (CBSE'07), Springer-Verlag, Berlin, Heidelberg, pp. 90-106, 2007.

[10] J. Hatfield, and F. Kojima, "Substitutes and stability for matching with contracts", *Journal of Economic Theory*, Vol. 145, No. 5, pp. 1704-1723, September 2010.

[11] B. Klaus, and M. Walzl. "Stable many-to-many matchings with contracts", *Journal of Mathematical Economics*, Vol. 45, No. 7-8, 20 July 2009, pp. 422-434, .2009.

[12] P. Collet, "Functional and Non-Functional Contracts Support for Component-Oriented Programming", *Proceedings of the 1st OOPSLA Workshop on Language Mechanisms for Programming Software Components*, ACM, 2001.

[13] K. Behrends, "Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages". Ph. D. Dissertation. Michigan State University, East Lansing, MI, USA. Advisor(s) R. E. Stirewalt. 2003.

[14] R. Stirewalt, L Dillon, and K. Behrends, "Using Views to Specify a Synchronization Aspect for Object-Oriented Languages", *Software Engineering Workshop*, 2006. SEW '06. 30th Annual IEEE/NASA, pp.272-281, April 2006.

[15] V. Petrascu, D. Chiorean, and D. Petrascu, "ContractCML - A Contract Aware Component Modeling Language", Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC '08. *10th International Symposium on SYNASC*, pp.273-276, 26-29 Sept. 2008.

[16] OASIS Web Services Business Process Execution Language (WSBPEL), Standards Specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, 2007.

[17] L. Jellema and L. Dikmans, "Oracle SOA Suite 11g Handbook", Product Manual, http://www.oracle.com/technetwork/middleware/soasuite/documentation/index.html, 2008.

[18] J. Garcia and G. Goldszmidt, "Building SOA composite business services", Part 1 of Develop SOA composite applications to enable business services, Online Article, http://www.ibm.com/developerworks/webservices/library/ws-soa-composite/, 2006.

[19] C. Molina-Jimenez, S. Shrivastava, and J. Warne, "A method for specifying contract mediated interactions", 9th IEEE International *EDOC Enterprise Computing Conference*, pp. 106- 115, 19-23 Sept. 2005.

[20] M. Bartoletti, and R. Zunino, "Primitives for Contract-based Synchronization", *In Proceedings ICE 2010*, EPTCS , 2010.

[21] Y. Huang, L. Dillon and R. Stirewalt. "Contract-based synchronization of IP telecommunication services: a case study". *In Proceedings of the 5th International Conference on Communication System Software and Middleware (COMSWARE '11)*. ACM, New York, NY, USA. 2011.

[22] L. Lamport, "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers", Addison Wesley, 2002.

[23] K. Czarnecki and U. Eisenecker, " Generative Programming - Methods, Tools, and Applications", Addison-Wesley, June 2000.

[24] A. Kumari, "Synchronization and Quality of Service Specifications and Matching of Software Components, MS Thesis", *Department of Computer and Information Science, Indiana University Purdue University Indianapolis*, http://www.cs.iupui.edu/uniFrame/pubs-openaccess/Anjali_Thesis.pdf, 2004.

[25] R. Raje, P. Katuri, A. Kumari, and O. Tilak, "Multi-level Matching of Distributed Software Components", *Proceedings of the International Conference on Computer, Communication, and Instrumentation*, Mumbai, India, 2009.

ABOUT THE AUTHORS

Ketaki A.Pradhan was a former Masters student in the Computer and Information ScienceDepartment at IUPUI. During her education at IUPUI, she was actively involved in the UniFrame project. Ketaki has also successfully completed her thesis titled 'MDE-URDS- A Mobile Device Enabled Service Discovery System' under supervision of Prof.Rajeev R. Raje in the year 2010. Ketaki was a member of the distributed computing group in the CIS department, IUPUI.

Lahiru S. Gallege is a graduate student in the Department of Computer and Information Science at Indiana University-Purdue University Indianapolis (IUPUI) and currently reading for his PhD under the guidance of Prof. Rajeev R. Raje. Lahiru's research focuses on techniques for evaluating and validating trust in service selection in distributed systems. Lahiru is a member of the distributed computing group in the CIS department, IUPUI.

Rajeev R. Raje is a Professor in the Department of Computer and Information Science at IUPUI. His research interests are in the fields of distributed computing, software engineering, and service oriented systems. He has published more than hundred papers in various conferences and journals and has participated in many externally funded projects. He has also served on numerous program committees. Rajeev is a member of ACM and IEEE.