A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components

Khaled Khan
School of Computing and IT
University of Western Sydney
NSW 1797 Australia
E-mail: k.khan@uws.edu.au

Jun Han, Yuliang Zheng
School of Network Computing
Monash University
Frankston, Vic 3199 Australia
{jhan, yuliang}@monash.edu.au

Abstract

This paper presents a framework for constructing compositional security contracts (CsC) based on the security property exposed by the atomic component. The framework uses interface structure of components in order to determine the CsC of software components. An active interface provides the component a basis for reasoning and assessing a component's suitability to meet certain security requirements of a particular application. Based on the security information available from the component interface, an active interface can reason whether the candidate component meets the security requirements for an envisaged systemwide application. Any security mismatches or discrepancies between components can be identified by the participating components before an actual composition takes place. Exposing the security properties of software components can be the basis for a trust relationship among components, and the exposed security could affect the underlying security of the enclosing system.

1. Introduction

Software components are receiving a great deal of interests from both industries and academia as the component based software development paradigm promises maximum benefits of component reusability and distributed programming. A software component is independently developed and delivered as an autonomous unit and that can be composed with other components to become a part of a larger application [15]. Research efforts are currently being carried out mostly in defining component models and compositions with technologies such as OMG's CORBA, Sun's Enterprise JavaBeans (EJB), Microsoft COM, and the most recent development of dot Net.

By contrast, the major issue of concern regarding the security mismatches of software components has received less attention. The mismatches of security properties of components may have serious consequences if they are discovered after a composition took place. To avoid this late discovery of security properties, the component should be given the capability of knowing and reasoning the precise security requirements and assurances of the candidate components before an actual composition takes place. In a distributed dynamic system, when two components interact to achieve a certain functionality, both participating components need to know upfront each others' security properties as well as the impact of those properties on the enclosing system. When a component is discovered on the net dynamically by other components for an intention to use it at run-time, it might be unclear at which level of trust should be placed on the component because its security properties are often unknown. In current practice, what a user knows is the component's interface structure on how to connect the component with the user's system and how to get the functionality that the component offers. It is crucial to know whether the information sent over an open untrusted network is protected, and how this could be actually achieved by the component and so on. The degree of conformity between the required security properties of one component and the ensured security properties of another is the ultimate compositional security contracts (CsC) of the enclosing system participated by several components. The need for such a security characterisation model in both human and machine comprehensible terms has been long due as reported in [10], [11], [12].

The characterisation of component security includes a systematic understanding of security properties of components and their impact on the global composed system [6]. Exposing security properties is important when different developers produce different components in a system [3]. A component needs to be able to make a run-time test with other candidate components to find the possible security matches and mismatches. At present, no such security characterisation framework exists in the open literature.

One of the primary objectives of security characterisation is to build a trust relationship among software components. The attributes that most affect a trust relationship are the identity and origin of the components and the security properties that components offer to and require from other components. If the trust related attributes of a component are missing, or unknown then the component is not trustworthy at all. Unfortunately, these trust related properties are often neither expressed nor communicated to other interested parties [3]. Today, most systems do not broadcast their security properties and their origin identities, rather they only publicise whether the system is secure or not. Telling whether a component is secure or not can only lull other components or users into a false sense of security, which may not have any qualified basis. The existing approach does not tell what is the basis for such claims. Judging a system secure or not is somewhat a subjective matter depending on the use context, the security properties provided, the magnitude of the data sensitivity, and the mode of operations among others.

In component based system, a variety of common security threats can be posed to a component such as unauthorised disclosure of information passed between two components, unauthorised modification of the data, retransmitting the modified data by a third party, non-repudiation, and unauthorised access. In this paper we discuss the security issues such as confidentiality and authenticity. The paper, however, does not present any new security design or architecture, rather it proposes a model to characterise the existing security properties of software components. We propose what constitutes security properties of atomic software components, and how these can be exposed to others. We then present a framework for constructing CsC based on the security properties exposed by atomic components. The framework uses interface structure of components to determine the CsC of software components. The work reported in this paper builds on and relates to our earlier efforts reported in [8] and [9].

In the next section, we outline our approach to characterise the component security using component interface structure. In section 3, we present a framework for active interface along with the structure of compositional security contracts (CsC). In section 4, the proposed framework is applied to an example to demonstrate its applicability. Section 5 outlines the possible use of the framework and the major limitations of its current state. We close with a conclusion in section 6.

2. Component interface and the approach

The interface of a software component makes all compositional structural elements available to other interested components. The availability of interface description is used for the component's interaction with the outside world. A structural composition consists of components (blackbox entities that export and import functionality), architectural style such as formalisation of component interface and composition rules, and glue code [4]. Current frameworks for software component models such as EJB, CORBA, COM, and dot Net are limited for the specification of structural interface definition and matching of interfaces. Interface description languages (IDLs) deal basically with the syntactic structure of the interface such as the forms and types of the interface elements like attributes, operations and events. These meta-data are primarily static in nature. Interface provided by the existing component technologies such as CORBA, EJB, and COM comprises attributes, operations and events. Recently proposed richer interface specification addresses the issues of software component interface signature (syntax), interface configurations (structure), interface behaviour (semantics), and interaction protocol (constraints) [1]. All these are aimed mainly at components' functionality.

To get a clear understanding of a component's security properties, this paper extends the model of interface structure proposed in [1] a bit further to make it dynamic or 'live' in a sense that the interface will have certain reasoning capability. An active interface not only contains the operations and attributes that the component provides to serve a functionality, but it also embodies security properties associated with a particular operation or functionality. In our approach, the essence of active interface is that a component knows its security properties, and can communicate this knowledge to other interested components. An active interface involves the ability of the component to reason about and to deduce the compositional security properties offered by other components.

It would be unrealistic to store an exhaustive list of pre defined compositional security properties for all possible use context in the interface of a component. By contrast, our approach provides an incremental security specification or introspection based on the security properties that a component exposes. This approach is based on the notion of 'light-weight' security specification advocated in [2]. A 'light weight' characterisation exposes some externally observable properties to other components. The active interface of a component can capture the security properties of another component or an existing composition, and compute the CsC of the dynamic system configuration and re-configuration.

The principle objective is to generate computational reflection to let components identify and capture the various security properties of other components with which they cooperate. In such a setting, components not only read the meta-description of others' security properties but also deduce the compositional impact of those properties on the ultimate composed application. This active interface supports a two-level negotiation model for component composition as proposed in [5]. In the first level, a component negotiates for a possible compositional contract with other interested candidate components. If it is successful, the negotiation results would be used to configure and re-configure the components dynamically. This augments a pragmatic approach for implementing a self-configuring and composable component framework, relying partly on an active interface structure.

3. Framework for an active interface

In component based systems, a distributed application is composed of a set of individual software components each having a local memory and its own executable code. A component is autonomous as it uses its own data and files, and usually it does not preserve state. A component that broadcasts an event to receive a service from other is called *focal component* [1]. The components that respond to the event are usually called *candidate components* residing on different remote locations. The basic entities that perform operations on components in a distributed system are processes. An event denotes an execution of an operation on components, and is attributed to the process that performs the operation [13]. In an event-based component interaction, a component generates events, other candidate components may choose to respond to the event. Based on the agreed contract for an event a dynamic composition is established between the focal component and the candidate component. We also sometimes refer a focal component as client component, and the candidate components as server components. A focal component receives certain services from candidate components. A focal component can also play the role of a candidate component when it serves functionality to others.

An active interface comprises a *component identity*, a static *interface signature*, a static *security knowledge base* (read-only) of the component, and a *CsC* (read-write) which is dynamic based on the security information available from the security knowledge base and the security information available from other components. CsC properties are dynamic in a sense that it can only be derived from the exposed security properties of the focal component and the candidate components related to a particular functionality. The following is a skeleton of such an interface.

```
COMPONENT UID{
  INTERFACE SIGNATURE{. . .}
      SECURITY {
         REQUIRED {. . .}
         ENSURED  {. . .}
         CsC {. . .}
      }
  }
```

The structure of the active interface is diagrammatically shown in Figure 1.

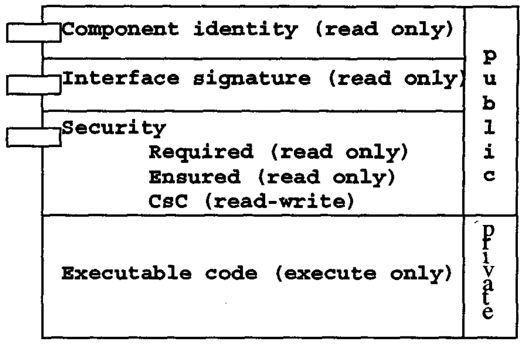
Component identity (read only)
Interface signature (read only)
Security
Required (read only)
Ensured (read only)
CsC (read-write)
Executable code (execute only)
public
private

Figure 1. A skeleton of an active interface

3.1 Component identity

The identity of the component not only includes a **unique identity (uid)**, but it also shows its origin from where it was originated, its current residing address, its current owner, the developer of the component, and the certification authority which approved all information that are available from the active interface. The identity is unique over its lifetime and is provided by a certifying authority. The current residing address is the URL where it is located and the owner of the component is the owner of the URL[8]. The following is a structure of the component identity template.

```
identity(uid, origin_URL, developer_URL,
                                certificate)
```

It can be further decomposed with more identity related information such as details about the certifying authority, component sealing template, validity period and so on.

3.2 Interface signature

An interface signature consists of operations and attributes. These properties are static in a sense that these are 'read only' properties. No components can make any modification to this. This interface is intended to make a structural match before two components are composed. Significant work on the structure of interface can be found in [1] and a comprehensive treatment on the structural matching of software components is available from [14].

3.3 Security

The goal of the *security knowledge base* is to specify the security properties of atomic components. The publishable security-related characteristics of any atomic component can be categorised as *required* security properties and *ensured* security properties. A required security property is a precondition in a sense that other interested parties should satisfy this during a composition to get ensured security services. An ensured security property is a post-condition in a sense that it is the responsibility of the component to maintain the committed security assurances during the composition.

In order to express security properties of a component, first we need to model what a security property does. Any ensured or required security property of individual component can be characterised with three basic elements: (i) ***operations*** performed by the components to enforce security properties, (ii) ***security attributes*** used to do the operation in (i), and (iii) ***data*** used or manipulated in a compositional contract. Using these elements we can formulate a simple structure that can capture the security requirements and assurances of individual software components. Thus, the security properties of individual component can be characterised with a predicate-like structure such as

$$f(O_i,\ K_j,\ D_k)$$

Where

- f is the name of the *security function* formed with three associated arguments as defined in [9],
- O is the security related *operation* performed by the component $_i$ in a compositional contract, subscript $_i$ is the identity (uid) of the component,
- K is a set of *security attributes* used by the component, and the subscript $_j$ contains additional information about the K such as key type, owner of the key and so on. Plus (+) or minus (-) signs as postfix of $_j$ denote a public key or a private key of $_j$ respectively, and
- D is an arbitrary set of data or *information* that are affected by the operation O. The subscript $_k$ contains additional information attached with D such as digital signature used or not, and so on.

An example of such structure can be

$$\texttt{protect\_in\_data}(\texttt{encrypt}_Q, \texttt{key}_{P+},\ \texttt{'amount'})$$

In this example, the security property is declared in a component's interface and visible externally. This is a public and read-only property in a sense that other components or human user can read this readily available from the component interface. It states that data '**amount**' is to be *encrypted* by the component Q with the *public key* of component P. The name of the entire security function is **protect_in_data.** More on this structure can be found in [9]. The security characterisation must be based on the actual security functions that a component employs to accomplish a functionality. The accomplishment of a functionality could be any services such as receiving functionality from another component or offering services to other components. The exposed security properties must be mapped with the functionality that it supports and based on the security functions implemented in the component.

3.4 Compositional security contracts (CsC)

A CsC is based on the degree of conformity between the required security properties of a component and the ensured security properties of another component. The resulting security property of the composition participated by two components is a new security property called CsC. The derived CsC is a security effect generated from the combined security properties of the participating components related to a particular functionality [7]. A CsC defines rules for composing security properties on the basis of conformance between the required security property of one component and the ensured security property of the another. With the security characterisation structure of individual components, a CsC between two components such as a and b can be modelled as

$$C_{a,b} = (E_b \Rightarrow R_a) \wedge (E_a \Rightarrow R_b)$$

Examples of E and R are:

$$R_a = f_1(\texttt{verify}, \texttt{password}_b,\ \texttt{file1})$$
$$E_a = f_2(\texttt{encrypt}, \texttt{key}_{b+}, \texttt{file1}_{a.digisign})$$

where

- C is a *compositional security contract* between two components subscripted with the identities of the participating software components separated by a comma in the compositional contracts such as $C_{a,b}$, $C_{c,d}$, and so on,
- E and R are the *ensured* and *required* security properties of the participating components in a composition contract respectively. In an expression such as $(E_b \Rightarrow R_a)$, required property R will always be on the right-hand side in an expression
- $x \Rightarrow y$ denotes implication such as x implies y. The evaluation of each required and ensured pair would result a Boolean *true* or *false* value,
- The *required* or *ensured* security property of a compositional contract can be referred from an existing CsC such as $C_{a,b}.R_b$ or $C_{a,b}.E_a$ respectively, where
  - the identity of the CsC and its associated security properties are separated by a *dot* when referred by another compositional contract or component
  - prefix $C_{a,b}$ in the argument $C_{a,b}.R_b$ denotes the *identity* of an existing CsC in where a is the focal component and b is the candidate component. Note that a focal component always follows a candidate component. The ordering of the components identities indicates the role of the components in a

composition such as whether a component is a focal or candidate component.

  - a postfix **R** in a CsC denotes the *required* security property of the component identified with the corresponding subscript and taking part in the CsC, and
  - a postfix **E** in a CsC denotes *ensured* security property of the component identified with the corresponding subscript and taking part in the CsC.
- The operators $\wedge$ and $\vee$ denote Boolean "and" and "or" respectively.

3.5 Executable code

A complete structure of an active interface is outlined as follows.

```
<begin COMPONENT> { <UID> }
      <begin INTERFACE SIGNATURE>
            <operation> {
                  <argument1,
                        , . . . ,
                  argumentn> }
      <end INTERFACE SIGNATURE>

      <begin SECURITY>
            <begin REQUIRED>
               <security_function1>{
                  <security_argument1,
                        , . . . ,
                   security_argumentn>}
                        , . . . ,
               <security_functionn>{
                  <security_argument1,
                        , . . . ,
                   security_argumentn>}
            <end REQUIRED>

            <begin ENSURED>
               <security_function1> {
                  <security_argument1,
                        , . . . ,
                   security_argumentn>}
                        , . . . ,
               <security_functionn> {
                  <security_argument1,
                        , . . . ,
                   security_argumentn>}
            <end ENSURED>

      <end SECURITY>
      /*  The  following  is  the
      structure   of   the   'live'
      executable  section  of  the
      interface.

      <begin CsC>
           RUID = get(<operation>,<REQUIRED>,
                     <UID>);
           EUID = get(<operation>, <ENSURED>,
                     <UID>);
           QualFrom=conform(<EUID>,<REQUIRED>);
           QualTo =conform(<RUID>,<ENSURED>);
           CsC=conform(<QualTo>,<QualFrom>);
           display = out(<CsC>);
      <end CsC> <end COMPONENT>
```

A binary executable code calculates and generates CsC. The basic algorithms of such executable code are listed in the above structure between **<begin CsC>** and **<end CsC>**. A **get** function *reads* the security properties from the interface of a candidate component and *stores* it in R_{UID}. The subscript **UID** is the identity of the candidate component. Similarly, the ensured security properties of the candidate component are read and stored in E_{UID}. The variable $\mathbf{Qual}_{from}$ stores the conformity result between the required property of the *focal component* (**REQUIRED**) and the ensured security property of the *candidate component* (E_{UID}). $\mathbf{Qual}_{To}$ stores the conformity between the required property of the *candidate component* (R_{UID}) and the ensured property of the *focal component* (**ENSURED**). A built-in function called **conform** generates these conformity results. If a non-conformance between the required and ensured properties is identified, it concludes a security mismatch. A Boolean true and false value is to be implicitly assigned to each required and ensured pair after the evaluation as shown in the following example.

$$C_{X,Y} = ((E_X \Rightarrow R_Y)=\mathrm{TRUE}) \wedge ((E_Y \Rightarrow R_X)=\mathrm{TRUE})$$

The above CsC simply shows that a composition between two components identified as **X** and **Y** has complete compliance. Required property of **Y** is ensured by the property of **X**, and **X**'s required property is satisfied by the ensured property of **Y**. The resulting CsC is stored in the interface structure of the focal component. When a component is composed with another component, the derived CsC is automatically attached with the interface. The entire CsC is to be added to the CsC slot in the interface and remains in the interface as long as the composition is valid. It should be noted that a partially valid CsC can be accepted by a component, in such case, the actual derived CsC needs to be stored in the interface as well. However, a partially satisfied CsC might have a negative security impact on the entire global system. Even a complete mismatch CsC can be accepted by the participating components if they decide so; in such case, the security risks would be much higher for the composition.

4. E-health care system: an example

In this section, we will describe a fictious distributed system topology as a vehicle to discuss how our proposed active interface would work in a distributed environment. The applicability of the proposed framework is examined with this example system.

Consider the information system of an e-health care system in a country where patients' clinical information is considered confidential. All information passed among the stakeholders such as GPs, specialists, patients and pharmacist must be confidential. Assume a software system identified as **G** running on a machine at a general practitioner's (GP) office is trying to connect with a trusted software component **S** chosen from a number of similar systems running at various specialists' office. **G** would provide patient's diagnosis reports to a specialist's system **S** to get a specialist prescription. After receiving a prescription from the component **S**, **G** sends this prescription to a component **P** residing on a pharmacist's system. There are many such components developed and managed by various developers available from various distributed sources delivering the same functionality that **G** wants. However, the security properties of all these components vary significantly from one another. Component **G** is not only interested in specific functionality of the components but also wants to know the security properties that are provided by those components. All information passed between **G** and the specialist system **S** is considered confidential. **G** requires following security properties from a specialist component **S**.

i. Authenticity of the specialist prescription
ii. Confidentiality of information exchanged with **G**.

The issue of access control is not included in this paper.

4.1 A binary compositional security contract

The main security goals of the stated scenario are confidentiality or secrecy of information, and the authenticity of the components regarding their origins and identities. The aim of confidentiality is to ensure that the data is not accessed by an unauthorised entity. The aim of authenticity is to make sure that the identity and the origin of the component are correct. These security policies of **G** can be transformed into our active interface framework as

```
COMPONENT G {
 INTERFACE SIGNATURE {, . . . ,}
  SECURITY {
   REQUIRED {
    RG=protect_in_data(encryptS,keyS-,
      'prescription'S.digisign) }
   ENSURED {
    EG=protect_out_data(encryptG,keyS+,
'diagnosis') }
CsC{ NULL } } }
```

The security properties attached with the functionality **receive_prescription(arg$_1$,arg$_2$,…,arg$_n$)** state that **G** will provide a specialist component with a **diagnosis** report of a patient. The diagnosis would be *encrypted* by **G** (**encrypt$_G$**) with the *public key* (**key$_{S+}$**) of the specialist system **S**. **S** can *decrypt* the message using its *private key*. In return, **G** expects from **S** that the **prescription** sent by **S** must be *digitally signed* ('**prescription'$_{S.digisign}$**) and *encrypted* by the component **S** (**encrypt$_S$**) with the *private key* of **S** (**key$_{S-}$**).

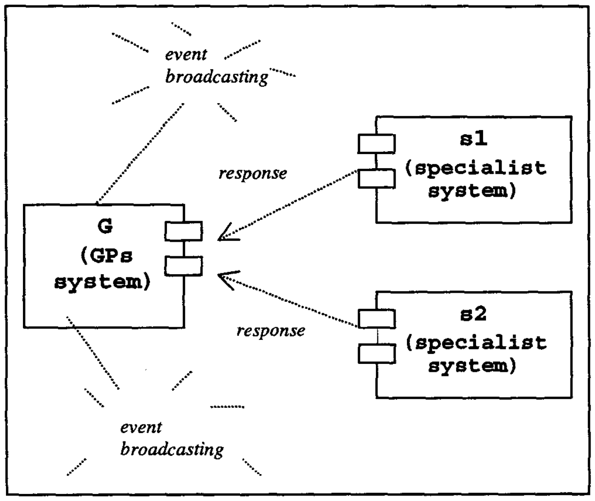

Figure 2. Events broadcasted by G

Based on these, **G** is looking for a component **S** that will satisfy the following CsC.

$$C_{G,S} = ((E_G \Rightarrow R_S)=\mathrm{TRUE}) \wedge ((E_S \Rightarrow R_G)=\mathrm{TRUE})$$

Assume **G** broadcasts an event to receive responses from other interested components, which could offer the functionality that **G** needs. In return **G** receives responses from components **s1** and **s2**. All these components offer the same functionality. These components are running on different machines for different specialists as depicted in Figure 2.

First **G** makes a query to a specialist's component **s1**. Component **s1**'s interface exposes the following security properties, which are stored in its *static knowledge base*.

```
COMPONENT s1 {
      INTERFACE SIGNATURE {
      make_comments(arg1,arg2,...,argn) }
   SECURITY {
    REQUIRED {
      Rs1=protect_in_data(encryptG,keys1+
      , 'diagnosis')     }
    ENSURED {
      Es1=protect_out_data(encrypts1,
      keyG+, 'prescription') }
   CsC { NULL } } }
```

The security properties attached with the functionality `make_comments(arg1,arg2,...,argn)}` state that **s1** will provide a specialist **prescription** to a requesting component. Component **s1** expects from **G** that the **diagnosis** report sent by **G** must be *encrypted* by the component **G(encrypt$_G$)** with the *public key* of **s1** (**key$_{s1+}$**). In return, the prescription would be *encrypted* by **s1** (**encrypt$_{s1}$**) with the *public key* of **G** (**key$_{G+}$**), but the *prescription* data would not be *digitally signed* by **s1**. Based on these security properties, the following algorithms of the **G**'s active interface now execute and generate a CsC.

```
COMPONENT G {
    ,...,
CsC {
  Rs1=read(make_comments,required, s1);
  Es1=read(make_comments, ensured, s1);
  Qual_to = conform(Rs1, EG);
  Qual_from = conform(RG, Es1);
  CsC = derive (Qual_to, Qual_from);
  display = out(CsC);
} , . . . ,}
```

The execution results the following CsC, which is not quite consistent with the requirement of the entire composition between **G** and **s1**.

$$C_{G,S} = ((E_G \Rightarrow R_{s1}) = \mathrm{TRUE}) \wedge ((E_{s1} \Rightarrow R_G) = \mathrm{FALSE})$$

We can see from the above CsC that the required security property of the component **G** cannot be satisfied with the ensured security property provided by **s1**. Thus the CsC will be partially compliance with the desired composition. The CsC failed because component **s1** can not provide the prescription with a *digital signature*, which could be verified by G., The authenticity of the prescription is one of the requirements that **G** expects from the component. **G** now decides to make a similar security test with another component called **s2**. Component **s2** also provides the same functionality that **G** is looking for, but the security properties of **s2** needs to be verified by **G** before it makes a composition with **s2**. The following is the security information that **G's** interface reads from **s2**'s interface

```
COMPONENT s2 {
  INTERFACE SIGNATURE {
      make_comments(arg1,arg2,...,argn)}
  SECURITY {
    REQUIRED {
      Rs2=protect_in_data(encryptG,keys2+
      , 'diagnosis')     }
    ENSURED {
      Es2=protect_out_data(encrypts2,
      keys2-, 'prescription's2.digisign) }
   CsC { NULL } } }
```

The above properties state that **s2** will provide a specialist **prescription** to a requesting component. Component **s2** expects from **G** that the **diagnosis** report sent by **G** must be *encrypted* by the component **G** (**encrypt$_G$**) with the *public key* of **s2** (**key$_{s2+}$**). In return, the prescription would be *digitally signed* and *encrypted* by **s2** (**encrypt$_{s2}$**) with the *private key* of **s2** (**key$_{s2-}$**). **G** can decrypt the message using the public key of **s2** to verify the signature. Based on these security properties and the following algorithms, the interface of **G** now computes a CsC such as

$$C_{G,S} = ((E_G \Rightarrow R_{s2}) = \mathrm{TRUE}) \wedge ((E_{s2} \Rightarrow R_G) = \mathrm{TRUE})$$

The generated CsC is now consistent with the requirement of the entire composition between **G** and **s2**. G finally makes a composition with **s2**. The resulting system is shown in Figure 3. The CsC is stored in the static knowledge base of **G's** interface for future reference.

$$C_{G,s2} = ((E_G \Rightarrow R_{s2}) = \mathrm{TRUE}) \wedge ((R_{s2} \Rightarrow R_G) = \mathrm{TRUE})$$

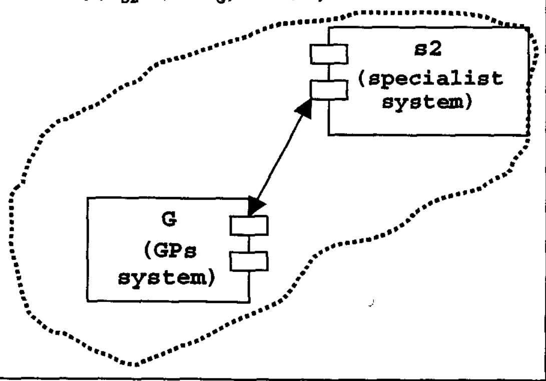

Figure 3. Composition between G and S

4.2 A multiple compositional security contract

We extend the same scenario a bit further to examine how a ternary or a composition with multiple components can be supported with our framework.

After **G** has composed itself with **s2**, it further looks for a third component **P** that would provide the price of the medicine based on the prescription produced by the component **s2** to **G**. All components that responded to the event generated by **G** are identified as p_1, p_2, and p_3. **G** now makes a test with **p1** to verify whether **p1** delivers the security property that **G** expects, or whether **p1**'s required property can be satisfied by the component **G**. **G**'s security properties are

```
COMPONENT G {
 INTERFACE SIGNATURE {
 get_price(arg_1,arg_2,. . .,arg_n) }
  SECURITY {
    REQUIRED {
   R_G=protect_in_data(encrypt_P,key_P-,
     'Price'_P.digisign)          }
    ENSURED {
 E_G=protect_out_data(encrypt_G,key_P+,C_G,s2.E_s2)
      }
   CsC { NULL } } }
```

G wants a price of the medicine from a component **P** for the prescription provided by **s2**. **G** will provide **P** the prescription of the specialist that was received from the component **s2** specified as $C_{G,s2}.E_{s2}$. Note that the previous CsC made between **G** and **s2** is stored in the **G**'s interface. **G** will also attach the digital signature of **s2** to component **P** to ensure that the prescription was authenticated by a specialist. However, in return, the **price** data must be *digitally signed* (**'Price'**$_{P.digisign}$), and *encrypted* ($\mathbf{encrypt}_P$) by the component **P**. An acceptable CsC from this composition can be worked out as

$$C_{G,P} = ((E_G \Rightarrow R_P)=\mathrm{TRUE}) \wedge ((E_P \Rightarrow R_G)=\mathrm{TRUE})$$

On the other hand, **p1** exposes its security properties to **G** as

```
COMPONENT p1 {
 INTERFACE SIGNATURE {
 get_price(arg_1,arg_2,. . .,arg_n)}
  SECURITY {
    REQUIRED {
   R_p1=protect_in_data(encrypt_G,key_G-,
      (C_G,S.E_S)_G.digisign)        }
    ENSURED {
      E_p1=protect_out_data(encrypt_p1,key_P-,
      'Price'_p1.digisign)   }
  CsC { NULL } } }
```

Component p1, in fact, does not require that the specialist component must be identified as **s2** with which **G** has composed. What actually it means that the signature must be from a specialist **S**. The verification of the signature by **p1** would reveal the actual identity and the validity of **S**. In this case, the identity of the specialist **S** would be obviously **s2**.

The algorithms of **G**'s interface now executes and generates a ternary compositional contract which is not quite consistent with the requirement of the entire composition between **G** and **p1** based on their security requirements as shown below.

$$C_{G,p1} = ((E_G \Rightarrow R_{p1})=\mathrm{FALSE}) \wedge ((E_{p1} \Rightarrow R_G)=\mathrm{TRUE})$$

The above CsC shows that the required property of the component **G** is satisfied by the ensured security property of **p1**, but the required property of **p1** cannot be satisfied by **G** because **p1** requires a digital signature of **G** in addition to a digital signature of **S**, before it services to component **G**. Component **G** does not have any such digital signature for itself. **G** 's security test with **p1** fails due to non-compliance security properties provided by **G**. **G** now decides to make similar test with another component called **p3**. Component **p3** also provides the same functionality that **G** is looking for, but the security properties of **p3** needs to be verified by **G**'s interface before it makes a composition with **p3**. The following is the security information that **G** reads from **p3**'s interface .

```
COMPONENT p3 {
 INTERFACE SIGNATURE {
 get_price(arg_1,arg_2,...,arg_n)   }
  SECURITY {
    REQUIRED {
   R_p3=protect_in_data(encrypt_G,key_p3+,
     C_G,S.E_S)          }
    ENSURED {
      E_p3=protect_out_data(encrypt_p3,key_p3-,
      'Price'_p3.digisign)   }
   CsC { NULL } } }
```

The interface of the component **G** executes and generates the following CsC, which is now consistent with the requirement of the entire composition between **G** and **p3**.

$$C_{G,p3} = ((E_G \Rightarrow R_{p3})=\mathrm{TRUE}) \wedge ((E_{p3} \Rightarrow R_G)=\mathrm{TRUE})$$

G composes itself with **p3** by using the interface signature of **p3**. In fact $C_{G,p3}$ involves three components as shown below.

$$\begin{aligned} C_{G,p3} &= (((C_{G,s2}.E_{s2}) \wedge (E_G)) \Rightarrow R_{p3}) \wedge (E_{p3} \Rightarrow R_G)) \\ &= ((((C_{G,s2}.E_{s2} \Rightarrow C_{G,s2}.R_G) \wedge \\ &\quad (C_{G,s2}.E_G \Rightarrow C_{G,s2}.R_{s2})) \wedge E_G \Rightarrow R_{p3}) \wedge \\ &\quad (E_{p3} \Rightarrow R_G)) \end{aligned}$$

The entire resulting system composed of components **G, s2**, and **p3** is shown in Figure 4. There are two CsCs in this system, one is between **G** and **s2**, and the other one is between **p3** and **s2, G** together. It should be noted that if a composition is broken after a functionality is complete then the associated CsC would not be available to any participating components. The obsolete CsC might be stored in a log for a future audit.

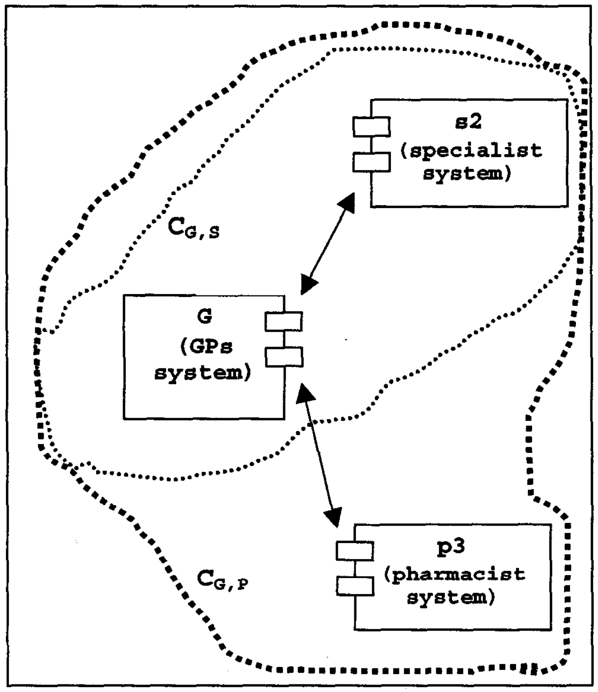

Figure 4. A CsC based on multiple components

Based on our framework, we could build a complete CsC for a system composed of several autonomous components. However, we believe that the framework may require some adjustments and modifications to accommodate more complex security properties for more complicated compositions.

5. Use of the framework and limitations

We are currently working to examine the framework in a real application scenario using one of the component models available from the commercial market. IDL of CORBA can be extended with this framework so that security properties of a component can be specified in its interface and stored in the interface repository. Such an extended CORBA's dynamic interface support could be used by client components to retrieve the security-related information of the candidate components. A client component even might run a security composability test with the candidate components. Similarly, the framework could be codified with the JavaBean's introspection mechanism (BeanInfo) and Java's reflection capability for JavaBeans.

We recognise that the proposed framework has some limitations. Firstly, in this framework we made a number of assumptions. We assume certain low level security properties are already in place by the supporting infrastructure such as protocols, middleware and the operating systems. Secondly, complex compositions and associated security features have not been discussed. Finally, the existence of a global certifying authority is assumed. The certification authority approves the component with its valid interface information including the security properties that a component exposes. How such a certification authority approves a component in respect to its claimed security properties is beyond the scope of this paper. More on the certification issue can be found in [7].

6. Conclusion

This paper has demonstrated that an active interface can provide the basis for reasoning and assessing a component's suitability to meet certain security requirements of a particular application. Active interface defines what should be expected from a component, and what the component expects from the outside world. An active interface not only exposes its own security properties but it may also show what it requires from a third component. It is almost undeniable that a software component should be clear enough about what the security across the dynamic composition with other components is, what security provisions each component requires and ensures, and what would be the ultimate security behaviour of the entire composed system. From a security point of view, it is unrealistic to tell the component users or the system composers whether a software component is secure or not, rather it is much useful to expose what security properties are implemented. In a distributed environment, it would not be realistic to expect that all components would provide same degree of security to others. The proposed framework lets the human users and software components judge the trustworthy of a component by reasoning the security properties that it exposes.

One of the secondary benefits of our framework is to separate the interface code from the application code of the component. This framework enforces a clear separation of concerns between the interface introspection and the application of the functionality.

We conclude with our belief that a security characterisation mechanism providing a full disclosure of security properties in both human and machine comprehensible terms could build a confidence and trust on a viable software component market.

References

[1] Han, J.,"A Comprehensive Interface Definition Framework for Software Components", IEEE Proceedings of the 1998 Asia-Pacific Software Engineering Conference, Taipei, December 1998, pp. 110-117.

[2] Perry, D., "Software Evolution and 'light' Semantics", IEEE Proc. Of the 21th International Conference on Software Engineering, Los Angeles, USA, May 1999, pp 587-550.

[3] Viega, J., Kohno, T., Potter, B., "Trust and Mistrust in Secure Applications", Communications of the ACM, Feb. 2001, Vol. 44, No. 2., pp 31-36.

[4] Keller, R., Lague, B., Schauer, R., "International Workshop on Large-Scale Software Composition", ACM Software Engineering Notes Vol. 24, No. 1, January 1999, pp. 49-54.

[5] Ben-Shahul, I., Gidron, Y., Holder, O., "A Negotiation Model for Dynamic Composition of Distributed Applications", Proc. International Workshop on Large-Scale Software Composition, Vienna, August 28 1998, pp. 820-825. http://www.iro.umontreal.ca/labs/gelo/iw-lssc98

[6] Beugnard, A., et al., "Making Components Contract Aware", IEEE Computer, July 1999, pp. 38-46.

[7] Voas, J., "Certifying Software for High Assurance Environments", IEEE Software, July/August 1999, pp. 48-54.

[8] Khan, K., Han, J., Zheng, Y., "Security Characterisation of Software Components and Their Composition", IEEE Proceedings 36th Int'l Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Asia 2000), Oct. 30- Nov. 4 2000, Xi'an, China, pp. 240-249.

[9] Khan, K., Han, J., Zheng, Y., "Characterising User Data Protection of Software Components", IEEE Proc. Australian Software Engineering Conference 2000, Canberra, April 28-29 2000, pp. 3-12.

[10] Thomson, C.: Workshop Reports. 1998 Workshop on Compositional Software Architectures, organised and sponsored by OMG, DARPA, MCC, OSC, Monterery, California.
http://www.objs.com/workshops/ws9801/report.html

[11] Szyperski, C.: Component Software -Beyond Object-Oriented Programming, Addision-Wesley, 1998.

[12] Michener, J., Acar, T., "Managing System and Active-Content Integrity", IEEE Computer, 33-7, July 2000, pp. 108-110.

[13] Ko, C., Ruschitzka, M., Levitt, K., "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach", IEEE Proc. Symposium on Security and Privacy, 1997, pp. 175-187.

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

[15] Repenning, A., et al., "Using Components for Rapid Distributed Software Development", IEEE Software, March/April 2001, pp. 38-46.