Electronic Notes in Theoretical Computer Science 54 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume54.html 9 pages

Security Issues in Component-based Design

A. Bracciali, A. Brogi, G. Ferrari, E. Tuosto [1,2]

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

Abstract

We propose a behavioural extension of the concept of interface of components. We aim to uniformly reason about correctness properties of both closed and open component-based systems. The characterizing feature of our approach is that we perform a local analysis over finite fragments of interactions naturally modeling mobility and coordination aspects. We present a semi-automatic technique that reduces the verification of security properties of protocols to the verification of correctness in component-based systems.

1 Introduction

Networked heterogeneous applications are nowadays becoming the primary part of modern software environments. These applications (e.g., web-browsers for cellular phones) require stationary servers and mobile devices to cooperate while the application is running, and the dynamic integration of network services is a crucial aspect. In this scenario, programming techniques adopt the abstraction of components to encapsulate services inside black boxes. Services are described by means of interface description languages. Recent developments have focussed on extending these languages in order to specify the interaction protocols of components. Service integration is then obtained by means of composition or coordination languages that describe the way in which components are glued together. Moreover, networked applications strongly emphasize the problem of security in software composition. Components may indeed be malicious and may have been designed to steal and/or corrupt information on the hosts where they will run.

In this paper we extend the formal framework presented in [6] to describe and reason on systems consisting of autonomous, interacting components. In particular, we will show how such framework can be suitably applied to analyse security protocols, viewed as behavioural features of components.

[1] This work has been supported by MURST National Project TOSCA.
[2] Email: {braccia, brogi, giangi, etuosto}@di.unipi.it

Our starting point is the notion of *interaction pattern*, which is an abstraction of the interactive behaviour of a software component, expressed with a subset of the π-calculus [10]. Differently from most existing approaches based on process algebras, we restrict ourselves to considering descriptions of finite behaviour of components. Indeed, we argue that trying to describe all the aspects of component-based programming in one shot unavoidably leads to complex formulations of low practical usability. On the other hand, interaction among components often follows recurrent finite patterns. This simplification leads to a formal framework suitable for locally and uniformly reasoning on properties of both static and dynamic compositions.

Following [6], we therefore introduce a simple interface description language to specify the interaction pattern of a component. A set of interaction patterns forms a context which may evolve either because of interactions occurring within the context, or because new components dynamically join the context.

The main contribution of this paper is to show that it is possible to treat uniformly both composition and security issues of component-based programming within the same formal framework. We propose a novel, semi-automatic approach to the verification of security protocols by extending the notion of interaction pattern. Our approach can be summarized as follows:

- We extend interaction patterns with cryptographic primitives, in the style of the spi-calculus of Abadi and Gordon [1].
- We express security properties of protocols by means of a revised version of the idea of *magic instance* of a protocol, originally introduced in [1].
- We develop a proof technique to verify the correctness of security protocols. The proof technique consists of two main steps:
  (i) The properties to be verified are first expressed in terms of the integrity and secrecy of certain critical data exchanged among the protocol principals. This corresponds to defining a *magic instance* of the protocol which fixes the values that such data are expected to assume if the desired property holds.
  (ii) Protocol correctness is then verified by checking the non-existence of an intruder capable of cheating the protocol principals with communications that cannot be accepted by the principals of the magic instance of the protocol.

2 Interaction patterns

Interaction patterns describe both the behaviour of components in terms of the communications they can perform, and the communication channels the components initially offer to the environment. We already remarked that interaction patterns can express only finite behaviours. The abstraction of the composition environment is called *context*, i.e. a multiset of connected

components. If the components are completely connected, i.e. they do not have open names, the context is *closed*, otherwise it is *open*. The full language of interaction patterns is derived from π-calculus, the main difference being the absence of recursion and of an explicit restriction operator (actually, all local names are required to be disjoint). By communicating channel names, the connection topology of a context may dynamically change. In this paper, we will present many of the features of the framework only informally. Formal semantics, composition operations, and the formal definition of other aspects may be found in [6].

Suppose that we are specifying a distributed system, with a simple server, which provides a reference (a name) to a given resource. Every time the server interacts with a possible client, it follows the same protocol: it receives a request on a communication channel (`_c` say, channel names are usually preceded by underscore), and then sends an answer (the name `_z`) over the same channel, unless it receives some sort of interruption from the same channel. Such a behaviour can be expressed by the following interaction pattern:

```
(_c) [ in(_c,query).  ( out(_c,_z).0 + in(_c, break).0 ) ]
```

where `in(_c,query)` (resp., `out(_c,_z)`) is a synchronous input (resp., output) action on the channel `_c`, where `query` and `_z` are the transmitted data, and `(_c)` denotes the channel that can be used to connect the server to other components.

Definition 2.1 An *interaction pattern* consists of a term $(\bar{\mathtt{X}})[E]$, where $\bar{\mathtt{X}}$ is the set of *open names* of E, a subset of the names occurring in the expression E. The *behavioural expression* E is defined by the following grammar:

$$E ::= 0 \mid \alpha.E \mid E||E \mid E+E \qquad D ::= C \mid ...$$

$$\alpha ::= \mathtt{in}(C,D) \mid \mathtt{out}(C,D) \mid \tau \qquad C ::= \mathtt{net} \mid ...$$

A component may send and receive data (D) over channels (C) by performing synchronous communication actions (`in` and `out`). As in π-calculus, channel names can be exchanged in communications. The signature D of data can be arbitrarily complex by allowing, for example, structures for method invocations as well as encryption mechanisms (see Section 3). Input actions on a complex term d synchronize with outputs that provide a term compatible via pattern matching with d. The set of names C contains the distinguished name `net`, which is known by every component in the system. The name `net` has been introduced to model all the untrusted communications that may occur over the net and to which every component can participate. The silent action τ denotes an internal computation a component may perform independently of the external environment, and it can be used to explicitly describe local choices. As usual, processes are built through the prefix, parallel and choice composition operators of π-calculus.

Components are then composed in a system by linking together some of their channels. Syntactically, this consists of identifying some of the open

names of those interaction patterns that belong to the same context. Once an open name has been connected, it is not anymore available to the environment, and it is removed from the set of open names. Besides the static incremental construction of a system, the same connecting mechanisms models also the dynamic interplay of components that intend to join a running environment, like, e.g., in the case of code mobility. The following example shows the composition of components inside a context.

Example 2.2 A client, expressing a behaviour compatible with the behaviour of the server previously illustrated, may send a request and then either accept an answer (`y` is a name, used here in place of a variable, in the π style) or autonomously decide to send a break, e.g. after a certain time out:

`(_s) [ out(_s,query).  ( in(_s, _y).0 + τ.out(_s, break).0 ) ]`

This illustrates the use of τ to model internal (local) choices. The two interaction patterns can be combined together by means of the substitution [`_x/_c,_x/_s`], obtaining the following closed context:

```
{ () [ out(_x,query).  ( in(_x, _y).0 + τ.out(_x, break).0 ) ],
  () [ in(_x,query).  ( out(_x,_z).0 + in(_x, break).0 ) ] }
```

The interaction patterns in the context, since their channels have been connected, can communicate together. After the first communication, the context evolves into:

```
{ () [ ( in(_x,_y).0 + τ.out(_x, break).0 ) ],
  () [ ( out(_x,_z).0 + in(_x, break).0 ) ]}
```

Depending on the client's choice, the two components can interact in two ways: Either the reference `_z` or `break` can be communicated next. Note how, as expected, in both cases the two components can completely execute their tasks, being in a sense compatible and correctly composed components.

The above framework permits us to describe finite fragments of the behaviour of a generic component, to compose components in a system (context) by appropriately connecting them, and to observe the behaviour of the resulting system. It is possible to uniformly model both static and dynamic systems, by means of closed and open contexts, respectively. The two different scenarios need different formulations of correctness properties.

The notion of *correctness* for closed contexts is quite standard, and it can be summarised by the absence of deadlock. For each possible computation all the interaction patterns in the context reduce to the successful pattern `()[0]`. The finiteness of the interaction patterns sensibly contributes to the efficient verification of this property.

Open contexts may also evolve because a new pattern joins the running system. In this case the notion of correctness has to be refined to cope with the incompleteness of dynamically evolving contexts. We have introduced a weaker notion of correctness, *feasibility*, which intuitively corresponds to the absence of a deadlock which can not be resolved by the contribution of any

component that will eventually join the context. An open context is *feasible* if and only if there exists an interaction pattern that can join the context, and make it both closed and correct. Feasibility appears like a desirable, easy to verify, invariant in the life of an open system. For example, a mobile component should be accepted inside a site only if it does not spoil its feasibility. The following example illustrates a feasible context.

Example 2.3 We showed a correct context in Example 2.2. Consider now a context consisting only of a client not capable of generating break events; such as:

{ (_s) [out(_s,query). in(_s,_y).0] },

the context is feasible, since even a server capable of handling break events:

(_c) [in(_c,query). (out(_c,_z).0 + in(_c, break).0)]

can join the context (by the substitution [_x/_s , _x/_c]) making it closed and correct. Indeed the context only evolution, where a query and a reference are exchanged, leads to a successful context. On the contrary, if we consider the dual situation of a feasible context with a client capable of sending a break:

{ (_s) [out(_s,query). (in(_s,_y).0 + τ.out(_s,break).0)] }

it can not be joined by a server not capable of handling break events:

(_c) [in(_c,query). out(_c,_z).0]

since the latter would make the context *not* correct, as one evolution of the context leads to a (permanently) deadlocked context, the name _x, local and not open, can not be accessed by any new component joining the context:

{ () [out(_x,break).0], () [out(_x,answer).0] }.

3 Security properties and "magic" contexts

We now extend the setting of interaction patterns with mechanisms to express secure communication protocols as composition of components. As a major example we show how to specify and verify properties of security protocols.

A context must protect its data/resources from being corrupted (*integrity*) or accessed (*secrecy*) by untrusted components. We consider components that may be distributed over a network and use public channels to communicate to each other. Therefore, in order to face security issues, communication requires security protocols based on cryptography. Protocols may be implemented as interaction patterns in contexts where malicious intruders can operate. Intruders are interaction patterns that may dynamically join a context and interact with the intended principals of the protocol.

We present a methodology which reduces the verification of security protocols to the correctness of the contexts in which the protocols are executed. In particular we will seek for the existence of an intruder that is able to "crack" the protocol without affecting the correctness of the entire context. Due to the

existential nature of this analysis, a weaker notion than feasibility suffices. We say that a context is *may-correct* if and only if there exists at least one computation in which all the interaction patterns reduce to the successful pattern `()[0]`. Given a may-correct context C, an interaction pattern is *C-compatible* if and only if it can join C without spoiling its may-correctness.

The methodology presented consists in a revised use of the notion of *magic instance*, originally introduced in [1]. A *magic instance* of a protocol $\mathcal{P}$, written $\widehat{\mathcal{P}}$, is an instance in which some variables in $\mathcal{P}$ have been fixed to ground values. Informally, if $\mathcal{P}$ is secure, all possible successful executions of $\mathcal{P}$ in an untrusted environment should assign the fixed values to the selected parameters, in spite of any possible interference of an intruder. The problem of verifying the security properties of a protocol is reduced to comparing the compatibility of an intruder with both the protocol and its magic version. Formally, we say that $\mathcal{P}$ is not secure w.r.t. the properties defined by $\widehat{\mathcal{P}}$ if and only if there exists an intruder I that is $\mathcal{P}$-compatible but that is not $\widehat{\mathcal{P}}$-compatible. In other words, if the protocol is unsafe, an intruder I may succeed in cheating a principal of $\mathcal{P}$ by forging values different from the expected ones, while still preserving the may-correctness of $\mathcal{P}$. On the other hand, I cannot cheat the magic version of principals that (magically) knows the correct values and cannot accept different data (viz., $\widehat{\mathcal{P}} \cup I$ deadlocks). The methodology consists of two steps.

3.1 Specification of security properties

A secure protocol does not permit unintended accesses to, or modifications of its "sensible" data. In this sense, the security of a protocol is reduced to the identity of some communicated data. Determining the data and the values whose integrity implies the desired security property is the crucial task. To give a flavor of this step of the methodology, let us consider a simple protocol. The protocol states that a principal A first sends (or better, intends to send) a key k to principal B, then it sends B a nonce n encrypted with k:

$$A \triangleq \texttt{()[out(net,k).out(net,\{n\}}_{\texttt{k}}\texttt{).0]},$$
$$B \triangleq \texttt{()[in(net,x).in(net,\{y\}}_{\texttt{x}}\texttt{).0]}$$

where $\texttt{\{n\}}_{\texttt{k}}$, like in the spi-calculus, is the encryption of `n` with the key `k`. Communication actions encapsulate the encryption and decryption mechanisms: The attempt to receive an encrypted data by means of a key `k`, e.g. $\texttt{in(net,\{x\}}_{\texttt{k}}\texttt{)}$, will succeed only if the sent data are encrypted with `k`, otherwise a deadlock will occur. On the other hand, it is possible to receive an encrypted data as is, i.e. `in(net, x)`, without attempting to decrypt it.

To check whether the protocol $\{A,\ B\}$ respects the secrecy of `k` (that is evidently false, since A sends `k` over `net`), we require that the value of the variable `x` in the first input action of B is exactly `k`, independently of any possible interference in the execution of the protocol. This implies that nobody is able to forge a different key for B. The corresponding magic instance is:

$A \triangleq$ `()[out(net,k).out(net,{n}k).0]`
$\widehat{B} \triangleq$ `()[in(net,k).in(net,{y}k).0]`

3.2 Verification of security properties

Once a desired security property has been expressed in terms of a magic instance of the protocol, we can verify whether an intruder exists which can violate the integrity of the data by (maliciously) interfering with the protocol.

Such a check is performed by a nondeterministic algorithm which given a protocol $\mathcal{P}$ and a magic version $\widehat{\mathcal{P}}$, either returns an intruder that is $\mathcal{P}$-compatible and is not $\widehat{\mathcal{P}}$-compatible, or it returns `none` if such an intruder does not exist. Intuitively speaking, the algorithm consists of the following main loop:

(i) Try to construct an intruder which collects as many messages as possible among those sent on the channels it can access.

(ii) When no message is available, nondeterministically extend the intruder with an action sending to another component values different from the ones that have been fixed in the magic instance.

(iii) Exit the loop when
- all principals have been reduced to `()[0]`, and return the constructed intruder, or
- no message can be sent in point (ii), and return `none`.

Sketching the execution of the algorithm in the previous example, we obtain the following incremental construction of the intruder:

- The intruder grabs the first message sent by A (`k`) by means of the action `in(net,u)`.
- The intruder grabs the second message sent by A ($\{\texttt{n}\}_\texttt{k}$) by means of the action `in(net,v)`
- No more messages are available at the moment. The intruder (who now knows `k` and also `n`) sends `k'` to B, where `k'` $\neq$ `k` is a key generated by the intruder. The intruder then sends $\{\texttt{m}\}_{\texttt{k}'}$ to B, which reduces to `()[0]` (as A did at the previous step).
- All principals have been reduced to `()[0]`. The constructed intruder:

  $I =$ `()[in(net,u).in(net,`$\texttt{v}_\texttt{u}$`).out(net, k').out(net,`$\{\texttt{m}\}_{\texttt{k}'}$`)]`

  is returned.

It is easy to verify that $\{I, A, B\}$ is a may-correct context, while the magic instance $\{I, A, \widehat{B}\}$ deadlocks after the first two inputs of I, since `out(net, k)` and `in(net, k')` deadlock because of the different ground values `k` and `k'`. I is $\mathcal{P}$-compatible but it is not $\widehat{\mathcal{P}}$-compatible, that amounts to say that the protocol is not secure.

4 Concluding remarks

We have presented a uniform framework to describe both composition and security issues of component-based programming. A semi-automatic proof technique to verify correctness of security protocols expressed in terms of component composition and interaction has been presented.

Several formal techniques have been recently proposed for the analysis of security protocols (see e.g. [1,9] and the references therein). On the other hand, a large body of foundational models address the problem of component composition (see e.g. [2,3,4,7,11]). The distinguishing and novel feature of our approach is to provide formal mechanisms to locally and uniformly reason on both composition and security in open system. To the best of our knowledge, only [12,13] address the issue of reasoning about secure composition of components, by considering the coordination of components that may be (partially) untrusted. Components are enclosed into *wrapper programs* to encapsulate and enforce security policies.

Our recent work [5] has been devoted to extend the framework presented in this paper in order to handle multiple sessions and authentication protocols, and to generalise the notion of magic instance so as to allow the specification of arbitrary relations among data and principal instances. Security properties are hence expressed in general as logic formulae rather than as magic instantiations. In spite of its simplicity, the proposed logic permits to express other classes of security properties. Current work is devoted to validate further the proposed methodology by experimenting its application to the work-bench examples of [8].

References

[1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] F. Arbab, M. Bonsangue, and F. de Boer. A coordination language for mobile components. In *Proc. ACM Symp. on Applied Computing*, ACM Press, 2000.

[3] F. Arbab, M. Bonsangue, and F. de Boer. A logical interface description language for components. In António Porto and Gruia-Catalin Roman, editors, *COORDINATION'2000*, LNCS 1906, pages 249–266, Limassol, Cyprus, September 2000.

[4] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A formal model for componentware. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, 2000.

[5] A. Bracciali, A. Brogi, G. Ferrari, and E. Tuosto. Formal intruder identikit for open security protocols. Submitted to Foundations of Software Technology and Theoretical Computer Science, 2001.

[6] A. Bracciali, A. Brogi, and F. Turini. Coordinating interaction patterns. In *Proceedings of the ACM Symposium on Applied Computing*. ACM press, 2001.

[7] C. Canal, L. Fuentes, J.M. Troya, and A. Vallecillo. Adding semantic information to IDLs. is it really practical ? In *Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics*, Denver, Colorado, 1999.

[8] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature. Unpublished, August 1996.

[9] R. Focardi and R. Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1), 1995.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[11] E. Najm, A. Nimour, and JB. Stefani. Infinite types for distributed objects interfaces. In *Proc. third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99*. Kluwer, 1999.

[12] P. Sewell and J. Vitek. Secure composition of insecure components. In *Proceedings of the Computer Security Foundations Workshop, CSFW-12*, 1999.

[13] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.