SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems

Marco Autili, Paola Inverardi, Alfredo Navarra, Massimo Tivoli
Computer Science Department, University of L'Aquila
Via Vetoio I-67100 L'Aquila, Italy.
E-mail: {marco.autili,inverard,navarra,tivoli}@di.univaq.it

Abstract

SYNTHESIS *is a tool for automatically assembling correct and distributed component-based systems. In our context, a system is correct when it is deadlock-free and performs only specified component interactions. In order to automatically synthesize the correct composition code,* SYNTHESIS *takes as input an high-level behavioural description for each component that must form the system to be built and a specification of the component interactions that must be enforced in the system. The automatically derived composition code is implemented as a set of distributed component wrappers that cooperatively interact with each other and with their wrapped components in order to prevent possible deadlocks and make the composed system exhibit only the specified interactions. The current version of* SYNTHESIS *supports two possible development platforms: Microsoft COM/DCOM, and EJB (Enterprise Java Beans).*

1 Introduction

Reuse-based software engineering is becoming a widely used development approach for business and commercial systems. Nowadays, a growing number of software systems are built as a composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. *Component Based Software Engineering* (CBSE) is a reuse-based approach which addresses the development of such systems.

One of the main goals of CBSE is to compose and adapt third-party components to make up a system [5, 11]. Building a distributed system from reusable or COTS components introduces a set of problems, mainly related to compatibility and communication issues. Often, components may have incompatible or undesired interactions. A widely used technique to deal with these problems is to use adaptors. They are additional components interposed between the components forming the system that is being assembled.

Despite the growing number of research works in the area of component adaptation and assembly (see [3, 4, 10, 15] and references therein), very few tools have been proposed to support automatic synthesis of the *actual* composition code for a set of components. Moreover, many of them do not support *industrial* component technologies and frameworks such as, *e.g.,* Microsoft COM/DCOM, Enterprise Java Beans (EJB).

This paper introduces SYNTHESIS, a tool for automatically assembling correct and distributed component-based systems out of a set of already implemented black-box components. SYNTHESIS implements our adaptor-based approaches to component assembly (see [1, 2, 7, 8]).

SYNTHESIS has been already presented in our previous work [12]. That implementation of SYNTHESIS was the first one and it was concerned with only the approaches described in [2, 8]. These approaches are based on a *centralized adaptor* which enforces the system to exhibit only a set of *deadlock-free* or *desired behaviours*. However, in a distributed environment it is not always possible or convenient to insert a centralized adaptor. For example, existing legacy distributed systems might not allow the insertion of a new component (*i.e.,* the adaptor) which coordinates the information flow in a centralized way.

In this paper we describe a recent and maturer version of SYNTHESIS implementing the extension (described in [1]) of our previous work to concurrent and distributed systems. The implemented approach automatically generates a *distributed adaptor* for a set of black-box components. It is a specification-based and decentralized approach. Namely, given (a) the *interface* specification (*i.e.,* the IDL) of each component, (b) the specification of the *interaction behaviour* of each component with its environment, and (c) a specification of the *desired behaviour* that the system to be composed must exhibit, it generates *component wrappers* (one for each component). These wrappers suitably communicate in order to avoid possible deadlocks and enforce the specified desired interactions. They constitute the distributed adaptor for the given set of black-box components.

More precisely, starting from the components' IDL and

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

from the specification of the components' interaction behaviour, SYNTHESIS automatically builds a behavioural model, *i.e.,* an LTS (Labeled Transition System), of a centralized *glue adaptor*. This is done by performing a part of the synthesis algorithm described in [8]. At this stage, the adaptor is built simply for modeling all the possible component interactions. It acts as a simple router and each request/notification it receives is strictly delegated to the right component. By taking into account the specification of the desired behaviour that the composed system must exhibit, SYNTHESIS explores the centralized adaptor LTS in order to find those states leading to deadlocks or violating the specified desired behaviour. This process is used to automatically derive the set of component wrappers that constitute the *correct*[1] and *distributed* version of the centralized adaptor.

SYNTHESIS supports two possible implementations of the generated distributed adaptor. One implements the adaptor's actual code as a set of COM/DCOM component wrappers (one for each component). These wrappers act on the component registries in order to interpose themselves among the components, intercept their messages, and coordinate them as it has been specified. The other implements the adaptor as a set of EJB component wrappers. Each wrapper is developed by using *AspectJ* to intercept the component messages and correctly coordinate them. SYNTHESIS allows the system assembler to display both the generated models (in terms of LTSs) and the actual code implementing the distributed adaptor (in terms of the actual code of each component wrapper).

So far, SYNTHESIS has been applied to several case studies. As the most relevant ones, we mention a cooling water pipe management system [1], a computer supported cooperative work system [9], and a product data management system [13]. The case studies in [1, 13] are industrial case studies carried on in cooperation with PREXIS S.P.A. and Think3 companies, respectively.

The remainder of the paper is structured as follows: Section 2 gives an overall description of the method implemented by SYNTHESIS. Section 3 briefly describes the modular software architecture of SYNTHESIS. Section 4 concludes and discusses possible future extensions.

2 The SYNTHESIS tool

By referring to Figure 1, the method implemented by the current version of SYNTHESIS assumes the following data as inputs.

(a) The interface specification of the components forming the system to be built. It is given as a set of IDL files, one for each component, implementing a *server* logic (obviously, for components that implement only a client code there is no IDL). According to "*design by contract*" approaches [11], we can assume that each of these IDL files is augmented by the component developer through a commented header. Such an header contains a relative path which externally refers to an XML file encoding the specification of the interaction protocol (see input (c)). Note that, this is only an implementation choice and it can easily be changed. For a client, such an XML file is directly provided by the client developer. In our context, a component always respects its interaction protocol specification since it is provided by the developer of the same component, who is aware of the information needed to specify the component protocol.

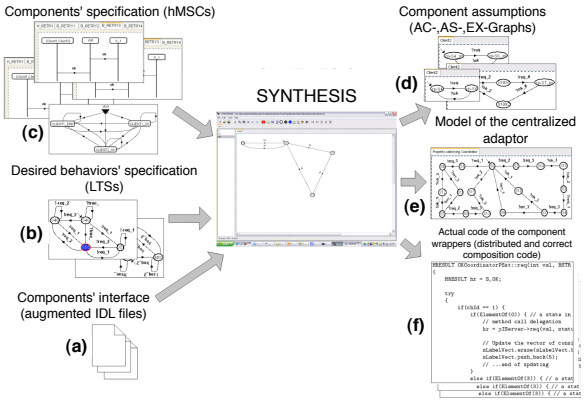

Figure 1. The SYNTHESIS tool

(b) The specification of the desired behaviours that the system to be built must exhibit. It is given in terms of a set of LTSs with a specific syntax for the transition labels. This syntax is expressive enough in order for a transition to model a precise component message (*i.e.,* usual action), a set of component messages but a specific one (*i.e.,* complemented action), and all possible component messages (*i.e.,* universal action). Each state of this LTS models a state of the system to be built and only one initial state exists.

(c) For each component (either client or server), the specification of its interaction protocol with the expected environment. It is an XML file that encodes an *high-level Message Sequence Chart* (hMSC). An hMSC specifies the possible scenarios for message exchanging between the component and its expected environment, and the possible continuations among the scenarios. For a server component, this XML file is referred within its augmented IDL.

These three inputs are then processed in two main steps:

[1] With respect to deadlock-freeness and the specified desired behaviour.

(1) by taking into account the inputs (a) and (c), SYNTHESIS automatically derives the LTSs (called AC-Graphs) that model the component interaction behaviour with the expected environment (*i.e.,* part of the output (d) in Figure 1). This is done by implementing a revisited version of the algorithm described in [14]. However, SYNTHESIS can also take these LTSs directly as input by providing the user with LTS drawing tools. From a component LTS, two other behavioural models are automatically derived: an AS-Graph and an EX-Graph (*i.e.,* the rest of output (d)). The former models the environment that the component expects in order not to block it. The latter is a partial model of the centralized glue adaptor (see Section 1). It is partial since it reflects the expectation of a single component. By "unifying" all component EX-Graphs, SYNTHESIS automatically derives the LTS K of the centralized glue adaptor (*i.e.,* output (e)). For supporting internal data traceability, as it is already said in Section 1, SYNTHESIS provides the user with displaying features for all generated models. At this stage, K models all the possible component interactions and it does not apply any adaptation policy. In other words, it simply routes component requests and notifications. In this way, K represents all possible linearizations by using an interleaving semantics. K is derived by performing the *graph unification* algorithm described in [8];

(2) After K has been generated, SYNTHESIS explores it looking for those states representing the *last chance* before incurring in an execution path that leads to a deadlock. The enforcement of a specified desired behaviour is realized by visiting the LTS modeling it (input (b)). The aim is to split and distribute this LTS in such a way that each component wrapper knows which actions the wrapped component is allowed to execute. The sets of last chance states and *allowed actions* are stored and, subsequently, used by the component wrappers as basis for correctly exchanging *synchronizing information* when strictly needed. In other words, the generated component wrappers interact with each other to perform the correct behaviour of K with respect to deadlock-freeness and the specified desired behaviours. By means of the K decentralization, the component wrappers preserve parallelism of the components forming the system. The SYNTHESIS's user can select which implementation of the wrappers must be enabled: COM/DCOM, or EJB.

It is worth mentioning that the systems assembled by using SYNTHESIS are *closed*. This implies that, for instance, if an additional non-wrapped client wants to connect a server then either the entire synthesis process must be re-executed or that action is disallowed.

3 Architecture of the SYNTHESIS tool

In Figure 1 we have shown the input and output data performed by SYNTHESIS. Note that, by means of capital letters, we obtain a direct mapping between Figure 2 and Figure 1. This mapping allows us to correlate each module with the I/O data it manages, performs, or builds.

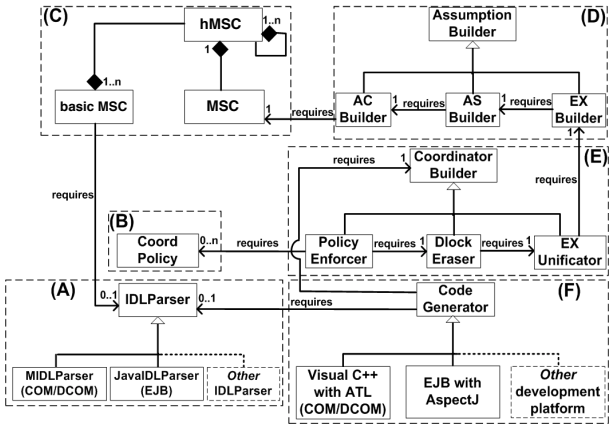

Figure 2. The SYNTHESIS's architecture

In the reminder of this section, we briefly describe each SYNTHESIS's module.

Module A, component interface parser: this module contains a superclass (*i.e.,* "IDLParser" entity in Figure 2) that manages a specific data structure which is for storing an abstract representation of an IDL file possibly given as input. That superclass has to be specialized in order to implement a parser of IDL files based on a particular IDL notation (*e.g.,* Microsoft IDL for COM/DCOM, DCE/IDL for CORBA, Java IDL for EJB, etc.). In the current version of *SYNTHESIS*, we specialized that class to implement two parsers: one for Microsoft IDL (MIDL) files and the other for Java IDL files.

Module B, LTS specification of the desired behaviours: this module is used to specify the desired behaviours for the system to be built in terms of LTSs. The current implementation of this module codes each LTS as a binary object by exploiting *Java* serialization. A corresponding version based on XML is still work in progress. Each LTS describes component interactions that must be cooperatively enforced by the component wrappers to be generated.

Module C, hMSC specification of the components: this module is used to specify/display/process the interaction behaviour of each component with its expected environment in terms of an hMSC specification. In doing that, this module exploits an *ad-hoc* library that we have developed to allow creation, validation and manipulation of hMSCs coded in *XML*. To check if these *XML* files are valid, an

ad-hoc XML schema is used. This module requires a suitable implementation of the "IDLParser" entity (see module **A** shown in Figure 2).

Module D, component assumption generation: by referring to Figure 2 this module exploits the module **C**. Taking as input the hMSC specification of a component, this module builds and outputs the component AC-Graph. According to the hMSCs data format, an AC-Graph is coded in *XML* too. By referring to Section 2, from a component AC-Graph, this module can also derive the component AS- and EX-Graph.

Module E, builder of the centralized adaptor model: this module is responsible for deriving the model of the centralized glue code. In particular, it is specialized by the "EXUnificator", "DlockEraser" and "PolicyEnforcer" entities shown in Figure 2.**E**. These entities respectively implement: (i) the EX-Graph unification algorithm that is for building the LTS of the centralized adaptor; (ii) the last chance states identification; (iii) the allowed actions (with respect to the specified desired behaviours) identification.

Module F, generator of the component wrappers actual code: this module implements a generator of the component wrappers actual code. It is structured analogously to module **A** and, hence, it refers to one or more specific development platforms. Currently, it only supports the generation of the code implementing COM/DCOM or EJB component wrappers. Note that a component wrapper does not care about checking the value of actual parameters of a component method. In fact, it is only a method call "delegator" whose delegation logic respects both the deadlock-freeness and the specified desired behaviours. This explains why, in drawing the MSC specification of the components, *SYNTHESIS* does not require that a user must specify the actual parameters of a method call.

4 Conclusion and future work

In this paper, we described our SYNTHESIS tool. SYNTHESIS is a tool for automatically deriving the correct and distributed composition code for a set of already implemented software components.

As ongoing work, we are currently re-engineering SYNTHESIS to plug it into the CHARMY framework core [6]. This will allow SYNTHESIS to interoperate with the other CHARMY's plugins and, in particular, with a *scenario-based* property specification plugin. It has been implemented by our research group as a wizard tool for guiding the user in the non-trivial task of specifying desired functional properties. This would make the desired behaviour specification easier.

Possible future work concerns a language-based extension of SYNTHESIS to real-time systems that are not currently supported. This extension would require to augment the component protocol specification in order to take into account also timing information. Other future work is focused on extending the SYNTHESIS tool in order to deal with (web-)services orchestration and choreography.

References

[1] M. Autili, M. Flammini, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis of concurrent and distributed adaptors for component-based systems. In *European Workshop on Software Architecture*, volume LNCS 4344. 2006.

[2] M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of correct adaptors for protocol enhancement in component based systems. In *Specification and Verification of Component-Based Systems (SAVCBS2004)*.

[3] A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. *10th International Conference on Algebraic Methodology And Software Technology (AMAST2004)*.

[4] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume LNCS 4037. 2006.

[5] I. Crnkovic and M. Larsson. *Building reliable component-based Software Systems*. Artech House, 2002.

[6] P. Inverardi, H. Muccini, and P. Pelliccione. CHARMY: An Extensible Tool for Architectural Analysis. In *5th ESEC/FSE - Research Tool Demos*, 2005.

[7] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Elsevier Journal of Systems and Software*, 65(3)(7346):173–183, 2003.

[8] P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. volume LNCS 2804. 2003.

[9] P. Inverardi, M. Tivoli, and A. Bucchiarone. Automatic synthesis of coordinators of COTS group-ware applications: an example. In *12th IEEE International Workshops on Enabling Technologies (WETICE2003)*.

[10] R. Passerone, L. de Alfaro, T. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility Verification and Converter Synthesis: Two Faces of the Same Coin. In *International conference on computer-aided design (ICCAD'02)*.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2004.

[12] M. Tivoli and M. Autili. SYNTHESIS: a tool for synthesizing correct and protocol-enhanced adaptors. *RSTI L'Objet journal*, 12(1):77–103, 2006.

[13] M. Tivoli, P. Inverardi, V. Presutti, A. Forghieri, and M. Sebastianis. Correct components assembly for a Product Data Management cooperative system. volume LNCS 3054, 2004.

[14] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *1st ESEC/FSE*, 2001.

[15] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.