Synthesis of Correct and Distributed Adaptors for Component-based Systems: An Automatic Approach

Paola Inverardi, Leonardo Mostarda, Massimo Tivoli, and Marco Autili
Dept. of Computer Science, University of L'Aquila
L'Aquila, Italy
inverard@di.univaq.it, mostarda@di.univaq.it, tivoli@di.univaq.it,
marco.autili@di.univaq.it

ABSTRACT

Building a distributed system from third-party components introduces a set of problems, mainly related to compatibility and communication. Our approach to solve these problems is to build an adaptor which forces the system to exhibit only a set of *safe* or *desired* behaviors. By exploiting an *abstract* and *partial* specification of the global behavior that must be enforced, we automatically build a *centralized* adaptor. It mediates the interaction among components by both performing the specified behavior and, simultaneously, avoiding possible deadlocks. However in a distributed environment it is not always possible or convenient to insert a centralized adaptor. In contrast, building a distributed adaptor might increase the applicability of the approach in a real-scale context. In this paper we show how it is possible to automatically generate a distributed adaptor by exploiting an approach to the definition of distributed IDS (Intrusion Detection Systems) filters developed by us to increase security measures in component based systems. Firstly, by taking into account a high level specification of the global behavior that must be enforced, we synthesize a behavioral model of a centralized adaptor that allows the composed system to only exhibit the specified behavior and, simultaneously, avoid possible unspecified deadlocks. This model represents a lower level specification of the global behavior that is enforced by the adaptor. Secondly, by taking into account the synthesized adaptor model, we generate a set of component filters that validate the centralized adaptor behavior by simply looking at local information. In this way we address the problem of mechanically generating *correct* and *distributed* adaptors for real-scale component-based systems.

Categories and Subject Descriptors: D.2 [Software Engineering]: Miscellaneous

General Terms: Design, Algorithms.

Keywords: Component Based Software Engineering, Component Assembly, Component Adaptation.

ASE'05, November 7–11, 2005, Long Beach, California, USA.

1. INTRODUCTION

Reuse-based software engineering is becoming the main development approach for business and commercial systems. Nowadays, a growing number of software systems are built as 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. In this context, one of the main goals of CBSE is to compose and eventually adapt loosely coupled independent components to make up a system [2, 8]. Adaptation of software components is an important issue in CBSE; building a distributed system from reusable or COTS components introduces a set of problems, mainly related to compatibility and communication. Often, components may have incompatible or undesired interactions. One 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. The intent of the adaptors is to moderate the communication of the components in a way that the system complies only to a specific behavior.

Our existent approach [7] (implemented in the *SYNTHESIS* tool) to solve these problems is to build an adaptor which forces the system to exhibit only a set of *safe* or *desired* behaviors. For example, the adaptor forces the system to exhibit only the subset of deadlock-free and/or explicitly specified wanted behaviors. By exploiting an *abstract* and *partial* specification of the global behavior that must be enforced, we automatically build a *centralized* adaptor. It mediates the interaction among components by both performing the specified behavior and, simultaneously, avoiding possible unspecified deadlocks.

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 addition of a new component (i.e., the adaptor) which coordinates the information flow in a centralized way. The coordination of an increasing number of components can cause loss of information or increase the response time of the centralized adaptor. In contrast, building a distributed adaptor might increase the applicability of the approach in a real-scale context.

In this paper we show how it is possible to automatically generate a distributed adaptor by exploiting an approach [6] (implemented in the *DESERT* tool) to the definition of distributed IDS (Intrusion Detection Systems) filters developed by us to increase security measures in component based sys-

tems. It is a specification-based approach to detect intrusions at architectural level. It is decentralized. That is, given a global policy for the whole system (i.e. an admissible global behavior) it automatically generates a monitoring filter for each component that looks at local information of interest. Filters then suitably communicate in order to carry on cooperatively detection of anomalous behavior and enforcement of the global policy.

The basic idea the *DESERT* approach is based on might be used to derive distributed component adaptor. By exploiting a *concrete* and *complete* specification of the global behavior that must be enforced, it might be possible to generate a set of component filters that, looking at local information, validate the specified behavior. In contrast with our existent approach to component adaptation, this second approach assumes that the specification of the global behavior (to be validated) includes also all possible deadlock-free interactions. In other words, unlike our existent approach, the second one cannot deal with deadlocks in an automatic way. Neither is ideal. Building a distributed adaptor requires a lower level and complete specification of the global behavior that must be enforced. For example, all possible deadlock-free interactions has to be explicitly specified as a part of the complete specification. Due to the high complexity of a real-scale system, often, it is impossible to provide one with such a specification. We address these problems by suitably combining our existent approaches to component adaptation and to the definition of distributed IDS filters.

Firstly, by taking into account a high level specification of the global behavior that must be enforced, we synthesize a behavioral model of a centralized adaptor that allows the composed system to only exhibit the specified behavior and, simultaneously, avoid possible unspecified deadlocks. This model represents a low level specification of the global behavior that is enforced by the adaptor. Secondly, by taking into account the synthesized adaptor model, we generate a set of component filters that validate the centralized adaptor behavior by simply looking at local information. In this way we address the problem of mechanically generating *correct* and *distributed* adaptors for real-scale component-based systems.

The two approaches take advantage of each other. In fact, the combined approach is applicable in situations in which considering only one of them is not practically feasible. *DESERT* permits to derive a *distributed* implementation of the *centralized* adaptor. Moreover, it allows us to remove the deployment's constraints that *SYNTHESIS* imposes on the adaptor in order to integrate it with the system [7]. This allows one to adapt complex applications where no centralized point of information flow exists or can be introduced. On the other hand, *SYNTHESIS* allows a higher level of abstraction in specifying the globally defined behaviors by avoiding the non-trivial (or often impossible) *DESERT*-user's task of specifying them. Moreover, *SYNTHESIS* enables *DESERT* to deal with deadlocks without requiring a specification of the deadlock-free interactions. This is possible because *SYNTHESIS* automatically derives a deadlock-free specification of the *globally* defined behaviors that must be enforced.

This paper is a short version of an existent technical report [4] where we report and compare related work, and validate the approach by means of an industrial case-study.

The reminder of the paper is structured as follows: Section 2 provides background to the problem. Section 3 describes our combined approach. Section 4 concludes and discusses future work.

2. BACKGROUND

In this section we discuss the background needed to understand the approach that we describe in Section 3.

2.1 The reference architectural style

This section describes our reference architectural style. Within this architectural style, we considered three kinds of system configuration concerning with (i) a system without adaptors, (ii) a system in which a centralized adaptor appears and (iii) a system in which distributed adaptor appears.

This architectural style is a layered architectural model in which components can request services of components above them, and notify components below them. We assume each component has a top and bottom interface. Connectors between components are synchronous communication channels defining a top and bottom interface too. The top (bottom) interface of a component may be connected to the bottom (top) interface of one or more connectors.

Components communicate by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. We will also distinguish between two kinds of components: *functional components* and *coordinators* (i.e., adaptors). Functional components implement the system's functionality, and are the primary computational constituents of a system (typically implemented as third-party components). Coordinators, on the other hand, simply route messages and each input they receive is strictly followed by a corresponding output. We make this distinction in order to clearly separate components that are responsible for the functional behavior of a system and components that are introduced to aid the integration/communication behavior.

Within this architectural style, we will refer to (i) a system as a *Coordinator-Free Architecture* (CFA) if it is defined without any coordinators. Conversely, (ii) a system in which centralized coordinators appear is termed *Coordinator-Based Architecture* (CBA) and is defined as *a set of functional components directly connected to one or more centralized coordinators, through connectors, in a synchronous way.* Moreover, (iii) a system in which distributed coordinators appear as sets of filters (one filter for each component) is termed *Distributed-Coordinator-Based Architecture* (DCBA) and is defined as *a set of functional components each of them directly connected to its local filter; each filter is connected to the other filters, through synchronous connectors, in a* peer-to-peer *fashion.* Our architectural style belongs to generic layered styles. In [5] it is showed that, under suitable assumptions, it is possible to decompose a n-layered CBA system in n single-layered CBA subsystems. Thus, in the reminder of this paper, we will only deal with single layered systems. To deal with multi-layered systems we apply the formalized approach for each single layered subsystem.

Figure 1.a) illustrates a CFA, Figure 1.b) its corresponding CBA and Figure 1.c) its corresponding DCBA. $C1$, $C2$, $C3$ and $C4$ are functional components. K is a centralized adaptor. $f1$, $f2$, $f3$ and $f4$ are the local filters. The communication channels denoted as lines between components

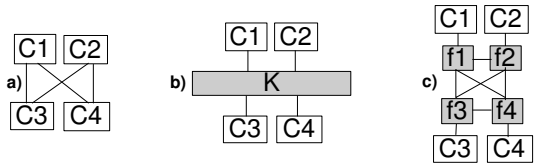

Figure 1: A sample of a CFA, the corresponding CBA and the corresponding DCBA

are connectors (e.g., ORB for CORBA, RPC for COM+ and RMI for EJB).

3. METHOD DESCRIPTION

Our approach aims at solving the following problem: *given (i) a CFA system T for a set of black-box interacting components and (ii) a specification P of the desired behaviors, automatically derive the corresponding deadlock-free DCBA system V which exhibits only the desired behaviors in P.*

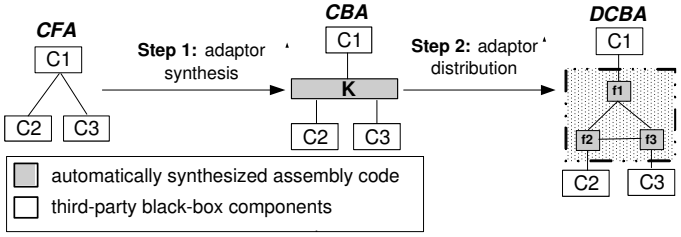

Figure 2: Method's steps

Our approach assumes that a specification of the system to be assembled is provided in terms of bMSCs (i.e.: *basic Message Sequence Charts*) and HMSCs (i.e.: *High-level* MSCs) specification. From the MSCs specification of the system, we can derive the *Labeled Transitions Systems* LTSs specification of each component. This is done by applying a suitable adaptation of the translation algorithm presented in [9]. We also assume that a specification P of the desired behaviors to be enforced exists in terms of Büchi automata[1] [1]. With these assumptions we are able to derive the DCBA system V which is obtained by automatically deriving the local filters that serve as *distributed glue code* for the components forming the CFA system T. These communicating filters mediate the interactions among components by allowing only the execution of the deadlock-free behaviors and the ones in P.

In the following, we discuss our method proceeding in two steps as illustrated in Figure 2.

(1) By tacking into account the specification of the components in T and following the CBA style constraints, the first step builds a deadlock-free centralized adaptor that exhibits only the behaviors in P (i.e., K in Figure 2). This is done by exploiting the coordinator synthesis algorithm we described in [5]. For the purposes of this work, we do not provide the reader with a detailed description of the phases involved in this step but entirely refer to [5] for them.

[1]A Büchi automata is an operational description of a desired behavior. It represents all the system behaviors that respect the logic of the specified desired behaviors.

In [5], by using a suitable notion of behavioral equivalence, we prove that the behavior of the CBA system is equivalent to the behavior of T without deadlocking interactions and the ones that do not belong to P.

(2) The second step exploits the basic idea of the approach to the definition of IDS filters [6]. This step distributes the adaptor K in the CBA to obtain the corresponding DCBA system V. K can be distributed by combining its LTS description with structural information obtained from the actual architectural configuration (i.e., CBA). This combination generates a set of local filters that are assigned one for each component. After the generation process each local filter is implemented as a wrapper that envelops the component it supervises. Therefore, a local filter can observe the messages (sent and received) of the component it resides on. Obviously, considering only the sequences of messages of the enveloped component is not sufficient to locally enforce the behaviors expressed by the LTS description of the adaptor. Therefore, a filter has to observe enriched sequences of messages that also contain context information provided by other filters. Correspondingly, a filter may provide other filters with the needed context information. Such information exchanged among filters is called *dependency information* [6]. These dependencies can be seen as synchronization messages exchanged among filters. Thus, a filter captures both the sequences of messages of the enveloped component and the dependencies information. It uses the dependency information to impose an ordering among the messages that are sent/received by the enveloped component.

We can prove that the behavior of V is equivalent to the one of the CBA system through CB-equivalence. This is done analogously to what we have done in [6] to prove that the behavior of a set of interacting IDS filters is equivalent to the specified one.

In the following section, by means of an industrial case study, we show an application of our approach. This is done by briefly sketching the first step and by focusing on the second one since it represents the main contribution of this paper with respect to our previous work [7].

3.1 Our approach at work

We start by taking into account the bMSC and HMSC specification of the components forming the CFA and their IDL (*Interface Definition Language*) files.

From the bMSC and HMSC specification, our tool automatically derives the LTSs for each component forming the CFA.

Each LTS is a 4-tuple (Q, q_0, I, δ) where Q is the set of nodes (each node denotes a state of the execution), $q_0 \in Q$ denotes the initial state (i.e., the state labeled with $S0$), I is the set of arc labels (each arc label denotes an I/O message), and δ is the transition function (i.e., $\delta \in (Q \times I \times Q)$ where $q_1 = \delta(q, \alpha)$ is the state reachable from the state q by performing the message α). For the sake of presentation, we will hereafter refer to a transition from q to q_1 labeled with α as "$q_1 = \delta(q, \alpha)$". The context allows to distinguish when we are talking of a state or a transition. Moreover, multiple transitions from the same source to the same target are indicated by using one directed arc (i.e., an arrow from the source to the target) with multiple labels that are separated by ";".

Within the LTS of a component, a message $?m$ ($!m$) denotes an input (output) message labeled with m. By per-

forming the *Step 1* of our approach, from the component LTSs, our tool automatically derives the adaptor LTS. In this LTS, a message $?m\_j$ ($!m\_j$) denotes an input (output) message labeled with m and received (sent) from (to) Cj.

For our purposes, let β be an I/O message in a LTS, we define $\overline{\beta}$ as the *complement* of β in such a way that if $\beta = ?\alpha$ ($\beta = !\alpha$) then $\overline{\beta} = !\alpha$ ($\overline{\beta} = ?\alpha$).

The LTS of the adaptor might contain filled nodes, which denote deadlocks. Deadlocks might occur, e.g., because of a race conditions among two components. Concluding the execution of the *Step 1*, our tool simply prunes the paths (of the coordinator LTS) which contain deadlocks in order to obtain the LTS of the deadlock-free adaptor K.

The *Step 2* distributes K in the set of filters (one for each component). As it is already said in Section 3, each filter implements a wrapper that envelops the component it resides on. The filters (by interacting with each other one) capture all messages sent/received by their enveloped components and delegate them by imposing an order that enforces the behavior expressed by K.

The basic idea for the filters generation can be summarized in two phases: *local filters generation* and *dependencies generation*.

Local filters generation

This phase considers the LTS of the centralized adaptor $K = (Q, q_0, I, \delta)$ and the LTS of a component $C_i = (Q^i, q_0^i, I^i, \delta^i)$. The output is a preliminary version of the LTS $f_i = (Q^{f_i}, q_0^{f_i}, I^{f_i}, \delta^{f_i})$ of the filter assigned to the component C_i.

The preliminary LTS of f_i is obtained by considering each transition $q_1 = \delta(q, m\_j)$ defined in K. In the case of $\overline{m\_j}$ is a message that can be performed by C_i (i.e., $\overline{m} \in I_i$ and $j = i$) such transition is reflected in a f_i-transition $q_1 = \delta^{f_i}(q, \overline{m\_j})$, the states q, q_1 are added to Q^{f_i} and the message $\overline{m\_j}$ is added to I^{f_i}. In other words, looking at K, the sequences of interaction that happen locally on a component C_i are projected on the preliminary LTS of the local filter f_i.

We recall that the preliminary version of a local filter is not sufficient to realize the correct enforcement. Our solution is to allow a filter to accept/provide context information by the other filters. We call such information exchanged among filters *dependency information*.

Let f_i be the LTS of the filter of the component C_i. Dependency information is of the form $!f(m, \{C_j\})$ or $?f(m', \{C_k\})$ where C_j and C_k range on the name of the application components. The message $!f(m, \{C_j\})$, *outgoing dependency*, is sent by f_i to filter f_j in order to communicate that C_i has performed the message m. The message $?f(m', \{C_k\})$, *incoming dependency*, is an incoming information sent by filter f_k. With this information f_k communicates to f_i that C_k has performed the message m'.

A dependency information allows a filter to synchronize and cooperate with each other to enable the delegation/acceptance of a message with respect to the order imposed by K.

The dependencies are placed by the *dependencies generation* phase that is performed locally to each component C_i.

Dependencies generation

The basic entities to place dependencies are *synchronization* and *enabling* states.

Let $K = (Q, q_0, I, \delta)$ be the LTS of the centralized adaptor and let $f_i = (Q^{f_i}, q_0^{f_i}, I^{f_i}, \delta^{f_i})$ be the preliminary LTS of the local filter f_i assigned to the component C_i. A state $q \in Q^{f_i}$ is a *synchronization state of* f_i if: (1) $\exists m\_i \in I^{f_i}$ and $\exists \delta^{f_i}(q, m\_i)$ where m_i is a message performed by the component C_i; (2) $\exists m\_k \in I$, $\overline{m\_k}$ does not belong to I^{f_i} (i.e., $k \neq i$) and $\exists q^+ = \delta(q, m\_k)$ and $q^+ \neq q$.

Let q be a state of K that is exited by different transitions projected in different filters. In this case the filters have to synchronize so that exactly one of those transitions will be performed. To simplify matters, let us consider the case in which from q two transitions exit: $q_1 = \delta(q, m\_1)$ and $q_2 = \delta(q, m\_2)$ ($\overline{m\_1}$ and $\overline{m\_2}$ are messages performed by C_1 and C_2, respectively) and $q \neq q_1 \neq q_2$. From the f_1 point of view q is a synchronization state since it is exited by a transition labeled with both a message (i.e., m_1) whose complement message (i.e., $\overline{m\_1}$) is performed by the component it envelops (i.e., C_1) and a message (i.e., m_2) whose complement message (i.e., $\overline{m\_2}$) is performed by a component that is enveloped by a different filter (i.e., C_2 enveloped by f_2). In this case the dependencies generation phase adds to f_1 the transitions $q_2 = \delta^{f_1}(q, ?f(\overline{m\_2}, \{C_2\}))$ and $q_1 = \delta^{f_1}(q, !f(\overline{m\_1}, \{C_1\}))$. The former is performed when f_2 communicates to f_1 that C_2 has performed the message $\overline{m\_2}$ and, in so doing, from the state q the global execution reaches the state q_2. The latter when f_1 communicates to f_2 that C_1 has performed the message $\overline{m\_1}$ and, in so doing, from the state q the global execution reaches the state q_1. Symmetrically, from the f_2 point of view q is a synchronization state and the dependencies $q_1 = \delta^{f_2}(q, ?f(\overline{m\_1}, \{C_1\}))$ and $q_2 = \delta^{f_2}(q, !f(\overline{m\_2}, \{C_2\}))$ are added to f_2. From the point of view of K if it is in the state q then either the transition $q_1 = \delta(q, m\_1)$ or $q_2 = \delta(q, m\_2)$ can be performed. From the filters point of view the dependencies $!f(\overline{m\_1}, \{C_1\})$ and $!f(\overline{m\_2}, \{C_2\})$ sent between f_1 and f_2 (respectively) are a means to overcome the problem.

To generalize the above discussion, let $K = (Q, q_0, I, \delta)$ be the LTS of the centralized adaptor and let q be a state of K, we denote by $f^q = \{f_i : \exists m\_i \textit{ in such a way that } q' = \delta(q, \overline{m\_i})\}$ the set of filters where a transition of K, exiting from q, has been projected.

Suppose that all filters in $f^q = \{f_1, ..., f_n\}$ are in the state q and try to acquire the right to accept/delegates a message performed by the component enveloped by each of them in the same time. The ordering $f_1 < ... < f_n$ among them establishes that only the highest filter in the ordering grants the right. In general, we denote the dependencies sent among the filters in f^q as *synchronization dependencies* since they are used to synchronize the filters in order to accept exactly one message.

To characterize the other type of dependencies, we introduce the concept of *enabling state* as follows.

Let $K = (Q, q_0, I, \delta)$ be the LTS of the centralized adaptor and let $f_i = (Q^{f_i}, q_0^{f_i}, I^{f_i}, \delta^{f_i})$ be the LTS of the local filter f_i assigned to the component C_i. A state $q \in Q$ is an *enabling* state if: (1) $\exists m\_i \in I^{f_i}$ and $\exists \delta^{f_i}(q, m\_i)$ where m_i is a message performed by the component C_i; (2) $\exists m\_k \in I$, $\overline{m\_k}$ does not belong to I^{f_i} (i.e., $k \neq i$) and $\exists q^- \in Q$ such as $q = \delta(q^-, m\_k)$ and $q^- \neq q$.

An enabling state of K defines one that is entered and exited by transitions projected on two different filters. For instance, K can perform two subsequent transitions from a state q to a state q_1 and subsequently to q_2 in such a way

that $q_1 = \delta(q, m\_1)$ and $q_2 = \delta(q_1, m\_2)$, where $\overline{m\_1}$ and $\overline{m\_2}$ are performed by C_1 and C_2 (respectively) and $q \neq q_1$. From the K point of view this sequence of two transitions defines a constraint among the messages m_1 and m_2 (i.e., m_1 must be acceped/delegated before m_2). However, from the local filters point of view this constraint is lost since that transitions' sequence is *divided* onto the filters f_1 and f_2. The problem is solved by adding dependencies. Since q_1 is an enabling state of K the dependencies generation phase adds the transition $q_1 = \delta^{f_1}(q, !f(\overline{m\_1}, \{C_2\}))$ to f_1 and the transition $q_1 = \delta^{f_2}(q, ?f(\overline{m\_1}, \{C_1\}))$ to f_2. Therefore f_2 can allows the global execution to reaches the state q_1 by means of the transition $q_1 = \delta^{f_2}(q, ?f(\overline{m\_1}, \{C_1\}))$. However this transition can be performed when f_1 sends the outgoing dependency $!f(\overline{m\_1}, \{C_1\})$ (i.e., f_1 can impose the right ordering among the messages $\overline{m\_1}$ and $\overline{m\_2}$). We denote such type of dependencies as *enabling dependencies* since they are used (by a filter) to correctly enable the filter-local computations.

The assignment of dependencies, on each local filter, is iterated until all enabling and synchronization states are covered. However, after this phase a filter (i.e., its LTS) may still have disconnected parts (i.e., it may be constituted by a set of disconnected sub-LTSs). We use ε-transitions [3] to link the parts of the LTS in the right ordering and to build a whole connected LTS of the local filter. The final version of a filter is obtained by applying usual technique to remove ε-transitions of each filter.

It is worth mention that (due to the ε-transitions elimination step) the state labels of each filter LTS are automatically renamed by our tool. Moreover, the tool makes use of unique identifiers automatically generated to codify the dependency messages.

As already discussed above, a filter uses dependencies to enable the delegation/acceptance of a message with respect to the order imposed by K. We remark that, only looking at local information, we maintain deadlock-freedom as well as all the behaviors globally specified by means of K.

The overhead of messages generated by the filters is strictly related to the behavior defined by the LTS of the centralized adaptor. A filter adds dependency messages when *non-interacting components* behavior has to be related. Let q be a state of the adaptor. Let $m_1 m_2 \dots m_n$ be n messages, related to n different components residing on n different hosts $H_1 H_2 \dots H_n$. Suppose that each message of $m_1 m_2 \dots m_n$ labels a transition exiting from the state q. In the worst case when a filter on the host H_i moves from a state q to a state q', with $q \neq q'$, then at most n dependencies can flow on the distributed system. In practice dependency synchronization messages are relatively small in size and, depending on the system architecture, it is possible to bound the number of the messages exiting from a state q related to different components/hosts.

Actual code derivation

The LTS of a local filter constitutes the basis formal specification to build our distributed adaptor equivalent to the centralized one, which is modeled by the LTS of K. Each filter is implemented as a component wrapper whose implementation logic reflects the structure of the state machine represented by the filter LTS. This wrapper is interposed between the environment and the component whose behavior has to be enforced as dictated by the structure of the filter LTS. It contains a message buffer used to store component messages and dependencies. To delegate a component message, the wrapper has to (i) send the synchronization dependencies in order to acquire the right of delegating that message; (ii) if it grants this right then it has to send the enabling dependencies to correctly preserve the behavior of the centralized adaptor K.

4. CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to automatically assemble component-based systems by synthesizing distributed adaptors. Our approach suitably combines two different methods we have previously developed. One method, implemented in the *SYNTHESIS* tool [7] allows one to automatically synthesize centralized adaptors for component-based systems. The second one, implemented in the *DESERT* [6] tool, allows one to generate a set of IDS filters that look at local information in order to validate globally defined security policies.

The two methods take advantage of each other. The latter allows the derivation of a distributed implementation of the centralized adaptor and, hence, it enhances *scalability*, *fault-tolerance*, *efficiency* and *deployment*. On the other hand, the former allows (i) a higher level of abstraction in specifying the globally defined behaviors by simplifying the non-trivial user's task of specifying them and (ii) detection of *deadlocks*. In [4], we successfully validated the approach on a real-scale case study in the domain of industrial applications (*PREXIS S.r.L. company - ITALY - http://www.prexis.it*).

Currently *SYNTHESIS* derives the actual COM/DCOM code implementing a centralized adaptor. However, the approach might still suffer of the well known state-explosion problem since we automatically derive the distributed adaptors by constructing a behavioral model of a centralized one. Synthesizing that model has an exponential space-complexity. As future work we plan to automatically synthesize the implementation of the distributed adaptors by using a *compositional* and *on-the-fly* technique which may avoid to produce the model of the centralized adaptor.

5. REFERENCES

[1] J. Buchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Method and Philosophical Sciences*, 1960.

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

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-wesley publishing company, 1979.

[4] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. Technical report, Dep. of Computer Science, University of L'Aquila - http://www.di.univaq.it/tivoli/trcs_07.pdf, 2005.

[5] P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly*. Springer, LNCS 2804.

[6] L. Mostarda and P. Inverardi. A distributed intrusion detection approach for secure software architecture. *2th European Workshop on Software Architecture. To appear.* 2005.

[7] M.Tivoli and M.Autili. Synthesis: a tool for synthesizing "correct" and protocol-enhanced adaptors. *to appear on L'Object Journal, http://www.di.univaq.it/tivoli/LastSynthesis.pdf*, 2005.

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

[9] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.