A Meta-protocol and Type system for the Dynamic Coupling of Binary Components

Ralf H. Reussner[1] and Dirk Heuzeroth[2]

[1] Chair Informatics for Engineering and Science
Universität Karlsruhe
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
reussner@ira.uka.de
[2] Institute for Programming Structures and Data Organization
Universität Karlsruhe
Am Zirkel 2, D-76128 Karlsruhe, Germany
heuzer@ipd.info.uni-karlsruhe.de

Abstract. [1]We introduce a new type system, where the type of a component consists of two *protocols* — a call and a use protocol. We model these protocols by finite automata and show how those reflect component enhancement and adaption. Coupling is controlled by a meta-protocol, which calls the adaption and enhancement algorithms. These algorithms require type information of the components involved. This type information is provided by the meta-protocol using reflection. This mechanism allows automatic adaption of components in changing environments.

1 Introduction

The aim of software-engineering in general is to support the efficient development of high-quality software products in such a way, that success is repeatable. Motivations for doing this are increasing quality, reducing time-to-market and, as a consequence lowering development costs. The reuse of approved work is one technique to reach this goal [MJ97][BP89]. Several reuse techniques have been proposed and employed. None of them has been satisfying, mainly because units of reuse have not been easily adaptable to several specific application contexts (or only with an enormous effort).
The development of complex applications can only be accomplished by composing, and thus reusing, approved parts. This composition requires, that reuse units must only have explicit external dependencies. To reuse components in assembling an application, they have to be adapted to the specific requirements. For the sake of an easy composition, adaptions should take place automatically. To support this, we introduce component types (see section 4), describing the services a component offers *and* the services it wants to use from other components. This is opposed to other works in the field of type systems [Mey92] [MMHH95], which only view types as specifying allowed operations on entities of the type. Automatic adaptions and extensions are in our approach reflected by dynamic changes

[1] accepted at OOPSLA 99 Workshop on Object Oriented Reflection and Software Engineering Denver, COLORADO Monday, November 1st, 1999

of the component type. This kind of type modification is not supported by current type systems [WZ88], because they often are too strict.
When a system is enhanced by new components, or components are replaced, we want to avoid halting or recompiling the system, for the sake of convenience. Thus, we have to compose the system dynamically.
Existing source code reuse techniques do not offer the required flexibility, are highly complex and require an enormous effort of understanding a unit of reuse in order to really reuse it. To reduce the effort of understanding a reuse unit, it should only have explicitly specified dependencies. No look into the source code should be necessary in order to reuse it. Therefore it should be delivered in binary form.
The above argumentation motivates why we focus on binary reuse units satisfying the above requirements. We call these units *binary components* and give a more precise definition of this notion in section 2 (fundamentals). This reuse technique may thus be described as *grey box reuse* opposed to black and white box reuse. Although the idea of grey box reuse is not new (see for example [Pae96]), many details of how it should work are still unknown. Even the idea of *software components* is not new [McI69]. But its precise definition is still concern of debates.
Despite of the advantages of component technology mentioned above, the extensive usage of binary components is impeded by several problems.
The functionality-reuse problem: Generally, problems with the 'adding functionality strategy' are: firstly, it is difficult to determine all requirements of a component in advance. Even when this can be done, a second problem arises: Adding new functionality does not in general improve the reusability of a software component, since the added functions translate into new requirements to the environment into which the component is to be embedded. Thus, the component becomes less reusable, contrary to the original intention. For example, imagine you are designing a printer management component. If you restrict its functionality to handle only local printers, it will not be very reusable, because it will not handle network printers. However, if you design the component for network printers, it will require a network even for managing the local printer, and that will not make it very popular with users.
The type-extension problem: Problems with the 'dynamic enhancement strategy' are that today type systems handle this dynamic binding only rudimentarily. Usually a plug-in is a parameterized extra application. Using it as a plug-in just means that it can be launched automatically with correct parameters (e.g., viewers in browsers or file converters in printer controls). Today type systems cannot handle more sophisticated interfaces to plug-ins. Especially, it is not clear how the functionality of a plug-in-component enhances the functionality of the component using the plug-in. The functionality of the plug-in does not really appear as new functionality at the interface of the plug-in using component. For example, this would be necessary when a user-interface has to adapt after inserting a plug-in component into a component collaborating with the user interface.
In our approach, we concentrate on components, that have to be installed only *once* in a system, i.e., there are no copies of a component in a system and there is only one component of a certain type installed in the system. But this is not a real restriction for practical cases, since it makes no sense to install for example several identical office components in a system.
In the sequel we first introduce the fundamental terms for understanding the rest of the paper (section 2). Section 3 describes our approach and illustrates it by an

example scenario. In section 4 we explain our solution to the dynamic coupling problem by introducing a type system for software components. Section 4.3 describes the meta-protocol we use to couple components. In section 5, we give an overview of more loosely related work, than the one we just compared with in this section. We finish by drawing our conclusions in section 6.

2 Fundamentals

In this section we define the basic terms used in this paper. As we have seen in the introduction 1, we deal with reuse units, more precisely components. So we have to define this term first. The other terms we define, result from our component definition and our introduction of component types.

Definition 21 (software-component) *The most important term is that of a* software component. *Unfortunately there is still no commonly agreed definition of this term. But some properties seem to be characteristic:*
'A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed indecently and is subject to composition by third parties.' ([Szy98]) No look onto the sources of a component should be necessary to reuse it, i.e., composing it with other components.
Dynamic composition requires components in a binary form. To avoid misunderstandings, for us binary *just means a non-source-code form of a component, which need not necessarily be the machine code of a certain hardware processor. As an example, we consider a component in JAVA byte code also as binary.*
To be applicable in several infrastructures, a component has to be easily adaptable or should even adapt itself automatically.
We basically recognize two strategies which are used to maximize the number of contexts a component is reusable in: (a) 'Adding functionality strategy': *The more functionality a component offer, the more users will be satisfied with this component and will reuse it. (b)* 'Dynamic enhancement strategy': *In this approach it is not tried to anticipate during design stage all functionality a component should have in its entire life-time. Instead of that, certain connectors for further enhancements are defined. When during employment of a component new functionality is required, the component can be enhanced at run-time by so-called 'plug-ins'.*

Definition 22 (functionality) *The* functionality *of a component does not only consist of the functions it offers, but also of the protocol to use them. For example, certain services are only available after initialization, some services may exclude others, etc. More concrete: functionality = functions + the protocol to use them.*

Definition 23 (component type) *The type of a component describes the applicability of the component [Nie93]. Information of the applicability of a component is represented by its interface.*

Definition 24 (applicability) *One aspect of the applicability of a component is the availability of its services. This availability depends on (a) the call protocol of functions, (b) the services the infrastructure offers the component, and (c) access restrictions due to security policies.*

Definition 25 (component interface) *The* component interface *represents the applicability of the component, i.e., it consists of the offered functions with their* call protocol *(the functionality)* and *the used functions with their* use protocol*.*

Definition 26 (call protocol) *The* call protocol *specifies the allowed call sequences of the functions offered by a component.*

Definition 27 (use protocol) *The* use protocol *specifies how a component wants to call the functions offered by another component.*

Definition 28 (coupling) *The* coupling *of components is the composition of two or more components, which involves either type adaption (see section 4.1) or type extension (see section 4.2).*

3 Coupling binary components

3.1 Our Approach

In this paper we describe our approach to support run-time coupling of binary components. We think that the problems mentioned in section 1 arise from the missing of a definition of a component type. According to Nierstrasz a type should describe the typed entity's applicability [Nie93]. This means the type of a component depends on its applicability.
One aspect of a component's applicability is the *availability* of services it offers to the applications it is used by. Two conditions of service availability are:

1. Another service has been called before — e.g., calling an initialization routine is a prerequisite for using other services.
2. Services form certain other components are available.

Point 1 specifies the protocol of allowed calls (the *'call protocol'*) of a component A. To handle condition 2 we additionally have to know how a component B uses a component A (*'use protocol'*). With both protocols together one can check, whether two components can work together or not. Basically the two protocols form the interface of a component and thus its *type*.
When coupling components, certain classes of errors can be detected immediately by matching the use protocol and the call protocol of the components to be coupled. Actually, this is a type check. A unique feature of our system is the automatic adaption of the call protocol of a component, in case the infrastructure does not support all services exactly as required by the component. The extension of component functionality by plug-ins can also easily be described by these types.
In order to perform these type-checks, adaptions, and extensions we require a certain run-time support, which is controlled by a meta-protocol.

3.2 Scenario

In this section we demonstrate by an example scenario which issues in component composition are not possible today but are supported by our type system.
Consider a framework for a mail user agent with a text-reader, a sound-player, and a video-player as pre-compiled components. These components should be added dynamically (i.e., at run-time) to the mail user agent component. Note, that the framework component, i.e. the mail user agent, should not need to know in advance which kinds of mails it has to handle, so it leaves this aspect open. But nevertheless, we want it in compiled form, because the user should not need to recompile the mail user agent only because he receives an unknown kind of mail he wants to perceive. The current solution of MIME-types is not satisfying, because the functionality of the required components (plug-ins) for handling the different MIME-types is not properly integrated into the mail user agent.
Figure 1 shows an UML class diagram of our mail user agent example. Note that the exact function signatures are omitted, for the sake of brevity.
On a first glance, the introduction of a common superclass for all mail types seems reasonable. So we could have introduced a class `Mail` with subclasses `TextMail`, `SoundMail`, and `VideoMail`. `Mail` could be used by `MailUserAgent` to represent all kinds of mails. The problem with this solution is, that we do not know all kinds of mails in advance. The superclass `Mail` could only contain methods and attributes common to all mail types. We even have to check dynamically the type of a concrete `Mail` object in order to use all its features. But this implies that we have to know all kinds of mails in advance. In contrast to this, in our approach neither the user nor the developer has to know all types of mails in advance, i.e., the mail types are not restricted.
In a concrete example the user receives a video mail, which is then listed in the inbox like the other mails. This video mail knows which sound players and video players could be used to present the mail. When the user selects the video mail, it uses the run-time system to get the actual paths to required players. In case they are not available, the run-time system may be able to automatically down-load them via the internet and install them. But for our example imagine the computer has no hardware support for sound, and the required sound player is missing. So the run-time system only returns the path to the video player. The video mail couples with this player component and adapts its functionality so that it can play videos but has no ability to play and control the sound of the mail. This functionality is integrated into the mail user agent. The mail user agent shows this new functionality in the menu and automatically starts playing the mail. The user can use the interface of the mail user agent to pause or to abort the video. Even the whole video player can be controlled by the mail user agent.
The following issues mentioned in the above scenario cannot be handled by todays technology.
pre-compilation: The `MailUserAgent` has no specific viewer for the different mail formats. In current framework-technology, one solution would be an abstract superclass for viewers, where actual viewers have to be derived from. This would hinder the compilation of at least parts of the framework. Furtheron, this viewer-superclass has to anticipate the functionality of all possible viewers in advance. A task which really cannot be accomplished. Additionally, viewers must be derived from this superclass, what impedes the reuse of existing viewers.
dynamic enhancement of functionality: Another way to make the `MailUserAgent` view different mail formats is to enhance it with several plug-ins. This is

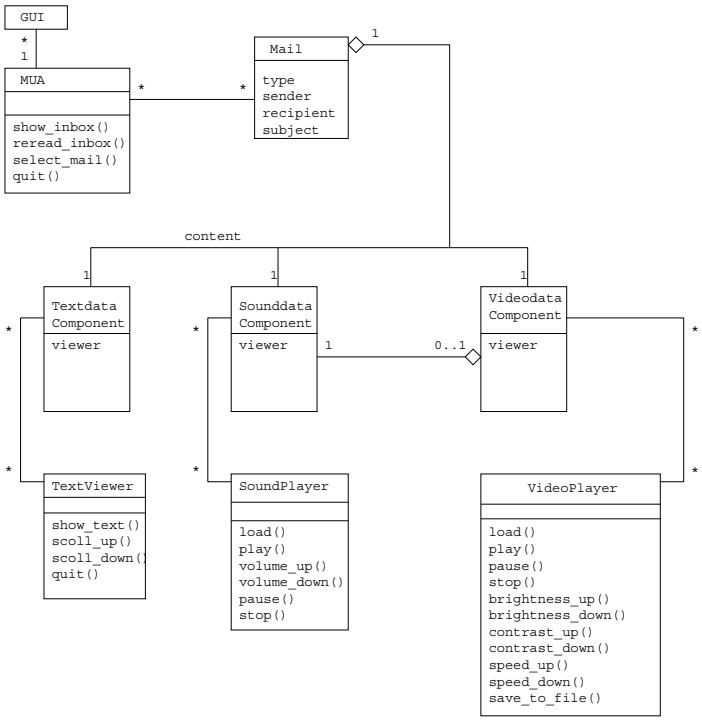

Fig. 1. Class Diagram of the Mail User Agent Example

possible only when one is willing to accept an extra GUI for each plug-in. In fact, the `MailUserAgent` gives up the control to the user-interface of the viewer. This may be acceptable, when using components which can have their own user interface. This traditional extension by plug-in components cannot be used, when the plug-in components are not controllable by user interfaces. In this case the plug-in component should be controlled by the extended component (here the `MailUserAgent`) and not directly by the user. Our type system allows the extension by plug-ins having no user interface.

dynamic adaption of functionality: A video requires sound support and graphic support to be presented. With current component technology `VideoMail` could not be viewed when the sound support is missing, unless the programmer of `VideoMail` has explicitly tackled this case. In general: currently it is not possible to deploy a component, when the environment does not offer all resources and services the component requires. But in many cases a component can still offer a subset of its services, when not all of the required resources are available. In our example the `VideoMail` can be viewed with `VideoPlayer` although there is no sound support given. So `VideoMail` adapts its functionality to the environment.

4 Dynamic Types for Software Components

There are two reasons for adding types to a programming language: (a) to ease memory layout during compilation, and (b) to add information to the program, which allows the compiler to detect certain errors statically, i.e., before the execution of the program. Our type system for software components makes the compilation of generic components possible and provides enough information to check for certain errors during the coupling of components. Here, we concentrate on protocol errors due to coupling non-fitting components, or installing a component in an insufficient infrastructure, etc. These errors should be detected during the coupling of components. This coupling may happen during compile-time (when it is statically known, which components will couple), or during installation (when components are installed), or even at run-time (when components are added dynamically to a running system). However, protocol errors are detected during a defined coupling time, when they may be expected. This is superior to the occurrence of errors at undefined later times, e.g., when the user starts an operation which causes an error.

Our approach defines a *component type* as a tuple consisting of (a) a *Component-Automaton (C-Automaton)* and (b) a set of *Function-Automata (F-Automata)*. The C-Automaton describes the *call-protocol* of a component as a formal language, i.e., the set of all legal sequences of calls to the component's functions. Each function of the component has a F-Automaton. This F-Automaton describes the set of all possible sequences of calls to internal and external functions this function could perform. The C-Automaton and the F-Automata together form the *use-protocol*. This interface describes all possible call sequences to external functions as a formal language.

All automata, C-Automaton and F-Automata, are deterministic finite automata, as described for example in [Nel68]. Their alphabets consist of the function names of a component, respectively of the names of the functions a component calls. Although not necessary in our example, we need some extensions to finite automata, mainly to describe related calls like push and pop on a stack, and to overcome other modeling problems with finite automata [All97].

It is important, that the added type information is no burden for the software developer. In our type system the required interface description of the use protocol can be partially computed by the compiler by control-flow analysis [Muc97]. The description of the call protocol can be given by simple pre- and postconditions of functions and extend the idea of contracts [Mey92].

The VideoMail component of our example has the C-Automaton shown in figure 2. The function `play` just uses the `play` functions of the `VideoPlayer` and the `SoundPlayer` components, thus only having a rather short F-Automaton, shown in figure 3.

The advantage of a finite automata based approach is the decidability of the equivalence problem and intersection problem (opposed to more expressive notations, such as push-down-automata [Sch92]). Furtheron, elaborated theoretical results for finite automata [Nel68] and protocol validation [Hol91] are usable and it is possible to merge automata theoretic algorithms with graph theoretic algorithms.

We distinguish two kinds of component composition, *type adaption* and *type extension*, which are described below.

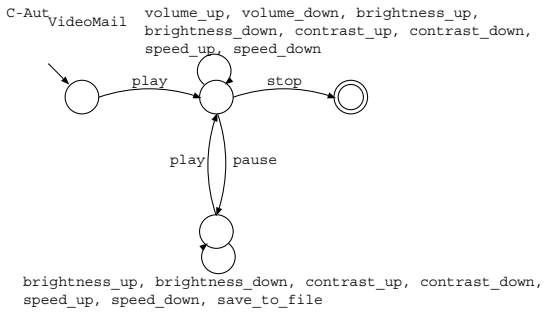

Fig. 2. C-Automaton of the VideoMail component

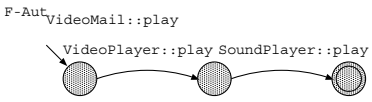

Fig. 3. F-Automaton of the play-function of the VideoMail component

4.1 Type Adaption

Type adaption is used when a component is inserted in a certain context. The automatic adaption problem (cf. section 1) occurs when a component has to be adapted to offer its services using different infrastructures. Type adaption guarantees that the component can offer a (possibly restricted) functionality in many different environments. If the environment does not offer all functionality required by the component, the component adapts correspondingly its offered functionality.

In terms of our type system, when the use interface of component A matches the call interface of component B, component A can use component B. In cases of an inexact match, a new call protocol of A is calculated, so that the new depending use interface of A matches exactly the call interface of B.

Note that these protocol computations happen when coupling components. After the coupling, when the components actually run (again), a correct use of component A according to its (possibly new) call interface ensures that component A uses component B according to B's call interface. That is A calls functions of B only according to the call protocol of B. Due to the dynamic changes of A's call interface, component A can offer its services (or a subset of them) even when B does not offer all services A requires from B.

The automaton of the use interface is built by plugging in the F-Automata into the C-Automaton, whereby each transition of the C-Automaton is replaced by the F-Automaton of the function corresponding to the transition, as shown in figure 4. Since the new automaton is an enhancement of the C-Automaton, it is called *Enhanced-Component-Automaton (EC-Automaton)*. It describes all possible sequences of calls to external functions. Note that it is also possible to construct the C-Automaton of a component, when only the EC-Automaton of the component is given.

As a finite automaton, each interface not only defines a protocol [Hol91], but also a regular language [Sch92]. To match the use interface of A and the call interface of B, we build the intersection of the languages defined by the EC-Automaton of

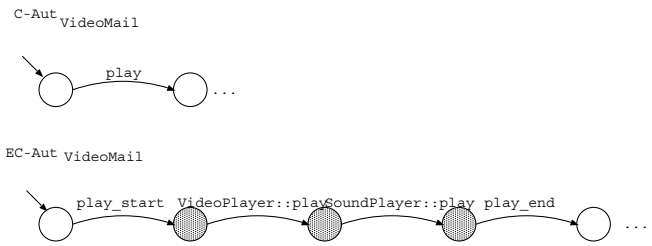

Fig. 4. Construction of a EC-Automaton out of the C-Automaton and the F-Automaton of figure 3.

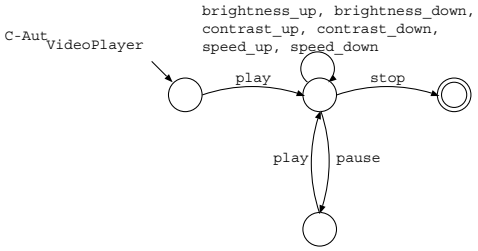

Fig. 5. Call interface of the `VideoPlayer`

A and the C-Automaton of B. This intersection of languages corresponds to the construction of a so called *cross product automaton (CPA)* [Sch92]. This CPA is the new EC-Automaton of the component created by coupling A and B, and it contains the new C-Automaton$_{A \times B}$. If B fulfills all requirements A posed, then the new C-Automaton$_{A \times B}$ is equivalent to the old C-Automaton$_A$, otherwise it just describes a new call interface, which can be offered by A when using B's incomplete functionality. The space- and time-complexity of this algorithm is determined by the complexity of the CPA construction, i.e., the product of the number of states of the involved automata.

In our example the functionality of a certain type of mail (text mail, sound mail, or video mail) adapts to the functionality of the corresponding viewers (the infrastructure). Opposed to type extension (described below) a video mail can know the most general functionality (including the use protocol) of a video player during design, because it is clear what functionality a video player must have to show a video mail. It is also clear which functionality of a video mail is still possible when a sound player is missing. As a consequence the mail is coupled with its corresponding viewer/player with type adaption.

Figure 2 shows the C-Automaton of the `VideoMail` component. Consider, that `VideoMail` needs a `SoundPlayer` component and a `VideoPlayer` component. This would be expressed in its F-Automata, and hence in its EC-Automaton. They are omitted here. Imagine that the `SoundPlayer` is missing in a concrete system, because of missing hardware sound support, and that the call interface of the `VideoPlayer` is the one of figure 5.

Then the new call interface of `VideoMail` coupled with `VideoPlayer` is given in figure 6. Note, that not only the sound services are not available (which

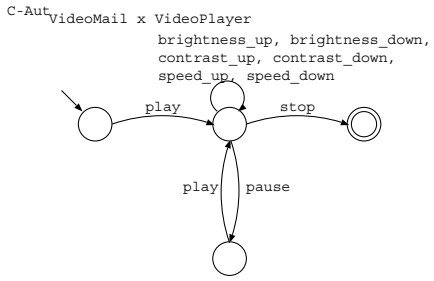

Fig. 6. New call interface of `VideoMail` coupled with `VideoPlayer`.

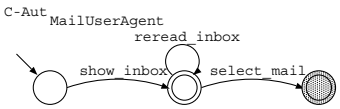

Fig. 7. Unextended call interface of the MailUserAgent component.

would not need a protocol to model), but also the availability of the video control services (for brightness, contrast, and speed) also changes (after pause pressed). To express these changes of availability protocol information is required.

4.2 Type Extension

Type extension is used when the functionality of a component cannot be foreseen during development. In our example this is the ability of the mail user agent to handle several mail formats, which are possibly not known during design of the mail user agent. In this case, the functionality of a viewer/player of this format is used to extend the functionality of the mail user agent at a defined point. The user does not recognize which functionality belongs to which component. All functionality is integrated in the user interface.
Figure 7 show the C-Automaton of the `MailUserAgent` component, before any extension. In our example this C-Automaton is coupled with a GUI. Note the shaded state in the unextended C-Automaton. This is the *coupling state*. After the user selected a mail, the C-Automaton of the corresponding mail component is plugged in the unextended C-Automaton of `MailUserAgent` at this state. Imagine the user selects a VideoMail (which has coupled with `VideoPlayer` before). Then the C-Automaton of `VideoMail x VideoPlayer` (as shown in figure 6) extends the C-Automaton of `MailUserAgent`. The resulting extended C-Automaton of `MailUserAgent` is shown in figure 8. The time complexity of this algorithms linearly depends on the number of coupling states, when copying of the inserted automata can be avoided.

4.3 Meta-protocol

In this section we sketch a meta-protocol which supports the above type adaption and coupling. The scenario of section 3.2 can be described in terms of our type system as follows.

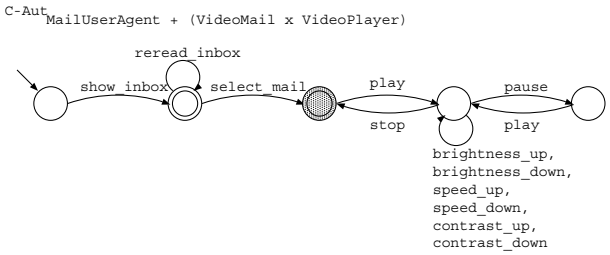

Fig. 8. Call interface of the MailUserAgent extended with VideoMail.

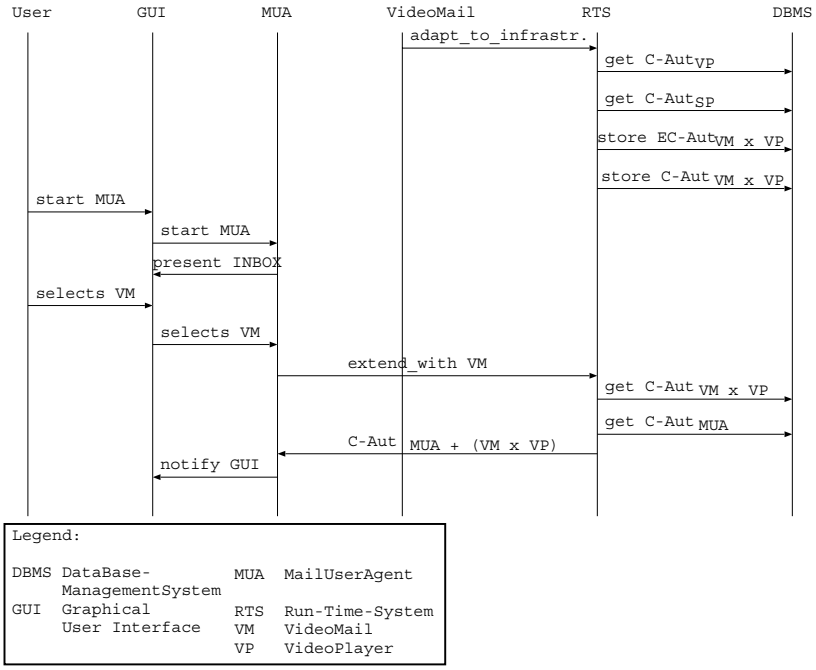

Fig. 9. Event-trace diagram of the scenario

1. An arrived `VideoMail` asks the run-time system to couple with its supporting infrastructure (i.e., `SoundPlayer` and `VideoPlayer`). The `VideoMail` includes in this request also URL's specifying where to get the required components. Additionally the `VideoMail` sends its EC-Automaton to the run-time system.
2. The runtime-system contains a database of the installed components and a description of the underlying system hardware. This description denies the installation of a sound player, since our hardware system contains no sound support, as mentioned in section 3.2. We assume that the required `VideoPlayer` is already installed. The run-time system retrieves its current C-Automaton from its database, and performs the type adaption algorithm with the EC-Automaton of `VideoMail` and the C-Automaton of `VideoPlayer`. The new C-Automaton of `VideoMail` is extracted from

the new EC-Automaton. Both (new C-Automaton and new EC-Automaton) are stored in the database entry associated with `VideoMail`. The new C-Automaton is returned to `VideoMail`. Note that this happens after the arrival of `VideoMail`. The user is not involved into these actions.

3. Later, the user starts the `MailUserAgent` via a GUI component. The `MailUserAgent` reads the inbox, and lists sender, subject, and type of each mail.
4. The user selects the `VideoMail` to view it.
5. The `MailUserAgent` requests type extension with `VideoMail` from the run-time system. It is not necessary, that `MailUserAgent` sends its C-Automaton, since C-Automata of already installed components are stored in the database.
6. The run-time system retrieves the C-Automaton of `VideoMail` from the database and performs the type extension algorithm with C-Automaton$_{MailUserAgent}$ and C-Automaton$_{VideoMail \times VideoPlayer}$. The resulting C-Automaton$_{MailUserAgent+(VideoMail \times VideoPlayer)}$ is returned to `MailUserAgent`.
7. The `MailUserAgent` notifies the GUI about its new functionality.
8. The GUI presents this new functionality to the user.
9. The user starts viewing the `VideoMail`.

Figure 9 shows a corresponding event-trace diagram.
This protocol uses a repository, which stores the component types. This repository need not necessarily be a database. The type information can also be attached at the component itself.
To realize the above scenario additional meta-information is necessary. (a) Information how to present the new functionality of the `MailUserAgent` in the GUI, and (b) information telling the `MailUserAgent` how to start the `VideoMail`. This information is not necessary to describe the above algorithms, but also has to be included in the call interface of a component.

5 Related work

Basically, the binary form of components may be seen as the major difference to the software modules described in [Par72]. Language support for this modularization of source code, e.g., in MODULA-2 [Wir85], can be seen as an early attempt of software reuse. Beside source code modules, other approaches of software reuse recently gained attention[TC97]. Software architectures allow the reuse of high-level software designs. They are relevant to software component technology, because they focus on the connection of several components [GS93]. Similarly design patterns [GHJV95] package lower level design information for reuse. Several benefits of design reuse are presented in the literature [BMR$^+$96] [Pre95] [GHJV95]. Despite of the well-known advantages of design reuse versus plain code reuse [BP89] [Joh97], several problems occur, when design information is visible in the units of reuse [GAO95]. One of the expected major benefits of software components is, that a software component is deployable in several different design contexts. Therefore, design decisions have to be encapsulated and hidden behind a component interface.
The COMPOST project[Aßm] uses another approach to adapt components to several contexts. The idea is somewhat inverse to ours. In COMPOST the contexts

of a component are defined by the applications which want to use or reuse it. If a component does not fully satisfy these context requirements, it is adapted appropriately. That is, the application composer develops and executes a meta-program, which invasively modifies the component, adding code to fulfill the requirements. The vision of COMPOST is, that components can automatically adapt themselves to different application contexts, by enhancing or modifying themselves. Currently only source code modifications are possible by this approach. Opposed to that, our adaption mechanism concerns dependencies from the underlying infrastructure, that is the opposite direction as in COMPOST. Moreover, we do not add any functionality to components that is not available in any of the underlying infrastructure components.

Frameworks are an attempt to reuse designs manifested in code. Frameworks are enhanced to a specific application by providing specific modules which are plugged in certain defined interfaces of the framework [Joh97]. Unfortunately, the interface descriptions do not contain enough information, so that a compiler can translate an uninstantiated framework into binary code. As a consequence most frameworks are delivered in source code. The relations between components and frameworks are many fold. With an interface description allowing the compilation of frameworks, we can regard such a binary framework as a component. The modules used to specialize a framework to an application can be seen as components, too. Changing these components during run-time allows system adaption and configuration. Furtheron, even the run-time system which supports the loading and coupling of components can be seen as a general framework, with the components as plug-ins.

Todays technology allows a limited composability of binary components. Well-known are plug-ins in internet browsers, such as special viewers. In principle, this is not a real composition to integrate the functionality of the viewers into the one of the browser. Basically the browser just calls the viewers with suitable parameters passing control to the respective viewer. Opposed to that, real composition comprises the integration of the services of the viewers into the browser. Thus, another component associated with the browser (e.g., a GUI) cannot distinguish the original browser functionality from the ones integrated from the viewers. This integrating composition is much more flexible than todays calling of plug-ins, because the latter usually is only useful when the functionality of the plug-ins can be modeled by *one* rough granulated procedure call. Additionally, allowing each plug-in to open a separated user interface limits the applicability and comfort of this approach.

Another technique of binary composition is the use of mobile objects [Nel99]. This approach is based on object composition [ND95] well known from patterns and frameworks [GHJV95] [Pre95] [Pre97]. Java objects in byte-code are used to configure other objects. While this composition is surely important for component technology, the interface description of an object only specifies static properties (such as the type of methods, and so on). Information concerning the dynamic coupling of components are not described by the interface. This is why many errors due to coupling non-fitting components cannot be checked by the compiler or the coupling system in advance.

6 Conclusions

We presented a meta-protocol and a type system for software components, which support dynamic coupling of components. This support consists of checking protocol errors of collaborating components and, as a unique feature of our type system, the dynamic adaption of a component's type to its environment. We identified two important ways of dynamic coupling: (a) type adaption and (b) type extension, and explain when to apply each.
The mail user agent example shows the insufficiencies of current technology, the importance of dynamic coupling and the advantages of automatic component adaptions.
In this paper we concentrated on type information necessary for protocol checks and adaption. However, to fully realize the above example scenario, a component type must include additional information.
Our further work comprises an implementation of the meta-protocol and type system in form of a compiler and run-time system for an enhanced form of JAVA.

References

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[Aßm] Uwe Aßmann. The COMPOST project main page. http://i44www.info.uni-karlsruhe.de/~assmann/compost.html.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, New York, 1996.

[BP89] T. J. Biggerstaff and A. J. Perlis. *Software Reusability*, volume I & II. ACM Press Addison-Wesley, Reading, MA, 1989.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[McI69] M. D. McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MJ97] Stephen J. Mellor and Ralph Johnson. Why explore object methods, patterns, and architectures? *IEEE Software*, 14(1):27–30, January/ February 1997.

[MMHH95] R. Mitchell, I. Maung, J. Howse, and T. Heathcote. Checking Software Contracts. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 — Technology of Object-Oriented Programming*, pages 97–106. Prentice Hall, August 1995.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, San Mateo, CA, 1997.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Nel68] R. J. Nelson. *Introduction to Automata*. John Wiley & Sons, New York, NY, 1968.

[Nel99] Jeff Nelson. *Programming Mobile Objects With Java*. John Wiley & Sons, New York, NY, 1999.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices 28(10)*, pages 1–15, October 1993.

[Pae96] Andreas Paepcke. Open Implementations and Metaobject Protocols, 1996. Tutorial Book.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Reading, MA, 1995.

[Pre97] Wolfgang Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt.verlag, Heidelberg, 1997.

[Sch92] Uwe Schöning. *Theoretische Informatik kurz gefasst*. Bibliographisches Institut Mannheim, 1992.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, Reading, MA, 1998.

[TC97] William M. Tepfenhart and James J. Cusick. A unified object topology. *IEEE Software*, 14(1):31–35, January/February 1997.

[Wir85] N. Wirth. *Programming in MODULA-2*. Springer-Verlag, 3rd Edition, 1985.

[WZ88] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and what isn't like. In *ECOOP'88*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer Verlag, 1988.