Dynamic Coupling of Binary Components and its Technical Support

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

[1]Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Am Zirkel 2, 76128 Karlsruhe, Germany
heuzer@ipd.info.uni-karlsruhe.de

[2]Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler, Universität Karlsruhe, Am Fasanengarten 5, 76128 Karlsruhe, Germany
reussner@ira.uka.de

Abstract

The aim of todays software development is to build applications by the reuse of binary components. This requires the composition of components and as special cases component enhancement as well as adaption. We demonstrate how to deal with these cases by furnishing components with a *type* consisting 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. This mechanism allows for automatic adaption of components in changing environments.

In order to obtain binary components we have to compile corresponding sources. In view of the required features of the binary components and with the problems of compiling generic classes in mind, we describe an approach to generate such *pre-compiled components* by appropriate compiler extensions.

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 techique 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 explict 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.

When a system is enhanced by new components, or components are replaced, we want to avoid halting or recompiling the system, for 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.

In the sequel we first introduce the fundamental terms for understanding the rest of the paper (section 2). In section 3 we discuss current problems of component technology. Section 4.1 contains an application scenario, showing todays technologies insufficiencies, and the problems we tackle. In section 5 we explain our solution to the dynamic coupling problem by introducing a type system for software components. Section 5.3 describes the meta-protocol we use to couple components. In section 6 we present an approach to solve the pre-compilation problem already mentioned in the abstract. We finish by giving an overview of related work in section 7 and draw our conclusions in section 8.

2 Fundamentals

In this section we define the basic terms used in this paper.

software-component: The most important term is that of a *software component*. Unfortunatley 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 indepently 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 ceratin hardware processor. E.g., 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.

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

component type: The type of a component describes the applicability of the component [Nie93]. Information of the applicability of a component is given in its interface.

component interface: The interface of a component describes its applicability, i.e., it consists of the offered functions with their *call protocol* (the functionality) and the used functions with their *use protocol*.

call protocol: The protocol specifying the allowed call sequences of the functions offered by a component.

use protocol: The protocol specifying how a component wants to call the functions offered by another component.

3 Current problems of component technology

Software component technology is becoming increasingly popular, since it is considered to be a candidate to accomplish the expected benefits of reuse. [Szy98][ND95]. But despite of the advantages

of binary component technology, the extensive usage of binary components is impeded by several problems:

pre-compilation problem: Currently many important portions of frameworks have to be delivered as source code, since they leave many aspects open for the sake of flexibility and thus cannot yet be compiled into a binary form. The use of generic classes is an example realization of leaving aspects open, which generally prohibits the pre-compilation of frameworks. Framework developers however would prefer to hide their sources and deliver the framework as a binary component. The application programmer using a binary framework then "only" has to plug in his own binary components. To do this comfortably, i.e. so that he can watch how system behavior changes by the plugged-in components, system composition should be possible dynamically.

automatic adaption problem: Components often have to be adapted or enhanced before being ready for reuse. Even in component based software development adaptions and enhancements will be necessary, because

1. not all functionality of a component can be anticipated during development — e.g., processing of future file formats — and
2. a component has to be deployable in various contexts, where each context may offer a different infrastructure the component has to adapt to during coupling.

Especially point 2 is important for a component market, because adding functionality to a component will widen its range of potential customers, although usually a single customer will not need the whole functionality. Unfortunately, a large functionality often implies strong dependencies of a component to its infrastructure. But most customers will not be willing to invest in such a powerful infrastructure, especially, when it supports component functions they do not need. Hence, a large functionality often complicates or even impedes the widespread use of a component.

When a component is able to adapt its functionality according to its infrastructure, a component manufacturer can add new functionality — and thus requirements to the component's infrastructure — without restricting the set of potential customers. When binary components are coupled dynamically these adaptions and enhancements have to be done automatically. Support for this automatic dynamic coupling is still very weak. Especially errors due to coupling non-fitting components are not detected as early as possible, that is during coupling-time. This is because component interfaces do not specify how to use the component.

missing uniform component model: Due to the absence of a commonly accepted, precisely defined software component model, difficulties in combining several vendors' components arise. Current software component models leave too many details of interaction open or define them differently. Since one defining property of components is their reusability in various contexts (e.g., [Szy98]), a component model has to specify a standardized meta-level protocol for retrieving the interface information[1] of a component.

4 Coupling binary components

4.1 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 3 arise from the missing of a definition of a component type.

[1] More precisely we should talk of the applicability information.

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*.

This type systems allows pre-compilation because unknown parts of a framework component can be described by a typed interface. The information given to the compiler by this interface is sufficient to do pre-compilation.

When coupling components, certain classes of errors can be found 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 in the way required by the component. 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 controled by a meta-protocol.

4.2 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 sake of brevity.

On a first glance, the introduction of a common superclass for all mail-types seems reasonable. So we could have intoduced 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 show or more precisely play 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

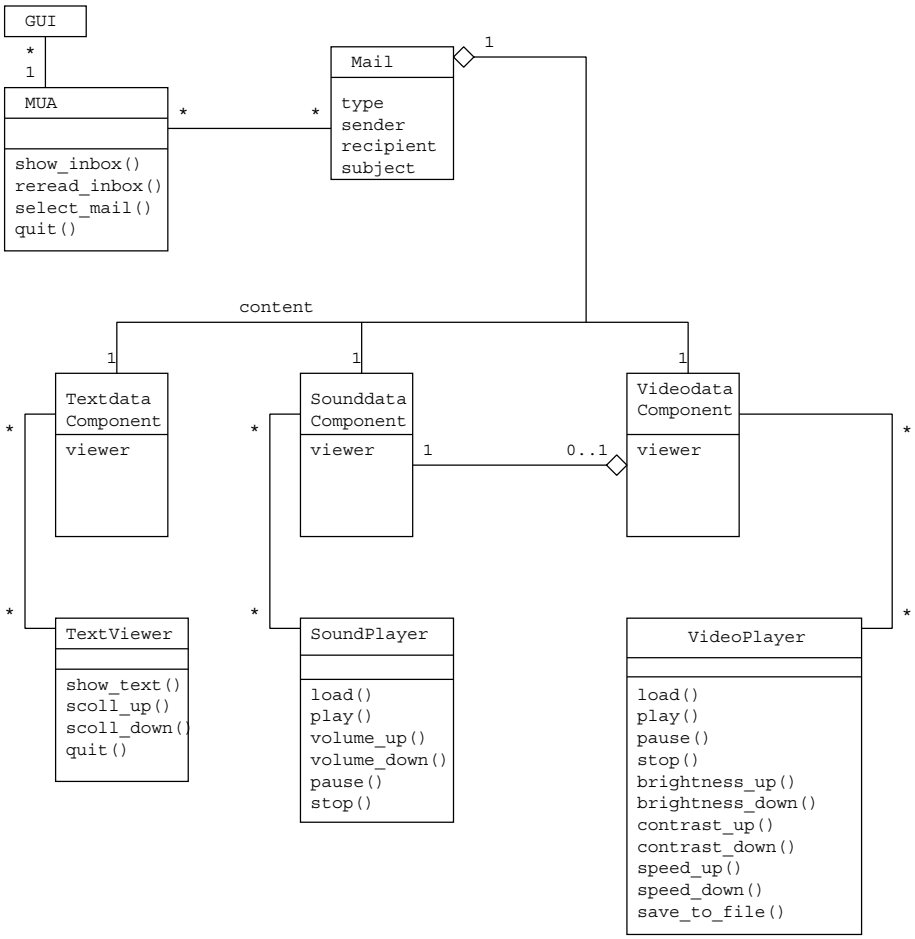

Figure 1: Class Diagram of the Mail User Agent Example

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.

5 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

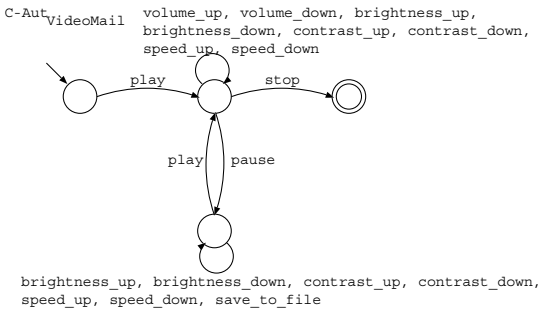

Figure 2: C-Aut of the VideoMail component

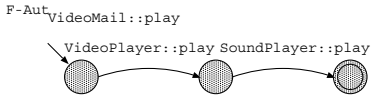

Figure 3: F-Aut of the play-function of the VideoMail component

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 occurance 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-Aut)* and (b) a set of *Function-Automata (F-Auts)*. The C-Aut 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-Aut. This F-Aut describes the set of all possible sequences of calls to internal and external functions this function could perform. The C-Aut and the F-Aut's together form the *use-protocol*. This interface describes all possible call sequences to external functions as a formal language.

All automata, C-Aut and F-Auts, 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-Aut 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-Aut, shown in figure 3.

The advantage of a finite automata based approach is the decidability of the equivalence prob-

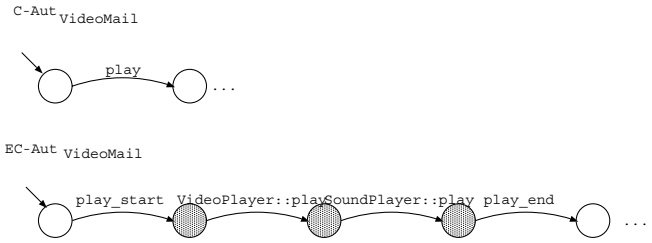

Figure 4: Construction of a EC-Aut out of the C-Aut and the F-Aut of figure 3.

lem 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.

5.1 Type Adaption

Type adaption is used when a component is inserted in a certain context. The automatic adaption problem (cf. section 3) 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.

The automaton of the use interface is built by plugging in the F-Auts into the C-Aut, whereby each transition of the C-Aut is replaced by the F-Aut of the function corresponding to the transition, as shown in figure 4. Since the new automaton is an enhancement of the C-Aut, it is called *Enhanced--Component-Automaton (EC-Aut)*. It describes all possible sequences of calls to external functions. Note that it is also possible to construct the C-Aut of a component, when only the EC-Aut 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-Aut of A and the C-Aut 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-Aut of the component created by coupling A and B, and it contains the new C-Aut$_{A \times B}$. If B fulfills all requirements A posed, then the new C-Aut$_{A \times B}$ is equivalent to the old C-Aut$_A$, 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

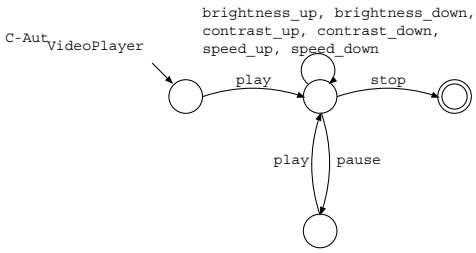

Figure 5: Call interface of the `VideoPlayer`

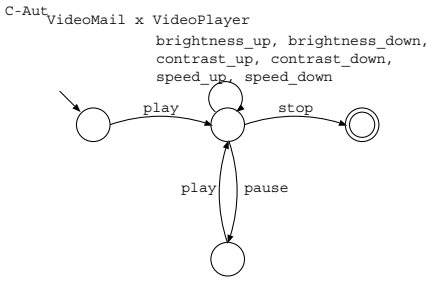

Figure 6: New call interface of `VideoMail` coupled with `VideoPlayer`.

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-Aut of the `VideoMail` component. Consider, that `VideoMail` needs a `SoundPlayer` component and a `VideoPlayer` component. This would be expressed in its F-Aut's, and hence in its EC-Aut. 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 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 avaliability protocol information is required.

5.2 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-Aut of the `MailUserAgent` component, before any extension. In our example this C-Aut is coupled with a GUI. Note the shaded state in the unextended C-Aut. This is the *coupling state*. After the user selected a mail, the C-Aut of the corresponding mail component

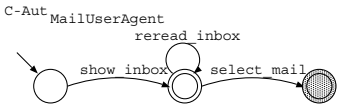

Figure 7: Unextended call interface of the MailUserAgent component.

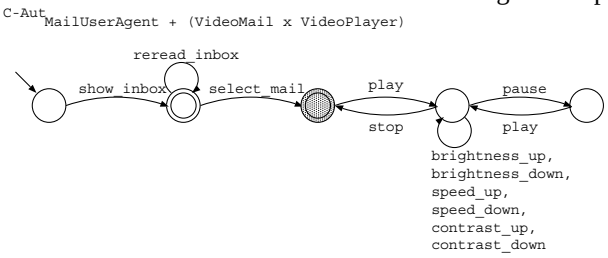

Figure 8: Call interface of the MailUserAgent extended with VideoMail.

is plugged in the unextended C-Aut of `MailUserAgent` at this state. Imagine the user selects a VideoMail (which has coupled with `VideoPlayer` before). Then the C-Aut of `VideoMail x VideoPlayer` (as shown in figure 6) extends the C-Aut of `MailUserAgent`. The resulting extended C-Aut 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.

5.3 Meta-protocol

In this section we sketch a meta-protocol which supports the above type adaption and coupling. The scenario of section 4.2 can be described in terms of our type system as follows.

1. An arrived `VideoMail` asks the run-time system to couple with its supporting infrastructure (i.e., `SoundPlayer` and `VideoPlayer`). The `VideoMail` incluedes in this request also URL's specifying where to get the required components. Additionally the `VideoMail` sends its EC-Aut 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 4.2. We assume that the required `VideoPlayer` is already installed. The run-time system retrieves its current C-Aut from its database, and performs the type adaption algorithm with the EC-Aut of `VideoMail` and the C-Aut of `VideoPlayer`. The new C-Aut of `VideoMail` is extracted from the new EC-Aut. Both (new C-Aut and new EC-Aut) are stored in the database entry associated with `VideoMail`. The new C-Aut 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-Aut, since C-Auts of already installed components are stored in the database.

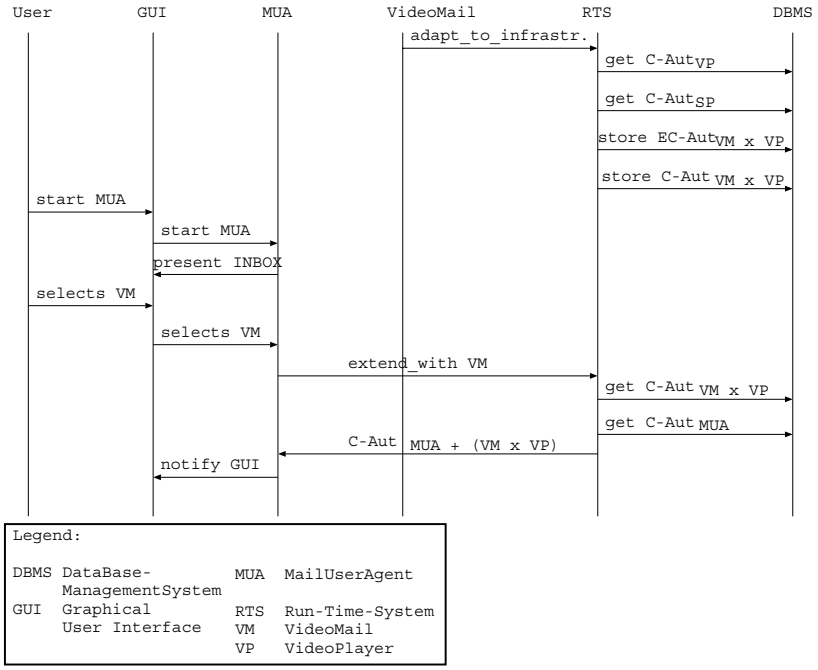

Figure 9: Event-trace diagram of the scenario

6. The run-time system retrieves the C-Aut of `VideoMail` from the database and performs the type extension algorithm with C-Aut$_{MailUserAgent}$ and C-Aut$_{VideoMail \times VideoPlayer}$. The resulting C-Aut$_{MailUserAgent+(VideoMail \times VideoPlayer)}$ 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.

To realize this 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.

6 An approach to solving the pre-compliation problem

As we already mentioned in section 3, reuse units, which do not constitute a complete program and contain generic parts, in general cannot be compiled into binary form. The reason for this problem is a lack of information, i.e., the concrete application contexts, which correspond to the instantiation of the generic parameters, are not or only vaguely known.

The solution to this problem is to generate code for the dynamic determination of the application context. This code has to generate the appropriate instantiation code dynamically. As a consequence, the code generating code constitutes a meta-program acting according to our meta-protocol for dynamic coupling. Since we deal with binary components, this meta-program has to query dynamically for the actual (component) type, in order to accomplish the appropriate instantiation.

Some information may already be available at compile time, because we know the use interface of the generic component. This tells us, how the generic component wants to use the currently unknown (plug-in) component specified by the generic parameter. We can use this information to derive some minimal requirements about the plug-in component.

Now, there are two possibilities to perform pre-compilation of generic components, which should both be incorporated into our coupling and runtime system.

1. We compile as usually done by seperated compilation but additionally include coupling code for the generic parts. This is similar to the dynamic dispatch code in conventional compilers for object-oriented languages like C++. We thus obtain conventional (relocatable) object code. The coupling code is in fact a meta-program obeying our meta-level protocol for coupling. It is thus able to instantiate generic components. Unkown parts are realized by references to objects or components, which are filled in at coupling time.

   We call the points where a component can be inserted into another one *composition points* or *coupling points*.

   This approach prohibits global optimizations, but makes further compilation steps at coupling time unnecessary.
2. We only perform as many analysis steps during compilation as possible without knowing the concrete instantiations of generic parts. All information gained this way, we store in a database, which is part of our compilation system as well as of our coupling and runtime system. This means, we have fragments of a partially analysed and attributed abstract syntax tree, as well as pieces of intermediate or maschine code stored in our database, for example. Instead of demanding a database at the customer site, the corresponding information may be stored in object files. Retrieving information may then again be accomplished by an appropriate meta-level protocol, using reflection for example.

   When components are coupled and we thus know the concrete form of a generic parameter, we perform a corresponding compilation step exploiting the new information. We thus compile incrementally. This step may again be specified by a meta-program.

   The advantages of this approach are

   - the possibility for global optimizations,
   - the possibility to reuse fine-grained information, and
   - the uniform handling of aspect-oriented programming, meta-programming and other modern program development and manipulation techniques and paradigms, as will be shown in [Heu].

To illustrate the approaches, we take a further look at our mail user agent example of section 4.2.

Suppose, the class `Mail` is realized as shown in figure 10. We tag the type of `content` to be generic and thus define a composition point.

Following the seperated compilation approach 1, we compile this class as usual and implement `content` as as reference to an unknown object. At coupling time, that is, when we know the concrete type (and interface) of `T`, our coupling mechanism recognizes, that `content` has to be realized by the mail content component of type `T` offered for coupling. Our mechanism thus enhances the public interface of `Mail` by the public functions of the `content` component and redirects every call to these functions to the corresponding functions on `content`. In general, problems arise from name conflicts, occuring when several components with non-disjoint function names are plugged into the same other component. But the usual name conflict resolving techniques do a good job here, too.

Following the incremantal compilation approach 2, we finish compilation, when we know all participating components. This complicates or even impedes “real” dynamic coupling, but it works fine with “half-dynamic” coupling and allows for global optimizations.

```
class Mail
{
  @generic T;

  public MailParticipiant getSender();
  public MailParticipiant getRecipient();
  public String getSubject();
  public abstract getContent();

  private MailParticipant sender;
  private MailParticipant recipient;
  private String subject;
  private T content;
}
```

Figure 10: Implementation of the class `Mail`

7 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.

8 Conclusions

We have presented our view on component technology and explained the properties and mechanisms necessary to make it useful, easy and comfortably applicable. We outlined the problems of current approaches and sketched our ideas to tackle those problems. We showed that dynamic coupling of binary components is the method of choice for easy, efficient and high-quality reuse. To support this method, we introduced a typing system for components, a meta-protocol for necessary adaptions and sketched two ways to obtain binary components despite of the problems arising from genericity.

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.

[Heu] Dirk Heuzeroth. Eine Software-Architektur für flexible Übersetzer für Sprachen mit modernen Programmierparadigmen. PhD thesis in preparation.

[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.

[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.