Dynamic Types for Software Components

Ralf H. Reussner

Chair Informatics for Engineering and Science
University of Karlsruhe
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
++49 721 608 7504
reussner@ira.uka.de
http://liinwww.ira.uka.de/reussner

Abstract
Component based software development requires run-time coupling of binary software components. We present a type system for software components, which allows to check for certain errors when coupling binary components. A unique feature of our type system is the dynamic adaption of a component’s type according to the environment the component is plugged in.

Keywords
software components, dynamic composition, type system, introspection.

Introduction
A defining property of components is their applicability in various contexts [6]. 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'.

Problems of current technology
Generally, problems with the 'adding functionality strategy' are: firstly, it is difficult to formulate 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 a software component’s reusability, since the added functions translate into new requirements to the environment where 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 above problem we call the 'functionality-reuse 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 plug-in using component. The functionality of the plug-in does not really appear as new functionality of the plug-in using component at its interface. For example, this would be necessary when a user-interface has to adapt after inserting a plug-in in a component collaborating with the user interface. This is the 'type-extension problem'.

Our solution
We tackle both problems, the 'functionality-reuse problem' of the 'adding functionality strategy' and the 'interface-extension problem' of the 'dynamic enhancement strategy', with a new type system for components. According to Nierstrasz a type should describe the applicability of the typed entity. So the type of a component
describes which service is available under which conditions [4]. Two conditions for the availability of services are: (a) another service has been called before (e.g., calling an initialization routine is a prerequisite for using other services), and (b) services form certain other components are available. Point (a) is the protocol of allowed calls (the 'call protocol') of a component A. To handle condition (b) we additionally have to know how a component B uses a component A ('use protocol').

Basically these two protocols form the interface of a component. An important feature of our approach is the linkage between these two interfaces of a component. This linkage is the basis for two algorithms: the 'type coupling algorithm' and the 'type extension algorithm'.

Benefits

The 'type coupling algorithm' allows us to solve the 'functionality-reuse problem'. Assume component B uses functionality of component A. Given the call protocol of A and the use protocol of B, we can not only check whether A is suitable for B or not (this can be done with traditional protocol checking [2]). Additionally, our 'type coupling algorithm' allows us to compute a new call protocol for B in cases where A fails to meet all requirements of B. An extreme example of this would be the total absence of B. In that case the new call protocol of B would be computed so as to maximize the services that B can offer without A. As explained above, this is the common case when applying the adding functionality strategy; in a certain context only a certain fraction of a component's functionality is required.

The 'interface extension problem' is handled by the 'type extension algorithm', where at certain defined 'points' the calling interface of a plug-in is inserted in a defined way into the calling interface of the component using the plug in.

Since the calling interface is considered as a part of the type, and it can change during the compile-time (which not necessarily is the compile-time), we call our type system "dynamic types for software components".

Our type system bases on deterministic finite state machines. Opposed to common dynamic modeling of objects with state machines (e.g., Statecharts[1], SDL[7], UML[5]), where the behavior of objects is specified, we use state machines to describe the availability of services. The advantage of state machines is the usability of elaborated theoretical results for finite automata [3] and protocol validation [2] and the possibility to merge automata theoretic algorithms with graph algorithms.

Example

On the poster we present a realistic example (mail user agent), where the 'functionality-reuse problem' and the 'interface-extension problem' arise. A mail user agent has to handle several types of mails (e.g., video mails, sound mails, text mails). Since during design of the mail user agent not all future mail type can be anticipated, the 'interface-extension problem exists'. Each mail needs a viewer specific for its type. This viewer is not a part of each mail. Instead, the mail couples with an available fitting viewer during run-time. On the one hand, a mail should not request to much functionality of its viewer (because this would narrow the range of usable viewers). On the other hand, a mail would like to make use of a viewers functionality. So the functionality-reuse problem arises.

Concrete type definitions of all components are shown and the underlying algorithms for coupling and extension are demonstrated.

Acknowledgments

I would like to thank Dirk Heueroth, Thomas Worsch, and Kathrin Paschen for many fruitful discussions.

References