Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 105–124
www.elsevier.com/locate/entcs

Only the Best Can Make It: Optimal Component Selection

Lars Gesellensetter[1] Sabine Glesner[1]

[1] *Institute for Software Engineering and Theoretical Computer Science, Technical University of Berlin, Germany*
Email: {lgeselle|glesner}@cs.tu-berlin.de
URL: http://pes.cs.tu-berlin.de/

Abstract

In Component-based Software Engineering (CBSE), the construction of cost-optimal component systems is a nontrivial task. It requires not only to optimally select components and their adaptors but also to take their interplay into account. In this paper, by employing methods from the area of compiler construction and especially optimizing code generation, we present a unified approach to the construction of component systems, which allows us to first select an optimal set of components and adaptors and afterwards to create a working system by providing the necessary glue code. With our two case studies, we demonstrate that our approach is efficient and generally applicable in practical scenarios.

Keywords: component-based software engineering, component selection, adaptor code generation, term rewriting, cost functions

1 Introduction

Component-based software engineering (CBSE) has emerged as a method of choice to cope with more and more complex software systems and constantly increasing application fields. CBSE requires software functionality to be encapsulated into software components which in turn can be reused in further systems. In this paper, we address the problem of generating component systems from specifications of individual components. In general, there may be different alternative components that can be selected, each coming at their own cost. We aim at a selection method that guarantees the optimality of the generated component system. Especially in the presence of huge component repositories, the best solution is not trivial to obtain.

We require our solution to fulfill the following requirements: The components selected for the target system shall be optimal. In the classical case, the cost measure mirrows the speed of the target component system. In addition, we want to be able to deal with arbitrary cost functions, in particular with those that are relevant in the area of embedded systems, e.g. code size or power consumption. Moreover, we

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.02.034

require our method to be able to build component systems from existing components without modifying the components themselves, i.e. by regarding the components as black boxes. Merely the adaptor code shall be generated to connect the selected components into a working target system.

Our solution approach transforms the problem of constructing component systems from existing components to the problem of code generation in compiler backends, which is typically tackled with term rewriting methods. During code generation, we have to find an optimal sequence of machine instructions that not only accomplish the computations specified in the intermediate compiler representation but also adhere to the constraints imposed by the target processor, e.g. limited number of registers or functional units. This code generation problem in compilers can be solved by representing the program as a tree that can be reduced in bottom-up order while simultaneously generating semantically equivalent machine instructions for the reduced intermediate operations. Here in the context of component-based software engineering, we need to select components and adaptor code such that the generated component system offers the required services at optimal cost. For this purpose, we specify the desired behavior of the target component system as a term. Rewrite rules map the required services in the specification to concrete components. In the optimal solution, all required services are bound to components without conflicts at minimal cost. Moreover, the components are connected by appropriate adaptor code, if necessary.

Our work is a major contribution to the field of component-based software engineering as it improves reusability of components and automation in the construction process of component systems. Moreover, our method is generally applicable, not only for software components but also for hardware components or in the area of hardware/software-codesign as well. In all these cases, components are described by the functionality that they offer and need to be selected depending on their individual cost and on the context of the overall system in which they are employed. Adaptor code (which can be both software or hardware) may be needed for the integration of the selected (software or hardware) components.

In this paper, we discuss two case studies. The first considers the situation when designing the electronic system of a car. Typically, several subsystems are needed, let us assume, we need e.g. a navigation system, a CD player and a car theft protection system. Each of these subsystems can be realized in different ways, each coming at its own cost. For example, a car theft protection system may work by constantly sending the current position of the car. In this case, a GPS signal sensor is necessary. The cost for this design decision can be compensated if the GPS sensor is used by other subsystems as well, e.g. by the navigation system. In general, the overall cost of the car system depends not only on the costs of the individual components but also on their interplay. We discuss the car scenario in more detail in Section 7. Our second case study considers a chatterbot system realizing a virtual partner for talks as an entertainment application for mobile phones [8]. This system offers a range of services, from a user interface to a dialog system to a visualization component. For each service, different realization possibilities of different costs

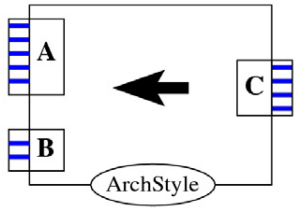


Fig. 1. A component

exist. In Section 7, we show how our construction method can automatically select components such that the resulting component system offers all required services at optimal cost.

This paper is structured as follows: In Section 2, we introduce our notion of a component. Based on this definition, Section 3 gives an overview of our approach of component selection and integration. Section 4 deals with preliminaries concerning term rewriting and optimal code generation, which we need to explain our concept in detail. Section 5 gives a comprehensive presentation of our method. In Section 6, we describe the implementation of our approach, and Section 7 presents the introduced case studies in more detail. After discussing related work in Section 8, we conclude in Section 9.

2 Component Model

We describe system and component behavior by the means of *services*. A service encapsulates a certain functionality and is formally described by the interface of the methods that implement it. *Each method is identified by its name and a list of named arguments. At the moment, we do not consider typing issues and consider the arguments simply as strings.* Furthermore, we assume a black-box component model and describe components by four characteristics. Components are specified by the services that they provide (*provided services*) as well as by the services that they require in order to work properly (*required services*). Moreover, each component may pose *architectural constraints* (the required underlying platform in case of software components, communication standards, etc.). Furthermore, certain *component properties* (utilized for the cost measure) are known about the components, e.g. their code size, their energy consumption or their price. Figure 1 shows a component: A and B are the provided services and C is the required service. ArchStyle denotes the architectural constraints.

We assume the following situation: Given a repository of components and a specification of the component system that we want to construct, we need to choose components and to connect them, if necessary, by suitable adaptor code such that the target component system fulfills the specification.

For this, we need a specification, if and how services can be mapped to each other. As illustration, this information can be thought of as (but not necessarily implemented as) a huge table, as depicted in Table 1. In the table, we see such a mapping, specified for a system of n components providing m services. The m

Components/ Services		C_1		C_2	$\cdots$	C_n	
		R_1	R_2	R_3	$\cdots$	R_{l-1}	R_l
C_1	P_1	–	–	–	$\cdots$	–	M_{1l}
	P_2	–	–	–		–	–
	P_3	–	–	–		–	–
C_2	P_4	–	id		$\cdots$	–	id
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\ddots$	$\vdots$	
C_n	P_{k-1}	–	$M_{k-1\,2}$	–	$\cdots$	–	–
	P_k	M_{k1}	–	–		–	–

Table 1
Specification of the Mapping

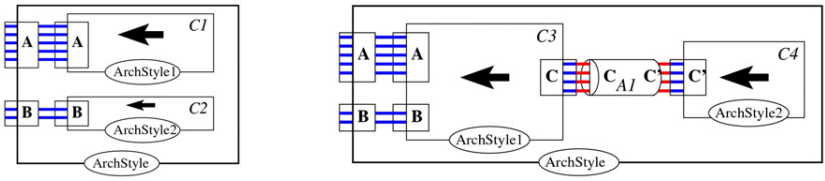
services can be subdivided into k provided services (denoted as P_i) and l required services (denoted as R_j). Talking about adaptability, we always think of a provided service that is mapped to a required service. Such a pair of services can be incompatible, adaptable or identical. Adaptable means that the provided service's methods can be mapped onto the required one's. The mapping does not need to be one-to-one, and method names as well as arguments can be transformed. As we can see in the table, P_4 and R_2 as well as R_l are identical and hence can directly interoperate. Service P_k can be adapted to service R_1, and the same hold for P_{k-1} and R_2, and for P_1 and R_l, respectively. In all other occasions, adaptation is not possible, the services are incompatible, indicated by a dash in the corresponding table cell [1]. From the specification, we can see that the required service R_2 can be provided directly by service P_4 as well as via adaptation by service P_{k-1}. Depending on the costs of each alternative, one has to decide which choice to make.

In our implementation and in the case studies described in this paper, the mapping is efficiently given by a functional specification, from which the adaptor code can be generated by generic rules. The advantage is that these rules have to be specified only once for each considered architecture.

In case of several possibilities, we are interested in the optimum. We describe a component system (that we aim to construct) by the desired services. Additionally, we can define architectural constraints. Then, components have to be chosen according to their provided services. In the process of system construction, the selection of a component might entail the selection of another one, since not all required services may be available. It has to be affirmed that the architectural constraints of the selected components fit to each other and to the initially given system constraints. Additionally to the given constraints, new constraints may evolve dynamically during the selection process. To select the optimal choice of components, we need a notion of optimality. To this end, a cost function has to be given. Usually, it assigns costs to the components by taking their properties into account. Moreover, the cost function assigns costs to the adaptor code that connects components.

Different services may be functionally equivalent but nonetheless might their interfaces be not directly compatible. In this case, an adaptor specification states how the interface of one service can be mapped to the interface of another one. If an adaptor maps the interface of a service A to the interface of service A', it can be

[1] For certain fields of the table, no mapping is allowed or reasonable to exist: Provided services of a component must not be mapped to required services of the very component. If this was the case, the corresponding services are redundant, anyway.



(i) (ii)

Fig. 2. Possible component choices for a system description

regarded as a component which requires service A and provides service A'. From the functional adaptor specification, an instantiation of the adaptor can be generated. This concept of adaptors allows us to make two components interoperable without intrusion, i.e. without changing the components, and, hence, fits well into our model of black-box components.

For the description of target component system to be constructed and the repository of available components, we specify the following information:

(1) the description of the services in the target component system, specified by their interface descriptions,

(2) the component specifications, which consist of the provided and required services as well as the architectural constraints and the respective properties defining the cost measure,

(3) the adaptor specifications, which specify for each pair of components if and by which adaptor code their services (i.e. their interfaces) can be mapped onto another.

As an example, consider Figure 2: The system has to provide services A and B. Depending on the given repository, we might have components at hand, which provide service A and B, respectively. If their constraints match each other and also the initial constraints, we have one solution as depicted in Figure 2(i). However, we might choose a single component that provides A and B, but which requires another service C. Then it can be the case that we only have a component available that provides service C'. In case we have an adaptor specification for the mapping from C' to C, we arrive finally at the solution depicted in Figure 2(ii). It depends on the cost function which solution is preferable. If e.g. the left component in Figure 2(ii) is cheaper than both components in Figure 2(i) together, the second choice can be better. However, the cost of having to select an additional component for service C might even out this benefit or even lead to a suboptimal solution. In the following section, we give an overview about our approach to the construction of cost-optimal component systems.

3 Constructing Optimal Component Systems by Term Rewriting

In this section, we give an overview about our approach, especially about the idea of employing methods from the area of compiler construction. We start with the following concrete problem: We assume that the desired system behavior is given in form of the wanted services together with a specification of the available services, components and adaptors together with their cost function and the architectural constraints. Given this description, we want to find an optimal set of components that fulfill the requirements and achieve interoperability (by regarding the constraints and generating adaptors).

In general, this problem is NP-complete [10,11] so that we cannot hope for an algorithm that always solves each instance of this problem efficiently, i.e. in polynomial time. Nevertheless, we can investigate heuristic strategies that may help to find optimal solutions in many practical cases. For this purpose, we map our problem to the problem of optimal code selection in compiler backends. This is especially useful as in both cases, we have a description of the system behavior from which we need to generate the target system. In case of component systems, the description specifies the desired services. In case of compiler backends, it expresses the desired functionality as intermediate compiler representation. In both cases, we have parts of the system which achieve a certain functionality (components or machine instructions of the target platform, resp.). In both cases, the mapping is rather $1 : n$ than $1 : 1$ since a component can provide many services, and likewise a machine instruction can perform a composed functionality[2]. Finally, in both cases we are interested in the optimal solution. Hence it is a promising approach to transfer solutions from the code generation problem to component selection.

For optimal code generation, one standard technique is term rewriting. It finds provably the optimal solution. In term rewriting, we start with an intial term and try to rewrite it in bottom-up order to a goal term, by using a set of given rewrite rules. Whenever a rewrite rule is applied, semantically equivalent machine code is emitted. The total sequence of emitted machine code makes up the target machine program. Each rule is adherent with a cost. In code generation, bottom-up term rewriting is used, to both restrict the search space and ensure the correct ordering of the generated instructions (cp. [9]; [1] proves optimal code generation for expression trees with shared subexpressions as NP-complete). Since there are usually many ways to rewrite a term, we deal with a classical search problem. Heuristic search is a valuable mean to cope with the complexity of the search space. The A*-search guarantees to explore a minimal part of the search space, while still finding the optimal solution, in the sense that no other algorithm searching for the optimum is guaranteed to expand fewer nodes than A*, cf. [6,3]

To model the problem of component selection as term rewriting problem, we

[2] Consider complex load operations, which perform addition and multiplication to determine addresses. Hence, a load with address calculation can be mapped to multiple simple commands or to the complex load instruction. Another example is multiplication by 2 which can be realized by a multiplication operation or by the much cheaper shift-left operation.

rule "select GPSLoc" {
match: $carsystem(\ldots, \triangle^{ATP}, \ldots)$
replace: $carsystem(children') \mid$
where children' is obtained by replacing $\triangle^{ATP}$ *by* $GPSLoc^{ATP}$ *and inserting* $\triangle^{GPS-sensor}$, *if none of the children has type* GPS-sensor
cost:50 }

Fig. 3. Example of a rule for component selection

state the desired system behavior as a term. The rewrite rules correspond to selection of a component (together with its provided and required services) or of an adaptor. The cost function for each rule expresses the cost of the selected component. Hence, we denote a system specification in the following way:

$$system_{constraints}(\triangle^{A}, \triangle^{B})$$

This term represents a system, which should provide services A and B under the given architectural *constraints*. The $\triangle$ denotes that the given service is still unbound. During the rewrite process, each service will be bound to certain components, and further services might be introduced, if they are required by a selected component. At the end, we arrive at a final state, namely a term in which all components are bound. The collected information during the rewrite process tells us which components to select and how to achieve interoperability (by generating adaptors).

Reconsidering our previous example, we can specify the desired system behavior by the following term:

$$carsystem(\triangle^{anti-theftprotection}, \triangle^{stereo}, \triangle^{navigationsystem})$$

One rewrite sequence could be (abbreviating the services for convenience):

$$\begin{aligned} & carsystem(\triangle^{ATP}, \triangle^{stereo}, \triangle^{NavSys}) \\ \rightarrow\ & carsystem(GPSLoc^{ATP}, \triangle^{stereo}, \triangle^{NavSys}, \triangle^{GPS-sensor}) \\ \rightarrow\ & carsystem(GPSLoc^{ATP}, CDman100^{stereo}, NAV200^{NavSys}, \triangle^{GPS-sensor}) \\ \rightarrow\ & carsystem(GPSLoc^{ATP}, CDman100^{stereo}, NAV200^{NavSys}, GPS10^{GPS-sensor}) \end{aligned}$$

Note the introduction of a new service in the first rewrite step. For the first rewrite step, the applied rule is shown in Figure 3. We see that the rule is applicable, since we have an unbound service of type ATP [3]. As a result of the rule application, the service ATP is bound to *GPSLoc*, and $\triangle^{GPS-sensor}$ is added as new required service, since it appears neither as bound nor as required service in the current system descripton. The remaining rewrite steps are performed similarly by corresponding rules.

After the general presentation of our concept and some motivating examples, we are now ready to describe our approach in detail. To this end, we introduce some preliminaries in Section 4, which we need for our concept, namely term rewriting and A* search. Having established these prerequisites, Section 5 presents our approach.

[3] In this example, there are no further architectural constraints.

4 Term Rewriting and A*

In our approach, we use the idea of coupling term rewriting with A* as proposed by [9]. This guarantees an efficient search. We give a short overview of term rewriting with respect to cost functions, and introduce then the concept of A* search. Finally, we describe the coupling between both and discuss complexity issues.

4.1 Term Rewriting

Term rewriting [2] is a standard technique for code generation. Terms are defined inductively in the usual way: We start with a finite set of function symbols $\mathcal{F}$ which is called *signature*. Each function symbol has an *arity* given by the function $r : \mathcal{F} \to \mathbb{N}$. Function symbols with arity 0 are also called *constant symbols*. We add a set of variables $\mathcal{V}$ ($\mathcal{F} \cap \mathcal{V} = \emptyset$). The set of terms, denoted as $\mathcal{T}(\mathcal{F}, \mathcal{V})$, is defined in the usual inductive way. For a term t, $Var(t)$ denotes its variables. If $Var(t) = \emptyset$, t is called a *ground term*.

Definition 4.1 (**Costed term rewrite system**) A costed term rewrite system (CTRS) is defined as a triple $((\mathcal{F}, \mathcal{V}), R, C)$ with

- $\mathcal{F}$, a non-empty signature
- $\mathcal{V}$, a finite set of variables
- R, a non-empty, finite subset of $\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$
- $C \in R \to \mathbb{R}^+$, a cost function

satisfying for all $(l, r) \in R$: (i) $l \neq r$, (ii) $l \notin \mathcal{V}$, (iii) $Var(r) \subseteq Var(l)$ ◇

We enrich the notion of a term by allowing arbitrary arity for certain function symbols (to model lists adequately) and attach a type[4] and attributes to the function symbols. Besides, we extend the rules by preconditions, which can refer to any of this information. Note that these extensions do not introduce additional complexity, since they can be straightforwardly expressed in classic term notion (model attributes as children, model n-ary function symbols by introducing n new function symbols and modifying the rules accordingly). We extend and modify the given definitions by the following:

Definition 4.2 (**n-ary Terms**) We extend the domain of the arity function to $\mathbb{N} \cup \{*\}$, $*$ meaning that the arity is arbitrary but finite. The inductive definition of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ has to be modified accordingly. ◇

Definition 4.3 (**Attributed Terms**) We introduce a function $t : \mathcal{F} \to T$ that assigns a type to each function symbol. The attributes of a function symbol are determined by its type. To each function symbol in a concrete term, an attribute function is associated that returns the attributes of the term. ◇

[4] Our notion of type is for now very basic and mainly denotes a special attribute, referring only to the given function symbol (and not the complete subterm).

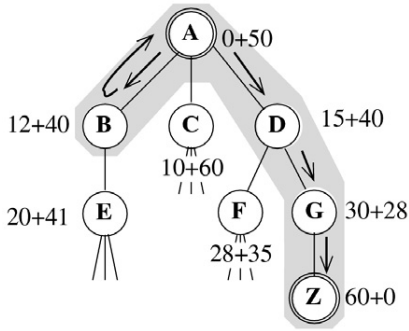


Fig. 4. Search space and visited nodes for A* search

The type information allows us to write more generic rules. E.g. in the context of code generation, the function symbols for the arithmetic operation + and − would be both of type *BinOp*. This allows us to write a generic rule dealing with binary operators.

For notation, we write the type of a function symbol superscribed and its attributes subscribed. If the type is irrelevant or clear from the context, we simply omit it, as we do with the attributes. A component *comp* of type *type* with attributes $a1, a2$ is hence denoted as follows:

$$comp^{type}_{a1=val1,a2=val2}$$

A rewrite rule specifies how terms can be rewritten, which preconditions have to be fulfilled and which costs arise. Additionally, it can be stated which code should be generated.

In general, there are many ways to rewrite a given term. To find the optimal rewrite sequence, all different possibilities have to be considered. The complexity of the search space has motivated the utilization of heuristic search methods, whereof the A* search is an instance.

4.2 A Search*

A search can be guided by a heuristic function to approximate the cost for the remaining transitions. The A* search ([6]) guarantees minimal exploration (w.r.t. the heuristic function) of the search space while finding an optimal solution, given that the heuristic function is *admissible*. *Admissible* means that the function always underapproximates the actual cost. If we take the zero function as heuristics, we end up with the same behavior as uninformed search. The more precise the heuristic function predicts the costs, the less expensive the search will be. The heuristics provides a valuable mean for dealing with the problem of combinatorial explosion.

A common and descriptive example for search is the problem of finding an optimal route from one location to another. The nodes correspond to the locations, and the edges represent connections, e.g. streets. The cost of a given edge is its distance. The heuristic function for a node defines its distance (here meant literally)

to the goal node. Intuitively, we choose air-line distance[5]. As example, consider Figure 4. We would like to travel from A to Z, and our search tree consists of all possible paths starting from A and possibly leading to Z. At every node, we annotate the accumulated cost (the distance travelled so far) and the estimated remaining cost (how far is it). A* is a breadth-first search, i.e., we consider all possibilities but look at the cheapest first. For the given example, we first try B, since $12 + 40 = 52$ is the cheapest choice. However, having reached B, we have a cheaper alternative to E. Hence backtracking occurs and we continue at D. The remaining search leads straightforwardly via G to the goal Z, encountering an actual distance of 60 instead of the initially assumed 50. The shaded area in Figure 4 depicts the visited section of the search space.

*4.3 Coupling Term Rewriting with A**

It is a general search problem to find the optimal solution for a given term. The search space contains all terms, the transitions are given by the rules. If we take A* search with an admissible heuristic function, it is guaranteed that we find the optimal solution in minimal time with respect to that heuristics. This application of A* to term rewriting in the context of code generation in compilers has been initially proposed by [9].

4.4 Complexity

As mentioned before, selecting a subset of components for a given set of services is NP-complete (this problem is in NP and is reducible to SAT), even without the constraint to find an optimal solution. Considering term rewriting, in general it is undecidable whether a given term is reachable (corresponding to finding a solution), even termination alone is undecidable[6]. Restricting the rewriting to ground terms (i.e. forbidding variables in the rules), reachability of a goal term is decidable. Least-cost instruction selection on DAGs is NP-complete [1]. While ground term rewriting has to deal with the problem of combinatorial explosion, it nevertheless yields a constructive algorithm for finding a solution, and by restricting the class of allowed rules and appropriate reduction of the search space, the problem becomes feasible for practical scenarios.

5 Composing the System: The Algorithm

The composition of the system is realized in two steps: First the component selection is accomplished (which includes selection of adaptors). In the second step, the required adaptor code is generated. The system specification is represented as a term, and component selection as well as adaptor code generation are performed by term rewriting. The rules are partly general (especially for the adaptor code generation), and partly specific for the concrete system (especially for the component

[5] Trivially, this heuristics always underestimates the actual cost.

[6] This holds even for quite simple systems with only one rule; shown by reducing it to the word problem.

rule "general rule for component selection" {
  cond: $constr(System) \wedge constr(C)$ satisfiable
  match: $system[services] \mid \exists \triangle^S \in services : S \in prov(C)$
  replace: $system[services']$, *where* $services' =$
    $\{C^{S_{prov}} | \triangle^{S_{prov}} \in services \wedge S_{prov} \in prov(C)\}$
    $\cup \{\triangle^{S_{req}} | \_^{S_{req}} \notin services \wedge S_{req} \in req(C)\}$
    $\cup \{\hat{C}^S \in services \mid (\hat{C} \neq \triangle) \vee (S \notin prov(C))\}$
  assert: $constr(System) \wedge constr(C)$ satisfiable
  cost: $cost(props(C))$
}

Fig. 5. Generic rule for component selection

selection). However, the specific rules are fully automatically generated from the system specification.

5.1 Component Selection

A system description consists of the descriptions of the components, of the mapping from provided services to required services, and of the desired functionality. Additionally, a cost measure has to be specified.

Generally, in describing a system, the relevant services have to be identified. By identifying we mean that the functionalities of the components are grouped into self-contained, meaningful aggregates referred to as services.

The description of the system behavior is mapped to a term, the arguments of which specify the required services. In the rewrite process, as components are selected, the corresponding provided services are bound accordingly, and new services might be introduced. Remember that selecting a component entails that all services in the system specification that are provided by this component are bound to it, given they were unbound before. To this end, component and adaptor descriptions are mapped to rewrite rules. The rules generated from the adaptor specification replace a service with another one. Therefore adaptors can be seen as a special case of a component. Hence in the following, if we talk about component selection, we equally mean adaptor selection. The generation of corresponding adaptor code is performed in the next step.

As example, reconsider the following term from our running example:

$$carsystem(\triangle^{ATP}, \triangle^{stereo}, \triangle^{NavSys})$$

This term describes a system specification, which states that the three services anti-theft protection, stereo, and navigation system are required. Each required service is modelled as argument of the root, whereas the type of the argument denotes the required service, and the name states which component (if any) is bound to the service. The special name $\triangle$ means that the service is still to be bound.

The selection of components is achieved by the application of rewrite rules, which are automatically generated from the component specification. A rule for a component C with required services $req(C)$, provided services $prov(C)$, constraints

$constr(C)$ and properties $props(C)$ is depicted in Figure 5 (with *cost* being the given cost function, built upon the properties). A rule is constituted by a condition, a matching expression, a replace expression, an assert, and a cost function. For a rule to be applicable, its condition has to be fulfilled, and its match expression has to match the considered term. For component selection, the condition states that the constraints of the component and the current system constraints are consistent. The match expression states that there should be an unbound service, which is provided by the considered component. If an applicable rule is selected and applied, the matched expression is replaced with the replace expression. In case of component selection, the list of services which constitute the system behavior is modified. Unbound services which are provided by the considered component are bound to this component. Services which are required by the component are added as unbound services, unless they are already bound in the previous system configuration. And finally, services which neither are unbound nor are in the list of provided services of the considered component are simply kept. This yields a new list of services (compare the replace-expression in Figure 5). Then, after the application of the rule, we can assert that the constraints of the component and the current system constraints are consistent.

Example 5.1 If we recall the system in fig. 2(i), this solution might be reached by the following steps: select component C1, select component C2. This is represented by the following rewrite sequence:

$$\begin{aligned} & system_{arch=i386}(\triangle^A, \triangle^B) \\ \rightarrow\ & system_{arch=i386}(C1^A, \triangle^B) \\ \rightarrow\ & system_{arch=i386}(C1^A, C2^B) \end{aligned}$$

On the other hand, the solution in Figure 2(ii) could be established by first selection component C3, and then choosing component C4 (after choosing the adaptor A1), which is expressed as the sequence:

$$\begin{aligned} & system_{arch=i386}(\triangle^A, \triangle^B) \\ \rightarrow\ & system_{arch=i386}(C3^A, C3^B, \triangle^C) \\ \rightarrow\ & system_{arch=i386}(C3^A, C3^B, A1^C, \triangle^{C'}) \\ \rightarrow\ & system_{arch=i386}(C3^A, C3^B, A1^C, C4^{C'}) \end{aligned}$$

Note that the second rewrite step (changing C to C') corresponds to the selection of adaptor A1, and demonstrates that adaptors are dealed with as components. ⋄

Since all rewrite steps come at a certain cost, first the best options (i.e. the ones with the lowest costs) should be considered. If we specify a heuristics, we can optimize the search. The A* search requires a heuristic function, which estimates the cost from a given term to the goal term. Note that the function must always underestimate the actual cost to guarantee the optimality of the A* search. If we know that each service comes at least at a cost of c_{min}, a simple heuristics would be the following: From a given term with n unbound services, the expected cost is

at least $n * c_{min}$. Although this heuristics seems quite simple, it is very effective in reducing the number of visited nodes.

Figure 6 specifies our algorithm for component selection. Our search space does not only contain the mere terms but instead terms with information about the current state of the rewrite process annotated, called *xterms*. For each xterm t, we have the following:

- **term**(t): the current term
- **constr**(t): the currently posed constraints
- **seq**(t): the rewrite sequence encountered so far, each step consists of the application of a rule r at a position p
- **estcost**(t): the estimated total cost of the term (contains actually encountered and estimated remaining cost)

Additionally, we have a heuristic function $heur : xterm \rightarrow \mathbb{R}^+$, which estimates the remaining costs for a node.

The algorithm works as follows: We keep a list of currently active xterms. In each iteration step, we select the minimum (with respect to the cost function) of this list and replace it with all successors which could be generated by application of a rewrite rule. We finish if we reach a goal term, in which case we report success, or if no terms are remaining to be examined, which means failure. If we end up with a goal term, the selection of the minimum in the loop guarantees that we found the optimal solution.

The determination of the successors of a xterm t considers all rules and all positions of t to find possible rewrite steps. If a rule r matches at a given position p, and if the constraints of the rule meet the current system constraints, according to the replace-expression a new term t' is generated. Additionally, we extend its rewrite sequence, its constraints, and its estimated cost. Since we only remember the estimated cost, a little calculation has to be done to this end: The previously encountered costs of t result to $estcost(t) - heur(t)$. Therefore at node t', to get the actually cost we simply have to add the cost of the rule application. By finally adding $heur(t')$, we arrive at the estimated cost for t', as we see in the line denoted with $[*]$.

Having esablished the algorithm for component selection, in the following we sketch how the generation of adaptor code works.

5.2 Adaptor Code Generation

After we have selected components and also adaptors (as a special case of the former) in the first step, in the second step we generate code for the chosen adaptors. For this, we need the adaptor specification which states how the interfaces of two services can be mapped to each other (cp. Table 1). The description covers binary directed mappings. The adaptor specification can be rewritten by general rules to the desired target platform.

For the specification of an adaptor, we need a notion of how communication

```
func selectComponentsAndAdaptors: (t:XTerm) → XTerm

  func filter (terms:XTerm list, newterms:XTerm list) → XTerm list
    (* eliminate redundant xterms in newterms (w.r.t. terms)
       redundancy is meant modulo equivalence of rewriting sequences *)
  func result (term:XTerm) → ResultType
    (* returns the desired result. This can be the list of generated code,
       or (in the case of component selection) the final term that
       provides all information about the component selection.*)

  func getNextTerms (t:XTerm) → XTerm list
    newTerms=[];
    for all rules r=(cond,match,replace,assert,cost) do
      for all positions p of term(t) do
        if "match matches term(t) at position p and
            (constr(t) ∧ cond) is satisfiable" then
          t'=t;
          "modify term(t') according to replace, add (p,r) to Seq(t'),
            set estcost(t')=estcost(t)−heurCost(t)+cost+heurCost(t'),     [*]
            add cond to constr(t')"
          newTerms=newTerms ∪ [t'];
        endif
      endfor
    endfor
    return newTerms;
  endfunc

  var terms: XTerm list;
  terms=[t]; selected=t;
  while (terms≠[]) and (not (goalTerm(selected))) do
    newTerms=filter ( getNextTerms(selected) );
    terms=(terms \ [selected] ) ∪ newTerms;
    if (terms ≠ []) then
      selected=selectMinimum terms;
    endif
  endwhile
  if (goalTerm(selected)) then (* we found the optimal solution *)
    return result(selected);
  else return failure;
  endif
endfunc
```

Fig. 6. Algorithm for component selection

between components is modelled. We assume that the components communicate by message exchange. Each message has a name (or a type) and a number of named arguments (i.e. a list of attribute-value pairs). If two services are functionally equivalent, but not per se interoperable, the cause can be one of the following:

- related messages have different names
- related messages have different names for their arguments
- n messages of one service can be mapped onto one message of the other one
- the domains of related arguments are different

Under these circumstances, a mapping between the corresponding services can be specified. This mapping can be straightforwardly transformed to an implementation. Since for now, an adaptor specification consists basically of this mapping, rules for generating the adaptor code can be easily written. One simply has a fixed frame for each adaptor and generates *if-then*-statements for each mapping. For now the adaptor specification has to be developed manually. However, one can think of generating this information automatically as well, by considering the given components and figuring out how they can interoperate.

6 Implementation

We have implemented our approach for the construction of component systems based on term rewriting in ML. ML is particularly suited for this purpose because of its higher-order power. In our implementation, a higher-order function realizes the generator generator. Given a concrete specification of the available components, it outputs a function that, given the desired component system specification as input, outputs a set of components and adaptors that implement the desired target system.

Terms are represented via a generic datatype. The term rewriter starts with the input term, computes all possible successors and puts them into a candidate list. During the following, the least-expensive (w.r.t. the previous cost and the expected cost) candidate is removed from the list and replaced by its successors. This process continues until a goal term is reached.

The system specification is given by the following datatypes:

```
datatype Message=
    Msg of Name * (Argument list)
datatype Service=
    ServiceSpec of Name * Message list
```

```
datatype System=
    SystemSpec of Service list
                * Constraint list
                * CostFunction
                * HeurFunction
```

Components and adaptors are described by the following datatypes (making it clear that an adaptor is modelled as a special case of a component):

```
datatype Component =
      ComponentSpec of Name
    * Constraint list
    * ProvidedService list
    * RequiredService list
    * Property list
```

```
datatype Adaptor =
      AdaptorSpec of Name
    * ProvidedService
    * RequiredService
    * MsgMapping list
```

The mapping specifies how an incoming message is transformed to the message to be sent. In the transformation, arguments can be transformed, renamed, the

component	offered services	required services	cost
GPS Locator	anti-theft protection	GPS sensor	50
CDman 100	cd-player		70
NAV-200	navigation	GPS sensor	80
NAV-300	navigation		100
NAV-1000	navigation, cd-player		160
GPS 10	GPS sensor		30

Table 2
Automotive Infotainment Unit: Component repository

message can be renamed, and an internal state can be updated.

The term rewriting rules have the following origin:

- The rules for component selection are generated from the component specifications. For each component, a rule is generated and corresponds to the selection of that component (as described in detail in the previous section).
- The rules for adaptor selection are also generated from the specification, as a special case of component selection.
- The rules for adaptor code generation depend only on the target platform and have to be specified manually. However, having them specified once allows for automatic generation of adaptor code for a given specification.

The role of the heuristic function must not be neglected. As an example, let us consider a first toy example we have modeled, consisting of 4 services and 7 components. Even for such a small setting, the search space contains 174 nodes. To find the optimum, 25 nodes were visited with uninformed search. If our earlier explained heuristics is used (minimal cost for a component selection presumed), only 10 nodes were visited, while still yielding the same result.

7 Case Studies

In this section, we discuss our two case studies. We first revisit the example from the introduction for an automotive system. Then, we present a case study in which we have modelled a software component system in the EJB (Enterprise Java Beans) framework [4]. Taking a repository of components for granted, a running system has been created (inclusive adaptors) from the specification.

7.1 Automotive Infotainment Unit

Let us reconsider the first example from the introduction. We want to select components in an automotive context. Our system should contain the following functionality: navigation system, cd player, car-theft protection. We assume that we have certain components at our disposal and are now to select the best composition. Table 2 gives an overview of the components. For each component, the offered as well as the required services are denoted. Additionally, the associated cost (e.g. the price) is shown.

Given our requirements, the optimal solution is the component selection *GPS Locator CDman 100, NAV-200, GPS 10* with a cost of 230 (note that the selection of either of *NAV-200* and *GPS Locator* forces to select a component which provides

component	proviced services	required services	constraints	cost
text input	UserInterface		b/w, text	10
text input color	UserInterface		color, text	13
graphical input	UserInterface		color, graphic	100
eliza	DialogSystem	UserInterface, Visualization		80
smiley color	Visualization		color, text	20
face color	Visualization		color, graphic	150

Table 3
Entertainment Software: Component repository

rule "select 'text input color' for the Service 'UserInterface'" {
cond: $color{=}yes, graphic{=}no$
match: $system[services] \mid \exists\, \triangle^{UserInterface} \in services$
replace: $system[services \backslash \{\triangle^{UserInterface}\}$
cost: $codesize{=}13$
assert: $color{=}yes, graphic{=}no$ }

Fig. 7. Selection of a concrete component for the chatterbot system

the service *GPS sensor*). However, if we omit the requirement for the anti-theft protection, the integrated cd-player/navigation system (*NAV-1000*) will be the best solution (cost: 160).

If we model the initially given example, our compacted search space consists of 39 nodes. To find the optimum, 23 (20) nodes are visited (with heuristics presuming minimal component cost) [7].

7.2 Entertainment Software for Mobile Devices

As second case study, we have modelled an application intended for entertainment on handheld devices. The application should act as a virtual conversational partner for the user, as a so-called *chatterbot* [8]. We used the Enterprise Java Beans (EJB) component framework as target platform.

The services which can be identified are: DialogSystem, Visualization, UserInterface. The available components are listed in Table 3, each component corresponds to a Java Bean. As cost measure, the size of the binaries (in kilobyte) was chosen. The constraints in this context describe the display type, namely whether it is black-and-white or color, and whether it is text-based or graphical. Mutually exclusive constraints have to be modelled as one parameter, e.g. black-and-white and color are mapped to the boolean value *color*. The specification for two components are given below.

```
val compEliza=Component(
  Name            "eliza",
  Constraints     [],
  ProvidedService ["DialogSystem"],
  RequiredService ["UserInterface",
                   "Visualization"],
  Properties      [("codesize",80)]
)
```

```
val compTextInputColor=Component(
  Name            "text input color",
  Constraints     [("color","yes"),
                   ("graphic","no")],
  ProvidedService ["UserInterface"],
  RequiredService [],
  Properties      [("codesize",13)]
)
```

The components are automatically transformed into rewrite rules. As exam-

[7] Without search space compaction, out of 163 nodes 48 (35) were visited (with heuristics).
[8] The setting is a simplification of a system presented in [8].

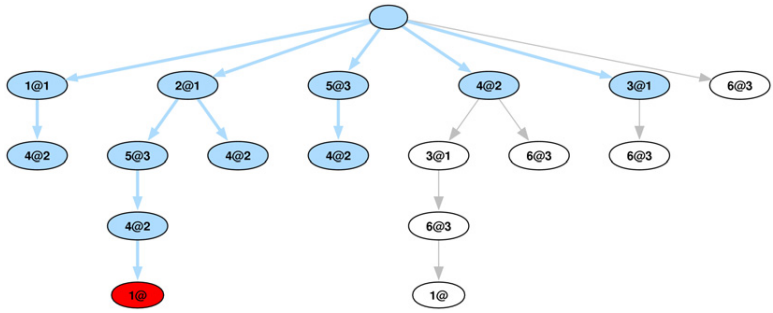


Fig. 8. Search space for the chatterbot system. Visited nodes are shaded, the goal node is darkly shaded.

ple, we regard the rewrite rule for the selection of the component *text input color*, cf. Figure 7. As we see, this component can only be selected if the already chosen architectural constraints do not contradict to the constraints of the component. After selection, the constraints of the system are extended by the constraints of the component.

The process of component selection can start with no constraints at all. In this case, constraints are dynamically selected during the rewrite process. On the other hand, we can preselect certain constraints. In the given example, if we start with no constraints, the optimal solution is the following: Select components *eliza*, *text input color*, *smiley color*, at the cost of 113. If we start with the constraint `graphic=yes`, we find this optimal solution: *eliza*, *graphical input*, *face color*, leading to a cost of 330.

For brevity, the description of adaptor specification has been ommitted. However, in the case study, the interfaces of the different components did not match to each other, and interoperability was only achieved by adaptor specification. Starting with the specification of the system (services, components, adaptors) and with the given Java Beans for the components, components were selected and integrated by adaptor code generation in a complete automatical process, leading to a deployed application.

For this case study, the compacted search space contained 18 nodes at maximum depth 5. For the optimal solution, 12 (11) nodes were visited (with heuristic function) [9]. The search space is depicted in Figure 8.

With these two case studies, we have demonstrated that our term-rewriting based approach to the construction of component systems is efficient and suited for practical applications.

8 Related Work

Our approach uses ideas from the algorithm proposed in [9] for optimal code generation in compilers. However, their approach puts strong constraints on the rewrite sequence to reduce the search space. In a precomputation step, the given term is annotated with possible rewrite sequences. As a result, the algorithm cannot cope

[9] The full search space contained 45 nodes, 16 (15) nodes being visited (with heuristics).

properly with dynamically added subterms. We have modified their method by allowing not only bottom-up term rewriting but also more general rewrite orders.

An analysis of the complexity of component selection is given in [10,11]. The authors establish that component selection is NP-complete. [7] considers as well the problem of component selection and models the problem as retrieval problem.

Concerning the problem of the generation of adaptors, [14] makes an important contribution. They model protocols with finite state automata and provide a formal definition of properties like dead-lock-freeness and liveness. Starting from a functional description of an adaptor, they show how to generate the corresponding adaptor code.

Several approaches consider EJBs as framework for component systems. [13] describes how to build composed systems via an XML script. [12] gives a formal modelling of java beans.

Most related work deals with either component selection or adaptor code generation. The contribution of our approach is the unified view of both processes.

9 Conclusion

We have presented a framework for the construction of optimal component systems based on term rewriting strategies. By taking these techniques from compiler construction, especially optimizing code generation, we have been able to develop an algorithm that finds a cost-optimal component system in minimal (wrt. the applied heuristics) time. The importance of the heuristics is very important as it directly determines the efficiency of the construction algorithm. In our case studies, we have demonstrated that a good heuristics can reduce the search space drastically.

Our method is very general as it does not put many requirements on the kinds of components and adaptors that can be selected and composed into a working system. In future work, we want to apply our construction method to hardware and mixed hardware/software systems as well. The constraints that these kinds of components need to fulfill can then be expressed in the architectural constraints of our formalism. Moreover, we want to extend our method such that not only two components but also larger numbers of components can be connected by suitable adaptors. And finally, we want to formally verify that the constructed component systems behave correctly as described in their specification. For this purpose, we also want to apply methods from compiler construction, especially verification methods [5] that can be used to show that transformation algorithms as well as their implementations are correct.

References

[1] Aho, A. V., S. C. Johnson and J. D. Ullman, *Code Generation for Expressions with Common Subexpressions*, J. ACM **24** (1977), pp. 146–160.

[2] Baader, F. and T. Nipkow, "Term rewriting and all that," Cambridge University Press, New York, NY, USA, 1998.

[3] Dechter, R. and J. Pearl, *Generalized Best-First Search Strategies and the Optimality of A**, J. ACM **32** (1985), pp. 505–536.

[4] DeMichiel, L., "Enterprise JavaBeans^{TM} Specification," SUN Microsystems, November 2003, version 2.1, Final Release.

[5] Glesner, S., G. Goos and W. Zimmermann, *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers)*, it - Information Technology **46** (2004), pp. 265–276.

[6] Hart, P. E., N. J. Nilsson and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics SSC4 (2) (1968), pp. 100–107.

[7] Hemer, D., *A formal approach to component adaptation and composition*, in: *ACSW '05, Australasian Computer Science Week*, 2005.

[8] Kopp, S., N. Krämer, L. Gesellensetter and I. Wachsmuth, *A conversational agent as museum guide – design and evaluation of a real-world application*, in: *5th Int. working conference on Intelligent Virtual Agents (IVA'05)*, 2005.

[9] Nymeyer, A. and J.-P. Katoen, *Code generation based on formal BURS theory and heuristic search*, Acta Informatica **34** (1995), pp. 597–635.

[10] Page, E. H. and J. M. Opper, *Observations on the complexity of composable simulation*, in: *WSC '99: 31st conf. on Winter simulation* (1999), pp. 553–560.

[11] Petty, M., E. W. Weisel and R. R. Mielke., *Computational complexity of selecting models for composition*, in: *Proceedings of the Fall 2003 Simulation Interoperability Workshop*, Orlando FL, 2003, pp. 517–525.

[12] Upadhyayaand, B. Z. L., *Formal support for development of javabeans component systems*, in: *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004, pp. 23–28.

[13] Weerawarana, S., F. Curbera, M. J. Duftler, D. A. Epstein and J. Kesselman, *Bean markup language: A composition language for javabeans components*, in: *COOTS 2001*, 2001.

[14] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM Trans. Program. Lang. Syst. **19** (1997), pp. 292–333.