Enhancing dependability through flexible adaptation to changing requirements*

Michel Wermelinger[1], Georgios Koutsoukos[2], Hugo Lourenço[2], Richard Avillez[2], João Gouveia[2], Luís Andrade[2], and José Luiz Fiadeiro[3]

[1] Dep. de Informática, Univ. Nova de Lisboa, 2829-516 Caparica, Portugal
mw@di.fct.unl.pt
[2] ATX Software SA, Alameda António Sérgio, 7, 1C, 2795-023 Linda-a-Velha, Portugal
{firstname.lastname}@atxsoftware.com
[3] Dep. of Computer Science, Univ. of Leicester, Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract. This paper describes an architectural approach that facilitates the dynamic adaptation of systems to changing domain rules. The approach relies on "coordination contracts", a modelling and implementation primitive we have developed for run-time reconfiguration. Our framework includes an engine that, whenever a service is called, checks the domain rules that are applicable and configures the response of the service before proceeding with the call.
This approach enhances dependability in two essential ways: on the one hand, it guarantees that system execution is always consistent with the domain logic because service response is configured automatically (i.e., without any need for programmer intervention); on the other hand, it makes it possible for changes to be incorporated into existing domain rules, and from new rules to be created, with little effort, because coordination contracts can be superposed dynamically without having to change neither the client nor the service code.
Our approach is illustrated through a case study in financial systems, an area in which dependability arises mainly in the guise of business concerns like adherence to agreed policies and conditions negotiated on a case-by-case basis. We report on an information system that ATX Software developed for a company specialised in recovering bad credit. We show in particular how, by using this framework, we have devised a way of generating rule-dependent SQL code for batch-oriented services.

1 Introduction

This paper describes an architectural approach to system development that facilitates adaptation to change so that organisations can effectively depend on a continued service that satisfies evolving business requirements. This approach has been used in a real project in which ATX Software developed an information

* This paper is a considerably extended version of [1].

system for a company specialised in recovering bad credit. The approach is based on two key mechanisms:

- the externalisation of the domain rules from the code that implements core system functionalities;
- the encapsulation of the code that enforces those domain rules into so-called coordination contracts that can be created and deleted at run-time, hence adapting computational services to the context in which they are called.

In the concrete case study that we present, the domain rules define the dependency of the recovery process on business concerns of the financial institution and product (e.g., house mortgage) for which the debt is being recovered. At any given time, this business configuration defines the context in which services are called.

These two mechanisms are aimed at two different classes of stakeholders. Domain rules are intended for system users, who have no technical knowledge, so that they can adapt the system in order to cope with requirements of newly or already integrated financial institutions or products. Coordination contracts are intended for system developers to add new behaviour without changing the original service implementation. This is made possible with the ability of coordination contracts to superpose, at run-time, new computations on the services that are being execute locally in system components.

Coordination contracts [2] are a modelling and implementation primitive that allows transparent interception of method calls and as such interfere with the execution of the service in the client. Transparent means that neither the service nor its client are aware of the existence of the coordination contract. Hence, if the system has to be evolved to handle the requirements imposed by new institutions or products, many of the changes can be achieved by parameterising the service (data changes) and by superposing new coordination contracts (behaviour changes), without changing the service's nor the client's code. This was used, for instance, to replace the default calculation of the debt's interest by a different one. The user may then pick one of the available calculation formulae (i.e., coordination contracts) when defining a domain rule.

To be more precise, a coordination contract is applicable to one or more objects (called the contract's participants) and has one or more coordination rules, each one indicating which method of which participant will be intercepted, under which conditions, and what actions to take in that case. In the particular case of the system that we are reporting in this paper, all coordination contracts are unary, the participant being the service affected by the domain rule to which the coordination contract is associated. Moreover, each contract has a single rule. We could have joined all coordination rules that *may be* applicable to the same service into a single contract, but that would be less efficient in run-time and more complex in design time due to more intricate rule definitions. The reason is that once a contract is in place, it will intercept *all* methods given in all the contract's rules, and thus the rule conditions would have to check at run-time if the rule is really applicable, or if the contract was put in place because of another coordination rule.

In this project we used an environment that we have built for developing Java applications using coordination contracts [3]. The environment is freely available from www.atxsoftware.net. The tool allows writing contracts, and to register Java classes (components) for coordination. The code for adapting those components and for implementing the contract semantics is generated based on a micro-architecture that uses the Proxy and Chain of Responsibility design patterns [4]. This microarchitecture handles the superposition of the coordination mechanisms over existing components in a way that is transparent to the component and contract designer. The environment also includes an animation tool, with some reconfiguration capabilities, in which the run-time behavior of contracts and their participants can be observed using sequence diagrams, thus allowing testing of the deployed application.

In the context of this work, we are concerned mainly with the maintainability attribute of dependability, and with faults that are accidental, human-made, and developmental, whereby our approach could be classified as fault prevention [5]. To be more precise, instead of hard-wiring and duplicating domain rules across multiple system services, which makes maintenance of the system error-prone when rules change, we provide support to keep rules separate from the services and to apply them only when needed. Our approach guarantees that whenever a user changes a domain rule, any future invocation of a service (whether it is a web-based interactive service or an SQL-based nightly batch service) that is affected by it will automatically take the rule in consideration. Hence, dependability, i.e., "the ability to deliver service that can justifiably be trusted" [5] is maintained.

The structure of the paper is as follows. The next section provides the wider context of our work, namely our architectural approach to the separation of computation, coordination, and configuration. This explains the design rationale for the framework illustrated, through its application to the debt recovery system, in the following sections. Section 3 starts describing the case study by introducing some example domain rules, taken from the credit recovery domain, and shows how coordination contracts are used to change the default service functionalities according to the applicable domain rules. Section 4 sketches the framework we implemented, describing how the service configuration is done at run-time according to the rules. Section 5 explains how the same framework is used to generate rule-dependent SQL code to be run in batch mode. The last section presents some concluding remarks.

2 The Three Cs of Architectures

The architectural approach that we will be describing is based on the crucial separation between "three Cs": Computation, Coordination, and Configuration. This separation needs to be supported at two different levels. On the one hand, through semantic primitives that address the "business architecture", i.e., the means that need to be provided for modelling business entities (Computation), the business rules that determine how the entities can interact (Coordination),

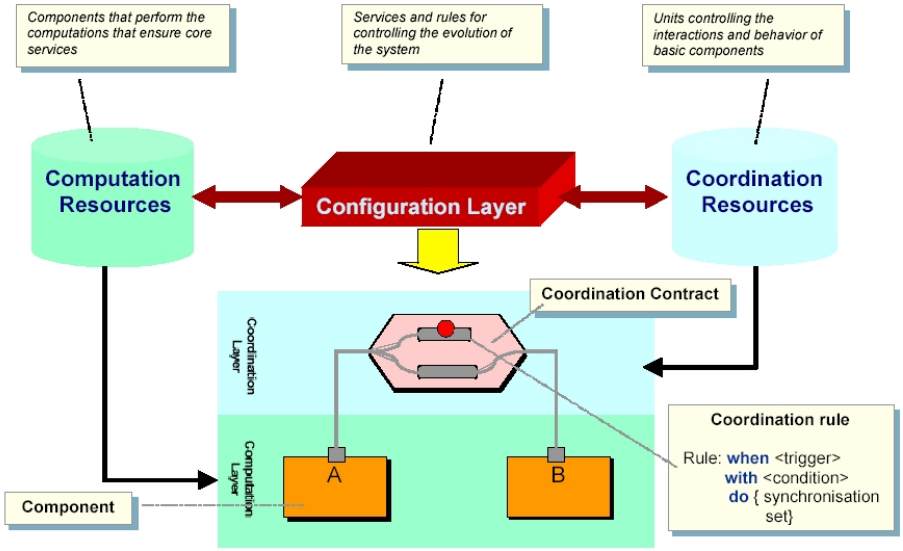


Fig. 1. The configuration framework

and the business contexts through which specific rules can be superposed, at runtime, to specific entities (Configuration). On the other hand, the architectural properties of the deployment infrastructures that can carry this distinction to the design and implementation layers, and support the required levels of agility.

2.1 The CCC System Architecture

As already mentioned, the rationale for the methodology and technologies that we have been building is in the *strict* separation between three aspects of the development and deployment of any software system: the *computations* performed locally in its components, the *coordination* mechanisms through which global properties can emerge from those computations, and the *configuration* operations that ensure that the system will evolve according to given constraints such as organisational policies, legislation, and other. Such layering should be strict in order to allow for changes to be performed at each layer without interfering with the levels below.

The Computation Layer should contain the components that perform the computations that ensure the basic services provided within the system. Each component has two interfaces [6]: a functional interface that includes the operations that allow one to query and change the encapsulated component state; and a configuration interface that provides the component's constructors and destructors, and any other operations that are necessary to the correct management of dynamic reconfiguration. One such operation is querying whether the component is in a "stable" state in which the component may be deleted or its "connections" to other components can be changed; another example is an

operation that can temporarily block the component's execution while a reconfiguration that involves it is processed. The reason for separate interfaces is to be able to constrain the access that the various parts of the architecture have to each other and, hence, achieve a cleaner separation of concerns. In the case of the coordination layer, we require that no component should be able to invoke another component's configuration operations: components should not create other components because that is a change to the currently existing configuration and, as such, should be explicitly managed by the configuration layer.

The Coordination Layer defines the way computational components are interconnected for the system to behave, as a whole, according to set requirements. In the terminology of Software Architecture, this layer is populated by the *connectors* that regulate the interactions between the components of the layer below. We call such connectors *coordination contracts* or, for simplicity, *contracts*. We also require each contract to provide a functional and a configuration interface; each constructor in the configuration interface must include as arguments the components that the connector instance to be created will coordinate. We impose two restrictions: a contract may not use the configuration interface of any contract or component; and a contract may not use another contract's functional interface. The rationale for the first condition is again that configuration operations should only be performed by the configuration layer. The reason for the second condition is to make it possible to evolve the system through (un)plugging of individual contracts between components, which is only possible if there are no dependencies among contracts. The coordination effects that contracts put in place are described in terms of trigger-reaction rules as illustrated in Sec. 3.

At each state, the interconnections put in place among the population of basic components via contracts define the current configuration of the system. The Configuration Layer is responsible for managing the current configuration, i.e., for determining, at each state, which components need to be active and what interconnections need to be in place among which components. This layer provides a set of high-level reconfiguration operations that enforce global invariants over the system's configuration. The actual implementation of the configuration layer may follow the technical architecture given in [7]: a configuration database containing updated information about the current configuration, a consistency manager that enforces a "stable" state in which reconfiguration can occur, and a reconfiguration manager that executes the reconfiguration operations, using the services of the database and the consistency manager. The implementation of the reconfiguration operations makes use not only of the configuration interfaces provided by the components and contracts, but also of the functional interfaces because some changes to the configuration may depend on the current state of components and contracts, and may trigger state modifications to restore application-wide consistency.

Systems whose design architecture supports this separation of concerns through a strict layering can be evolved in a compositional way. Changes that do not require different computational properties can be brought about either by reconfiguring the way components interact, or adding new connectors that regu-

late the way existing components operate, instead of performing changes in the components themselves. This can be achieved by superposing, dynamically, new coordination and configuration mechanisms on the components that capture the basic business entities. If the interactions were coded in the components themselves, such changes, if at all possible thanks to the availability of the source code, besides requiring the corresponding objects to be reprogrammed, with possible implications on the class hierarchy, would probably have side effects on all the other objects that use their services, and so on, triggering a whole cascade of changes that would be difficult to control.

On the other hand, the need for an explicit configuration layer, with its own primitives and methodology, is justified by the need to control the evolution of the configuration of the system according to the business policies of the organisation or, more generally, to reflect constraints on the configurations that are admissible (configuration invariants). This layer is also responsible for the degree of self-adaptation that the system can exhibit. Reconfiguration operations should be able to be programmed at this level that enable the system to react to changes perceived in its environment by putting in place new components or new contracts. In this way, the system should be able to adapt itself to take profit of new operating conditions, or reconfigure itself to take corrective action, and so on.

According to the nature of the platform in which the system is running, this strict layering may be more or less directly enforced. For instance, we have already argued that traditional object-oriented and component-based development infrastructures do not support this layering from first-principles, which motivates the need for new semantic modelling primitives as discussed in the next subsection. However, this does not mean that they cannot accommodate such an architecture: design techniques such as reflection or aspect-oriented programming, or the use of design patterns, can be employed to provide the support that is necessary from the middleware. In fact, we have shown how the separation between computation and coordination can be enforced in Java through the use of well known design patterns, leading to what we called the "Coordination Development Environment" or CDE [4, 3, 8].

The design patterns that we use in the CDE provide what we can call a "micro-architecture" that enforces the externalisation of interactions, thus separating coordination from computation. It does so at the cost of introducing an additional layer of adaptation that intercepts direct communication through feature calling (clientship) and, basically, enforces an event-based approach. In this respect, platforms that rely on event-based or publish-subscribe interaction represent a real advantage over object-based ones: they support directly the modelling primitives that we will mention next.

2.2 The CCC Business Architecture

The separation of coordination from computation has been advocated for a long time in the Coordination Languages community [9], and the separation of all three concerns is central to Software Architecture, which has put forward the

distinction between components, connectors and architectures [10]. The Configurable Distributed Systems community [11], in particular the Configuration Programming approach [12], also gives first-class status to configuration. However, these approaches do not provide a satisfying way to model the three concerns in a way that supports evolution. Coordination languages do not make the configuration explicit or have a very low-level coordination mechanism (e.g., tuple spaces); architecture description languages do not handle evolution from first principles or do it in a deficient way; configuration programming is not at the business modelling level.

For instance, the reconfiguration operations that we provide through *coordination contexts* correspond more to what in other works is called a reconfiguration script [13] than the basic commands provided by some ADLs to create and remove components, connectors, and bindings between them [14]. Coordination contexts also make explicit which invariants the configuration has to keep during evolution. It is natural to express these invariants in a declarative language with primitive predicates to query the current configuration (e.g., whether a contract of a given type connects some given components). Such languages have been proposed in Distributed Systems (e.g., Gerel-SL [13]) and Software Architecture approaches (e.g., Armani [15]). However, all these approaches *program* the reconfiguration operations, i.e., they provide an operational specification of the changes. Our position is that, at the modelling level, those operations should also be specified in a declarative way, using the same language as for invariants, by stating properties of the configuration before and after the change. In other words, the semantics of each reconfiguration operation provided in this layer is given by its pre- and post-conditions.

On the other hand, it is true that modelling languages like the UML [16] already provide techniques that come close to our intended level of abstraction. For instance, "use cases" come close to coordination contexts: they describe the possible ways in which the system can be given access and used. However, they do not end up being explicitly represented in the (application) architecture: they are just a means of identifying classes and collaborations. More precisely, they are not captured through formal entities through which run-time configuration management can be explicitly supported. The same applies to the externalisation of interactions. Although the advantage of making relationships first-class citizens in conceptual modelling has been recognised by many authors (e.g., [17]), which led to the ISO General Relationship Model (ISO/IEC 10165-7), things are not as clean when it comes to supporting a strict separation of concerns.

For instance, one could argue that mechanisms like association classes provide a way of making explicit how objects interact, but the typical implementation of associations through attributes is still "identity"-based and does not really externalise the interaction: it remains coded in the objects that participate in the association. The best way of implementing an interaction through an association class would seem to be for a new operation to be declared for the association that can act as a mediator, putting in place a form of implicit invocation [18]. However, on the one hand, the fact that a mediator is used for coordinating the

interaction between two given objects does not prevent direct relationships from being established that may side step it and violate the business rule that the association is meant to capture. On the other hand, the solution is still intrusive in the sense that the calls to the mediator must be explicitly programmed in the implementation of the classes involved in the association.

Moreover, the use of mediators is not incremental in the sense that the addition of new business rules cannot be achieved by simply introducing new association classes and mediators. The other classes in the system need to be made aware that new association classes have become available so that the right mediators are used for establishing the required interactions. That is, the burden of deciding which mediator to interact with is put again on the side of clients. Moreover, different rules may interact with each other thus requiring an additional level of coordination among the mediators themselves to be programmed. This leads to models that are not as abstract as they ought to be due to the need to make explicit (even program) the relationships that may exist between the original classes and the mediators, and among the different mediators themselves.

The primitive — *coordination law* - that we have developed for modelling this kind of contractual relationship between components circumvents these problems by abandoning the "identity"-based mechanism on which the object-oriented paradigm relies for interactions, and adopting instead a mechanism of superposition that allows for collaborations to be modelled outside the components as connectors (coordination contracts) that can be applied, at run-time, to coordinate their behaviour. From a methodological point of view, this alternative approach encourages developers to identify dependencies between components in terms of *services* rather than identities. From the implementation point of view, superposition of coordination contracts has the advantage of being non-intrusive on the implementation of the components. That is, it does not require the code that implements the components to be changed or adapted, precisely because there is no information on the interactions that is coded inside the components. As a result, systems can evolve through the addition, deletion or substitution of coordination contracts without requiring any change in the way the core entities have been deployed.

3 Business Rules and Coordination Contracts

ATX Software was given the task to re-implement in Java the information system of Espírito Santo Cobranças, a debt recovery company that works for several credit institutions, like banks and leasing companies. The goal was not only to obtain a Web-based system, but also to make it more adaptable to new credit institutions or to new financial products for which the debts have to be collected. This meant that business rules should be easy to change and implement.

The first step was to make the rules explicit, which was not the case in the old system, where the conditions that govern several aspects of the debt recovery process were hardwired in tables or in the application code itself. We defined a

business rule to be given by a condition, an action, and a priority. The condition is a boolean expression over relations (greater, equal, etc.) between parameters and concrete values. The available parameters are defined by the rule type. The action part is a set of assignments of values to other parameters, also defined by the rule type. Some of the action parameters may be "calculation methods" that change the behaviour of the service to which this rule is applicable. The priority is used to allow the user to write fewer and more succint rules: instead of writing one rule for each possible combination of the condition parameter values, making sure that no two rules can be applied simultaneously, the user can write a low priority, general, "catch-all" rule and then (with higher priority) just those rules that define exceptions to the general case. As we will see later, rules are evaluated by priority order. Therefore, within each rule type, each rule has a unique priority.

To illustrate the concept of business rule, consider the agreement simulation service that computes, given a start and ending date for the agreement, and the number of payments desired by the ower, what the amount of each payment must be in order to cover the complete debt. This calculation is highly variable on a large number of factors, which can be divided into three groups. The first one includes those factors that affect how the current debt of the ower is calculated, like the interest and tax rates. This group of factors also affect all those services, besides the agreement simulation, that need to know the current debt of a given person. The second group includes those factors that define what debt should be taken into account for the agreement: the debt on the day before the agreement starts, the debt on the day the agreement ends, or yet another possibility? The last group covers factors concerned with internal policies. Since the recovery of part of the debt is better than nothing, when a debt collector is making an agreement, he might pardon part of the debt. The exact percentage (of the total debt amount) to be pardoned has an upper limit that depends on the category of the debt collector: the company's administration gives higher limits to more experienced employees.

As expected, each group corresponds to a different business rule type, and each factor is an action parameter for the corresponding rule type. The condition parameters are those that influence the values to be given for the action parameters. As a concrete example, consider the last group in the previous paragraph. The business rule type defines a condition parameter corresponding to the category of the debt collector and an action parameter corresponding to the maximum pardon percentage. A rule (i.e., an instance of the rule type) might then be `if category = ‘‘senior’’ or category = ‘‘director’’ then maxPardon = 80%`. The priorities might be used to impose a default rule that allows no pardon of the debt. The lowest priority rule would then be `if true then maxPardon = 0%`.

However, a more interesting rule type is the one corresponding to the calculation of the debt (the first group of factors for the agreement service). The debt is basically calculated as the sum of the loan instalments that the ower has failed to pay, surcharged with an amount, called "late interest". The rules

for calculating this amount are defined by the credit institution, and the most common formula is: instalment amount * late interest rate * days the payment is late / 365. In other words, the institution defines a yearly late interest rate that is applied to the owed amount like any interest rate. This rate may depend only on the kind of loan (if it was for a house, a car, etc.) or it may have been defined in the particular loan contract signed between the institution and the ower. In the first case, the rate may be given as an action parameter value of the rule, in the second case it must be computed at run-time, given the person for whom the agreement is being simulated. But as said before, the formula itself is defined by the institution. For example, there are instutions that don't take the payment delay into account, i.e., the formula is just `instalment amount * late interest rate`. For the moment, these are the only two formulas the system incorporates, but the debt recovery company already told us that in the forseeable future they will have to handle financial institutions and products that have late interest rates over different periods of time, e.g., quarterly rates (which means the formula would have the constant 90 instead of 365).

In these cases, where business rules impose a specific behaviour on the underlying services, we add an action parameter with a fixed list of possible values. Each value (except the default one) corresponds to a coordination rule that contains the behaviour to be superposed on the underlying service (which implements the default behaviour, corresponding to the default value of the parameter). However, from the user's perspective, there is nothing special in this kind of parameter; the association to coordination rules is done "under the hood". For our concrete example, the late interest rule type would have as condition parameters the institution and the product type, and as action parameters the interest rate (a percentage), the rate source (if it is a general rate or if it depends on the loan contract), and the rate kind (if it is a yearly rate or a fixed one). The last two parameters are associated to coordination rules and the first parameter (the rate) is optional, because it has to be provided only if the rate source is general. Two rule examples are

- `if institution = 'Big Bank' and productType = 'car loan' then rate = 7%, source = 'general', kind = 'fixed';`
- `if institution = 'Big Bank' and productType = 'house loan' then source = 'contract', kind = 'yearly'.`

As for the coordination rules, we need one for each computation that differs from the default behaviour, which is implemented directly in the service because it is assumed to be the case occurring most often. For the example, we need a rule to fetch the rate from the database table that holds the loan contract information for all processes handled by the debt recovery company, and another rule to calculate the late interest according to the fixed rate formula.

Continuing with our example, the service has (at least) the following methods:

- `void setRate(double percentage)`, which is used to pass the value of the `rate` action parameter to the service;

– `double getRate()`, which is used by clients of the service, and by the next method, to obtain the rate that is applicable;
– `double getInterest()`, which uses auxiliary methods implemented by the same service to calculate the late interest to be paid. Its implementation is `return getInstalment() * getRate() * getDays() / 365;`.

Given these methods, the coordination rules are as follows:

Fixed Rate This rule intercepts the `getInterest()` method unconditionally, and executes: `return getInstalment() * getRate()`.

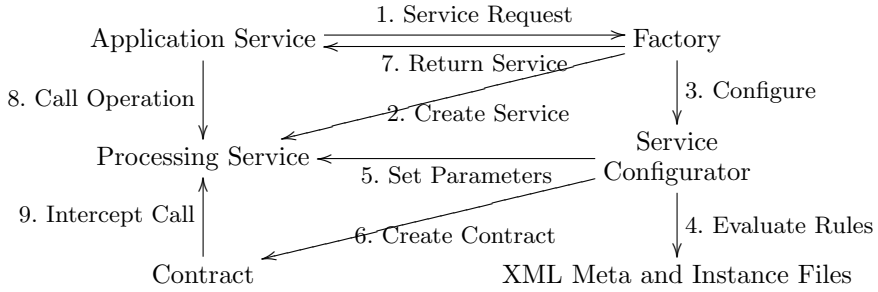
Contracted Rate This rule intercepts the `getRate()` method under the condition `!calculated`, and executes: `r =` the rate obtained by consulting the database`; setRate(r); calculated = true`.

The second rule requires the coordination contract to have a local boolean attribute `calculated`, initialized to false. The idea is that, no matter how often the service's clients call the `getRate()` method, the database lookup will be done only for the first call, and the rate is stored into the service object, as if it were given directly by a business rule. This "shortcut" works because we know that the rates stored in the database may only change at the beginning of the nightly batch, not during the interest calculation.

The next section explains how the three parts (business rules, coordinations contracts, and services) work together at run-time in order to ensure that the correct (business and coordination) rules are applied at the right time to the right services.

4 Architectural Framework

The architecture of the configuration framework, and the steps that are taken at run-time, are shown next.



The process starts with the creation of an application service object to handle the user's request, e.g., the request for the simulation of the agreement. This object contains the necessary data, obtained from the data given by the user on the web page, and will call auxiliary processing services. Each service is implemented by a class, whose objects will be created through a factory (step 1 in the figure). After creating the particular instance of the processing service

(step 2), the factory may call the service configurator (step 3), if the service is known to be possibly subject to business rules. The configurator consults two XML files containing information about the existing business rules. The one we called meta file defines the rule types (see Fig. 2 for an example), while the instance file contains the actual rules (see Fig. 3).

The configurator first looks into the meta file to check which business rules are applicable for the given processing service. For each such rule, the meta file defines a mapping from each of the rule type's condition (resp. action) parameters into getter (resp. setter) methods of the service, in order to obtain from (resp. pass to) the service the values to be used in the evaluation of the conditions of the rules (resp. the values given by the action part of the rules). There is also the possibility that an action parameter is mapped to a coordination contract. For our example, the mapping could be the one given in Table 1. Notice that the default values `general` and `yearly` are not mapped to any coordination contract.

Parameter	Value	Method	Coordination Contract
institution		getInstitution	
productType		getProductType	
rate		setRate	
source	general		
source	contract		Contracted Rate
kind	yearly		
kind	fixed		Fixed Rate

Table 1. Example mapping of parameter (values) to methods and contracts

With this information (which of course is read from the meta file only once, and not every time the configurator is called), the configurator calls the necessary getters of the service in order to obtain the concrete values for all the relevant condition parameters. Now the configurator is able to evaluate the rules in the instance file (step 4), from the highest to the lowest priority one, evaluating the boolean expression in the if part of each rule until one of them is true. If the parameter values obtained from the service satisfy no rules' condition, then the configurator raises an exception. If a suitable rule is found, the configurator reads the values of the action parameters and passes them to the service (step 5) by calling the respective setters. If the action parameter is associated with a coordination contract, the configurator creates an instance of that contract (step 6), passing to the contract constructor the processing service object as the participant. Continuing the example, the configurator would call the `getInstitution` and `getProductType` methods. If the values obtained were "Big Bank" and "car loan", respectively, then, according to the example rules in the previous section, the configurator would call `setRate(0.07)` on the service, and create an instance of the "Fixed Rate" coordination contract.

At this point the configurator returns control to the factory, which in turn returns to the application service a handler to the created (and configured) pro-

cessing service. The application service may now start calling the methods of the processing service (step 8). If the behaviour of such a method was changed by a business rule, the corresponding contract instance will intercept the call and execute the different behaviour (step 9). To finish the example, if the application service calls `getInterest`, it will be intercepted by the fixed rate contract, returning `getInstalment() * 0.07`.

Of course, the application service is completely unaware that the processing service has been configured and that the default behaviour has changed, because the application just calls directly the methods provided by the processing service to its clients. In fact, we follow the strict separation between computation and configuration described in [6]: each processing service has two interfaces, one listing the operations available to clients, the other listing the operations available to the configurator (like the getters and setters of business rule parameters). The application service only knows the former interface, because that is the one returned by the factory. This prevents the application service from changing the configuration enforced by the business rules.

The user may edit the XML instance file through a tool we built for that purpose to browse (Fig. 4), edit (Fig. 5) and create business rules. The tool completely hides the XML syntax away from the user, allowing the manipulation of rules in a user-friendly manner. Furthermore, it imposes all the necessary constraints to make sure that, on the one hand, all data are consistent with the business rules' metadata (i.e., the rule types defined in the XML meta file), and, on the other hand, that a well-defined XML instance file is produced. In particular, the tool supplies the user with the possible domain values for required user input, it checks whether mandatory action parameters have been assigned a value, facilitates the change of priorities among rules and guarantees the uniqueness of priorities, allows to search all rules for a given institution, etc. You may notice from the presented XML extracts that every rule type, rule, and parameter has a unique identifier and a name. The identifier is used internally by the configurator to establish cross-references between the instance and the meta file, while the name is shown by the rule editing tool to the user (as shown in the screenshots). The `valueType` attribute of a parameter is used by the rule editor to present to the user (in a drop-down list) all the possible values for that parameter. In Fig. 5 the drop-down list would appear by clicking on the button with a down-arrow in the upper right part of the active window.

Notice that the user is (and must be) completely unaware of which services are subject to which rule types, because that is not part of the problem domain. The mapping between the rules and the service classes they affect is part of the solution domain, and as such defined in the XML meta file. As such, each rule type has a conceptual unity that makes sense from the business point of view, without taking the underlying services implementation into account.

```
<service class="ComputeDebt">
  <ruleType name="Late Interest" id="LateInterest">
    <condition>
      <conditionGroup>
        <conditionParameter name="Financial Institution"
                            id="Inst" type="string">
          <valueType name="Institution" />
          <getter name="getInstitutionCd" returnType="String" />
          <SQL>
            <expr>AT_LATE_INTEREST_CALC.INSTITUTION_CD</expr>
            <from>AT_LATE_INTEREST_CALC</from>
          </SQL>
        </conditionParameter>
        <conditionParameter name="Credit Type"
                            id="CredType" type="string">
          <valueType name="CreditType" />
          <getter name="getCreditType" returnType="String" />
          <SQL>
            <expr>ST_PROCESS_CONTRACT.CREDIT_TYPE_CD</expr>
            <from>ST_PROCESS_CONTRACT,AT_LATE_INTEREST_CALC</from>
            <join>ST_PROCESS_CONTRACT.PROCESS_NBR =
                  AT_LATE_INTEREST_CALC.PROCESS_NBR</join>
          </SQL>
        </conditionParameter>
        <!-- the current phase of the recovery process -->
        <conditionParameter name="Phase" id="Phase" type="string">
          <valueType name="ProcPhase" />
          <getter name="getProcessPhase" returnType="String" />
          <SQL>
            <expr>AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD</expr>
            <from>AT_LATE_INTEREST_CALC</from>
          </SQL>
        </conditionParameter>
        <!-- other condition parameters -->
      </conditionGroup>
    </condition>
    <!-- the action parameters would be given here -->
 </ruleType>
</service>
```

Fig. 2. An extract of the XML meta file

```
<service class = "ComputeDebt" name = "ComputeDebt">
  <ruleType id = "LateInterest">
    <!-- other rules with higher priority -->

    <rule name = "Big Bank, judicial phases" id = "3" priority = "3">
      <conditionset type = "AND">
        <comparison id = "Inst" serviceValue = "0916"
           userValue = "Big Bank" operator = "equal"/>
        <conditionset type = "OR">
          <comparison  id = "Phase" serviceValue = "0005"
            userValue = "External judicial phase" operator = "equal"/>
          <comparison  id = "Phase" serviceValue = "0007"
            userValue = "Internal judicial phase" operator = "equal"/>
        </conditionset>
      </conditionset>
      <!-- the values for the action parameters come here -->
    </rule>

    <!-- remaining rules, with less priority -->
  </ruleType>
</service>
```

Fig. 3. An extract of the XML instance file

5 Batch-oriented Rule Processing

The approach presented in the previous section is intended for the interactive, web-based application services that are called on request by the user with the necessary data. These data are passed along to a processing service. The configurator queries the processing service for the data in order to evaluate the conditions of the rules.

However, like most information systems, the debt recovery system also has a substantial part working in batch. For example, the calculation of the debt is not only needed on demand to project the future debt for the simulation agreement service, it is also run every night to update the current debt of all the current credit recovery processes registered in the system. In this case, the debt calculation is performed by stored procedures in the database, written in SQL and with the business rules hard-wired.

Hence, when we have a large set of objects (e.g., credit recovery processes) for which we want to invoke the same processing service (e.g., debt calculation), it is not very efficient to apply the service to each of these objects individually. It is better to apply a "batch" strategy, reversing the configuration operation: instead of starting with an object and then choosing the rule that it satisfies, we take a rule and then select all the objects that satisfy it. This is much more efficient because we may use the same configured processing service instance for objects A and B if we are sure that for both A and B the same rule is chosen.

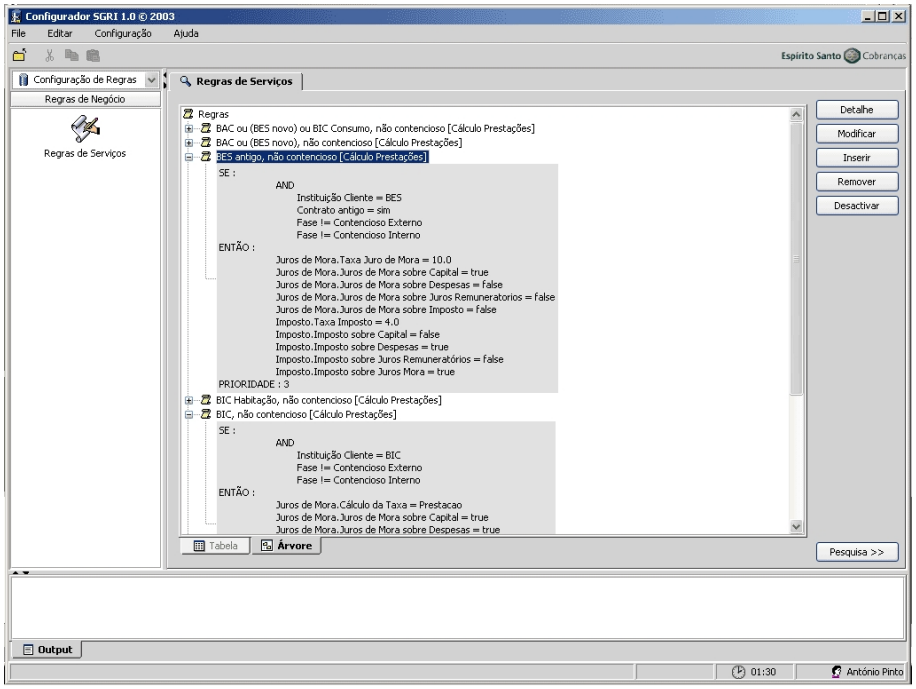


Fig. 4. Browsing rules for the debt calculation

We thus have the need to be able to determine for a given rule the set of objects that satisfy it. Pragmatically speaking, we need a way of transforming the if-part of a rule into an SQL condition that can be used in a SELECT query to obtain those objects. Therefore we extended the rule type information in the XML meta file, adding for each condition parameter the following information:

- an SQL expression that can be used to obtain the parameter value;
- the list of tables that must be queried to obtain the parameter value;
- a join condition between those tables.

Fig. 2 shows a fragment of the meta information for the debt calculation service. There we see, for example, that in order to obtain the value of the product type parameter we have to write the following query:

```
SELECT ST_PROCESS_CONTRACT.CREDIT_TYPE_CD
FROM ST_PROCESS_CONTRACT,AT_LATE_INTEREST_CALC
WHERE ST_PROCESS_CONTRACT.PROCESS_NBR =
      AT_LATE_INTEREST_CALC.PROCESS_NBR
```

Using this information we can now take a rule condition and transform it into a SQL fragment. As an example, consider the rule condition (for the same service) in Fig. 3: it is applicable to all recovery processes of “Big Bank” that

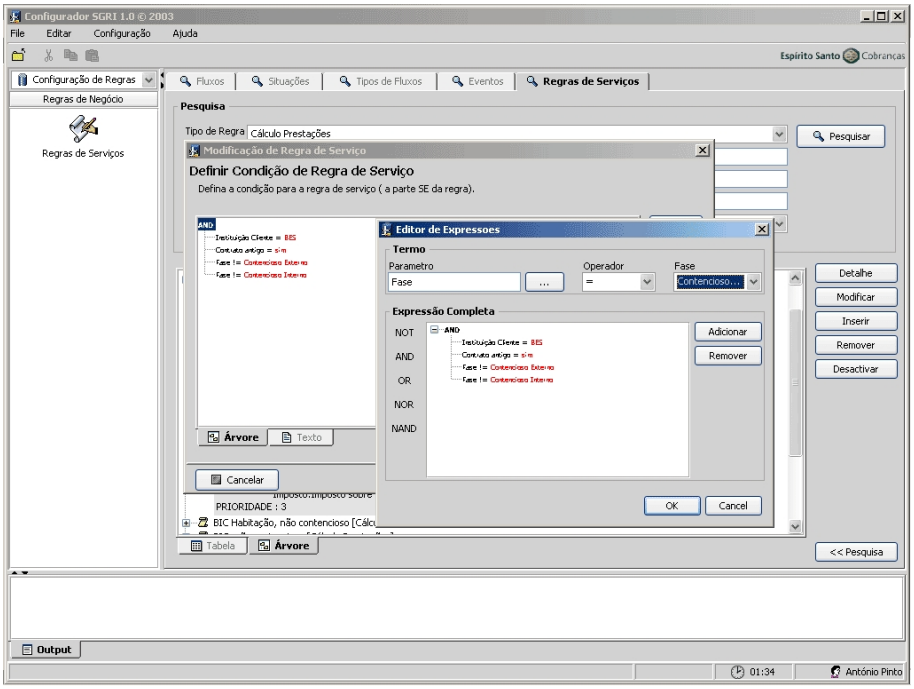


Fig. 5. Editing the condition of a rule

are in the internal judicial phase (i.e., the company's lawyers are dealing with the process) or the external one (i.e., the case has gone to court). We may compose the information for each of the rule parameters in order to obtain a single SQL fragment for the rule condition. This fragment contains the following information:

- the list of tables that must be queried in order to evaluate the rule condition;
- an SQL condition that expresses both the join conditions between the several tables and the rule condition itself.

For our example, the meta file specifies that `Inst` and `Phase`, the two parameters occurring in the condition, only require the table `AT_LATE_INTEREST_CALC` to be queried. As for the rule condition, the meta file specifies that `AT_LATE_INTEREST_CALC.INSTITUTION_CD` corresponds to the usage of the `Inst` parameter, and `AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD` to the `Phase` condition parameter. By a straightforward replacement of these names in the boolean expression of the rule condition, we get the following SQL expression:

```
((AT_LATE_INTEREST_CALC.INSTITUTION_CD = '0916') AND
((AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0005') OR
(AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0007')))
```

The SQL representation of a rule is conveyed by an instance of class `SQLRule`, which is contained in the service configurator, because the XML files are accessed by the latter.

```
public class ServiceConfigurator {
  public class SQLRule  {
      public String getId() { ... }
      public String getName() { ... }
      public String getWhere() { ... }
      public String getFrom() { ... }
  }
}
```

Each processing service class provides a static method for obtaining all of its rules in this "SQL format". This method simply calls a method of the service configurator, passing the service identification, which returns all rules for that service in decreasing order of priority. In the example below we show how we can generate a specialized query for a rule. In this example we first obtain all the service rules in "SQL format" and then generate a query that returns the first object that satisfies the condition of the third rule.

```
ServiceConfigurator.SQLRule[] SQLrules = ComputeDebt.getSQLRules();

ServiceConfigurator.SQLRule rule = SQLrules[2];
System.out.println("Rule : " + rule.getId() + " - " + rule.getName());
String sql = "SELECT TOP 1 AT_LATE_INTEREST_CALC.PROCESS_NBR " +
  " FROM " + rule.getFrom() +
  " WHERE " + rule.getWhere() +
  " AND PROCESSED = false";
System.out.println(sql);
```

The output generated is the following:

```
Rule : 3 - Big Bank, judicial phases

  SELECT TOP 1 AT_LATE_INTEREST_CALC.PROCESS_NBR
  FROM AT_LATE_INTEREST_CALC
  WHERE ((AT_LATE_INTEREST_CALC.INSTITUTION_CD = '0916')
  AND ((AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0005')
  OR (AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0007')))
  AND PROCESSED = false
```

Similar queries are generated for each rule and each query is executed. In this way we obtain, for each rule, one object satisfying its condition. This object is basically a representative of the equivalence class of all objects that satisfy the given rule conditions. Now, step 1 of the run-time configuration (section 4) is executed for each object. In other words, we execute the same call as if it

were an application service, but passing one of the already created objects as data. The steps then proceed as usual. This means that after step 7, we obtain a service instance that has been configured according to the rule corresponding to the given object.

The generation of the SQL code for the batch version of a service proceeds as follows, for each i from 1 to n (where n is the number of rules for that service). First, generate the SQL query to select all objects satisfying the condition of the i-th rule. This is done as shown above, but without the `TOP 1` qualifier. Second, call a special method on the i-th service instance. This method will generate the SQL code that implements the service, based on a template that is customized with the action parameters that have been set by the configurator on the service.

In summary, the SQL code that will be executed in batch is made up of n "modules", one for each rule. Each module first selects all objects to which the rule is applicable, and then executes the parameterized SQL code that corresponds to the Java code for the interactive version of the service. The modules are run sequentially, according to the prioritization of the rules.

This raises a problem. If some object satisfies the conditions of two or more rules, only the one with the highest priority should be applied to that object. To preserve this semantics, each batch service uses an auxiliary table, initialized with all objects to be processed by the service; in the case of the debt calculation service, it is the `AT_LATE_INTEREST_CALC` table. This table has a boolean column called `processed`, initialized to false. As each batch module executes, it marks each object it operates on as being processed. Hence, when the next module starts, its query will only select objects that haven't been processed yet. In this way, no two rules will be applied to the same object.

The last, but not least, point to mention are coordination contracts. As said above, one service instance has been created for each rule, and configured accordingly. This means that coordination contracts may have been superposed on some service instances (step 6). Hence, the SQL code generated from those service instances cannot be the same as for those that haven't any coordination contracts. The problem is that services are unaware of the existence of contracts. The result is that when the code generation method of a service object is called (step 8), the service object has no way to know that it should generate slightly different code, to take the contract's behaviour into account. In fact, it *must* not know, because that would defeat the whole purpose of coordination contracts: the different behaviours would be hard-wired into the service, restricting the adaptability and flexibility needed for the evolution of the business rules. Since the Java code (for the web-based part of the system) and the SQL code (for the batch part) should be in the same "location", to facilitate the maintenance of the system, the solution is of course for each contract to also generate the part of the code corresponding to the new behaviour it imposes on the underlying service. For this to be possible, the trick is to make the code generation method of the service also subject to coordination. In other words, when a contract is applied to a service object, it will not only intercept the methods supplied by

the service to its clients, it will also intercept the code generation method (step 9) in order to adapt it to the new intended behaviour.

6 Concluding Remarks

This paper reports on the first industrial application of coordination contracts, a mechanism we have developed for non-intrusive dynamic coordination among components, where "dynamic" means that the coordination may change during execution of the system.

Although the examples given are specific to the debt recovery domain, the framework we developed is generic and can be used to help organisations maintain their dependence on business rules and achieve flexibility of adaptation to changing rules. The framework provides a way to separate business rules from services and to apply one or more rules to services depending on the exact service call. Moreover, the approach is applicable to systems with an interactive and a batch part (based on SQL procedures), both subject to the same rules. This avoids the traditional scenario of duplicated business rules scattered among many system services and entangled within their code, thus helping to prevent unintentional faults by developers during software maintenance.

Flexibility of adaptation to change was achieved by two means. The first is the definition of parameterised business rule types. The condition parameters can be combined in arbitrary boolean expressions to provide expressivity, and priorities among rules of the same type allow to distinguish between general vs. exceptional cases. The second means are coordination contracts to encapsulate the behaviour that deviates from the default case. At run-time, from the actual data passed to the invoked service, a configurator component retrieves the applicable rules (at most one of each rule type), parameterises the service according to the rules, and creates the necessary contract instances. The contracts will intercept some of the service's functionalities and replace it by the new behaviour associated to the corresponding business rule.

The architectural framework we designed can be used both for interactive as well as batch application services. The difference lies in the fact that the batch application service has to get one representative data object for each rule, and only then can it create one processing service for each such data. The application service then asks each of the obtained configured services to generate the corresponding SQL code. Coordination contracts will also intercept these calls, in order to generate code that corresponds to the execution of the contract in the interactive case.

This approach has proved to work well for the system at hand. On the one hand it guarantees that the system will automatically (i.e., without programmer intervention) behave consistently with any change to the business rules. On the other hand, it makes possible to incorporate some changes to existing rule types and create new rule types with little effort, because coordination contracts can be added in an incremental way without changing the client nor the service code. Furthermore, the code of the services remains simple in the sense that it

does not have to entangle all the possible parameter combinations and behaviour variations.

The main difficulty lies in the analysis and design of the services and the rules. From the requirements, we have to analyse which rules make sense and define what their variability points (the parameters) are. As for the services, their functionality has to be decomposed into many atomic methods because coordination rules "hook" into existing methods of the contract's participants. As such, having just a few, monolithic methods would decrease the flexibility for future evolution of the system, and would require the coordination rule to duplicate most of the method code except for a few changes.

The approach is also practical from the efficiency point of view. The overhead imposed by the configurator's operations (finding the rules, passing action parameter values, and creating coordination contract objects) does not have a major impact into the overall execution time of the application and processing services. This is both true for the interactive and batch parts of the system. In the former case, the user does not notice any delay in the system's reply, in the latter case, the time of generating the SQL procedures is negligible compared to the time they will execute over the hundreds of thousands of records in the database. Moreover, the execution time of the generated SQL code is comparable to the original batch code, that had all rules hard-wired.

To sum up, even though we used coordination contracts in a narrow sense, namely only as dynamic and transparent message filters on services, and not for coordination among different services, we are convinced that they facilitate the evolution of a system that has to be adapted to changing business rules.

7 Acknowledgments

This work was partially supported by project AGILE (IST-2001-32747) and the Marie-Curie TOK-IAP action 3169 (Leg2NET), both funded by the European Commission; by project POSI/32717/00 (Formal Approach to Software Architecture) funded by Fundação para a Ciência e Tecnologia and FEDER; and by the research network RELEASE (Research Links to Explore and Advance Software Evolution) funded by the European Science Foundation.

References

1. Wermelinger, M., Koutsoukos, G., Avillez, R., Gouveia, J., Andrade, L., Fiadeiro, J.L.: Using coordination contracts for flexible adaptation to changing business rules. In: Proc. of the 6th Intl. Workshop on the Principles of Software Evolution, IEEE Computer Society Press (2003) 115–120
2. Andrade, L., Fiadeiro, J.L., Gouveia, J., Koutsoukos, G.: Separating computation, coordination and configuration. Journal of Software Maintenance and Evolution: Research and Practice **14** (2002) 353–369
3. Gouveia, J., Koutsoukos, G., Wermelinger, M., Andrade, L., Fiadeiro, J.L.: The coordination development environment. In: Proc. of the 5th Intl. Conf. on Fundamental Approaches to Software Engineering. Volume 2306 of LNCS., Springer-Verlag (2002) 323–326

4. Gouveia, J., Koutsoukos, G., Andrade, L., Fiadeiro, J.L.: Tool support for coordination-based software evolution. In: Proc. TOOLS 38, IEEE Computer Society Press (2001) 184–196
5. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. In: 3rd Information Survivability Workshop, Software Engineering Institute (2000)
6. Wermelinger, M., Koutsoukos, G., Fiadeiro, J., Andrade, L., Gouveia, J.: Evolving and using coordinated systems. In: Proc. of the 5th Intl. Workshop on Principles of Software Evolution, ACM (2002) 43–46
7. Moazami-Goudarzi, K.: Consistency Preserving Dynamic Reconfiguration of Distributed Systems. PhD thesis, Imperial College London (1999)
8. K.Lano, J.L.Fiadeiro, L.Andrade: Software Design Using Java 2. Palgrave Macmillan (2002)
9. Gelernter, D., Carriero, N.: Coordination languages and their significance. Communications of the ACM **35** (1992) 97–107
10. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17** (1992) 40–52
11. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press (1996) 3–14
12. Kramer, J.: Configuration programming—a framework for the development of distributable systems. In: International Conference on Computer Systems and Software Engineering, Israel, IEEE (1990)
13. Endler, M., Wei, J.: Programming generic dynamic reconfigurations for distributed applications. In: Proceedings of the First International Workshop on Configurable Distributed Systems, IEE (1992) 68–79
14. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
15. Monroe, R.T.: Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University (1998)
16. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
17. Kilov, H., Ross, J.: Information Modeling: an Object-oriented Approach. Prentice-Hall (1994)
18. Notkin, D., Garlan, D., Griswold, W., Sullivan, K.: Adding implicit invocation to languages: Three approaches. In: Object Technologies for Advanced Software. Volume 742., Springer-Verlag (1993) 489–510