Using Coordination Contracts for Evolving Business Rules[*]

Michel Wermelinger
Dep. de Informática
Univ. Nova de Lisboa
2829-516 Caparica, Portugal
mw@di.fct.unl.pt

Georgios Koutsoukos,
Hugo Lourenço, Richard Avillez,
João Gouveia, Luís Andrade[†]
ATX Software SA
Alameda António Sérgio, 7, 1C
2795-023 Linda-a-Velha, Portugal

José Luiz Fiadeiro
Dep. of Computer Science
Univ. of Leicester
University Road
Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract

This experience paper reports on the use of coordination contracts in a project for a credit recovery company. We have designed and implemented a framework that allows users to define several business rules according to pre-defined parameters. However, some rules require changes to the services provided by the system. For these, we use coordination contracts to intercept the calls to the underlying services and superpose whatever behaviour is imposed by the business rules applicable to that service. Such contracts can be added and deleted at run-time. Hence, our framework includes a configurator that, whenever a service is called, checks the applicable rules and configures the service with the given parameters and contracts, before proceeding with the call. Using this framework we have also devised a way to generate rule-dependent SQL code for batch-oriented services.

Based on our experience, we feel that coordination contracts facilitate the evolution of the system in order to accomodate new business rules that change the "normal" behaviour of the provided system's functionalities.

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

- the externalisation of the business rules that define the dependency of the recovery process on the financial institution and product (e.g., house mortgage) for which the debt is being recovered;
- the encapsulation of parts of behaviour into so-called coordination contracts that can be created and deleted at run-time, hence adapting computational services to the context (e.g., institution and product) in which they are called.

[*]Work partially supported by project AGILE (IST-2001-32747) funded by the European Commission; by project POSI/32717/00 (Formal Approach to Software Architecture) funded by Fundação para a Ciência e Tecnologia; and by the research network RELEASE (Research Links to Explore and Advance Software Evolution) funded by the European Science Foundation.

[†]E-mail: {hlourenco,ravillez,jgouveia,landrade}@atxsoftware.com

These two mechanisms have two different stakeholders as target. Business rules are intended for system users, who have no technical knowledge, so that they can parameterise the system in order to cope with requirements of new financial institutions and products. Coordination contracts are intended for system developers to add new behaviour without changing the original service implementation. This is made possible by the ability of coordination contracts to intercept calls to the service's methods and execute the contract's code instead.

Coordination contracts [1] are a modelling and implementation primitive that allows transparent interception of messages and as such replace the service's method by the code provided by the coordination contract. 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 creating 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 business 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 our approach to the system we developed, all coordination contracts are unary, the participant being the service affected by the business 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 lead to less efficiency and to more complex rule conditions. 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.

We should also mention that in this project we used our environment to develop Java applications using coordination contracts [3]. The environment is freely available from www.atxsoftware.com. 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 [2]. 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.

The structure of the paper is as follows. The next section introduces some example business rules, taken from the credit recovery domain, and shows how coordination contracts are used to change the default service functionalities according to the applicable business rules. Section 3 sketches the framework we implemented, describing how the service configuration is done at run-time according to the rules. Section 4 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 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 two 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 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()`.

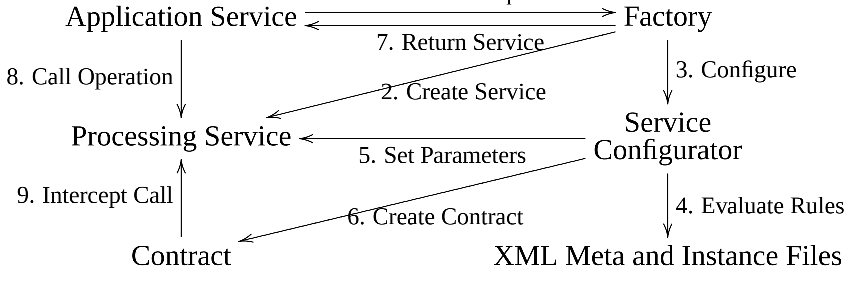
Contracted Rate This rule intercepts the `getRate()` method under the condition `!calculated`, and executes: `r =` the rate obtained by consulting the database`; setRate(r); calculated = true`.

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.

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.

3 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. 1 on page 6 for an example), while the instance file contains the actual rules (see Fig. 2 on page 7). 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. 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 to 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. 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) processing 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).

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 [4]: 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,edit 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 is 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.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.

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>
```

Figure 1: 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>
```

Figure 2: An extract of the XML instance file

4 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.

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. 1 on page 6 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. 2 on page 7: it is applicable to all recovery processes of "Big Bank" that 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 3) 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.

5 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.

One of the key requirements of the debt recovery system ordered to ATX Software was the flexibility of adaptation to new client institutions and new financial products. This flexibility 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.

6 Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] L. Andrade, J. L. Fiadeiro, J. Gouveia, and G. Koutsoukos. Separating computation, coordination and configuration. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):353–369, 2002.

[2] J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS 38*, pages 184–196. IEEE Computer Society Press, 2001.

[3] J. Gouveia, G. Koutsoukos, M. Wermelinger, L. Andrade, and J. L. Fiadeiro. The coordination development environment. In *Proc. of the 5th Intl. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 323–326. Springer-Verlag, 2002.

[4] M. Wermelinger, G. Koutsoukos, J. Fiadeiro, L. Andrade, and J. Gouveia. Evolving and using coordinated systems. In *Proc. of the 5th Intl. Workshop on Principles of Software Evolution*, pages 43–46. ACM, 2002.