Role-based Multi-purpose Workflow Engine Architecture

Sebastian Richly, Sebastian Goetz, Uwe Assmann and Sandro Schmidt

Software Engineering Group, Dresden University of Technology, 01062 Dresden, Germany

Abstract. The workflow management systems domain today is completely fragmented. For each purpose various solutions with different specializations exist. Even for standardized process languages, many different extensions and engines exist. If new requirements, domains or standards emerge, the engines have to be adopted. In this paper, we want to show how a workflow engine can be designed to support different workflow languages and different domains - an extensible multi-purpose workflow engine. Our approach for this kind of engine is based on a workflow net engine that allows us to support most of the existing workflow languages. To support different tasks of different specifications, we integrated *object roles* in our engine. This extension of the object-oriented paradigm allows flexible runtime adaptations and extensions. Thus, we are able to add new domain specific functions to our engine at runtime, even if the original process language does not support them.

1 Introduction

A workflow process is the executable representation of a business process in a workflow management system. A workflow management system does not only execute these processes. Most systems support modeling, monitoring, integration of external applications, organization structures and administration, too. Workflow management systems operate on workflow specifications. They all define a set of *activities* (also called *tasks*) and their order of execution. Over the years, a completely fragmented workflow language ecosystem has been developed. After the introduction of SOAP Web services, the number increased further and started a new wave of research. BPEL, BPMN, ebXML BPSS, WS-CDL are prominent representatives of these languages; XPDL, jPDL or YAWL are examples for classic process languages from the pre-SOA era. The corresponding workflow engines, which all differ in their specific description, deployment and configuration, are incompatible, although the language is standardized.

Another deficit of today's workflow engines is their missing runtime flexibility. For example, if a Web service invocation fails, current engines search for another adequate service or initiate the predefined exception handling. It is not possible to bypass the call to an Enterprise Java Bean because this runtime adaptation is not feasible in the engines architecture. Now, many specialized and inflexible workflow engines exist, but there is no workflow management system that can be used in a flexible way and for different workflow languages. Exception handling techniques react only, we intent to

Richly S., Goetz S., Assmann U. and Schmidt S.
Role-based Multi-purposeWorkflow Engine Architecture.
DOI: 10.5220/0003011300450054
In *Proceedings of the International Joint Workshop on Technologies for Context-Aware Business Process Management, Advanced Enterprise Architecture and Repositories and Recent Trends in SOA Based Information Systems (ICEIS 2010)*, page
ISBN: 978-989-8425-09-6

build a proactive solution. Hence, we present a workflow engine design with its focus on extensibility and runtime adaptation. The approach is based on two main techniques:

Workflow Net [1] / Petri Net: As pointed out in [2], the implementation of a workflow engine with workflow nets has several advantages compared to *Directed Acyclic Graphs*: (a) the formal semantics despite the graphical nature, (b) the state-based structure, and (c) the availability of analysis techniques (check deadlock, liveness, fairness).

Role-Oriented Programming (ROP)[3]: This approach of modeling and programming systems is an extension to the classic object-oriented paradigm. The term *role* (or object role) is not related and can not be compared to the classic role term in workflow systems - also called Role-Based Access Control model (RBAC). In a role universe core types (these are the classical objects) and a set of role types exist. A core object can play a role instance, this means core type and role type are linked by the *can-play-a* relationship. For example, a *Person* can play a *Father* and a *Customer* role. Players are able to start and stop playing roles at runtime. Notably, roles change the behavior of core objects (like aspects in aspect oriented programming) and are able to store data, which is only applicable for the role and not for the core.

By combining these two essential techniques, we build a multi-purpose workflow engine with five essential characteristics:

1. Support of all petri-net-compatible process languages.
2. Mix of concepts of different process languages.
3. Dynamic extension of new task types. (e.g. invoke geospacial web service)
4. Runtime flexibility of the task configurations and the execution semantics.
5. Customizable description language.

This paper is structured as follows. In the subsequent section, we describe role-oriented programming. In Section 3, we present our workflow model, and illustrate the architecture and functionality of the role based workflow engine and explain its implementation in Section 4. Section 5 takes a look at the current state of multi-purpose workflow engines and other related work.

2 Role-oriented Programming

Designing such a flexible workflow engine is not easy with object-oriented techniques. For example, to support new types of tasks, two main possibilities are available. *Multiple inheritance* is one of them, but is a rather static approach, because instances cannot change their type dynamically. *Delegation* is the second possibility, but delegation does not support type safety. Thus, we had to look for another opportunity and chose role-oriented programming [3].

Imagine the following scenario: In a workflow engine there is a *task* which should execute something. This could be a *WebService-Call*, *GeoWebService-Call* **and** *EJB-Call*. In object-oriented programming (OOP), at least four classes are necessary to model the scenario in Figure 1. The task can only be an instance of one of these three specializations at runtime. A *Task* either can be an instance of *WebService-Call* or, for example, of *EJB-Call*. There is no inheritance-based solution for this problem. The only way to

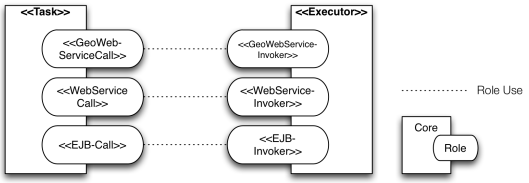

Fig. 1. Roles Example.

express this scenario is to use delegation, thereby losing type safety and the conceptual unity of the core and its roles. This is, because only the type of the core object, but not its role types are known when passing references to the player. A further feature of roles is that objects are able to play multiple roles as it is shown in Figure 1. By introducing the specific calls as roles having the *Task* as core, the *Task* can play all three roles at the same time.

Another characteristic of role types is that they are founded. Founded types are relative types that depend on a collaboration partner. The role *WebSerivce-Call* does not only describe which service has to be invoked, but needs some kind of executor. For example, a role instance of *WebService-Call* needs a core object of *Task* and the counter role *WebService-Invoker* of *Executor* to collaborate at runtime. The *WebService-Invoker* is responsible to invoke a single web service. It is the counter role of *WebService-Call* and vice versa, meaning that they communicate with each other. At runtime, the engine has to decide, which corresponding counter role to choose.

All these constraints are defined conceptually. No currently available mature language supports such constraints on the level of types. Hence, they are realized as additional code inside of methods. The more complex the collaborations get, the harder such implementations are to maintain. Section 3 presents a solution to this problem.

3 Role-based Workflow Engine

To build a general-purpose workflow engine, it is important to know the aspects that have to be considered. The so-called ”aspects” of workflows give a guideline to the main concepts encountered in workflow languages that should be supported by a multi-purpose engine. These aspects describe the essential functionality for a workflow engine/language. A good overview of the aspects is collected in [4] and the workflow meta-model by the Workflow Management Coalition (WFMC).

Based on these aspects, we developed our role model. The role space is partitioned by contexts, derived from the essential workflow aspects. Figure 2 shows the role and object space on a high abstraction level. The square forms in the behavioural context indicate core objects and the rounded forms in the contexts are symbols for roles. The elements of the behavioural context can play the roles of the contexts which is shown by the ”plus sign”.

3.1 Workflow Aspect Role Space

The core is represented by the behavioral context. This context contains the essential object model of our base workflow net, which was introduced in [2].

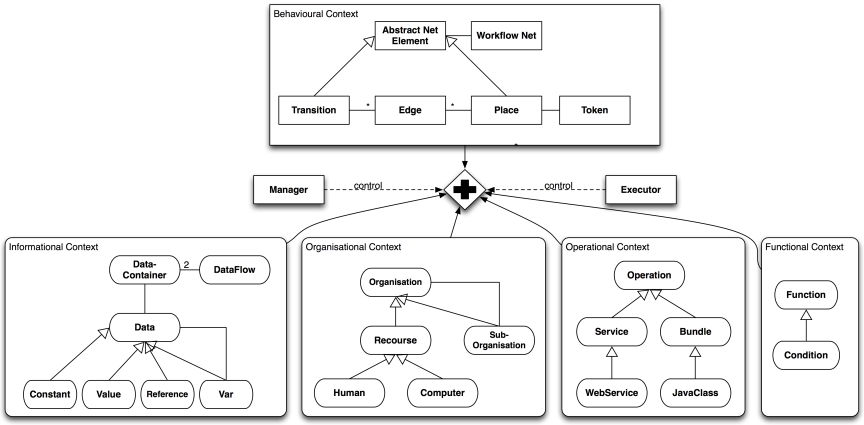

Fig. 2. Basic Role Space.

The behavioral context represents the core context and thus contains the individuals of the workflow system which can play roles. In all other aspects we expect changes, hence they must be flexible. We focused on the control flow aspect because the roles should only change the runtime behavior of several tasks, not the structure of the entire process definition.

The roles need these individuals (see Figure 2): without them, they cannot be instantiated. These are *Transition, Edge, Token, Place* and the *Workflow Net* itself. These five individuals are static, that is, they cannot be removed from the engine at runtime in contrast to roles that are able to refine these individuals. Considering the workflow application domain, a transition expression is often represented by a task execution. Transitions are atomic operations that, when fired, consume tokens from the input places and put tokens into the output place, which stores the token until the next transition can be fired. The tokens can be typed through the roles of the *informational context*.

The *informational context* is responsible for the data itself and the dataflow. All default entities in the context are roles, which are associated with each other. The base role interface is the *datacontainer*, which can refer to multiple *data* roles. The data construct is designed using the classic composite design pattern. Using the recursion of *Var*, structured data types can be constructed, which is necessary for BPEL or other Web service composition languages. The other basic roles are *constant*, a variable *value* and a *data reference*.The number of default implementations in these two contexts is much higher than in the other contexts because they represent the most common part of all workflow definitions.

The other contexts are not modeled in such detail and need to be refined to support the different languages. The *functional context* describes data for the execution of different functions in the workflow engine. There are only two role interfaces: *function* and *condition*. A function could be an invocation of a Web service. The available Web services are described in the *operational context*. It contains only one interface: the *operation*. The derived *service* and *bundle* are only sample implementations to support BPEL and JPDL.

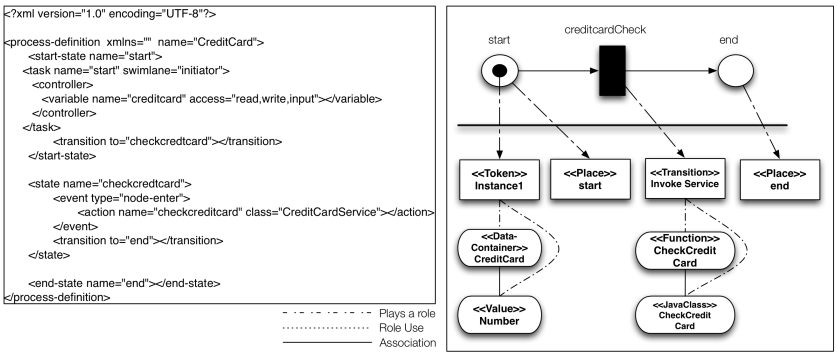

Fig. 3. From JPDL to role space.

The *organizational context* describes the structure of an organization and a general resource. This could be a human or a computer resource. Thus, we support user tasks and automated tasks by assigning the corresponding roles to the core identities.

In addition to the classic contexts, we added two components. The first one is the *manager*. This component monitors and controles all the core-role playing and checks all role model constraints. Because the role space should be extended at runtime, the manager is designed as an extensible component. Other managers can register themselves with the central manager and they will be invoked during validation. This enables to add new role constraints if new roles are added to the role space.

Figure 3 shows a simple JPDL process definition which had been transformed to our role space. It shows a process to check a credit card number. The process consists of a single transition and a token. The transition transports the token to the end. Thus, it knows the token and can access the data it transports. The transition itself plays the function CheckCreditCard which is associated to a Java class call. Both have access to the DataContainter (DC) CreditCard through CheckCreditCard.

The second component is the *executor*, which interprets the workflow net and serves as execution environment for the roles. Functions of basic roles can only be executed in conjunction with the assigned counter roles. For that purpose, the executor provides the counter roles for the basic roles. For example the executor provides the *WebSerivce-Invoker* role for the base role *WebSerivce-Call*. The ”call role” serves the configuration data which is interpreted by the ”invoker role”. This scenario is shown in Figure 1. But someone has to know, which is the correct counter role. Accordingly the role-counter role mapping is primarily instantiated at execution time and not at design time. Thus, the *manager* helps the executor allocates the referenced roles. The correct role mapping is expressed as simple key-value pair $role = counterrole$. This knowledge base is integrated in the manager and can be extended at runtime. In Figure 3 the role association between function *CheckCreditCard* and DataContainer *CreditCard* is created by the executor using the basic role interfaces. The concrete communication depends on the implementation of the roles.

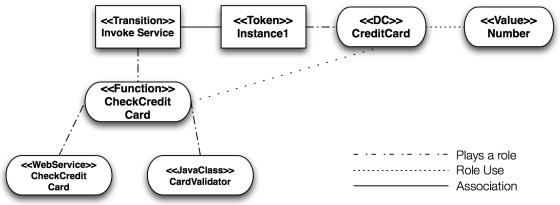

Fig. 4. Example Role Space for One Transition and One Token.

3.2 Dynamic Extension

The main advantage of using roles in this context is the dynamic extensibility of the workflow engine. In contrast to classic inheritance-based extensibility - provided by decorator, composite and other design patterns - roles can be attached or detached at runtime, with a component type. Using this characteristic, we build up a dynamic extensible workflow engine.

The basic roles and role interfaces are provided by the engine itself. Every extension can use or inherit them, except if they violate the role constraints. New roles can be added by registering a new role bundle to the engine during runtime. The role bundle has to provide two essential components: The role registry of the bundle, where all roles are listed and a role manager extension. The registry is used by the role runtime to instantiate the roles. The manager extension will be automatically registered with the central manager. This is necessary to provide a safe role space in the entire engine.

A role bundle can be detached at runtime. If so, the roles are no longer available for all designed and running processes. But this can lead to an invalid state of the complete engine. Thus, before the roles can be detached, every running process has to finish. This can be done through canceling or simply by waiting for the regular ending. Canceling is not an adequate procedure in productive systems. Hence, the solution is to delete the registration of the role bundle, but to let it stay active and connected until all related processes have come to an end. The advantage of this approach is that no new processes can be started if they use the unknow roles. The validator checks the registry at every request and returns an *invalid*-message to the executor if a constraint is violated or if a role is not available any longer. In such a case, the executor will not start the execution.

3.3 Dynamic Flexibility

Figure 4 shows a modified sample configuration of Figure 3. The transition *Invoke Service* itself plays the function CheckCreditCard which is now associated to two operational roles: WebService and JavaClass. This means that the function can be executed by both operational roles. Both have access to the DataContainter(DC) CreditCard through the function CheckCreditCard. The executor and manager can decide at runtime, which counter role they use for this function. If one call fails the executor can then call the other one in the exception handling strategy. Our dynamic role-based engine also allows us to adapt the designed processes at runtime. If we look at the example transition in Figure 4, the transition has one function and two available operations to invoke, namely a WebService and a JavaClass, which is located on the engines server. Both

can be invoked with the variables of the data container from the token. If the engine supports both roles, it can choose which one of them to invoke. This can happen with a random function or in accordance to quality-of-service parameters. For example, if the engine recognizes that the Web service answers very slowly to requests, it can use the Java class implementation. If the process gets transferred to another engine with only one of the two possible roles, the executor can only invoke the one activated on the system. Thereby, activities in the process can be designed in a more flexible way. Instead of compensation handling, if the Web service is not available or if the response time is bad, we can model variants at design time and avoid compensation handling.

This type of "flexibility by configuration" is the basic kind of adaptation. [5] presents an approach establishing flexibility/adaptation using workflow inheritance. This approach can be realized through our role based engine but in our approach it is flexible at runtime. An approach with inheritance is just too static. In our simple example it would be possible to design a Task which implements both types- WebService and JavaClass - but if a third invocation method is added we have to redesign the application. The role approach enables a dynamic type system, which is more flexible in adaptive systems.

3.4 Support of other Workflow Languages

As we stated in the last sections, a role-based workflow engine is very flexible at runtime and even during the execution of processes. But we can use these features to support and import different workflow languages, too. To import several workflow languages in our engine, we designed a two-step procedure. The first step is to map the control flow structures to the workflow net object structure. The second step is to map resources and the activity description.

Building on the workflow nets, we designed our generic importer that contains several interface methods to be implemented by the concrete importer for each language. These interface methods represent the two-step import. First, we import the control flow; second, if we have mapped the control flow to the behavioral context, all the activity and resource configurations have to be mapped. The mapping has to be provided by the importer, too. The configuration data has to be transferred to the roles. To ensure the smooth import, every importer has to provide a list of required roles for a complete mapping. If a role is not provided by the engine, it can be installed immediately (see 3.2 and 3.3).

3.5 Mix of Domain Concepts and Domain-specific Workflow Engine

Aside from importing different processes from different domains, another advantage of a role-based workflow engine appears. The different roles, deployed in the engine, represent different domains, such as Open Geospatial Web Service roles running side by side with SOAP or EJB roles. If a common BPEL process has been imported to the engine, it is no longer restrained by the BPEL corset. The process is now open for different domains, which are provided by the roles in the engine. For example, we can add EJB-RMI calls to a former invoke. With this mashup of different domain concepts, we can design unique process definitions.

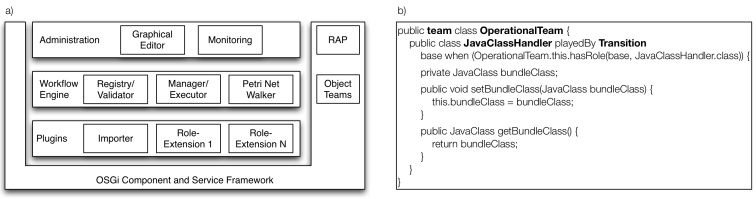

Fig. 5. a) Architecture and b) JavaClass Role definition.

On the basis of the presented new features of a role-based workflow engine, completely new application areas can be covered with one single engine design. These application areas are divided in two parts: language- and architectural-specific areas. First of all, the role-based engine can be adopted to many software and hardware architectures. It is possible to create different variants. For example, in embedded systems, only a basic set of roles is available because of the limited hardware; the same engine can also be configured for a full-fledged engine for high-end servers.

But if we can easily configure an engine for special hardware architectures only by configuring the role space for a use case, we can also specify our own domain-specific workflow language. As described in the previous section, with a role space, we can mix several domain concepts within one engine. The mix can be fixed and described in a special domain-specific workflow language. A company can now build a special adapted language and is no longer restricted to a language that does not fit the companies requirements.

4 Implementation

We realized our role-based workflow engine in the Open Service Process Platform [6, 7]. This engine is developed at TU Dresden as an OSGi application.

The role-based engine is a step towards a novel kind of workflow engine, which affects all three stated requirements. The base for all implementations is the OSGi framework. In this component framework, we can define so-called *bundles*, which are often called plug-ins, too. The OSGi framework is responsible for the runtime and lifecycle of each bundle. One central concept is the extension point. These points are slots in a bundle where other bundles can hook in to extend the original implementation. We used this concept to define extension points for our importer, the manager extensions and the workflow contexts themselves. One special feature of OSGi is the dynamic bundle integration at runtime, which allows for the dynamic extension and adaptation at runtime (see 3.2 and 3.3). Figure 5 a) shows the OSGi container including our bundles. The OSGi runtime contains *Object Teams* for the role support. The base workflow engine contains a workflow net walker (execution of the workflow net description), the manager and executor including all roles. Additionally users can add new role bundles containing the role description, executor role and a validator. But OSGi does not support roles as first-order programming paradigm. Thus, we had to integrate another framework that supports roles. We decided to use Object Teams (short OT) [8] for this purpose. OT is realized as an extension to the Java language that does not need a modified Java Virtual Machine and introduces two new elements - Teams and Roles which

you can see in Figure 5 b). Teams are modeled like classes and they are denoted by the additional keyword *team*. In our implementation, every workflow aspect is described as a team with all the roles we showed in 3.1. The workflow net engine is implemented in an extra OSGi bundle and is designed following the work in [2] and represents the core context. The different contexts are represented by teams, which contain the roles. In Figure 5 b) you can see the operational team containing one role - the JavaClassHandler. The JavaClassHandler can be played by a transition. This role only contains a description of the JavaClass (and Java methods) to call, but can not be executed alone. The executor has a corresponding counter role to execute this description (in this case just using the Java reflection API). With the described mechanisms, it is possible to build a role-based workflow engine. First, we implemented a basic importer and roles for BPEL. Then, we also built an importer for JPDL and implemented the JavaClass role to show the flexibility of our approach but the approach is not limited to these two languages.

5 Related Work

The subject of interoperability between workflow languages has been discussed in different ways. The Workflow Management Coalition [9], a standardization organization for the workflow domain, has defined several interfaces for workflow systems as well as one for the import and export of workflow definitions. They use a common workflow model for the exchange, like XMI for UML models. This common model is the well known XPDL.

The extensible routing language - XRL - also bases on XML with a DTD. In several papers [10, 11] the authors present the basic functions (comparable to BPEL) and several transformations to workflow nets. Processes are executed by a workflow net engine. New elements can be added through changing the DTD. In contrast to our approach the changes can only be done at design time and the engine does not support runtime flexibility through different configurations.

Runtime adaptation is also possible with some aspect oriented approaches like AO4BPEL[12]. These approaches allow to weave in new tasks or to replace designed behavior, but these approaches are limited to specific languages and engines and do not allow to produce customizable workflow definitions.

Other approaches can be found in different areas. In [13], the authors present a portal supporting the interoperation between different workflow languages in the grid domain. They use an XSLT converter, which translates between the workflow languages.

6 Conclusions

In this paper, we presented an approach for a new type of workflow engines. Based on the powerful combination of workflow nets and role-based programming, we showed an architecture for more flexible and integrative workflow engines. We do not introduce a new workflow language or a new interoperability model, but we present a simple workflow net model enhanced with a basic role model that can be easily extended

at runtime. Through this combination, we are able to support unique features: (1) Dynamic extension by new constructs or task definitions, (2) runtime adaptation of the role configurations and the execution semantic, (3) support for many workflow languages, and (4) the mix of concepts of different domains.

We implemented our approach in the Open Service Process Platform using OSGi and Object Teams. In our future work we try not only to adopt the runtime behavior of the worklfow but also on non-functional aspects such as security which where not in the focus of the paper. In a future work we will extend our workflow engine to support workflow nets, which overcome some disadvantages of pure workflow nets.

References

1. Van Der Aalst, W.M.P.: Verification of workflow nets. In: ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, London, UK, Springer-Verlag (1997) 407–426
2. Pellegrini, S., Giacomini, F.: Design of a petri net-based workflow engine. In: GPC-WORKSHOPS '08: Proceedings of the 2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops, IEEE Computer Society (2008) 81–86
3. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowl. Eng. (2000) 83–106
4. Jablonski, S.: Mobile: A modular workflow model and architecture. In: in Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems. (1994)
5. Van der Aalst, W.M.P., Jablonski, S.: Dealing with workflow change: identification of issues and solutions. International Journal of Computer Systems Science and Engineering (2000) 267–276
6. Richly, S., Habich, D., Ruempel, A., Buecke, W., Preissler, S.: Open service process platform 2.0. In: Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008, 8-11 Jul, Hawaii, USA). (2008)
7. Richly, S., Buecke, W., Assmann, U.: A bdi-based reflective infrastructure for dynamic workflows. Enterprise Distributed Object Computing Workshops, International Conference on (2008) 112–119
8. Herrmann, S.: Object teams: Improving modularity for crosscutting collaborations. In: NetObjectDays. (2002) 248–264
9. Hollingsworth, D.: Workflow management coalition specification: the workflow reference model. Technical report, WfMC specification (1994)
10. Van der Aalst, W.M.P., Kumar, A.: Xml - based schema definition for support of interorganizational workflow. Information Systems Research (2003) 23–46
11. Verbeek, H.M.W., Van Der Aalst, W.M.P., Kumar, A.: Xrl/woflan: Verification and extensibility of an xml/petri-net-based language for inter-organizational workflows. Inf. Technol. and Management (2004) 65–110
12. Charfi, A., Mezini, M.: Aspect-oriented web service composition with ao4bpel. In Zhang, L.J., ed.: ECOWS, Springer (2004) 168–182
13. Huang, L., Akram, A., Allan, R., Walker, D.W., Rana, O.F., Huang, Y.: A workflow portal supporting multi-language interoperation and optimization: Research articles. Concurr. Comput. : Pract. Exper. (2007) 1583–1595