DENEB: a platform for the development and execution of interoperable dynamic Web processes

J. Fabra*, †, P. Álvarez, J. A. Bañares and J. Ezpeleta

Aragón Institute of Engineering Research (I3A), Department of Computer Science and Systems Engineering, University of Zaragoza, Spain

SUMMARY

Service-oriented computing provides a suitable technological foundation for developing and executing dynamic business processes. However, most current approaches for composition languages and architectures for dynamic process integration do not provide the flexibility and dynamism required by interorganisational evolving environments. In this work, the DENEB platform for the development and execution of Web processes is presented. DENEB is based on the conversational approach and the Nets-within-Nets paradigm, allowing Web processes to acquire and execute new interaction protocols at run-time. This makes DENEB a well-adapted and flexible platform for dynamic service composition, interaction and integration, suitable for this kind of scenario. The scalability of the proposed solution is evaluated by means of the development of a set of experiments. Copyright © 2011 John Wiley & Sons, Ltd.

Received 30 May 2009; Revised 7 April 2011; Accepted 4 May 2011

KEY WORDS: service-oriented architectures; process composition; interaction protocols; dynamic approaches to integration; Petri nets; Linda coordination

1. INTRODUCTION

Service-oriented computing (SOC) is the computing paradigm that allows developers to create applications using services as basic elements. In terms of interaction models, the interoperability among components is achieved by means of the description of service capabilities and their interactions. Different standards have been specified for these purposes, such as the Web services standards (see, e.g. Business Process Execution Language (BPEL), Web Service Choreography Description Language (WS-CDL) or Web Service Description Language (WSDL)) or some semantic standards (see, e.g. the Web Ontology Language-based ontology for the Semantic Web (OWL-S), Web Service Modeling Ontology (WSMO), WSDL-S or the Semantic Web Services Framework (SWSF)).

Because of their independence to computing models, Web services represent a promising technology to deal with the requirements of service integration in dynamic and evolving environments. However, systems developed with current Web service technologies are often limited by the need of software developers to know in advance the interface of the required software components as well as the interaction protocols. The next generation of solutions requires more flexible interoperability capabilities in order to implement (Internet-based) applications combining services in an evolving environment, such as business management scenarios [1]. A particular case of such applications are Web processes. These processes run in an interorganisational environment and have to be able to dynamically adapt themselves to new business policies and strategies and to...
handle, at run-time, unexpected events and changes [1–4]. The level of flexibility allowed by the Information Technology infrastructure in which these processes are deployed determines the way an organisation can gear itself towards the real world of fully automated, complex electronic transactions.

A step towards this flexibility can be made by means of a separation between the business logic of a process and the interaction logic required in order to access and invoke the composing services (these two different aspects are commonly referred to as orchestration and choreography, respectively [5]). The conversational approach is a way of dealing with such a separation [6–8]. This approach allows developers to reuse the same business logic with different interaction protocols during the process life cycle, helping to reach the integration requirements of interorganisational environments. Unfortunately, some popular process description languages do not allow this level of flexibility to be reached because of the existing dependence between both orchestration and choreography aspects [9]. A clear example of this lack of flexibility is BPEL [10]. Although BPEL was designed in order to provide developers with an interoperable and portable language for both abstract and executable processes, BPEL was specially based on the flexibility provided by the Extensible Markup Language (XML) [11]. BPEL processes integrate the business logic and the interaction logic within the same process description, so a change in one of these logics requires the review and recoding of the whole process.

In this work, the DENEB platform for the Development and Execution of Web processes is presented as a valid and flexible approach for the development and execution of processes, decoupling orchestration and choreography aspects using the conversational approach [12, 13]. The business logic of a process is implemented by means of a workflow, which starts and executes roles (implementing the interaction logic by means of protocols or conversation policies) when some interactions with published services are required. DENEB uses Petri nets to implement both workflows and conversations from the perspective of the Nets-within-Nets paradigm [14].

The main contributions of this work are as follows: (i) the application of SOC patterns and blueprints and the conversational approach to promote the flexibility of the infrastructure when interacting in an evolving interorganisational environment by means of the development of DENEB, covering the critical aspects presented in [1]; (ii) the development of a flexible and dynamically extensible Linda-based message broker component to explicitly separate external communication details and concrete technological issues from modelling aspects; and (iii) a dynamic loading component able to manage the acquisition of both workflow and role implementations, and also new mediators, at run-time. This component allows processes to adapt to new business goals and environment requirements and also to consume or provide new services and functionalities in a nonagreed way.

The paper is organised as follows. Section 2 provides an overview of the requirements for the management of dynamic Web processes and proposals for process integration. The conversational approach is then introduced. In Section 3, the reference architecture of DENEB is depicted. Section 4 gives details of the implementation of the framework, describing the dynamic loading component architecture that supports the dynamic capabilities of DENEB and giving an overview on how workflows and protocols are constructed and implemented. An example showing the capabilities of the DENEB platform when dealing with interaction protocols in a dynamic way is described in Section 5. Finally, Section 6 depicts the set of experiments that were performed in order to evaluate the scalability of DENEB, and the obtained results are discussed.

2. BACKGROUND

Web process integration requires two main goals to be achieved: a descriptional approach and a valid infrastructure supporting the requirements of the integration process. Current approaches fall into static aspects, which cannot deal with the requirements of SOC for new trends of processes. In this section, the main problems of current proposals for process integration, the new requirements and the capabilities that a valid infrastructure should be able to deal with are analysed.
2.1. Management of dynamic Web processes

Business organisations offer their business processes as Internet accessible processes to form virtual partnerships with other organisations. In the highly competitive, rapidly changing and expanding global marketplace, these partnerships should be carried out by means of flexible dynamic business processes that integrate Web processes provided by other organisations. Unlike traditional business processes, dynamic processes must be able to accommodate the changing business policies and strategies of participating organisations to handle expected and unexpected events and also to support run-time modifications of their business and interaction logics [4].

Service-oriented computing provides a suitable technical foundation for developing and executing dynamic business processes. Nevertheless, many research challenges must still be tackled. From the implementation point of view, composition languages should provide abstractions to support the required flexibility and dynamism. Business processes should be able to discover and to interact with new partners to automatically select the most suitable partner candidate for a given business collaboration. All these aspects should be possible at run-time.

From the architectural point of view, the concept of Enterprise Service Bus (ESB) is emerging as an appropriate infrastructure for dynamic process integration. The possibility of plugging and unplugging new business applications and other components to the Enterprise Service Bus should allow the easy configuration of highly dynamic business scenarios where solutions are organised as sets of collaborations among autonomous partners looking for a common goal. Regarding this subject, research experiences in multiagent systems must be considered [15].

The very interesting SOC research roadmap published in [1] highlights some of the most notable research challenges that must be considered in the near future when looking for dynamic business process support. Let us now summarise the most important aspects:

- Process workflows should be able to configure themselves to cope with their inevitable evolution and the required adaptation to changes.
- Business requirements should be integrated into the workflow’s life cycle.
- Instead of service interactions being defined in advance, process workflows should be able to discover, select and interact with new partners at run-time.
- In a highly dynamic environment, processes involved in an interaction could be autonomously redefined by external organisations. Thus, a support for process interactions, more flexible than approaches based on process discovery and selection, should be provided.
- A process execution environment should have the critical ability to route process interactions (sequences of exchanged messages) through a variety of transport protocols and to transform from one protocol to another when necessary.

In this work a valid and flexible approach for the development and the execution of processes, which covers these critical aspects, is presented. Orchestration and choreography aspects have been decoupled using the conversational approach [12, 13], and a set of facilities have been provided in order to integrate current standards and technologies and to deal with flexibility and dynamic aspects.

2.2. An overview of proposals for process integration

Service-oriented computing relies on the construction of complex systems using services as the basic construction unit. Services represent independent entities that can be used in an independent way. The integration of such services and the management of the required infrastructure are the key elements of current Information Technology scenarios.

Interoperability among components requires the description of service capabilities and their interactions to be published. In this respect, a detailed description might allow the integration of almost any entity in a context-independent way. This process of publishing services involves two main aspects: composition and interaction. The terms orchestration and choreography have been widely used in this area [5]. Orchestration refers to the description of how services should be arranged in terms of the business logic of the process. This logic is hidden and represents the internals of the process. The term choreography denotes the description of the correct and valid sequences of messages...
that can be exchanged between processes in order to perform some task or operation. Choreographies are kept public, because choreographies define the rules of participation for collaboration and must be understood and agreed by all participants.

Composition and interaction aspects should be managed in a platform-independent manner, allowing the reuse of service descriptions by several different organisations. A natural solution of this issue is the use of standards. Currently, different standards have been specified for these purposes, being competing initiatives for the specification of processes. For example, the WS-CDL specifies the behaviour of participants in a collaboration [16]. On the other hand, BPEL targets orchestration and also some choreography aspects, being the most widely used for this aspect [10].

Current approaches are far from achieving a correct and automatised integration of processes. They mainly provide static descriptions of the logics involved in the process life cycle, and service interactions must be imposed at a previous stage. New trends in service composition require ad hoc and dynamic characteristics [1]. New approaches should provide adaptive and flexible methodologies for the construction of Web processes. Currently, there is a need for languages that allow not only the definition of the internal business logic but also the requirements of external services, providing an automatic way to link them with external and heterogeneous processes.

2.3. The conversational approach

The Internet infrastructure, together with Web based technologies, greatly facilitates the development of (distributed) processes that interact using that environment. Web processes are the applications that live in this world. A Web process is composed of a set of tasks to be executed according to a specified set of ordering constraints. From an implementation point of view, workflows are a suitable technology for the description and execution of Web processes, representing what is usually called business logic. On the other hand, the execution of each task usually corresponds to the invocation of either some internal process or some published service. Invoking an external service requires some interactions between the invoking and the invoked processes. These interactions can be as simple as making a request for the service to be executed or as complicated as executing a negotiation protocol in order to commit to a set of quality of service (QoS) parameters. The specification of the correct sequences of messages exchanged among communicating processes that are needed to accomplish the communication process is referred to as a (interaction) protocol or conversation policy [6, 7]. A protocol is organised as a set of roles; a role is the part of the protocol that a participant process must execute. In an interaction protocol, there are often many alternative and valid sequences of messages, each of them triggering one action or another. The execution of one of these possible sequences is called a conversation.

The conversational approach is based on a clear separation between business logic aspects (workflows) and interaction aspects (protocols), allowing easy and flexible working with interaction protocols keeping the same business logic. Flexibility requires a calling process to be able to select one from a predefined set of protocols as a function of the process execution history and the process to interact with. Depending on the process execution and also on the candidate selected for the service delivery, the calling process should be able to start the execution of a protocol (in fact, a role implementation of the protocol) selected from a set of protocols.

In the real world, not all processes follow the conversational approach schema. It is important to understand how this approach can collaborate with the existing different ones. A process implemented using the conversational approach must offer a way to receive messages from the external world and a mechanism in order to manage and execute a set of correlated messages, thus forming the role of a conversation. A set of endpoints, which allow interacting with its services as usually is done in the world of services, must be published. These endpoints must support several interaction styles and protocols, such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) in order to interact with external service providers and to allow incoming requests. As depicted in Figure 1, processes implemented with different technologies, such as BPEL, are able to cooperate with the conversational approach-based processes through the exposed endpoints. However, to use such solution (that is, from the internal point of view), it is required to know how to play in the world of conversations. This means that it is required to explicitly separate the modelling
and/or the implementation of the business logic (by means of workflows) and the interaction protocols. Moreover, the implemented workflow should be compliant with the conversations which are contained in the protocol and will be executed as part of the role played by the workflow. Support to import well-known standards can also be offered. Once imported, these process implementations and descriptions would be analysed and separated into business logic and interaction protocols, thus being integrated successfully in the internals of the platform. As depicted in Figure 1, processes A and B can interact following a conversational schema. In the represented state, the workflow corresponding to process A interacts with the one belonging to process B by means of a role interaction (roles B and C). Obviously, a message or communication facility must be provided in order to allow following the conversational approach.

2.4. Related work

Several research efforts have recognised the need of a clear separation between the composition logic of a Web process (i.e. its business logic) and its interaction logic (coordination protocols and their possible executions). Usually, interaction specifications are organised as (multiparty) conversations (organisation of message exchanges) in which participants play one or more roles. There is a consensus on the use of the BPEL specification for the modelling of the composition logic. However, very different proposals have been presented to specify and implement the coordination logic in the world of Web processes.

Traditional formalisms used for concurrent systems (e.g. state machines [17, 18] or coloured Petri nets (CPN) [19, 20]) and for XML-based languages (e.g. Web Services Flow Language, Choreography Description Language [21] and BPEL) coexist for the description of protocols. In general, these languages have a declarative nature and a lack of interpreters, except the case of BPEL and CPN-based approaches. This means that they cannot be directly executed by a conversation controller. Moreover, some executable specifications such as CPN-based representations are translated...
into Petri Net Markup Language (PNML; an XML-based language for Petri net description) and, therefore, cannot be considered as directly executable specifications.

A participant may play one or more roles in a given conversation. The most restrictive proposals only allow the server side to impose constraints on the way interactions must proceed (like the REST architectural style [22]), such as [17, 23]. One desirable property is the ability of allowing to define interacting protocols involving any number of participants. Several proposals have pointed in this direction, including [18–20, 24, 25] for instance.

There are some proposals that enhance their own conversation language with the addition of constraint descriptions (e.g. temporal constraints, conditions on requester profiles, transactional requirements, etc.) [18, 19]. However, only one of them offers a computer-aided software engineering (CASE)-like tool to help with the implementation of protocols and their conversations [18]. It is interesting to remark that Web-service coordination standards, such as the Web Service Choreography Interface (WSCI), WS-CDL, Conversation Policy XML or OWL-S, for instance, have been ignored by these approaches. Probably, there are two reasons for that. The first is that coordination standards and the BPEL specification (used to describe the composition logic of Web processes) have not been conveniently integrated yet [26]. The second reason can be the lack of CASE tools for the specification and execution of protocols described using coordination standards.

Regarding the architectural interaction patterns used by the considered proposals, in some cases, a centralised conversation controller manages the interactions among all the participant processes [20, 24]. A distributed coordinator is used in [20, 24], where each participant interacts with its own conversation controller. Independently of any architectural considerations, an asynchronous communication model is adopted in most cases.

From a functional point of view, in all the considered cases, conversation controllers are able to maintain the states of conversations, to check for sequencing constraints and also to return error information when a message is not compliant with the protocol in execution. A common restriction is that protocols interpreted by conversation controllers, except the BPEL-based or Business Process Modeling Language-based descriptions [24, 25, 27], cannot be directly executed and must be translated into some executable representation, such as control tables [18], hash tables [23] or dual processes [24], Petri nets are proposed in [20] as an executable specification, but this is done from a conceptual point of view, and no implementation is reported in the paper.

Unfortunately, these conversation controllers are still rather immature. Most of the proposals are prototypes [17, 18, 23] or complex architectural designs that will require further efforts for its implementation [20, 25]. On the other hand, [24] describes the implementation of a conversation controller as a message handler for Axis; that is, it can transparently intercept a copy of messages sent and received by services. Also, the controller contains the JOpera process engine for tracking the conversation in which services are involved. According to this implementation, protocols are not directly executed by the conversation controller, but this component only checks that the exchanged messages meet the protocol. Thus, the BPEL-based process used to model a protocol must be previously translated to an alternative process that is used by the JOpera engine for checking purposes.

2.5. The need for a platform for the development and execution of Web processes

As previously stated, a management infrastructure is required for supporting Web process integration. This infrastructure should provide a run-time environment for the integration of applications, resources and facilities into processes. At service level, the infrastructure should enable the process composition, whereas at integration level, external and heterogeneous entities should be considered. Unfortunately, current approaches are rather static, lacking the required flexibility. As stated in [1], future platforms and infrastructures for process integration should support not only data and process integration but also dynamic aspects. First of all, they should represent dynamically (re)configurable run-time architectures, meaning that they should be able to configure themselves in order to fit to specific application requirements and high-level policies. This would provide QoS benefits, policies for security, transactions, and so on. A special characteristic of this feature is the possibility of changing the protocol used to route interactions, when necessary. The infrastructure should also
be flexible and adaptive enough with respect to the mechanisms used for message selection and routing.

Another key aspect to be managed is related to the dynamic connectivity. This means that processes should be able to dynamically interact without using a separate static interface or proxy. Current applications mainly operate using a static connectivity model (derived from the use of static description languages). Dynamic connectivity would allow Application Programming Interfaces (APIs) to be developed independently of the service implementation protocols.

The last point relates to the ability of the infrastructure to execute processes. The best case would correspond to an infrastructure in which developers can model, develop and execute processes. From the point of view of flexibility, this would ensure that the capabilities associated to the developed processes could be executed, and it would not be necessary to move the process to an execution environment with different requirements.

In recent years, agent-based workflow management systems have been widely used to design business processes. Agents are autonomous and proactive entities, loosely coupled, which can normally adapt to certain parameters. This is a good reason for believing that agent platforms are suitable for flexible and dynamic process integration. Most initiatives have been focused on the use of the Java agent development framework (JADE) platform for the development of integration approaches [28, 29]. JADE supports agent mobility and also ontologies and content languages, which can be extended to fulfill specific application requirements. In this sense, agent platforms could fulfill many of the requirements just established. However, the main drawback of JADE and most of agent platforms is the lack of a description language for processes, because the use of programming languages is not as descriptive as BPEL or other process-oriented initiatives. A possible solution to overcome this drawback is represented by Workflows and Agents Development Environment (WADE), a software system for the definition of processes using the WOLF designer but has been developed as an extension of JADE [30]. However, in this work, the DENEB approach is presented as a fully integrated solution for the description, modelling, implementation and execution of Web processes.

3. THE REFERENCE ARCHITECTURE

The development and execution of Web processes should be provided by means of an infrastructure able to manage all the integration requirements described previously. In this section, the key points of a reference architecture are provided.

Figure 2 gives an overview of the DENEB reference model. This reference model defines and describes the main components and interfaces of an infrastructure based on service-oriented architecture able to integrate Web processes and execute their conversations. Similar architectural models have been proposed in, for instance, [31, 32]. Note that this reference architecture is based on the conversational approach, separating business logic aspects from interaction logic aspects (the workflows and its related protocol roles, respectively).

The core of the system is the workflow and conversation enactment service. This service provides the run-time environment that takes care of the control and execution of both workflows and roles. In order to support scalability issues, the workflow and conversation enactment services may use a workflow engine and a protocol engine, respectively, able to execute the business and interaction logic of Web processes. An in-execution workflow describes the activities that constitute the different steps to be completed to achieve a particular business objective and the constraints for the business activities. An activity can correspond to the invocation of an external service (e.g. a WSDL-described service) or an invocation of some private resource (e.g. an internal database or a legacy system). Any standard language (such as BPEL, for instance) could be used for the definition and description of the considered workflows. Support for importing BPEL and other related description languages is provided by means of translation tools through interface 1.

Similar to workflow engines, the workflow and conversation enactment services may use several protocol engines. These engines are responsible for the execution of the corresponding role for a given workflow. A role can be as simple as invoking an external service in order to consume a given
feature or as complex as deciding at run-time the best service to invoke (e.g. QoS issues) and then to invoke it. It is quite common that a workflow uses a conversation that implements synchronous or asynchronous calls to external services using a request–reply interaction schema, for example.

Both workflow and roles communicate in order to exchange information such as operation signatures and parameters and results from external invocations. This direct communication is performed through interface 2. A message broker is required in order to coordinate the technological aspects of interactions between roles and the external world. Only the protocol engine is allowed to communicate with the message broker by means of the interface 3 (the workflow engine is isolated from this communication because it represents business aspects).

The main component of the message broker is a message repository, where normal messages or control/synchronisation messages can be stored. For scalability and performance purposes, several instances of the message repository can also be executed. The message broker must contain not only a message repository but also all the means needed to send and receive messages via different transport protocols and message formats and to access external services and resources. This can be done by means of mediators inside the message broker. Two types of mediators can be differentiated in DENE: handlers and invokers. A handler is an entity that directly communicates with the message repository, catches special-tagged messages and processes them. Then it can process the contents of such messages and realise a nonstandard action to apply a special format or to decompose it, for example. On the other hand, an invoker catches a message in a similar way, but it is responsible for achieving an invocation to an external service using an interaction protocol such as SOAP, REST or Simple Mail Transfer Protocol (SMTP), for example. DENE provides a caching facility that is accessed through interface 4 for storing invokers. This allows reusing the same invoker, as for example performing SOAP invocations from different processes using the same SOAP invoker. All invokers are compliant with a predefined API and define the target endpoint, the operation name and the parameters required to perform the required service invocation. Also, invokers and handlers can cooperate to achieve a complex task. For example, Figure 3 depicts the trace in which a business rule engine handler uses the SOAP invoker in order to get the rules needed to process data from an external provider.

Therefore, the reference architecture can be extended with new mediators (providing new capabilities and functionalities) without having to modify the core of the architecture. The only requirement for adding a new mediator is that it must be compliant with the predefined API. Examples of

Figure 2. The reference architecture of DENE. BPEL, Business Process Execution Language; WSCI, Web Service Choreography Interface; SOAP, Simple Object Access Protocol; REST, Representational State Transfer; SMTP, Simple Mail Transfer Protocol; API, Application Programming Interface.
mediators are the use of an encapsulated rule engine in order to extend the architecture with decision-making features, or a virtual cart for e-commerce environments. Mediators are also responsible for the external invocation of services using different transport protocols, accessible by means of interface 4.

The message broker mechanism also serves to isolate the model of Web processes from the details of particular protocols, such as SOAP, Remote Procedure Call, Hypertext Transfer Protocol (HTTP) and SMTP, allowing communication with external entities that use a wide variety of communication technologies. Moreover, depending on the destination process, different transport protocols may be used.

Recently, the reference architecture has been extended in order to provide new features related to security and high-flexibility mechanisms. Such mechanisms may allow loading at run-time new mediators and components, giving adaptation characteristics to the proposed model and adding the feature of security mechanisms for the exchange of data and trust in dynamic acquisition scenarios.

4. IMPLEMENTATION OF DENEB

As indicated in the introduction, DENEB is a nets-within-nets-based infrastructure that decouples orchestration and choreography aspects using the conversational approach. The business logic of a process is implemented by means of a workflow, which starts and executes roles (implementing the interaction logic by means of conversations) when some interaction with published services is required. In this section, the implementation of DENEB is depicted.

4.1. The nets-within-nets paradigm

Petri nets [33] have been extensively used for the specification, analysis and implementation of workflows [34] and also communication protocols [35]. The Petri net family of formalisms is of interest because it provides a clear and precise formal semantics, an intuitive graphical notation and many techniques and tools for analysis, simulation and execution. Using the same formalism for both aspects, business logic and interactions, greatly simplifies dealing with integration aspects.
Ordinary Petri nets (either place/transition nets or generalised Petri nets) have a rather static structure that is not flexible enough to deal with the high flexibility required in the world of Web processes. As shown in [36, 37], the nets-within-nets [38] paradigm, belonging to the family of object Petri net formalisms, is a very suitable approach. nets-within-nets are composed of a static part, the environment (also called system net) and a dynamic part, composed of instances of object nets that move inside the system net. These instances can be created or destroyed in a dynamic way. Each object net can have its own internal dynamic behaviour and can also interact with the system net by means of interactions. The system net can also move (transport) object nets on its own.

Java expressions (with some peculiarities) can be introduced in arcs and transitions by means of inscriptions [39]. The possible kinds of tokens that move from a place to another can be Java values or references. By default, an arc will transport a black token, denoted by ]. But if an inscription is added to an arc or a transition (arc inscriptions and transition inscriptions, respectively), that inscription will be evaluated and the result will determine which kind of token is moved. Therefore, in Reference nets, tokens in the system net can be references to object nets, so it is possible that different tokens point to the same object net. Figure 4(a) depicts a system net and an object net class. Firing transition t10 creates two references to a new instance of objectClass1, moving the system to the state in Figure 4(b). In nets-within-nets, three different types of transition firings are possible. The first one corresponds to the initiative of the system net: in the state in Figure 4(b), transition t11 of the system net is enabled and can fire moving the reference from the input place of transition t11 to its output place, leading to the state in Figure 4(c) (the internal state of the object net does not change). This is the reason why these firings are called transports. The second one corresponds to the case in which an object instance executes an object autonomous action: in the state in Figure 4(b), transition t1 of the object net is enabled and can fire independently of the system net, leading to the state in Figure 4(d). The last case corresponds to the synchronised firing of a transition of the system net and a transition of an object net: in the state in Figure 4(b), transitions t12 and t2 can synchronise their firings (this is indicated by the common part in their inscriptions, :i()), whose firing will give the state in Figure 4(e). This way of firing is called an interaction.

Interactions are executed by means of synchronous channels, which impose the synchronised firing of transitions related to the same channel. Synchronous channels can also have parameters as the way of exchange information. In Figure 5, the firing of transition t allows object nets ON1 and ON2 to interchange parameter values 1 and 2.

Figure 4. (a) A reference net-within-net example with the system net and an object net. (b) The previous systems once transition t10 was fired. (c) Evolution from the state in (b) once t11 was fired (transport). (d) Evolution from the state in (b) once t1 is fired (autonomous object event). (e) Evolution from the state in (b) when the synchronised firing of t12 and t2 occurs (interaction).
In our work, nets-within-nets have been used to model dynamic business processes and their interaction protocols (i.e. as a composition and coordination language) and to implement a platform for process definition, integration and execution (as a powerful CASE tool). The interoperability of the platform, called DENEB, is guaranteed because it allows internal business processes to cooperate in a transparent manner with heterogeneous external processes (as is the case of processes implemented using Web service technologies) [12, 13].

The Reference Nets Workshop, Renew, is an academic open-source tool to model and execute Reference nets [39, 40]. Renew is based on Java, and it is completely integrated with the language, allowing to use Reference nets as Java objects and also to use Java code and invocations in the nets.

Renew supports the use of synchronous channels as a communication mechanism and as an abstraction concept to exchange information between nets. Also, these channels are used to synchronise the firing of transitions in a multilevel fashion.

Renew provides users with a GUI and several toolboxes to model nets and also a complete and powerful execution subsystem. The use of the Byte Code Engineering Library, BCEL [41, 42], and the Renew’s reflection API, provided to interact at run-time with the core, help to improve the execution performance of the tool.

4.2. Design considerations

Figure 6 depicts the high-level software architecture of DENEB. As shown, DENEB runs on top of the Renew tool [39], which is executed on the Java Virtual Machine. An instance of the architecture of DENEB integrates the composition and conversation components that contain the workflow and protocol engines, respectively. These components correspond to the workflow and conversation enactment service shown in the reference architecture (Figure 2). Regarding the conversation component, invoking an external service as an activity inside a workflow can be more complex than a simple synchronous or asynchronous call. Moreover, for the same activity, and depending on the past history of the workflow execution, different interaction protocols (conversation policies) may be necessary. These two considerations make a noncoupled execution of workflows and interaction flow controllers of interest, perhaps even necessary. To achieve this, the execution of an activity in a workflow may require a conversation between the service demander (the invoking workflow) and one or more service providers to be started. The logics of the message exchanges should be decided at invocation time, choosing one among all the conversation protocols previously agreed among the peers wanting to participate in the world of Web processes and available through the protocol storage facility (which corresponds to the protocol repository component in our reference architecture). A natural set of interaction protocols could be that defined for the multiagent community (Foundation for Intelligent Physical Agents (FIPA) compliant protocols [43], for instance) but could also be privately agreed protocols and also run-time agreed protocols. Finally, mediators extend the features of the platform, making the bridge between the message broker and the external world.
In accordance with the correspondence between DENEB design and its reference model, the implementation of DENEB is based on the following facts:

1. Both Web service composition and interactions have quite similar aspects, which are strongly related to concurrent elements. For instance, a workflow may be blocked waiting for an invoked service to be terminated, the same as a conversation may be waiting for a message to be received. In both aspects, typical concurrent programming constructs are needed (sequential composition, fork or join constructs for the starting and ending of parallel activities, etc.). Therefore, using the same formalism to deal with both aspects is quite a natural solution. Petri nets have been extensively and intensively used in the domain of workflow management systems, as well as in the domain of modelling, analysing and executing communication protocols [34, 35, 44, 45]. We have adopted this approach to develop the work explained in this article.

2. Interactions among processes have an inherent asynchronous nature, being organised by means of protocols in which the services involved agree to play different roles. Service architects can logically group one or two exchanged messages (like units of communication between services) to form simple interaction patterns (Message Exchange Patterns of WSDL) or, alternatively, to form complex conversations and protocols grouping several messages, which are associated with some well-defined behaviour for the participating services (as is the case of complex agent interactions). This is the approach followed in [6, 8], for instance, by means of conversation policies.

3. The message broker must be able not only to store messages but also to route them or to block calling processes until a specific message has arrived. These characteristics are typical of asynchronous message-passing systems. We have considered the Linda coordination model [46] as the intermediate language for the implementation of conversations among Web processes. This choice was motivated by the fact that its communication primitives are particularly well suited for Web process environments allowing an uncoupled communication and requiring a minimum prior knowledge between the cooperating peers. Besides, a Linda-based coordination
space is used as the repository where conversations and Web processes write/read/wait-for messages implemented by means of Linda tuples in an implementation-independent way. Additionally, the routing information may be easily represented by the matching functions used for the recovery of tuples/documents from the coordination space [47,48].

4.3. Implementation details

Let us now show a concrete implementation of DENEB according to the described design considerations.

4.3.1. The workflow and conversation enactment component. Figure 7 depicts an excerpt of the system net of the Reference-nets-based implementation of DENEB [12,13] that corresponds to the engine fragment of the workflow and conversation enactment service depicted in Figure 2 and where workflows and roles will be executed as object nets. The complete enactment service is much more complex and involves a set of transitions and nets representing subprocesses that manage the working cycle of the system net, the loaded nets and the status of the whole system, translation tools to import workflows and protocols described using different standards, such as BPEL or WSCI, and a set of managers to deal with contexts, communications and storage. However, Figure 7 contains enough information to describe the main characteristics and behaviour of the system net because it is directly related to the life cycle of workflow and role nets.

In nets-within-nets terminology, a workflow, once created and until its termination, will stay in place work-space, whereas conversations (always created by workflows) will stay in place conversation-space. The background image shows the correspondence between the simplified top-level architecture and the proposed implementation. Notice that interactions between the components are managed with the simultaneous firing of synchronisation channels.

Let us now briefly explain the purpose of the main elements in the system net.

- Transitions t01 creates a new workflow, putting it into the work-space. A Workflow Management Component is responsible for deciding to start a new workflow or to create it as the answer to an external request.
- Transitions t02 and t03 correspond to the beginning and termination of a workflow in execution, respectively.
- As previously stated, when some (simple or complex) interactions with other processes are needed, one or more roles of a conversation policy must be started by the workflow. This is the purpose of transitions t10, t11 and t15. In the first case (which involves transitions t10 and t11), a conversation policy was previously started (maybe by the same workflow or by another one), and the workflow just agrees to participate by playing the role demanded. In the second case, it is the workflow that starts the execution of a conversation policy (firing transition t15) and calls a set of services demanding the rest of the roles needed to complete the conversation policy to be played.
- Transitions t14 and t13 correspond to the starting and ending of the execution of roles, respectively.
- In some cases, the execution of a conversation policy requires some interactions between the role and the workflow that started it (some decisions should be taken by the workflow as a reaction to some information received by the role). Transition t12 is used for this communication by means of workflow-role interaction channels (wf-roleInt for short).
- Transitions t20, t21, t22 correspond to the Linda coordination primitives read, take (in) and write (out) and are used to check for, to withdraw and to put tuples in the message repository, respectively.

4.3.2. The message broker. In order to support message exchanges among in-execution processes, DENEB integrates a message broker. Figure 8 shows the main components of the message broker: the message repository and the mediators component (as was previously shown in Figure 2).
Figure 7. DENEK implementation: System Net. API, Application Programming Interface; BPEL, Business Process Execution Language; WSCI, Web Service Choreography Interface.

The message broker acts as a middleware in charge of coordinating the interactions of DENEK processes, implemented as Petri nets corresponding to the roles implementations in execution. According to the system net of DENEK, only these implementations can access the message broker.
Therefore, this allows keeping an explicit separation between the business logic of the process (implemented by means of a workflow) and its interaction logic. Moreover, as it will be depicted later, roles implementations must access external subsystems and expose their capabilities by means of mediators. Mediators are components that act as a bridge between the implemented platform and the external world, allowing not only the direct communication with the external world but also the message manipulation by acting as message wrappers. These mediators can be separated into two different kind of systems: invokers and handlers. Invokers allow accessing external services from a role implementation by means of a concrete technology (such as SOAP or REST), without having to explicitly specifying the implementation details in the role itself. Mediators expose more sophisticated functionalities and can be dynamically bound to the message broker through a dynamic loading interface, as it will be shown in next subsection. This way, protocols specify the interaction logic of the conversational approach, which means that they can be reused by different workflows. Also, they are isolated from concrete technological aspects, which also enhances their flexibility.

Let us briefly focus on the activities that occur in the message broker. The Linda coordination primitives are used by roles for sending/receiving messages to/from another participants. Exchanged messages are temporally stored in the message repository. If a message must be delivered to an external participant, then one or several mediators (invokers) are responsible for sending it using a specific communication technology (such as SOAP, Java Message Service, HTTP and Remote Method Invocation). In a similar way, messages received from external participants are received, processed and inserted into the message repository by a specialised mediator (invokers or handlers).

For supporting all these activities, a message repository based on the Linda coordination model, called *RLinda* [49], has been implemented using the nets-within-nets paradigm and the Renew tool. The details of the working implementation (that is, the executable model/net) are shown on the left side in Figure 8. A distributed implementation based on this, called *DRLinda*, has been also developed [50]. Consequently, messages exchanged among participants of a conversation are represented as normalised tuples. In this approach, a FIPA Agent Communication Language Message Structure [29, 51] has been adopted to represent tuples. This representation includes mainly the type of communicative act, the message receiver, the message content, the content description and the context identifiers used to properly associate workflows and corresponding roles nets (some clarifying examples have been presented in some previous work [12, 13, 52]).

Tuples also encode the information needed for sending the message in accordance to a specific communication technology. The corresponding set of specialised mediators is able to check the message repository for taking tuples of interest to them, to interpret the encoding information and, finally, to send the message. These interchanged messages are usually XML encoded. Currently, several mediators have been implemented for sending messages (SOAP, REST or SMTP, among others), offering a high-compliance level with most of the current transport protocols and interaction styles in the world of Web services. These mediators integrate the transport protocol repository component, shown previously in the reference architecture (Figure 2), thus extending the communication mechanisms of the platform.

On the other hand, processes in execution may be able to offer services to the external world. This is performed by means of a deployment mediator that is able to publish a process service description by means of its associated WSDL, using the Axis technology. These components parse invocations building tuples, which are inserted into the message repository in order to be processed by the corresponding process.

Mediators allow the functionality of the DENEb platform to be extended, adding new features to the defined model. In this respect, scalability and flexibility aspects are strongly related to the use of these components. For example, in an e-commerce environment (as an e-marketplace instance, for example), the main problem of reusing a nonspecialised platform is the lack of interaction standards for the understanding of the used semantics and protocols. The DENEb platform can be extended easily in order to support such protocols and semantics by adding a new mediator to the message broker component, compliant with the defined API. This mediator could bind to an external interface, process external requests (messages) and provide the mechanisms required for the translation
4.3.3. **DENEV dynamic loading component.** From an operational point of view, the dynamic loader component (CL component), shown in Figure 6, is responsible for performing the required steps in order to dynamically acquire a role at run-time, to add it to the conversation engine, and to put it in execution in DENEV. The dynamic loader interacts with other entities (administrators or loaded classes and nets) through an API, whereas the loaded roles are registered in the Renew engine by means of the direct interaction using the Renew API. Figure 9 depicts a low-level view of the software architecture for the dynamic loader component. This component allows loading into DENEV several resources at run-time using a streaming mechanism or by means of an URI, and currently supports loading nets, Java classes and JAR packages. Optionally, Java code (without library dependencies, unless the required libraries are specified) can also be automatically loaded and compiled. Let us now describe that process by means of the analysis of the different ways a role can be loaded into DENEV.

When a new conversation policy is acquired at run-time, the participant providing the implementation of the corresponding role can deliver it in two ways: either by means of the direct sending of its PNML\(^2\) description or by means of an URL from which the role description can be retrieved. Interoperability issues regarding the exchange of both workflows and roles in heterogeneous environments are avoided by using the Petri Net Markup Language. In both cases, once the role has been received through the API, a control message (a message with a control header, a field to identify the mediator - handler - which is going to process it and the required parameters, if any, with the data and operations to execute) is put on the message repository specifying that a new role is available at run-time. Then, the dynamic loader engine (called class loader engine in Figure 9) identifies and receives these control messages using a message broker mediator and executes the required steps in order to load the corresponding role at run-time. As a previous step for the case in which the URL of the role description is provided, the spider wrapper must retrieve the PNML description from the specified address. The process from this point is the same for both cases.

First, the PNML RefNet parser checks the description and then generates a Renew net that is compiled into a Java class (DENEV’s core supports the use of Java-compiled nets rather than PNML-based nets in order to improve performance at the execution stage), suitable for its integration with the DENEV engine. Then, using the Java Reflection API and BCEL, the bytecodes are optimised and finally loaded into the DENEV core using the bootstrap loader, which directly interacts with the Renew engine through its API [41, 42]. We must remark that the current implementation is running on its first release, and we assume a trusted environment which imply loaded roles or workflows have a correct and trusted behaviour. Further developments will add the required security issues.

The dynamic loading of mediators is also allowed in order to support possible new interaction styles or sequences of complex operations for external service invocation, in the case of invokers, and to allow adding or extending new features to the platform, in the case of handlers. Loading a mediator is performed in a similar way to the interaction protocol case but, since a mediator is implemented as a reference net or as a Java class, a basic release of a code checker has been developed in order to support the latter possibility.

4.4. **Modelling and implementing workflows and roles**

Both workflows and roles are executed in DENEV by means of the system net depicted on Figure 7. Let us now briefly introduce some notions about the modelling of these entities. Figure 10 depicts the life cycle of workflows and roles in DENEV, and their corresponding interactions through the system net. Due to the fact that both workflows and roles are also modelled using Reference Nets, these interactions are implemented using synchronisation channels, which are instantiated with the \(:channel\,Name()\) notation. Channels are called uplink or downlink depending on the way the

---


communication is performed. These channels communicate with the corresponding channel in the system net, and allow also achieving a multiparty interaction, as the :wf\(_{-}\)roleInt() channel, for example. Channel invocations are coded in transitions, so firing the corresponding transition enables the uplink/downlink of the channel, depending if it initiates the synchronisation or if responds to a requested one.

A workflow is put in execution in the system net by means of a :new Workflow() channel firing. Once it has been instantiated, it remains idle until its setup has been performed and the :begin() channel is synchronised. An in-execution workflow can perform several tasks until it finishes by means of the firing of the transition containing the :end() channel. These tasks can be the following:
• Initiate a new conversation (instantiating the role it must play) or participate in an existing one in order to execute some interaction logic, such as invoke an external service or receive some data from a provider (which can be either an external or internal process). This is done by means of :initiateConv() and :participateConv() inscriptions. Also, workflows can manage the starting and ending of these roles.

• Interact with a related conversation (concretely, with the corresponding role) in order to exchange data, by means of the :wf-roleInt() inscription (which stands for workflow-role interaction).

• Process some received data or create new information by means of Java code and libraries. Renew fully supports Java integration, so using Java code in the transitions is allowed. A transition could execute, for example, a urllist = XMLmashup.extractURL(data) instance, assuming that the XMLmashup library has been loaded in the system.

DENEB’s workflows are constructed recursively by aggregation of these actions. Notice that parallel flows can be modelled easily using the Petri nets formalism too. Figure 11 depicts an example workflow and a fragment of the corresponding PNML code. This workflow starts its execution, creates a conversation to retrieve a message from an external echo service and then continues its execution. Note that the model in Figure 11 is directly executable in the DENEB platform, so no post-processing or recoding steps are required. Firing of transition t01 depicts also the creation of the parallel execution flow, which is easily done by means of Petri nets. When both branches have finished, they can join in transition t04.

A role is instantiated by a request from a workflow in the system net by the :createConv() channel, and then bound to the corresponding workflow by means of the :initiateConv() or :participateConv() inscriptions. A role starts its execution when the uplink channel :beginRole() is enabled and fired. Until its termination, a role can usually perform the following actions:

• Interact with the bound workflow in order to exchange data, by means of the :wf-roleInt() inscription.

• Process some received data or create new information by means of Java code and libraries. This is similar to the workflow case and allows creating complex conversation policies.

Figure 11. An example of a DENEB’s workflow and a PNML excerpt.
Interact with a mediator by means of the broker’s message repository. This is done using the Linda primitives read, take and write, which can be instantiated by means of the :read(), :take() and :write() channels, respectively.

In this last case, Linda’s read, take and write directives contain a tuple message as a parameter. This tuple starts with a header field which identifies the required mediator. The format of the parameters of the rest of the tuple is specified by the mediator’s API. Invokers allow roles to access and invoke external services, whilst handlers allow roles to perform a complex manipulation of data. A REST invoker, for example, defines an API in which a message to invoke a REST service requires the following fields, forming a tuple: ["REST_mediator", <roleRequester>, <roleReplier>, <URL specifying the operation and the parameters>, <conversation identifier>, <context identifier>]. This tuple must be introduced in the message repository by means of a write operation. Then, the REST invoker will be monitoring for tuples identified by a REST_mediator header and will catch this tuple using the weak matching mechanism [49], process it and perform the REST invocation. Once the invocation has returned a result, the REST invoker will place the result in the message repository with the following syntax: ["REST_mediator", <roleReplier>, <roleRequester>, <conversation identifier>, <context identifier>, <result>]. Since the role which put the request message knows the required fields, it can retrieve this message by means of a read or a take operation. In both cases the role execution will be blocked until the message has been placed in the tuple space. If an error occurs, the <result> field will contain the information about the error. Figure 12 depicts an example of an interaction of a role with the Amazon E-Commerce Service (ECS) external service using the REST invoker.

DENEB’s roles are constructed recursively by aggregation of these actions. Figure 13 depicts an example role to achieve SOAP invocations and a fragment of the corresponding PNML code (the original PNML code has been parsed in order to replace some tags for more expressive ones). As shown, the role starts its execution and receives the endpoint and parameters in transition t01 in order to invoke an external service using SOAP (by means of a SOAP invoker). The results from such invocation (firing transitions t02 and t03, which correspond to the write and take transitions in Figure 12, respectively) are passed to the workflow by means of the :wf-roleInt() downlink channel, firing transition t04. Then, firing transition t05 the workflow can pass a new endpoint and new parameters in order to perform a new invocation or just to finish the execution of the role, depending on the value of the variable continue. Finally, the role finishes when this variable is set to false and the firing of transition t06 is enabled.
Figure 13. An example of a DENEK’s role and its Petri Net Markup Language excerpt. SOAP, Simple Object Access Protocol.

A more detailed use case regarding the class of workflows and roles used in DENEK, which was partially presented in [13], is described in Section 5.

4.5. DENEK in the management of dynamic Web processes

DENEK provides the required level of dynamism described in Section 2.1 for the management of dynamic Web processes. The presented environment allows an in-execution net to dynamically select and load a new net instance to be started based on run-time parameters. This ability can be used to dynamically create the flow of activities within a workflow or to start a new conversation (maybe defined at run-time) within an in-execution workflow. This means that the first three challenges described in Section 2.1 can be achieved.

Besides, in our environment implementation, new workflow and protocol implementation nets can be dynamically loaded and changed by means of the dynamic loading component using the reflection and introspection characteristics of Renew (which uses the Reflection API of the Java language for such purposes). Thus, some of the advanced forms of interaction required by the fourth challenge are also supported. For example, an external partner can send, at run-time, a nonpreviously agreed role, which can be subsequently loaded and started by an in-execution workflow (this aspect will be shown in the second example in the next section).

Finally, the mediator component system of DENEK can provide the dynamic connectivity capabilities required by the last two challenges. Our conversations can select the transport protocol to be used for sending messages based on run-time parameters. Moreover, specialised mediators could be incorporated into the binding component system for protocol translation when necessary; maybe at run-time. On the other hand, the Linda coordination system used as a message repository,
more specifically, its associative mechanism for tuple matching, allows new forms of content-based routing. Some previous work regarding these issues and some real case studies can be found in [13].

5. A CASE STUDY: DYNAMIC RUN-TIME PROTOCOL ACQUISITION

In order to show the capabilities of the DENEB environment when dealing with interaction protocols in a dynamic way, this section presents a case study of a service requiring a role to be sent and executed at run-time.

Let us suppose that an electronic book dealer offers a service to purchase a book over the Internet by means of the following procedure. First, a client provides the ASIN code of the book he wants to buy. Then the dealer looks for the most adequate electronic book store selling the book and sends to the client process all the needed information to execute the purchasing steps. The idea is to offer the service in such a way that the client does not need to know anything about the interactions with the book store because each electronic store could use its own processing steps. In the concrete case of Amazon, for instance, the purchasing cycle requires the customer to create a remote shopping cart, which previously requires an Amazon ECS (Amazon E-Commerce Service Web API) account to be created. On the other hand, the information returned by Amazon contains much more data than necessary to show the interface in Figure 17, which is our objective. Therefore, the information returned by Amazon must be filtered in order to extract the payment URL, which can be achieved using an external URL-filtering service. Analogous processes should be required for any electronic book store. From the customer process point of view, the question is always the same: I need the book with ASIN [anyASIN]. The differences are just related to the interactions with the book stores, that is related to interaction protocols.

Let us now assume that the customer’s application uses the DENEB framework. For the business process of a book store, this issue would be as easy as starting its specific interaction protocol, similarly to the process depicted in Figure 14. Unfortunately, it would be impossible to know all the possible interaction protocols used for the different book stores. The flexibility provided by DENEB features the following solution: the book dealer (whose implementation will not be described here for the sake of simplicity) provides the customer with not only the information about the concrete book store but also the protocol implementation (in fact, the client role) to be executed in order to interact with it. In order to purchase the book, the client process just executes the dynamically received role.

Figure 14 sketches a buyer process that interacts with the book dealer’s service in order to get the required purchasing protocol’s role to interact with an electronic store, the Amazon ECS (Amazon.com ECS) in this case. The purchasing role is dynamically generated by the dealer and sent to the buyer, who loads and executes it. This role, in execution, accesses the Amazon ECS using the DENEB’s REST mediator. Then an electronic remote shopping cart is created using the ECS operation cartCreate for the book whose ASIN code has been provided. As a result, Amazon returns, among other data, the purchasing URL. The purchasing role then invokes a Web service in order to extract the purchasing URL. In the normal case, the book dealer would provide this service because it really knows which information must be filtered in order to shown the desired interface. Once the result is received, the role launches the browser with the filtered URL, as shown in Figure 17. The user can then execute the last step, proceeding to fill in the payment information and the delivery address to finish purchasing the desired book.

From an implementation point of view, the client’s workflow and the role (the implementation of the required interaction protocol played by the client) are implemented using Petri nets, which produces a direct executable implementation using the DENEB framework. Figure 15 depicts the purchasing role generated by the book’s dealer as a response to the buyer’s request, and which implements the DENEB interface. As shown, the role is composed of four main blocks, which implement the interactions with the chosen electronic book store, Amazon ECS, the interactions with the filtering service and the interactions with the buyer’s workflow.

First, a request to the Amazon ECS is performed in order to create a cart with the book identified by its ASIN (cartCreate Request). Then the answer of the ECS service is retrieved
Figure 14. Diagram showing the different interactions that take place in the purchasing process. ECS, Amazon E-Commerce Service; API, Application Programming Interface; REST, Representational State Transfer.

(Amazon Response). Transitions $t_{112}$ and $t_{113}$ implement the interactions with the external services through the DENEB’s message broker. The answer, composed of an operation and the purchasing URL provided by Amazon, is then passed to the (dealer’s) filtering service through transitions $t_{114}$, which finally returns the extracted purchaseURL (transition $t_{115}$). This URL is then passed to the customer’s workflow in order to let the customer to finish the purchasing process (Purchase Cart task, transition $t_{116}$). After that, the execution of the role ends.

Mediators join the external world and the message repository, managing interactions with external services and accepting external requests in the case of the invokers. For example, the inscription of transition $t_{112}$ will write a tuple in the message repository that will be processed by the invoker that knows how to deal with the specified interaction style protocol (REST in this case). The tuple message format accepted by this invoker corresponds with the one used in the FIPA SC00026H protocol [43]. Therefore, a request is going to be performed playing the role of a roleAmazonRequester, waiting for a party playing the role of roleAmazonReplier in order to process the REST statement specified in the message. A protocol/conversation identifier and a context identifier ($idCn$ and $idCx$, respectively) are passed too in order to identify the messages traces. Fields that are not required to be specified for this interaction using the FIPA format are left blank intentionally.

Let us now concentrate on the load and execute purchasing process task from the buyer’s point of view. This task is responsible for the dynamic loading and execution of the received purchasing role, which interacts with the Amazon ECS service and the (dealer’s) filtering service. The buyer knows neither information about the electronic store nor about the filtering service because he or she simply leaves all the complexity in the received role. The left part of Figure 16 shows the part of the client’s workflow starting the execution of the purchasing role just received from the book dealer, whereas the right part sketches the interactions between that part of the workflow and the purchasing role shown in Figure 15. The purchasing role is created by firing transition $t_{205}$. The synchronised firing of transitions $t_{206}$ and $t_{111}$ starts the role execution. The role invokes first the Amazon
Figure 15. Instance of the role generated by the dealer in order to interact with Amazon E-Commerce Service corresponding to the ASIN 1195214119.

Figure 16. Buyer’s workflow corresponding to the load and execute purchasing process task.

d-service (transitions t112 and then t113) and then invokes the filtering service (transitions t114 and then t115). The next step is the synchronised firing of transitions t207 and t116 so that the filtered URL is passed from the role to the workflow, which then invokes the browser with the filtered URL (transitions t208 and then t209). Note that the Misc library has been previously loaded using the dynamic loading subsystem, depicted in Section 4.
It is important to remark the fact that such high flexibility opens a new challenge. It is possible, in some cases necessary, for a role in execution to interact with the workflow that started it. This is so at the start and end instants, as sketched in Figure 15, but also, in some cases, during the execution of the role (this is the aim of transition $t_{12}$ in Figure 7 in Section 4.3.1).

Another application scenario in which the DENEF framework is currently being applied is related to e-commerce schemas and B2B interactions [52]. At run-time, a seller process can load different auction protocol implementations and integrate the product in different e-marketplaces without having to recode or modify the general process workflow, thus providing a flexible and highly adaptive solution for such evolving scenarios.

6. PLATFORM PERFORMANCE ANALYSIS

In order to measure the efficiency of the approach presented in this work, a set of experiments focusing on scalability was performed simulating different configurations for the same process. The flexibility provided by DENEF relies on three different levels (business logic, interaction and message broker levels). At first sight, it seems that an overloading cost has to be paid for this, making more difficult to deal with scalability questions.

In these experiments, a simple workflow acts as a consumer of an external SOAP service that returns a simple processing of service’s input, received as a parameter. After the reception of the result, the workflow evaluates the correctness of the answer by means of a comparison using a static Java method.

Figure 18 depicts the four different scenarios used in the experiments. In the first one (Figure 18(a)), the process is fully coded in Java. This means that neither mediators nor separation between business logic and interaction logic has been established. From the infrastructure requirements and code complexity’s point of view, this scenario may correspond to an optimised one, where time is the most important performance measure and no flexibility or scalability is expected. A change in the business logic, the interaction logic or the used transport protocol would mean recoding all the process internals. The second configuration (Figure 18(b)) corresponds to the implementation of the process using the Renew tool. This would correspond to the direct modelling of the Java-coded process using Reference Nets and its execution by means of the Renew tool. The main advantage of this approach is the visual and more expressive representation of the process, albeit the same problems as for the direct Java-coding approach may be encountered. The third possibility (Figure 18(c)) is to implement the process over the DENEF platform but without using the message broker. This means that the SOAP invocation is hardwired in the role net. The advantage of this approach is that the process will benefit from the separation between business logic and interaction
logic, although a change in the transport protocol to be used requires a change in the implementation of the role. Finally, the last case (Figure 18(d)) corresponds to the implementation of the process using the fully functional DENEH platform. In this case, the conversational approach is used to separate logics into workflow and protocol, and a SOAP mediator allows low-level details related to the transport mechanisms to be isolated. This is the most flexible and scalable approach because every aspect of the process is managed separately and any change in the process can be performed with a minimum impact on the implementation and the required costs (time and resources).

For all the configurations, time has been taken at the start and the end of the workflow execution, and the execution time set as the difference between these points. In the experiments, the following measurements have been taken: the mean time for process execution scaling the number of concurrent processes executing on the system and the speedup of each approach taking as reference the best execution time (which corresponds to the Java Virtual Machine). Processes were run 2000 times for each configuration. Network latency has been discarded because simulations were carried out using virtual machines and all the accessed Web services were installed on a local virtual machine, using the local network interface for communication.

The benchmark was executed running several instances of the Web process in a virtualised dual processor 64 bit machine (2× AMD Opteron Processor at 2.4 GHz with a 64 bits architecture, 2 MB cache, 4,3 BogoMIPS, physical addressing using 40 bits and 8 GB random access memory) running a Debian GNU/Linux operating system and the Java 1.5.12 compiler and virtual machine. We used kernel 2.6.23 with several customisations in order to achieve CPU bound optimisation and preemptive multitasking. Additionally, we applied a low latency patch, and hardware RTC and RAID were used to improve input/output (SATA2) if pagination was needed. To avoid external or internal communication signalling, which could alter the results, we dedicated a single-user session with no external activity and no network signalling or interruptions (in fact, the consumed Web service was accessed locally), extra processes were killed and the JRE garbage collector was flushed in the same instant the benchmark execution started to avoid pending memory cleans.

Figure 19 shows the graphical representation of the mean time for process execution with respect to the number of concurrent processes of the configurations depicted in Figure 18. As expected, the direct Java-coded process showed the best performance evaluation, although the fully DENEH (the implementation that integrates the message broker) results are very close to the fully Java implementation. Both approaches support up to 16 000 processes executing concurrently, whereas the simple-DENEH and Renew versions cannot go further than 13 000 and 12 000, respectively. These two last configurations are so CPU consuming that for values beyond these thresholds, some processes started showing hazard results, so that it makes no sense to show results beyond this number for that cases. Figure 19 demonstrates that the DENEH approach can be almost as good as a pure Java implementation, but allowing much more flexible, scalable and expressive implementations. Notice that the DENEH results can be obtained for as many concurrent executions as using the Java implementation without degrading system performance.

It can be seen in Figure 19 that the simple separation between business logic (workflow) and interaction logic (protocol role) in DENEH does not help improving the results with respect to the simple
The use of levels in DENEB increases processing times because synchronising actions between workflows and protocols take place through the system net.

However, the use of two levels in DENEB without the message broker implies that SOAP invocations must be coded into the role net, which overloads the execution in such a way that global performances are seriously degraded.

Figure 20 compares the throughput obtained by the different configurations with the best one, pure-Java implementation (Figure 18(a)). As shown, performances significantly decrease for the simply DENEB and Renew versions, being more than three to five times worse for high-concurrency scenarios. As shown by means of the trend analysis functions (x being the number of concurrent processes and the coefficient of determination and $R^2$ being the square of the sample correlation coefficient between the outcomes and their predicted values, varying from 0 to 1), both the simply DENEB and Renew versions get a cubic cost. This result is below that of DENEB, whose performance is very close to pure-Java implementation, getting a linear trend line.

The results provided in Figures 19 and 20 demonstrate that the explicit separation between the business logic and interaction logic and the use of decoupled components (a message broker with a set of mediators) to manage the communication mechanisms allow direct executable models to be produced, which cannot only achieve a performance results as good as the processes directly
6.1. Comparing DENEB and JADE

Current use of object Petri nets include the specification, modelling and analysis of complex concurrent and distributed systems. Although the models are directly executable by Petri net interpreters, that is programs that trigger the firing of the net transitions observing the marking evolution rules, the use of object Petri nets as an implementation language is not so usual. Only agent-oriented programming environments provide the abstractions and infrastructure to support the development and the dynamic integration of autonomous processes and allow the configuration of highly dynamic business scenarios. Therefore, following a pragmatic route, the comparison with other implemented agent software platform such as JADE will show the viability of the DENEB as a software development platform.

JADE is a software platform based on software agent abstractions that provides basic middleware-layer functionalities that are independent of the specific application and simplify the realisation of distributed applications [53]. JADE implements this abstraction using Java and provides a simple and friendly API. JADE is quite an efficient environment limited mostly by the standard limitations of Java programming language, which is interpreted and executed in a Virtual Machine [54]. Both DENEB and JADE represent Java-based development platforms compliant with the FIPA standards and can implement and manage protocols and workflows similarly, so a comparison between both approaches was made.

JADE’s agents can be extended by application programmers; besides, a Behaviour class hierarchy is contained in the jade.core.behaviours subpackage. Behaviours implement the tasks, or intentions, of an agent. They are logical activity units that can be composed in various ways to achieve complex execution patterns and can be concurrently executed. Agent operations can be defined writing behaviours and agent execution paths interconnecting them, which represent an easy way for the implementation of processes in JADE using the approach presented in this work.

On the one hand, the business logic (the workflow) and the interaction logic of a Web process can be implemented as agent behaviours. This way, Web processes are implemented from the DENEB approach using the JADE platform. This approach is quite similar to the schema presented in Figure 18(c)). The main problem of this implementation is that as occurred in Figure 18(b)), all the overhead of the implementation of SOAP invocations are carried out by the agent whose behaviour is responsible for implementing the interaction logic.

To promote interoperability between different platforms, JADE implements all the standard Message Transport Protocols defined by FIPA. These protocols can be processed by a Message Transport Service (MTS) that manages message exchanges within and between platforms (JADE and externals) [53]. This JADE approach is similar to the scenario proposed in Figure 18(d)). In order to improve the JADE performance, the MTS used was the same message broker as that used for DENEB, RLinda, which was decoupled and accessed using external calls. Because both JADE and DENEB use FIPA-compliant messages, the integration of the same message broker was an automatic task.

Figure 21 depicts the results obtained from the execution of the scenarios (c) and (d) shown in Figure 18 using JADE versus the scenario (a) and (d) using Java and DENEB, respectively. Results demonstrate that the use of the message broker improves the execution time for processes using the JADE platform. However, the DENEB platform demonstrates a slightly better performance when process concurrency goes beyond 8000 processes. Therefore, the DENEB implementation improves the JADE implementation, also providing a more expressive description approach (Petri nets) and a platform not only for the execution of processes but also for their modelling.

7. CONCLUSIONS AND FUTURE WORK

In this work, the DENEB platform for the development and execution of Web processes has been presented as a very suitable approach for evolving and interorganisational environments, where
most current approaches for composition languages and architectures for dynamic process integration do not provide the flexibility and dynamism required. DENEB provides the implementation of a run-time environment based on the conversational approach and the Renew tool for the modelling, implementation and execution of dynamic Web processes. This environment allows the collaboration of heterogeneous processes by means of translation tools and mediators (both handlers and invokers). New mediation components can be plugged-in at run-time, making the environment much more flexible and allowing to extend the features and functionalities of the platform according to the requirements of the deployment environment.

DENEB also allows the integration of horizontal protocols in order to provide some required behaviour (e.g. a behaviour with transactional properties, as shown in [12]) or to support different process deployments (such as client-server, peer-to-peer and hierarchical).

From the development point of view, DENEB processes are able to flexibly cooperate in order to easily build composite added-value processes. From the execution point of view, DENEB extends the dynamic binding not only to transport protocol implementations but also to enactment aspects. Processes can reconfigure their business logics and their interaction logics at run-time, selecting and interacting with new partners and sharing, when necessary, nonpreviously agreed interaction protocols. Workflows can also be reconfigured at run-time according to business requirements and organisational strategies. Besides, although Reference nets have been used for modelling process workflows and interaction protocols, this decision is compatible with service-based standards (standard-based descriptions can be easily translated into Petri Nets and vice versa).

The characteristic of dynamic loading and acquisition of protocols opens new challenges: a process receiving a new interaction protocol to be executed must be able to check whether this protocol can properly interact with the workflow (business logic) that uses it. This is a problem of compatibility between a given workflow and a given interaction protocol. In the case of DENEB, this question has been addressed using the analysis capabilities of Petri net models in such a way that compatibility problems are stated and verified in terms of the Petri net model corresponding to the synchronised execution of the given workflow and role [55, 56].

In [57], the use of ontologies for the description of the information to be exchanged between a workflow (implementing a business process) and a role (view of an interaction protocol) has been applied. Also, this work shows how the use of Petri net analysis techniques can be applied for proving whether a given workflow and a given role are compatible in the DENEB environment.

As an immediate task, the DENEB platform is being extended in order to support QoS aspects. In this sense, service pools are created internally to register QoS parameters related to the invoked services, such as average, maximum and minimum response time and availability. This way, the service selection stage can be improved using this collected QoS information.

Regarding translation tools, the BPEL2DENEB tool has been developed in order to allow BPEL processes to be translated automatically to DENEB processes, keeping the semantics of the original...
Then the resulting Petri nets implement the conversational approach, and they can be directly executed in the DENEB platform. Further work is being carried out to enhance the BPEL language description in order to take advantage of DENEB dynamic features when applying this translation tool.

Finally, some work is being carried out in the field of Model-driven Engineering (MDE) to integrate DENEB with some SOC-based MDE approaches. The aim of such integration is to provide an automatic way to generate executable code implementing Web processes, starting from the very early stages of the software development lifecycle.

ACKNOWLEDGEMENTS

This work has been supported by the research project TIN2010-17905, granted by the Spanish Ministry of Education and Innovation.

REFERENCES


58. Fabra J, Álvarez P. BPEL2DENEB: translation of BPEL processes to executable high-level Petri nets. 5th International Conference on Internet and Web Applications and Services, 2010; 496–505, DOI: 10.1109/ICIW.2010.81.