Model-Driven Specification of Software Services

Boris Shishkov, Marten van Sinderen, Bedir Tekinerdogan
Faculty of Electrical Engineering, Mathematics and Computer Science; University of Twente
{B.B.Shishkov; M.J.vanSinderen; B.Tekinerdogan}@ewi.utwente.nl

Abstract

Aligning adequately business requirements and software functionality as well as achieving 'loose coupling' for service functionalities, are identified challenges relevant to service-oriented software design. Furthering previous related work, we propose in this paper an application design process that, taking the above challenges into account, addresses systematically and separately business requirements, the identification of (desirable) service functionalities, and their mapping onto technology platforms. With respect to service modeling, a communication pattern has been identified that is relevant and useful. As for the enforcement of social restrictions in the application functionality, semiotic norms are helpfully applied. And finally, 'loose coupling' is achieved in an orchestration-driven way.

1. Introduction

Software architecture design has crucial importance in the software development life cycle. This can be seen not only from the widely recognized claim that a software architecture forms key reusable (platform-independent) artifacts [2] but also from the software architecture's role in bridging between business logic and application logic, such that the final software system will support the desirable features required by the various stakeholders [10]. Hence, for delivering an adequate architecture design, it is important to appropriately close the semantic gap between real-life-level problems, on one hand, and technology-rooted solutions, on the other hand [11]. It is not surprising that a number of software development methods address this challenge; among them are SDBC, Catalysis, and KobrA [8]. Nevertheless, these methods do not support mapping to any specific platform technology, such as web services technology. Web services are (currently) the technology of choice for implementing the Service-Oriented Architecture – SOA [6]; SOA provides a structure for composing software applications based on the use of services as building blocks that can perform distinct functions through well-defined interfaces [1]. Being able to find, select and compose services without prior agreement results in a 'loose coupling', which together with the availability of (web services) standards brings the potential benefits of increased software reuse, easier application integration and higher business agility.

A usual software design starting point is to consider a (real-life) business scenario that is to be reflected in business modeling envisioning not only statics (entities and their relationships) but also dynamics (entities' behaviors). We should then refactor the business model, responding in this way to the user requirements which might include for example the introduction of a new entity. Then, considering again the requirements, a business-software mapping is to be conducted, including not only mapping between business and application entities but also delimitation of the automated support to be provided by the software system. The derived (high-level) application model is to undergo refactoring driven by the SOA technology architecture underlying the software application development. What we also might need to address at this phase are crosscutting concerns – concerns (such as security, for example) which appear at the business modeling level and then crosscut elements at software architecture level. It is considered useful defining such concerns as first-class abstractions at the business modeling level since architects would be able to treat them uniformly at this level and further distribute them to application components (this would facilitate the design process). All this output represents therefore a software specification model that is appropriately elaborated in terms of statics and dynamics. Desirably, this model denotes the software architecture components that can be further implemented using software component technologies, such as .NET or EJB [8].

We use SDBC as a framework through which to conduct our studies because, compared to other design methods, including Catalysis and KobrA, SDBC provides more thorough support to the software

0-7695-3003-6/07 $25.00 © 2007 IEEE
DOI 10.1109/ICEBE.2007.86

development lifecycle. SDBC not only explicitly considers most of the above phases in a component-based way but it has also useful SOA-related complements, particularly in achieving the desired property of 'loose coupling' [11].

What remains unsolved nevertheless is the full integration of this all, in a SOA-driven software design approach, where the business-software mapping is systematically elaborated. This should include a normative enforcement concerning the application model towards consistency with the business context. Furthermore, awareness is necessary that crosscutting concerns (as introduced above), not considered as such in SOA-related application design approaches, should be identified and at the design level, in facilitating the whole design process.

In the current paper, we derive SOA application desired properties, based on a SOA state-of-the-art analysis, and we focus from this perspective on software development lifecycle in general and particularly, on its phases related to the business-software mapping, using SDBC as a framework through which to conduct our studies. We elaborate on our SDBC-SOA result on achieving 'loose coupling', studying further how SOA concepts and challenges can be usefully reflected in software architecture design. This will include a useful consideration of the Theory of Communicative Action [4] for the purpose of specifying services. This will also include the consideration of Norm Analysis - NA [5] as a business-software-consistency enforcement tool and indirect consideration of crosscutting concerns, inspired by related work on aspect-oriented software development.

The contribution of this paper is thus three-fold:

- it analyzes SOA and its actual challenges, and derives, based on that, essential SOA application desirable properties;
- it approaches software architecture design from the perspective the identified SOA-related application desirable properties, namely (i) the enforcement of a proper 'alignment' between business requirements and software functionality and (ii) the achievement of 'loose coupling' concerning the functionalities of software components;
- in realizing this, an architecture design approach is proposed, that usefully addresses the above mentioned challenges, by appropriately combining existing modeling tools, including SDBC, LAP, and NA.

The proposed service-oriented architecture design approach is elaborated and demonstrated by means of an example: the Emergency Health-Care (EHC) case.

The paper is organized as follows: Section 2 not only elaborates our proposed views, by presenting relevant theories and methods but also derives (driven by an analysis) application desirable properties. Based on this, Section 3 outlines our approach. Section 4 introduces the EHC case. Section 5 considers the service-driven business-software alignment, by applying the approach with regard to the (EHC) case. Finally, Section 6 presents related work and also outlines conclusions.

2. Background and analysis

This Section contains: (i) a brief outline of SDBC that will be used as a general modeling framework; (ii) related Norm Analysis background; (iii) relevant analysis of SOA and actual related challenges, leading to the derivation of application desirable properties.

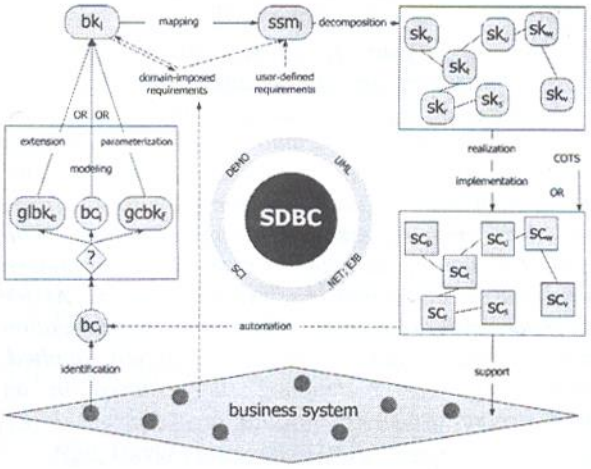

Figure 1: Outline of the SDBC approach [8]

2.1. SDBC

In summarizing the approach 'Software Derived from Business Components' – SDBC [9], we use the following abbreviations as applied in Figure 1: bc – Business Component (a business sub-system that comprises exactly one business process); bk – Business CoMponent (a model of a Business Component, which is elaborated in terms of *statics* and *dynamics*); glbk – general Business CoMponent (which is re-usable by extension); gcbk – generic Business CoMponent (which is re-usable by parameterization); ssm – software specification model; sc – Software Component (an implemented piece of software representing a part of an application); sk – Software CoMponent (a conceptual specification model of a Software Component). For more information on the above concepts, interested readers are referred to [8].

The figure shows that SDBC is about a component-based business-process-modeling-driven specification and realization of software. The starting point is the consideration of a business system that might be identified and elicited using a Scenario (as suggested in the Introduction) but it could also be derived from an abstract business modeling input [9]. Business Components are then identified (being denoted with textual descriptions), by applying the Semantic Analysis Method – SAM leading to the derivation of the so called 'SCI modeling output' [5,8]. They are then reflected in corresponding Business CoMponents, in supplying an adequate modeling foundation for the further software specification activities. Another way of arriving at a Business CoMponent is by applying re-use: either extending a general Business CoMponent or parameterizing a generic Business CoMponent. DEMO and other Language-Action-Perspective-(LAP)-driven modeling tools [4,9] are relevant as far as Business CoMponents' specification is concerned. Each Business CoMponent should be then elaborated with the domain-imposed requirements, for the purpose of adding elicitation on the particular context in which its corresponding Business Component exists within the business system. Then, a mapping towards a software specification model should take place, possibly driven by the DEMO-UML transformation mechanism [8]. The domain-imposed requirements as well as the user-defined requirements are to be considered here, since the derived software model should reflect not only the original business features but also the particular user demands towards the software system. The (UML-based) software specification model would need then a precise elaboration, achieved partially through its decomposition into a number of Software CoMponents reflecting functionality pieces [9,8]. Then these Software CoMponents are to undergo realization and implementation, being reflected (in this way) in Software Components. This final set of components might consist of such components which are implemented (using software component technologies, such as .NET or EJB, for instance) based on corresponding Software CoMponents and such components which are purchased. The (resulting) component-based application would support the target business system, by automating anything that concerns the initially identified Business Component(s).

We can therefore refer partially to SDBC in the context of whose design process we could place our SOA-driven business-software mapping procedure.

2.2. Extending SDBC with Norm Analysis - NA

In modeling business system's dynamics, SDBC uses workflow-like techniques, without explicitly formulating rules and conditions necessary for executing optional and conditional actions. NA is one of the semiotic methods [5] that are usefully complementable SDBC in this direction [8,9]. NA identifies responsibilities and rules that govern (human) behavior in an explicit and articulate manner. It recognizes conditions and constraints of the actions driven by their responsibilities. *Norms* in essence are a set of rules and regulations, an underlying protocol governing the behavior network. These norms are embedded within a social context transcending the boundaries of explicit, implicit, formal and informal states, collaborating to attain certain goals. Norms revolve around entities, which influences the entities to execute concerted actions to achieve a particular goal. In this respect, it can specify to a limited extent how an entity should or should not behave.

Five types of norms can be identified, each of which governs a certain aspect of (human) behavior. *Perceptual norms* deal with how entities receive signals from the environment via their senses. *Cognitive norms* enable one to interpret what is perceived, and to gain an understanding based on existing knowledge. *Evaluative norms* help explain why entities have certain objectives. *Behavioral norms* govern entities' behaviors within regular patterns. *Denotative norms* direct the choices of signs for signification.

In business modeling, most rules and regulations fall into the category of Behavioral norms. These norms prescribe what entities must, may, and must not do, which are equivalent to three deontic operators, namely 'obliged', 'permitted', and 'prohibited'. Hence, the following format is considered suitable for the specification of Behavioral norms:

whenever <condition>
if <state>
then <entity>
is <deontic operator>
to <action>

Adopting this form, a credit card company may state norms governing interest charges, for instance:

whenever an amount of outstanding credit	**whenever** an agreement for cr. card is signed
if more than 25 days after posting	**if** within 14 days after commencing
then the card holder	**then** the card holder
is obliged	**is permitted**
to pay the interest.	**to** cancel the agreement.

It is essential to recognize that norms are not as rigid as logical conditions. If a person does not drink water for certain time duration he can not survive. But an individual who breaks a group's working pattern, does not have to be punished. For 'permitted' actions,

whether the agent will take an action or not is seldom deterministic. This elasticity characterizes the business processes, therefore is of particularly value to understand the organizations.

For more information concerning norms, readers are referred to [5]. In the following Sub-section, we continue with outlining SOA as the context in which we are going to apply our business-software mapping.

2.3. Software components and SOA

The emergence of the *service-oriented computing* is widely considered as a move towards overcoming the business-software gap, envisioning a service (of a component/entity) as defining the goal, capabilities and/or behavior (of the component/entity) as observed by and relevant to the users (of the component/entity) [7].

A *web service* (WS, for short) is hence considered as self-contained, Internet-enabled service component capable of not only performing business activities on its own but also possessing the ability to engage other WS and form higher-order business transactions [12].

Further, we distinguish between composite and constituent WS – a *composite WS* consists of (is provided by an orchestration of) multiple constituent WS, and a *constituent WS* is an 'elementary' WS, i.e. a WS which can be used on its own or in a composite WS [6].

In order to be usable on a broad scale, WS (which are based on specific set of standards) should be somehow reflectable in certain abstractions, as an instrument for their application in any platform through which the Internet user accesses them. Moreover, WS usually should not require design 'from scratch' because this would make them expensive. They should instead be re-usable, using one WS as a basis for developing another, by making use of its core functionality [3].

We consider it innovative that multiple users are able to access WS, personalize them and finally use them. This usage of WS implies advanced infrastructures and application platforms that utilize and coordinate such (globally) re-usable services, which is our first conclusion. Finally, employing such generic WS for work in domain-specific business environments means that the service use has to be driven by appropriate underlying business models.

Prior to their use, WS would have to be discovered (by matching requirements to advertised names) and subjected to negotiation (since the user must of course accept using a particular WS).

All these views have actually contributed to the emergence of the *Service-Oriented Architecture* – SOA which goes beyond the sole consideration of WS [1], being a useful paradigm that can support engineers in their designing, building and using distributed software systems. SOA facilitates establishment of ICT support for business processes, which is readily available, flexible and easily maintainable across multiple organizations and platforms. The concept of service/WS adopted by SOA, has evolved from modular object/component middleware approaches, such as CORBA, DCOM and J2EE [6]. However, WS have become the technology of choice for implementing service-oriented software systems, primarily because they are based on ubiquitous Internet standards, such as HTTP and XML, and because they support 'loose coupling'. Whereas the uptake of WS based SOA is impressive, there are still important fundamental challenges not addressed by this technology.

Firstly, the 'plug and play' interoperability of WS to enable ad hoc cooperation of new partners is limited. For on-demand composition of services in an open service-oriented world, interoperability has to be ensured at different levels (syntactic and semantic) and in different dimensions (information and behavior). Current research in this direction is using, for example, Semantic Web and ontology technologies [12].

Secondly, the property of 'loose coupling' is not appropriate for many applications that involve stateful components. Hence, the benefits of WS and SOA would be limited for the developers of such applications if they themselves have to solve the issues of stateful interaction, notification of state changes, support for sharing and coordination [11]. It should thus be aimed that these concerns are placed at the service infrastructure level or that another solution is enforced. Thus, our second conclusion is that achieving 'loose coupling' should be a goal in developing SOA applications.

Finally, WS alone are not sufficient to guarantee a proper 'alignment' between business requirements and software functionality. What is needed is a structured approach for the development of service-oriented software solutions, in which consistency with business requirements, (de-)composition of application services, and mapping onto (alternative) technology platforms can be systematically and separately addressed [11,1]. And our third conclusion is that a business-software alignment is needed particularly in the SOA context.

This last challenge is essentially addressed in the current paper which proposes a SOA-driven application specification approach (introduced in Section 3) that systematically and separately addresses business requirements, identification of desired service functionalities and their mapping onto technology

platforms, taking into account the 3 identified desirable properties:

- Application architecture must allow usage of SOA infrastructure;
- 'Loose coupling' should be enforced;
- Application architecture must fit within the business context.

3. Architecture design approach

Inspired by *layered distributed system architectures* [6], elaborating on our research vision, stated in the Introduction, and enforcing the above formulated desired properties, we propose a layered design architecture illustrated in Figure 2.

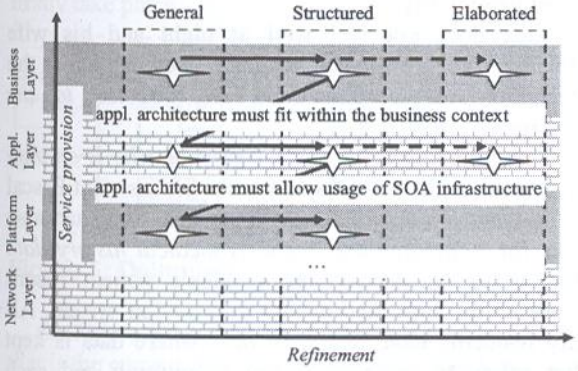

Figure 2: Proposed layered architecture

As seen from the figure, we distinguish 4 service provisioning layers, namely *Network Layer* (concerned with networking protocols), *Platform Layer* (concerned with (IT) infrastructures), *Application Layer* (concerned with the application logic), and *Business Layer* (concerned with the business logic that is not delegated to the application layer).

Further, we consider 3 'degrees' of refinement, namely General ('black-box') view, Structured (high-level 'white-box') view, and Elaborated (detailed 'white-box') view.

Next to that, we enforce two relevant desired properties (requirements) as identified in the previous section:

- The service(s) provided by the Application Layer must fit within the business context;
- The architecture of the Application Layer must be SOA compliant.

A top-down 'Waterfall'-driven [8] architecture design approach has been adopted, helpfully refining some SDBC phases and staying consistent with the layered structure (Figure 2), being driven accordingly by the two above-mentioned requirements.

In general, we firstly consider the overall business system and delimit the sub-system relevant to the software development task; then we refine the sub-system's model, to elaborate on structural aspects, including static and behavioral aspects, with possible further (normative) elaboration. This is followed by a consideration of the application model in terms of application service(s)-delivering entities. The latter represents a requirements- driven model of the (automated) functionality that can be expected from the software system. The application model is to be (analogously) elaborated, and related accordingly to relevant restrictions that concern the platform, the application is to comply with. Figure 3 outlines (using the notations of UML Activity Diagram) the approach process (where we assume the existence of the (SDBC-driven) initially identified and refactored business model), grey arrows indicating the second cycle through the process.

As the figure is suggesting, we consider the input (identified and refactored general business model) by firstly delimiting a sub-model that is relevant to the automated support to be provided by the software system. What we consider are *entities* and related *interactions*. We have to model both entities' statics (2), disclosing the entity's static relationships, and (related to this) the interactions' statics (3), disclosing the interactions' static relationships, assuming that an interaction can only concern the collaboration between two entities. However, considering statics only is not enough for adequately grasping a (business) system that has also dynamics – overall behavior, entities' behaviors, and so on. Therefore, we consider in parallel three interrelated types of system-related behavior, namely the overall system behavior (4), the behaviors associated with each interaction (5), and the internal behaviors of each entity (6). Furthermore, an interaction-associated behavior (5) should be not only consistent with the overall system behavior but it should also comply with socially-driven norms (for example, that a medical specialist cannot abandon a person who needs to be treated) which are enforced through a normative elaboration to interactions (7). Thus, through steps 1, 2, 3, 4, 5, 6, and 7, we arrive at an adequate and sufficiently elaborated input for the business-software mapping, that is to be however complemented with the restriction that the application architecture should be SOA-compliant (Figure 2). The SOA requirements are considered in our mapping between static entity business and application models (8). Then, we follow the same cycle, making sure however that each derived application model is consistent with its corresponding business model.

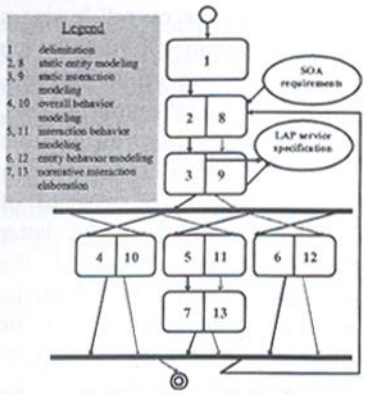

Figure 3: Proposed design process

The entities of the (derived) application model are therefore helpful for service specification because this specification should be driven by the identification of units of (composite) entities' behaviors that can be considered as self-standing application services. However, these services refer to corresponding business services associated with the business entities and if they appear to be 'tightly coupled' at the business modeling level, this should be resolved in the business-software mapping, for example through Orchestration [11].

As for norms, they are not only elaborating interactions, making them more comprehensive [5] but they are also helpful in enforcing social restrictions to which the business entities (hence the application entities as well) should comply while identifying these restrictions is goes often beyond just considering the scenario based on which the source business models have been specified [5,9]. For this reason, we claim that normatively enriching the business-software mapping adds value.

Finally, we use the Language-Action Perspective – LAP, that is driven by Habermas' Theory of Communicative Action [4] as a helpful method for service specification [11].

As suggested by the well-known LAP Pattern (Figure 4), a (real-life-level) communication invariance includes two roles, namely *Initiator* (I) and *Executor* (E). I *requests* (r) something and it is up to E to commit or not, by either *promising* (p) to execute what I requested, or declining (d). Having promised, E has to realize the production act and *state* (s) afterwards that it is done, to which I could react by either *accepting* (a) it – which marks the interaction completed, or declining (d). The coordination acts: r, p, s, a, and d are therefore useful for describing service delivery and we will specify services, by replacing an interaction with a LAP pattern, as suggested by Figure 3 and motivated by previous work [11].

We will elaborate on our approach in Sect. 5, facilitated by the EHC case, to be introduced in Sect. 4.

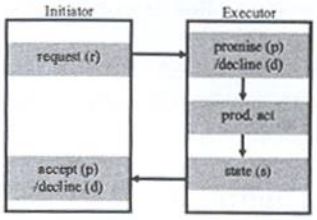

Figure 4: The LAP pattern

4. Emergency Health-Care (EHC) case

The EHC Scenario relates to a health-care case considered in [8]:

Alex does not feel well at night and his wife Heather takes him to the hospital. They explain at the Reception the problem and are *directed to the Emergency Dept.*

There they meet *Dr. med. Chapman who is on duty* during the night. Dr. Chapman asks his assistant Pascal to *conduct the medical anamnesis.* Dr. Chapman also asks for symptoms and *requests medical history data* that is to be derived from Alex's patient record accessed through the hospital information system. All city residents have health records where data is kept that relate to previous health monitoring and each resident's record is accessible from the hospital. The information derived from Alex's record is nevertheless insufficient and (for this reason) Dr. Chapman asks Heather to *elaborate Alex's condition.*

After this first familiarization with the patient's situation, Dr. Chapman sends Alex for *additional standardized examinations*, including ECG. The examination information is then sent to Dr. Chapman who analyzes it and *makes preliminary diagnosis*. This triggers the *patient's treatment plan* that is to be guided by treatment protocols.

According to the plan, Alex is *referred to other medical specialists*. If at the moment all specialists are occupied with other patients, the hospital secretary should *determine the patient's urgency status* that can be either 'green' (the patient can wait) or 'yellow' (the patient must receive help at the first possible occasion) or 'red' (the patient cannot wait). If there are more patients with urgency status 'red' than available specialists, then, applying 'FIFO' approach, first are treated (when specialists are available) those patients who were firstly registered. The other ones are to be approached by nurses.

This is the case with Alex who receives help from one of the nurses, Kelly. She can treat the patient only under the *guidance of a medical assistant.* The

assistant Mathew provides Kelly with instructions. Mathew applies specialized instructions after having 'personalized' them for Alex's situation. Treating a patient by following such instructions aims at just *keeping the condition of the patient stable* until a specialist is available.

Finally, becoming available, Dr. med. Jonkers, a medical specialist, starts treating Alex.

REQUIREMENTS: the treatment provision, diagnosis establishment and standardized examinations to be automated, by allowing limited intervention concerning (only) some patient conditions [8].

5. Applying the design approach

As suggested by Figure 3 – 1, delimitation should firstly take place.

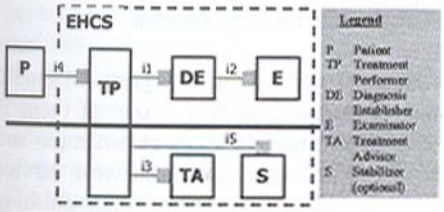

Figure 5: Delimitation model

On Figure 5, we have the *refactored business model* that is the current study's input. On the model, we have entities (named boxes), connections that indicate the need for interactions between entities (i5 is indicated as optional), and inspired by LAP (Figure 4); we indicate the entity that has the role of Executor in an interaction, by putting a grey box on the particular connection, on the side of the entity. The dashed line indicates the *system boundary*. Finally, the (solid) horizontal line separates the entities that are relevant to the automation (as according to the requirements (Section 4) from the rest of the entities (we will further fully abstract from S, since it is optional, as indicated in Figure 5).

We can now consider steps 2 and 3 (Figure 3) and focus particularly on the business-level statics of entities and interactions, depicted in Figure 6, left and right, respectively.

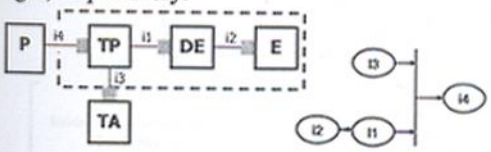

Fig. 6: Str. view on entities and interactions

The right-side model indicates that both Interaction 1 (i1) and i3 have to be completed before i4 can be completed, and also that i2 has to be completed before i1 can be completed. To model corresponding services, as already explained, we replace each interaction with a LAP Pattern, having done conformance checking [11], as partially shown in Figure 7:

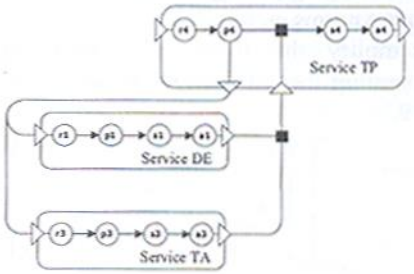

Figure 7: Identification of services

As seen from the figure, if we map one to one business entities to application entities, we arrive at 'tightly coupled' services, which contradicts with one of the identified (in Section 2) desirable properties. We therefore, need to enforce 'loose coupling'.

As concerns the dynamics views, steps 4, 5, and 6 (Figure 3), we start by building (using the notations of UML Activity Diagram) the system's overall behavior – Figure 8:

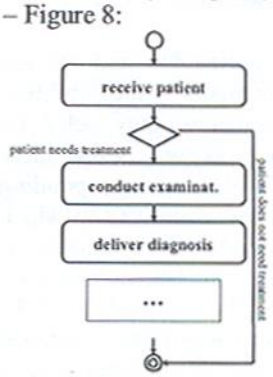

Figure 8: Dynamics view: overall behavior

This overall behavior model must 'govern' the two further identified (according to steps 5 and 6) ones, namely the interaction process specification – for each interaction (that can be expressed using UML Sequence Diagram) and the entity internal behavior specification – for each entity (this can be expressed through UML Activity Diagram). For brevity, we omit those steps.

As stated in Section 3, applying norms for further elaborating interactions can be helpful, especially for the enforcement of relevant social restrictions, not 'rooted' in the case scenario. We consider the theory behind Norm Analysis concerning the identification of norms for that purpose. According to this theory, from the norms identified in the business processes, some

refers to the major authorities and responsibilities. These norms (called '*Framing norms*') govern some trivial, relatively less important norms or those of lower priorities, from the perspective of organizational functionalities. This strongly suggests hierarchies of norms, with Framing norms on top.

We will exemplify the usage of norms, by normatively elaborating the interactions i4 and i1, as shown in Figure 9.

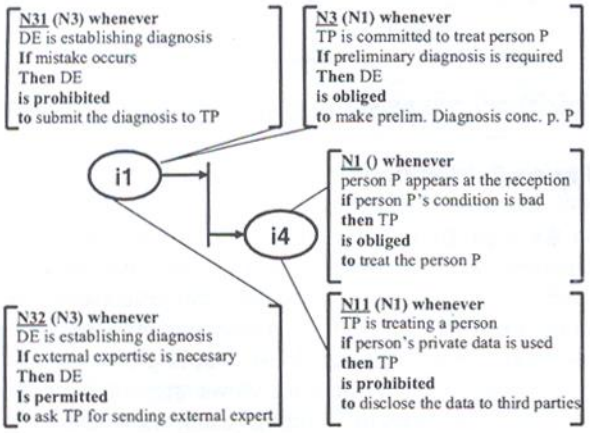

Figure 9: Elaborating interactions with norms

As seen from the figure: norms N1 and N3 are Framing norms – they refer to major responsibilities, being essential for the interactions they refer to, namely i4 and i1. What is seen as well is that these norms naturally 'inherit' their corresponding interactions' dependencies: in the same way in which i4 requires the completion of i1, N3's actuality is dependent on N1 (N1 is context for N3, indicated by putting 'N1' in brackets next to N3, in the label of N3). This is (obviously) because responsibilities associated with different interactions are part of the dependencies of the interactions. Finally, norms N11 from one side and N31 and N32, from another side are 'governed' by N1 and N3, respectively because they appear to elaborate the essential responsibilities as defined in N1 and N3. N11 is not derivable from the case scenario and it should be additionally identified by system architects, in their securing the software's adequacy. We therefore claim that this innovative normative elaboration of interactions usefully supports the enforcement of a business-software consistency, making the business-level norms govern the further application-level norms.

We now closed the first cycle of the design 'loop' (Figure 3) and proceed towards the business-software mapping (step 8), hence essentially aiming at enforcing: (i) requirements-functionality alignment (to be achieved by keeping each of our application models consistent with its corresponding application one); (ii) 'loose coupling'.

Inspired by related work, we resolve the latter through *orchestration* [11] that includes a new entity (Orchestrator) mediating all 'tightly coupled' interactions, invoking corresponding core functionalities as generic services, and handling the remaining application specific 'part' that concerns the interaction. The orchestration-driven entity model, representing actually the application entity model is shown in Figure 10.

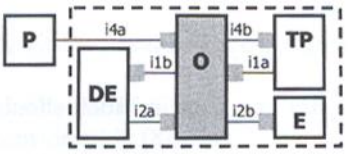

Figure 10: Application entity model

The Orchestrator (colored grey in Figure 10) should thus 'split' each interaction in two – one of them is executed by the application-specific orchestrator and the other one is executed using a generic service delivery. Hence, in this way – by distinguishing between application-specific and generic behaviors, the behavior specification would fulfill the 'loose coupling' requirement.

Nevertheless, to adequately fulfill (at the same time) the alignment requirement would mean specifying the application behaviors in such a way that keeps them consistent with the business context.

We should therefore put the condition from a business-level norm as the condition of the (corresponding) application-level norm that specifies the corresponding application-specific interaction, and we should put no condition in the (corresponding) application-level norm that specifies corresponding generic interaction, as shown in Figure 11 for norm N3.

This illustrates the usefulness of supporting the (application) specification with norms, demonstrating how they can be useful in designing the Orchestrator.

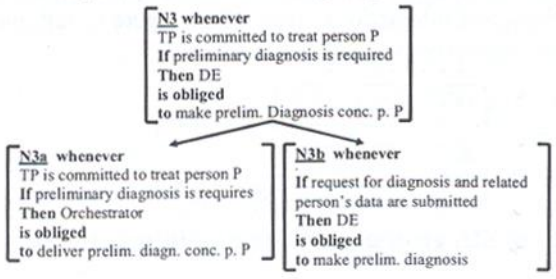

Figure 11: Norm split

The remaining 5 steps (Figure 3) will be omitted for brevity. However, they are analogous to the considered 3-to-7 steps, taking the Application Entity Model as a starting point. However, as mentioned before, it is essential that each application-level derived model is kept consistent with its corresponding business-level model.

6. Related work and conclusions

In this paper we have presented an architecture design process that, in the context of SOA, addresses systematically and separately business requirements, the identification of (desirable) service functionalities, and their mapping onto technology platforms.

Among the work that is focusing on core SOA concerns are [1,3,7], driven mainly by consideration of particular key problems, such as service composition and WS technologies, overlooking however the issue on requirements-functionality alignment. On the other hand, there is reported research concerning the requirements-functionality alignment, mainly related to relevant methods, such as Catalysis, KobrA and SDBC [8] which nevertheless lack the proper (SOA) focus concerning the 'loose coupling' desirable property, for instance

In the proposed architecture design approach that is embracing the SDBC business-software alignment vision (characterized by a clear separation between business modeling and application modeling, and also by restrictions imposed by the business model to the application model): services are modeled with the help of a generic (LAP-driven) communication pattern; social restrictions are enforced to service functionalities through semiotic norms; the desirable 'loose coupling' property is achieved in an orchestration-driven way.

Distinctive features of the proposed way of modeling thus are the systematic approach towards the design of SOA applications (coming through the key problem of business-software alignment) as well as the combination of (i) LAP-driven services specification, (ii) normative functionalities restriction, (iii) orchestration.

To further this research, we plan to address crosscutting concerns (mentioned in the Introduction), considering them as first-class abstractions at the business modeling level where they are treated uniformly, and distributing them to application components. Addressing crosscutting concerns would increase the practical value of the software design, while considering them in the way mentioned above would facilitate their adequate inclusion in the design process.

7. References

[1] Alonso, G., F. Casati, H. Kuno, and V. Machiraju, *Web Services, Concepts, Architectures and Applications*, Springer-Verlag, Berlin-Heidelberg, 2004.

[2] Aksit, M. (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001.

[3] A. Bosworth, "Developing Web Services", In *Proceedings of the 17th Int. Conf. on Data Engineering*, 2001.

[4] Habermas, J., *The Theory of Communicative Action*, Cambridge,1984.

[5] Liu, K., *Semiotics in Information Systems Engineering*, Cambridge University Press, Cambridge, 2000.

[6] Newcomer, E., *Understanding Web Services, XML, WSDL, SOAP and UDDI*, Addison-Wesley, Boston, 2002.

[7] J. Pasley, "How BPEL and SOA are Changing Web Services Development", *Internet Computing*, IEEE, 2005.

[8] Shishkov, B., *Software Specification Based on Re-usable Business Components*, TU Delft – Sieca Repro, Delft, 2005.

[9] B. Shishkov, J.L.G. Dietz and K. Liu, "Bridging the Language-Action Perspective and Organizational Semiotics in SDBC", In *Proceedings of the 8th Int. Conf. on Enterprise Inf. Systems*, INSTICC Press, 2006.

[10] B. Shishkov, D. Quartel, "Refinement of SDBC Business Process Models Using ISDL", In *Proceedings of 8th Int. Conf. on Enterprise Inf. Systems*, INSTICC Press, 2006.

[11] B. Shishkov, D. Quartel and M. van Sinderen, "SOA-Driven Business-Software Alignment", In *Proceedings of Int. Conf. on E-Business Engineering*, IEEE Press, 2006.

[12] World Wide Web Consortium, *Web Services Description Language 1.1*, W3C Note, http://www.w3.org/TR/wsdl,2005.