A Requirements-Based Model for Effort Estimation in Service-Oriented Systems

Bertrand Verlaine, Ivan J. Jureta, and Stéphane Faulkner

PReCISE Research Center, University of Namur
Rempart de la Vierge, 8, BE-5000 Namur, Belgium
{bertrand.verlaine,ivan.jureta,stephane.faulkner}@unamur.be

Abstract. Assessing the development costs of an application remains an arduous task for many project managers. Using new technologies and specific software architectures makes this job even more complicated. In order to help people in charge of this kind of work, we propose a model for estimating the effort required to implement a service-oriented system. Its starting point lies in the requirements and the specifications of the system-to-be. It is able to provide an estimate of the development effort needed. The latter is expressed in a temporal measurement unit, easily convertible into a monetary value. The model proposed takes into account the three types of system complexity, i.e., the structural, the conceptual and the computational complexity.

Keywords: Software Engineering, Service-oriented Computing, Development Costs Estimation.

1 Introduction

"*How much will it cost to develop a given Information System* (IS)?" remains one of the main issues for project managers. The rapid evolution of technologies as well as some new IS development paradigms do not often facilitate this work. In this paper, we focus on Service-oriented Systems (SoS), i.e., ISs based on the Service-oriented Computing (SoC) paradigm. Its main component, the service, is a black box: only messages sent and received are known. Consequently, some software features are no longer programmed while the exchanges of messages must be developed. As recently underlined, assessing the cost of SOA development deserves more attention: "Current approaches to costing [SOA] projects are very limited and have only been applied to specific types of SOA such as Service Development or SOA Application Development" [1]. In response, we propose a requirements-based model for estimating *a priori* the effort needed to develop a SoS. To do so, we adapt and extend an existing model to best suit to the service-oriented paradigm. The results provided consists of an estimation of the development effort required to carry out the SoS implementation. This estimate is based on the three types of software complexity, i.e., the structural, the computational and the conceptual complexity [2, Chap. 5]. The measurement unit of the estimate provided is temporal in order to avoid focussing on a specific social policy applied in a given country.

This paper proceeds by first analysing the related literature based on which we conclude that an adapted model for SoS is needed (§2). Then, the methodology followed

A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp. 82–94, 2014.

is detailed (§3) and the model is developed accordingly (§4). In §5, an example case illustrates the use of the model. Conclusion and future work are presented in §6.

2 Related Work

Existing methods used for estimating a priori the software development costs are either experts-based methods or model-based methods. Model-based methods use algorithms, heuristics computations and/or old projects data. Experts-based methods rely on human expertise and depend on experts' intuition, knowledge and unconscious processes. We decide to focus on a model-based approach in the scope of this work.

Model-based estimation techniques are principally grounded on analogies, empirical studies and/or system-to-be analysis. To be effective, the first kind of techniques needs lots of data collected during previous projects. The objective is to find the similarities with the current project. This technique is close to experts-based methods but it is applied with much more formalism and, often, the use of probabilistic principles. Analogy-based techniques, e.g., [3, 4], face a recurring issue: they need highly skilled workers and they cannot be applied in young organizations because of a lack of historical data. That could be a problem in SoC seeing that it is a young paradigm which evolves quickly.

The second kind of techniques is based on empirical research, whereby situation-based models are proposed. In some sense, they generalize analogy techniques. One well-known initiative is COCOMO [5]. The core idea is that the development costs grow exponentially when the system-to-be grows in size. The problem is that the development of a SoS often combines several development strategies and processes: the underlying services can communicate without any restrictions on their own development technologies. As a result, COCOMO models and similar techniques are often over-calibrated as underlined by Tansey & Stroulia [6]. These authors attempted unsuccessfully to propose an empirical model based on COCOMO to estimate SoS development costs. They were constrained to conclude that SoS development also involves developing and adapting declarative composition specifications, which leads to fundamentally different processes.

The third kind of techniques consists of an analysis of the system-to-be structure in order to measure its characteristics impacting the development costs. One well-known technique is the use of function points based on which the software size is estimated. It is a measurement unit which captures the amount of functionalities of an IS [7, 8]. In this way, Santillo uses the COSMIC measurement method and, actually, he mainly focuses on the determination of the boundary of an SoS [9]. He also identifies one critical issue: from a functional point of view, SoC is different from traditional software architectures. New measurement methods are therefore essential for sizing SoS: we need new rules and new attributes appropriate to the SoC paradigm [9]. Nevertheless, the idea of using the function points deserves further research, which is what we aim for this paper.

2.1 Software Development Costs Estimation in Service-Oriented Computing

In [10], the authors use the Work Breakdown Structure (WBS) for costing SoS. This is a decomposition technique that tries to make a granular list of planned tasks often represented as a tree. It helps to reduce the mean relative error and possible slippages in

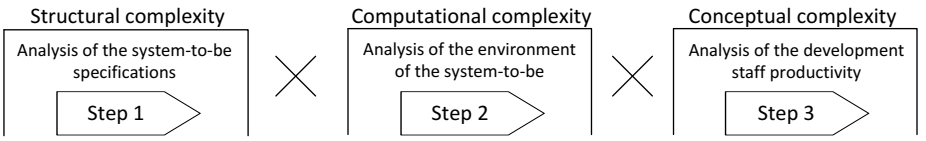

Fig. 1. Illustration of the proposed model structure and its main components

project deliverables. After the SoS decomposition in atomic tasks, the authors propose an algorithm to estimate the development costs of the system-to-be.

A second related work tackles the defect prediction issue in SoS [11]. To do so, the authors use COCOMO to estimate the size of the future SoS. The paper does not solve the main issue explained above, i.e., different strategies and processes can be used during a SoC project, and one variable used in their model –the infrastructure factor– is not clearly defined. It seems they use a COCOMO coefficient estimated based on common software.

In [12], the authors propose an estimation framework for SoS by reducing the total software complexity. They propose to decompose the SoS into smaller parts. Then, each of them is separately estimated. However, it is not clear how all the values resulting from the individual estimation are aggregated to provide a single figure.

3 Methodology Followed

Instead of measuring the SoS development costs –which depend on many unrelated variables such as the wage level– we propose to measure the effort needed, i.e., the number of staff per period needed to carry out the development tasks. To do this, we first evaluate the SoS complexity from which we can deduce the total effort needed. We take into account the three main sources of software complexity [2, Chap. 5]. The *structural complexity* refers to the software design and structure such as the quantity of data stored, the operations achieved, the user interfaces required and so on. As show in Fig. 1, the structural complexity is captured in our model through the analysis of the SoS specifications. This step is the starting point. Specifying the SoS could be achieved thanks to a modelling language, e.g., UML, or with a framework such as IEEE SRS[1]. In the scope of this work, that choice is not important as long as one is able to identify the significant factors –defined below– impacting the structural complexity. The *computational complexity* refers to the way that the computation is being performed. This kind of complexity is captured via an analysis of the system-to-be environment (the second step in Fig. 1). The *conceptual complexity* is related to the difficulty to understand the system-to-be objectives and its requirements. It refers to the cognitive processes and the capabilities of the programmers. In our model, the effort estimation is adjusted to the development staff productivity (the third step in Fig. 1).

[1] The IEEE SRS framework was consulted the last time in February 2013 at `http://standards.ieee.org/findstds/standard/829-2008.html`

Table 1. Characteristics of the SoS Complexity Model along with their acronyms

IC	(Input Complexity): Complexity due to data inputs received by the system-to-be.
OC	(Output Complexity): Complexity due to the data outputs that the system-to-be has to send to its environment.
DSC	(Data Storage Complexity): Complexity due to persistent data that the SoS has to store.
WS	(Weight Source): Weight allocated to an input or output source type (see Table 2).
WT	(Weight Type): Weight allocated to a specific type of input or output (see Table 2).
WST	(Weight Storage Type): Weight allocated to a type of storage destination (see Table 2).
IOC	(Input Output Complexity): Sum of the IC, OC and DSC.
FRC	(Functional Requirements Complexity): Complexity due to the implementation of functional stakeholder's needs.
NFRC	(Non-Functional Requirements Complexity): Complexity due to the implementation of non-functional stakeholder's needs.
FI	(Functions to Implement): Features that have to be entirely implemented in the system-to-be.
FS	(Functions as a Service): Features that will not be coded because services will be used instead.
QA	(Quality Attribute): Primary characteristics coming from the non-functional requirements which state how the functional requirements will be delivered.
QSA	(Quality Sub-Attribute): Secondary characteristics refining each QA.
RC	(Requirements Complexity): Sum of the FRC and NFRC.
PC	(Product Complexity): Complexity of the SoS due to the tasks that it will perform; it sums the IOC and RC.
DCI	(Design Constraints Imposed): Complexity due to constraints and rules to follow during the system-to-be development.
C	(Constraint): Any environment characteristic of the development work or of the system-to-be that limits and/or control what the development team can do.
IFC	(Interface Complexity): Complexity due to the interfaces to implement in the system-to-be.
I	(Interface): Integration with another IS or creation of a user interface.
SDLC	(Software Deployment Location Complexity): Complexity due to the type of users who will access to the SoS as well as their location.
UC	(User Class weight): Weight associated with a user class.
L	(Location): Number of the different access locations for a specific user class.
SFC	(System Feature Complexity): Complexity due to specific features to be added to the system-to-be.
FE	(Feature): Distinguishing characteristic of a software item aiming at enhancing its look or its feel.

4 A Model for Effort Estimation in SoS Development

4.1 Software-Intrinsic Complexity Estimation in Service-Oriented Systems

The SoS Complexity Model. The model proposed should first help to estimate the structural complexity of the SoS (see the first step in Fig. 1). To do so, we adapt and improve an existing model [13]. The latter allows to compute the software-intrinsic complexity before its coding. It analyses stakeholders' requirements expressed in natural language and categorizes them into three groups –critical, optional and normal

Table 2. Sources and type weights for the input, output and data storage complexity

Parameter	Description	Weight
Input/Output Sources	External Input/Output through Devices	1
	Input/Output from files, databases and other pieces of software	2
	Input/Output from outside systems	3
Input/Output Types	Text, string, integer and float	1
	Image, picture, graphic and animation	2
	Audio and video	3
Data Storage Types	Local data storage	1
	Remote data storage	2

requirements– according to eleven axioms. The "normal category" is the default category when the classification algorithm does not succeed to select one of the two other categories. From our point of view, this method faces two problems. First of all, the requirements categorization is complex and imprecise (cf. the default category used when no decision is made). Secondly, the complexity estimation does not take into account some specific features of the SoC such as the use of external services to provide system features. In [13], once a requirement is specified, all of its underlying features increase the software complexity. Despite these two flaws, this model performs well during the tests and comparisons with similar initiatives [13–15]. This is why it is a sound basis on which a specific model for the SoC could be built.

In the rest of this section, we identify the characteristics –defined in Table 1– of SoS and how they increase structural complexity based on the model proposed in [13].

Input Output Complexity. The Input Output Complexity (IOC) gathers the complexity of the input (IC), output (OC) and data storage complexity (DSC) together. Table 2 lists the different weights, picked up from [13], for the types and sources of IC, OC and DSC.

$$IC = \sum_{i=1}^{3} \sum_{j=1}^{3} I_{ij} \times WS_i \times WT_j \tag{1}$$

where I_{ij} is the number of inputs of the source i and being of the type j identified in the system-to-be specifications; WS_i and WT_j are respectively the weight of the input source i and the input weight of the type j as listed in Table 2. In order to compute the OC value, you substitute the variable I_{ij} by O_{ij} in Equation 1.

The use of services to perform some functionalities involves data exchanges between the providers and the consumers of services. The WSDL technology is commonly used for describing service capabilities and communication processes [16]. Two versions of the WSDL protocol currently exit (WSDL 1.1 and 2.0), but their relevant parts for our model are identical. *<operation/>* tags define service functions. Each operation consists of one or several input and output tag(s), i.e., messages exchanges, which must be considered as an input/output source from outside systems. Most of the time, the type to apply is "text" seeing that messages exchanged are XML documents.

Equation 2 states how to compute the DSC.

$$DSC = \sum_{i=1}^{2} S_i \times WST_i \tag{2}$$

where S_i is the number of data storage of type i and WST_i is the weight of the type i.

The IOC value is the addition of the IC, OC and DSC values.

Requirement Complexity. The Functional Requirements Complexity (FRC) value captures the complexity of a given functionality. As some functions can be fulfilled thanks to the use of (composite) services, they should not all be taken into account for the computation of the FRC complexity value. Let F be the set which includes all the SoS' functions. F contains two sub-sets: FI and FS for, respectively, the *Functions to Implement* set and the *Functions as Services* set which will not be fully developed because (composite) services will be used instead. They do not increase the RFC value as stated in Equation 3.

$$FRC = \sum_{i=1}^{n} \sum_{j=1}^{m} FI_i \times SF_{ij} + \sum_{k=1}^{k} FS_k \tag{3}$$

where FI_i is the i^{th} function of FI and SF_{ij} is the j^{th} sub-function obtained after the decomposition of the function FI_i. FS_k is the k^{th} function of FS outsourced as services. In this case, only the main function –i.e., the (composite) service being used– increases the FRC value. Although its computational complexity is hidden, developers have to implement the exchanges of messages between the service used and the SoS.

Non-functional requirements are criteria related to the way the functional requirements will be performed; its complexity value can be computed as stated in Equation 4.

$$NFRC = \sum_{i=1}^{6} \sum_{i=1}^{n} QA_i \times QSA_j \tag{4}$$

where QA_i is the main quality attribute i and QSA_j is the quality sub-attributes j related to QA_i. The quality attributes proposed are those of the ISO/IEC-9126 standard[2] [17].

The Requirement Complexity (RC) is the addition of the FRC and the NFRC.

Product Complexity. The Product Complexity (PC) captures the SoS complexity based on its overall computations. It is obtained by multiplying the IOC and the RC values [13].

Design Constraints Imposed. The Design Constraints Imposed (DCI) refers to the number of constraints to consider during the development of the SoS such as regulations, hardware to reuse, database structures, imposed development languages, etc. Of course, the constraints imposed on the software modules used as services are not taken into account. These services are black boxes for service customers, only the constraints concerning the communication are relevant for the computation of the DCI.

$$DCI = \sum_{i=1}^{n} C_i \tag{5}$$

where C_i is the i^{th} constraint type imposed; its value is to number of constraints i.

[2] The main quality attributes of the ISO/IEC-9126 standard are *Functionality, Reliability, Usability, Efficiency, Maintainability* and *Portability*. See [17] for more information.

Interface Complexity. The Interface Complexity (IFC) is computed based on the number of external integrations and user interfaces needed in the future software.

$$IFC = \sum_{i=1}^{n} I_i \tag{6}$$

where I_i is the i^{th} external interface to develop. I_i has a value ranging from one to x depending of the number of integrations to carry out: a user interface has a value of one while the value of an interface used to integrate multiple systems corresponds to the number of ISs to interconnect. Each service used counts for one interface.
Software Deployment Location Complexity. The Software Deployment Location Complexity (SDLC) is the software complexity due to the types of users accessing the system-to-be combined with the different locations from where they will access it.

$$SDLC = \sum_{i=1}^{4} UC_i \times L_i \tag{7}$$

where UC_i is the user class weight and L_i is the number of locations from which the user belonging to the user class i will access the software. User classes are [13]: casual end users occasionally accessing the SoS (weight of 1), naive or parametric users dealing with the database in preconfigured processes (weight of 2), sophisticated users using applications aligned with complex requirements and/or infrequent business processes (weight of 3), and standalone users working with specific software by using ready-made program packages (weight of 4).
System Feature Complexity. The System Feature Complexity (SFC) refers to specific features to be added to enhance the look and the feel of the system-to-be.

$$SFC = \sum_{i=1}^{n} FE_i \tag{8}$$

where FE_i is the feature i with a weight of 1.
Computation of the SOS RBC *value.* The Service-Oriented System Requirements-based Complexity (SOS RBC) value can be computed as follows:

$$SOS\ RBC = (PC + DCI + IFC + SFC) \times SDLC \tag{9}$$

Note Sharma & Kushwaha also include the "personal complexity attribute" (PCA) in their complexity measurement model [13, 18]. However, the structural complexity measure should only take into account the software structure and not the capabilities of the development staff. The latter should only impact the development effort needed.

Validation of the Complexity Model. Here is a theoretical validation of the model we proposed in this section based on the validation framework for the software complexity measurement process of Kitchenham et al. [19].

Property 1: *For an attribute to be measurable, it must allow different entities [i.e., different specifications of systems-to-be] to be distinguished from one other.*

All attributes used in the Equations 1 to 9 are clearly defined and distinguishable from each other (see Table 1). They cover the specifications of a SoS. Therefore, the SOS RBC model should give different values for different SoS specifications.

Property 2: *A valid measure must obey the representation condition, i.e., it must preserve our intuitive notions about the attribute and the way in which it distinguishes entities.*

This property refers to the psychological complexity, also called conceptual complexity, –i.e., the complexity due to the efforts needed for a given human being to understand and to perform a specific software development task– which cannot interfere with the structural complexity . The latter is the kind of complexity that the SOS RBC model has to capture. All the attributes used are only related to countable and distinguishable intrinsic characteristics of the system-to-be without any relations with the development staff capabilities. We conclude that this property is respected by the SOS RBC model.

Property 3: *Each unit of an attribute contributing to a valid measure is equivalent.*

Each identical attribute in the system-to-be will have the same weight and importance in the estimation regardless its position in the specifications.

These three properties are necessary to validate a complexity measurement process, but not sufficient [19]. Indirect measurements must also respect properties 4 and 5.

Property 4: *For indirect measurements processes, the measure computed must be based on a dimensionally consistent model, with consistent measurement units while avoiding any unexpected discontinuities.*

Our model aims at measuring the complexity of software specifications. All the attributes evaluated to compute the model are intrinsic features of the SoS impacting its complexity.

Property 5: *To validate a measurement instrument, we need to confirm that the measurement instrument accurately measures attribute values in a given unit.*

This property asks for a definition of the measured attributes and their unit. In this paper, we propose a semi-formal definition of the measurement instrument –the best solution is to propose a formal one– based on both mathematical tools and literal definitions.

4.2 Estimation of the Total Intrinsic Size of the System-to-be

In order to estimate the total development effort needed, the model is adjusted with the Technical Complexity Factors (TCF) [7, 8]. They are used to capture the computational complexity (see the second step in Fig. 1). The TCFs are significant characteristics of the software development project which influence the amount of work needed. Each TCF is associated to a Degree of Influence (DI) ranging from 0 (*no influence*) to 5 (*strong influence*). They must be estimated by the development team based on the requirements and on the system-to-be environment[3].

Equation 10 expresses TCF value (TCFV) in a mathematical form [8].

[3] The sixteen TFCs are Complex processing, Data communication, Distributed functions, End user efficiency, Facilitate change, Heavily used configuration, Installation ease, Multiple sites, On-line data entry, On-line update, Operational ease, Performance, Reusability, Security concerns, Third parties IS and Transaction rate. See [7, 8] in order to have more details about the TCF's and the process to follow in order to estimate the appropriate DI for a TCF.

$$TCFV = 0.65 + 0.01 \times \sum_{i=1}^{16} DI_i \tag{10}$$

where DI_i is the degree of influence of the i^{th} TCF.

The adjusted SOS RBC (A-SOS RBC) is the SOS RBC value times the TCFV [8].

$$A\text{-}SOS\ RBC = SOS\ RBC \times TCFV \tag{11}$$

4.3 Estimation of the Total Development Work Needed

The estimation of the SoS Requirements-based Effort (SOS RBE) value is based on the A-SOS RBC. The SOS RBE is significantly related to the productivity of the development staff –it captures the conceptual complexity (see the third step in Fig. 1). The staff productivity is the ratio between the number of code lines written and the time required. It depends on the language used since the latter can be more or less complex, expressive, flexible, etc. The Quantitative Software Management firm (QSM), specialized in quantitative aspects of software, makes available the productivity of development staff for many languages. These values result from empirical research achieved on more than 2190 projects. For all the studied languages, QSM proposes the average value, the median as well as the lowest and the highest value of the number of lines of code needed[4]. For instance, the values of the J2EE language are, respectively, 46, 49, 15 and 67.

Equation 12 states how to compute the SOS RBE.

$$SOS\ RBE = \frac{(A\text{-}SOS\ RBC \times L)}{P} \tag{12}$$

where L is the number of code lines needed per function point as stated by the QSM company. P is the productivity of the development staff express in lines of code per period. The SOS RBE value estimates the number of periods needed for the implementation of the SoS. The unit of the SOS RBE is the same than the period unit of P. The development productivity variables P and L may lack of precision. There are two more sophisticated approaches. The first one lies in calculating the ratio between the number of code lines and development time needed for previous internal projects (see, e.g., [20]). A second approach is to use a parametric estimation model built upon empirical data (see, e.g., [21]). A complete discussion of this topic is out of the scope of this paper.

5 Example Case of the Proposed Effort Estimation Model

A company active in the food industry would like a new IS in order to improve the purchase management. With the new IS, a significant amount of orders should be automatically sent. Currently, workers have to manually carry out all the orders. It exists a legacy IS which manages the outgoing orders. Only its main function will be kept and exposed as a service –it estimates the stock level needed.

[4] All the results of this research are available at `http://www.qsm.com/resources/function-point-languages-table`. Last consultation in July 2013, the 3^{rd}.

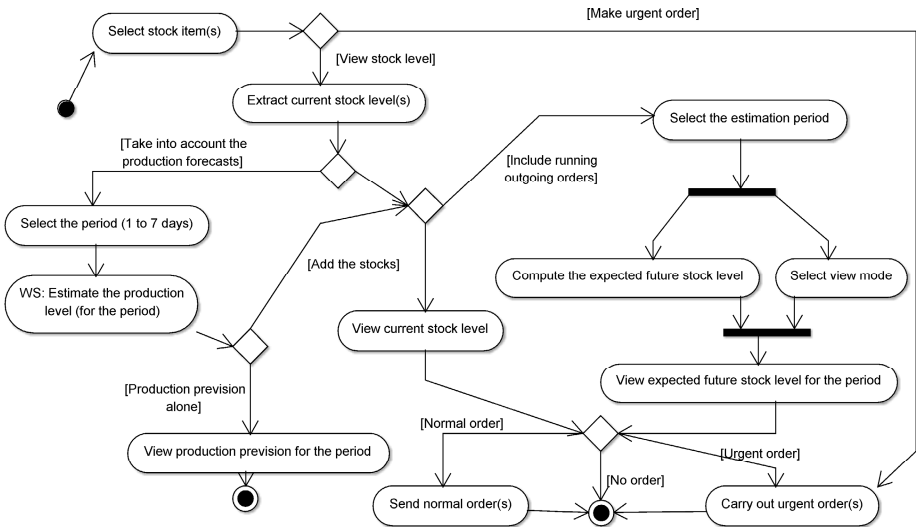

Fig. 2. Activity diagram of the use case *View stock level*

First, the system-to-be specified with UML has to satisfy the following main use cases. *View stock level*: the system-to-be should enable the purchase department to consult the stock levels for all existing products. *Carry out analysis of purchases*: the stock manager would like to have a specific interface to analyse the purchases made (mainly with descriptive statistics and underlying graphic illustrations). *Manage order error*: the purchase manager is in charge of the errors management detected when outgoing orders are delivered and encoded by a warehouse worker. *Send automatic order*: one of the main requirements of the company is to enable automatic sending of orders when a given threshold is reached. The use cases were refined with other UML diagrams. As an example, Fig. 2 represents the Activity diagram refining the use case: *View stock level*.

The IOC identified in the studied Activity diagram is 13: the IC is 5, the OC is 13 and the DSC is 0. E.g., for the activity "*Select stock item(s)*", the OC is 2×1 because of the request in the database (source weight is 2) allowing to display all the possible stock item(s) stored as string (type weigh is 1). The IC is 1×1 because of the selection made by the user through a device, e.g., the mouse or the keyboard.

Concerning the FRC, the use cases compose the functions set; their sub-functions are the steps of their respective Activity diagrams. The FRC value for the studied Activity diagram is 10 $(1 \times 9 + 1)$; 1 because we study here the sub-functions of only one main function, i.e., one use case, 9 because there are nine sub-functions –*send normal order(s)* and *carry out urgent order(s)* are extends use cases and thus refined in other Activity diagrams; the activity *Estimate the production level (for the period)* will be achieved through the use of a Web Service (WS) $(+1)$.

Based on the stakeholders' non-functional requirements, the NFRC value is 10. The total RC value identified in this Activity Diagram is 20 $(10 + 10)$.

The stakeholders explain they want to use the J2EE development platform (one constraint) and the WS technologies –WSDL, SOAP and HTTP (three constraints)– in order

to reuse the legacy application. Last but not least, the SoS will be hosted on the existing application server (one constraint). The total DCI value is $5\ (1 + 3 + 1)$.

The IFC value identified in the studied Activity Diagram is 16. There is one interface with the Warehouse Management IS, one with the 13 provider ISs, one user interface for the workers at the purchase department and one interface for the WS used.

In this example, two user classes were identified: the workers at the Purchase Department and their manager. Both of these two classes are parametric users (weight of 2). They should access the system-to-be from their company offices. The SDLC value is $2\ (2 \times 1)$.

No additional system features were required for this Activity. The SFC value is thus 0.

Once this work done for all the SoS specifications, the SOS RBC value can be computed. The result of this analysis based on Equations 1 to 9 is[5]: SOS RBC $= 5170$.

The SOS RBC value is then adjusted with the TCF's applicable to this system-to-be such as, e.g., *Distributed functions*, *Facilitate change* and *Third parties* IS, with a DI value of, respectively, 2, 1 and 5 evaluated as described in [7, 8]. The TCFV is: $0.65 + 0.01 \times 27 = 0.96$. The A-SOS RBC value is: $5170 \times 0.92 = 4756.4$.

The last step is the computation of the total work needed for the implementation of the system-to-be. The reference language used is J2EE: $L = 46$ (cf. §4.3). The productivity of the staff development has been estimated to 37 lines per hour thanks to an analysis of previous projects. So, the total development effort needed is: SOS RBE $= \frac{4756.4 \times 46}{37} \cong 5790$ hours. Once the average cost per hour known, the financial forecasting of the total development costs of the system-to-be can be drawn up.

6 Conclusions and Future Work

The model proposed, based on the specifications of a SoS, enables to compute the estimated development effort needed for its development. Eliciting, modelling and specifying correctly the requirements remain a significant success factor in the use of our model.

As underlined in §3, the three sources of software complexity –i.e., the structural, the conceptual and the computational complexity– are covered by the estimation model proposed. The analysis of the system-to-be specifications identifies the different software attributes of the structural complexity and put values behind each one (cf. Equations 1 to 9 from which the SOS RBC value can be computed). The TCFs used to adjust the SOS RBC value (cf. Equation 11) aim at adding the computational complexity to the1 model proposed. Indeed, they refer to the way that the stakeholders' requirements will be processed in the system-to-be according to its environment. Lastly, the third step in the model use takes into account the conceptual complexity. This is achieved thanks to Equation 12 in which the productivity of the development staff is added comparatively to the development language chosen for the project.

However, we put aside some difficulties. First, the system-to-be can be coded with more than one language while allowing the use of other programming languages for

[5] The detailed calculation is: $((IC + OC + DSC) \times (FRC + NFRC) + DCI + IFC + SFC) \times SDLC = ((21 + 33 + 5) \times (33 + 10) + 8 + 39 + 1) \times 2 = 5170$.

implementing the services used. Secondly, the productivity of the development staff deserves more attention. Although this problem is out of the scope of this work, one significant question remains unsolved: Is the productivity of development staff the same for SoC projects than for projects in line with other computing paradigms? To the best of our knowledge, there is no clear answer to this question.

References

1. O'Brien, L.: Keynote Talk: Scope, cost and effort estimation for SOA projects. In: Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference Workshop (EDOCW), p. 254. IEEE Computer Society (2009)
2. Laird, L.M., Brennan, M.C.: Software Measurement and Estimation: A Practical Approach. Quantitative Software Engineering Series. Wiley - IEEE Computer Society (2007)
3. Bielak, J.: Improving Size Estimates Using Historical Data. IEEE Software 17(6), 27–35 (2000)
4. Pendharkar, P.C.: Probabilistic estimation of software size and effort. Expert Systems with Applications 37(6), 4435–4440 (2010)
5. Boehm, B.: Software Engineering Economics. Prentice-Hall (1981)
6. Tansey, B., Stroulia, E.: Valuating Software Service Development: Integrating COCOMO II and Real Options Theory. In: Proceedings of the First International Workshop on Economics of Software and Computation, pp. 8–10. IEEE Computer Society (2007)
7. Albrecht, A.J.: Function points as a measure of productivity. In: GUIDE 53 Meeting (1981)
8. Symons, C.R.: Function Point Analysis: Difficulties and Improvements. IEEE Transactions on Software Engineering 14, 2–11 (1988)
9. Santillo, L.: Seizing and sizing SOA applications with COSMIC Function Points. In: Proceedings of the 4th Software Measurement European Forum (SMEF 2007), pp. 155–166 (2007)
10. Oladimeji, Y.L., Folorunso, O., Taofeek, A.A., Adejumobi, A.I.: A Framework for Costing Service-Oriented Architecture (SOA) Projects Using Work Breakdown Structure (WBS) Approach. Global Journal of Computer Science and Technology 11, 35–47 (2011)
11. Liu, J., Xu, Z., Qiao, J., Lin, S.: A defect prediction model for software based on service oriented architecture using EXPERT COCOMO. In: Proceedings of the 21st Annual International Conference on Chinese Control and Decision Conference (CCDC 2009), pp. 2639–2642. IEEE Computer Society (2009)
12. Li, Z., Keung, J.: Software Cost Estimation Framework for Service-Oriented Architecture Systems Using Divide-and-Conquer Approach. In: The Fifth IEEE International Symposium on Service-Oriented System Engineering (SOSE 2010), pp. 47–54. IEEE Computer Society (2010)
13. Sharma, A., Kushwaha, D.S.: Natural language based component extraction from requirement engineering document and its complexity analysis. ACM SIGSOFT Software Engineering Notes 36(1), 1–14 (2011)
14. Sharma, A., Kushwaha, D.S.: Complexity measure based on requirement engineering document and its validation. In: International Conference on Computer and Communication Technology (ICCCT 2010), pp. 608–615. IEEE Computer Society (2010)
15. Sharma, A., Kushwaha, D.S.: A complexity measure based on requirement engineering document. Journal of Computer Science and Engineering 1(1), 112–117 (2010)
16. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented Computing. Communications of the ACM 46(10), 24–28 (2003)

17. ISO/IEC: 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, The International Organization for Standardization (2010)
18. Sharma, A., Kushwaha, D.S.: An Improved SRS Document Based Software Complexity Estimation and Its Robustness Analysis. In: Computer Networks and Information Technologies. CCIS, vol. 142, pp. 111–117. Springer, Heidelberg (2011)
19. Kitchenham, B., Pfleeger, S.L., Fenton, N.E.: Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering 21(12), 929–943 (1995)
20. Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B.: A SLOC counting standard. In: The 22nd International Annual Forum on COCOMO and Systems/Software Cost Modeling (2007)
21. Cataldo, M., Herbsleb, J.D., Carley, K.M.: Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 2–11. ACM Press (2008)