2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement

Towards Component-Aware Function Point Measurement

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
email: luigi.lavazza@uninsubria.it

Valentina Lenarduzzi and Davide Taibi
Faculty of Computer Science
Free University of Bolzano/Bozen
Bolzano/Bozen, Italy
email: {valentina.lenarduzzi, davide.taibi}@unibz.it

***Abstract*—*Background*. Function Point Analysis is the most used technique for sizing software functional specifications. Function Point measures are widely used to estimate the effort needed to develop software, hence the cost of software. However, Function Point Analysis adopts the point of view of the end user, and –consistently– considers a software application as a whole. This approach does not allow for assessing the role of reusable components in software development. In fact, reusing available components decreases the cost of software development, but standard Function Point measures are not able to account for the savings deriving from component reuse.**
***Objective*. We aim at modifying the definition of Function Point Analysis so that the role of components can be taken into account. More specifically, we redefine the measurement so that when no components are used the resulting measure is the same yielded by the standard measurement process, but in presence of components, our modified measure is less than the standard measure (the bigger the role of components, the smaller the measure).**
***Method*. Components partly support the realization of elementary processes. Therefore, we split elementary processes into sub-processes, such that each sub-process is either totally supported by a component or it is not supported at all by any component; the size of the elementary process is defined to be inversely proportional to the size of sub-processes supported by components.**
***Results*. The proposed approach was applied to a Web application, which was developed in two versions: one from scratch and one using available components. As expected, the 'component-aware' measures obtained are smaller than the standard measures. We also compared the reduction in size with the reduction in development effort.**
***Conclusions*. The proposed method proved effective in taking into account the usage of components in the development of the considered application. However, the observed decrease in size is smaller than the decrease of development effort. The latter result suggests that this initial proposal needs further experimentation to support accurate effort estimation.**

I. Introduction

Function Points (FP) [1] are widely used to measure the functional size of software. Functional Size Measurement (FSM) is especially useful in the early stages of development, when an estimate of the development effort is needed, and the only description of the software to be developed is given by the specifications of the Functional User Requirements (FUR). Function Point Analysis (FPA) applies to FUR and delivers a measure that does not depend on technical and implementation-oriented considerations.

A problem with the current definition of FPA (i.e., the definition maintained by IFPUG [2] and standardized by ISO [3]) is that FPA does not take into account how the usage of Commercial Off The Shelf (COTS) components [4] may affect software development. In fact, having been conceived to measure just FUR, FPA applies to the end user view of software. Being the usage of COTS components an implementation detail, it is clearly not perceivable by users, hence, it is ignored by FPA.

However, the usage of COTS components can affect the development effort to a large extent. In fact, not accounting for the usage of COTS components may lead to misleading estimates concerning the amount of effort required, since a single COTS component may save a great deal of development effort.

In this paper we propose some initial considerations concerning the problem of adapting FP measurement to cases where COTS components are used. The final goal is to achieve measures that support reliable effort estimates also when COTS components are used.

In practice, we address two research objectives:

1) Define a functional size measure that is both compatible with IFPUG FP and is able to account for the usage of COTS components.
2) Make the definition of 'component-aware' functional size measure suitable for estimating the savings in development effort deriving from component reuse.

The paper is organized as follows. In Section II the basics of FP measurement are recalled. Related work and the state of the art concerning functional size measurement in presence of components are described in Section III. Some considerations concerning the nature of the measure and its support to effort estimation are given in Section IV. The core of our proposal, i.e., how to measure functional size when components are used, is described in Section V. The proposed approach is applied in a case study, as described in Section VI. The obtained results are discussed in Section VII. Finally, Section VIII draws some conclusions and outlines future work.

978-1-5090-4147-3/16 $31.00 © 2016 IEEE
DOI 10.1109/IWSM-Mensura.2016.13

II. A Brief Introduction to Function Points

The Function Point method was originally introduced by Albrecht to measure the size of data processing systems from the end user's point of view, with the goal of estimating development effort [1].

The initial interest sparked by FPA, along with the recognition of the need for improvement in its counting practices led to founding the International Function Points User Group (http://www.ifpug.org/), which provides guidelines for carrying out FPA [2], makes FPA counting rules evolve along with the evolution of software technologies, and oversees FPA's standardization.

IFPUG FPA is now an ISO standard [3] in its "unadjusted" version. So, throughout the paper, unless otherwise explicitly stated, we refer exclusively to Unadjusted Function Points.

The basic idea of FPA is that the "amount of functionality" released to the user can be evaluated by taking into account the data used by the application to provide the required functions, and the transactions (i.e., operations that involve data crossing the boundaries of the application) through which the functionality is delivered to the user. Both data and transactions are evaluated at the conceptual level, i.e., they represent data and operations that are relevant to the user. Therefore, Function Points (FP) are counted on the basis of the user requirements specifications. The boundary indicates the border between the application being measured and the external applications and user domain.

In Function Point Analysis, functional user requirements are modelled as a set of basic functional components (BFC), which are considered the elementary unit of functional user requirements. Each of the identified BFC is then measured; finally, the size of the whole application is obtained as the sum of the sizes of BFC.

FPA BFC are data functions, which are classified into internal logical files (ILF) and external interface files (EIF), and transactional functions, which are classified into external inputs (EI), external outputs (EO), and external inquiries (EQ), according to the main intent of the process. Each function, whether a data or transactional one, contributes a number of FP that depends on its "complexity," which is evaluated based on data and transaction details. Each function is weighted on the basis of its complexity according to given tables. Finally, the number of so-called Unadjusted Function Points (UFP) is obtained by summing the contribution of the function types. Details about FP measurement can be found in the manual [3].

III. Related Work

Software components help developers increase the productivity and ease the maintenance process. A specific characteristic of components is that they are conceived to be reusable, i.e., applicable in several different applications and contexts, typically not known when the component was developed.

With reference to their reusability, components can be considered as 'functional commonalities', as mentioned in the COSMIC Method Update Bulletin [5]: "*When a statement of Functional User Requirements (FUR) is implemented in software, any 'functional commonality' may or may not be developed as reusable software. The extent of actual or potential software reuse arising from functional commonality may therefore need to be taken into account when using functional size measurements for performance measurement or project effort estimating purposes.*"

However, in the IFPUG Guide to Software measurement, it is stated that "*COTS packages and SaaS packages are outside the boundaries of normal function point sizes of commercial applications.*" Therefore, in case of existing components, there are no function points to be counted [4].

In 2006, Maschino and Makar-Limanov [6] discussed about this rule, highlighting the need for a new approach to measure the impact of COTS components in an application, so as to understand the changes in terms of FP. They suggested measuring a set of properties that affect the economics of software development:

- Component features and support: the size of the component, the features of the component that are used "as is" and those that need to be modified.
- Process performance: changes in the productivity baseline.
- Track improvements: how much of the original system was replaced and how much of the original system was retained.
- Total cost of ownership.

They also highlight that the usage of COTS components involves a set of non-functional factors such as the need for configuring components and the integration of components into the software system.

In 2009, Brown described the "Zero function points" problem [7]. He proposed that in case of development (or maintenance) activities involving software that reuses components, functional size measurement can be applied, but requirements and their size should be allocated to "appropriate acquisition method categories" (including components and their tailoring, adaptation, etc.) so that estimates can account for the contributions by components.

In 2013, Trudel and Abran [8] conducted a case study aimed at measuring the impact of reusable software components in terms of COSMIC Function Points (CFP) [9]. They analyzed six existing projects developed for the insurance domain, during their maintenance phase. They isolated the Business Rules as isolated functional processes belonging to the service layer of the architecture and they analyzed the influence of reusable components to implement the business rules. Results show that most of the analyzed components reported a very low functional size (i.e. only 2 CFP) since they are related to receiving data from one layer and passing them on –in a validated format– to another layer. However, they considered 2 extra CFP every time the component is reused in the application, for a minimum of 4 CFP per component. This means that for every reusable component they propose an overhead of 4 CFP on the total project size. The measurement of the six applications reported that a significant proportion of the reusable software components (70.6%) have been used only once.

Lavazza [10] also raised the issue of measuring the functional contribution of COTS components, as components can affect the value of the software product and the cost of maintenance, as well as increase reusability. This paper is a follow up of the considerations reported in [10].

At the best of our knowledge, no other works investigate the contribution of COTS in terms of functional size measurement.

IV. Considerations on Component-aware Functional Size Measurement

In Section V, we present a method to measure the "component-aware" functional size of a software application. Before providing the details of the measure's definition, it is worth explaining a few ideas that underlie our proposal.

A. Independence from technology

Function Points provide a measure of size that is independent from technology, that is, it does not depend on how the software is built; therefore measuring the 'component-aware' fraction of size could be considered not coherent with the principles of FPA. However, there are a few considerations in favour of not strictly complying with this rule.

A first consideration is that when an application is built using components, we are naturally interested in knowing the size of the software to be developed, as opposed to the size of software to be reused. In fact, the development effort is expected to depend mainly on the portion of the functional specifications not covered by components, Similarly, the notion of defect density is typically referred to the portion of software specifically developed for implementing the application, i.e, excluding components.

A second consideration is that when we consider the standard IFPUG sizing of enhancement projects, IFPUG's approach is similar to ours. In fact, when you need to develop an application App' as an enhancement of an existing application App, according to IFPUG you do not ignore the fact that you build on App, and you do not consider building on App as a mere "implementation detail." For enhancement projects, IFPUG defines rules that measure the *difference* in size between the new App' and the existing App. We are proposing a similar practice: the component-aware size is the difference between the size of the whole application and the size of reused components.

In this case, strict conformance to total independence from implementation considerations appears neither useful nor advisable.

B. Effort estimation

Suppose that we need to estimate the effort required to develop App, by reusing some components.

In a traditional scenario, we use exclusively the standard IFPUG method: we measure App and we find out that its size is S FP. Then, suppose that we use COCOMO II [11] to estimate effort: according to COCOMO II, we have to compute a parameter named AAF (amount of modification), which accounts for the fraction of the software that needs to be implemented. The purpose of AAF is to assure that components reused without modification do not contribute to the development effort. Now, to compute AAF, we have to evaluate –at a quite fine level of granularity– the contribution of components.

We propose –just like COCOMO II– to evaluate the contribution of components, but in a different manner: instead of deriving a parameter like AAF, which is usable only in the context of COCOMO II formulae, we measure the component-aware size (Sca), which in principle can be used for multiple purposes, including effort estimation. If suitable historical data are available, we can define an effort estimation model Effort=f(S, Sca, P), where P is a set of parameters that take into account process and product characteristics that affect the development effort. In this way, we get a model that bases the estimate on the total size S of the application, on the size Sca to be built (i.e., not reused) and on a set of parameter that qualify the product and the development and reuse processes. Such model would be able –for instance– to estimate that the effort require when Sca = 0.5 S is very different (as noted in [11]) form the effort required when Sca = 0.9 S.

Deriving a proper effort estimation model is out of the scope of this paper. Nevertheless, it should be clear that we are posing the basis to build fairly sophisticated reuse-aware effort models. In any case, it should be noted that we do not imply that Effort = f(Sca, P), i.e., excluding S as an argument, or that the Effort etimation function f could be a linear function of Sca. Exploring effort models involving component-aware size is definitely a topic for future work.

V. Measuring Components

In this section we propose an approach to measure the component-aware size of a software application. The idea is that, given an application whose size –measured according to the IFPUG standard– is S, we are interested in quantifying the part of the application that has to be developed, as opposed to the part already available from reusable components. We call Sca the size of the fraction of application to be developed. By definition, it is always Sca $\leq$ S; Sca=S when development involves no reuse at all.

A. Component-aware FUR models

Figure 1 represents –as a data flow diagram (DFD)– an elementary process (EP), together with the elements needed for FP measurement. In fact, to correctly measure the functional size of the EP, we need to take into account the logic data files that are used while executing the EP (FTR) and the elementary data (DET) exchanged with the elements that lay outside the borders of the application. In Figure 1, the only FTR is the Logic data file, and the DET are exchanged with the External element and possibly the Logic data file, if it is an EIF.

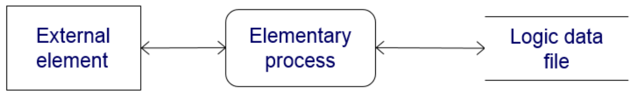

Figure 1. Representation of an Elementary Process.

It is easy to see that Figure 1 does not provide any information concerning the usage of components. Therefore, it is not possible to account for components when sizing the given EP.

To overcome this problem, we need a different representation of EP's. In particular, we need to represent that a component may support the EP. However, a component may support an EP only partly. In general, we would like that the "component-aware" size of an EP is inversely proportional to the role played by the component in carrying out the EP. Consider for instance an EP that is entirely supported by a component (so that the development effort needed to implement the EP is null, since the available component provides the entire functionality of EP): in this case, the component-aware size of the EP should be zero. On the contrary, if an EP is only partly supported by a given component (so that the development effort to implement the component-supported EP is not null, but smaller than the effort required for implementing EP from scratch): in this case the component-aware size Sca of the EP should be 0 <Sca<S.

Our proposal is to build component-aware models of EP's as shown in Figure 2.

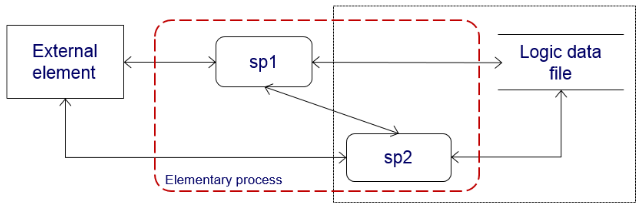

Figure 2. Representation of an Elementary Process and its sub-processes.

In Figure 2, the EP of Figure 1 has been split into two sub-processes: sp1 and sp2 (the dashed line indicates that these two sub-processes are actually perceived by the user –and are described in FUR– as a single EP). Moreover, the support provided by the available component is represented via the dotted line: in Figure 2, sp2 is supported by the available component, while sp1 is not.

Note that while building these component-aware models, we identify sub-processes that are either fully supported by components (i.e., we do not have to write code to implement them) or not supported at all by components.

B. Definition of component-aware size measures

In this section, we propose a method to differentiate the standard size of an EP [2], [3] from the component-aware size of EP. In what follows, we use the term $S(EP)$ to indicate the standard size of an EP (as defined by IFPUG), and $S_{CA}(EP)$ to indicate the component-aware size of EP.

Sub-components are measured *separately*, via the standard IFPUG FP measurement process. Note that –even though we use the standard measurement rules– we apply them to sub-process: this is not a standard practice.

With reference to Figure 2 we have that

- $S(sp1)$ is measured including in the DET's exchanged through the boundary those exchanged with the External element, those exchanged with sp2 and those exchanged with the data file if it is an EIF. The access to the data file is counted as a FTR.
- $S(sp2)$ is measured including in the DET's exchanged through the boundary those exchanged with the External element, those exchanged with sp1 and those exchanged with the data file if it is an EIF. The access to the data file is counted as a FTR.

Clearly, it may happen that $S(sp1) + S(sp2) > S(EP)$. As an example, suppose that we have an EP having size 7 FP. This EP includes three sub-processes:

- sp1, having size S(sp1) = 3 FP
- sp2, having size S(sp2) = 4 FP
- sp3, having size S(sp3) = 5 FP

$\sum_{j \in Subp(EP)} S(spj) = 3 + 4 + 5 = 12 > S(EP) = 7$, where $Subp(EP)$ is the set of sub-processes of EP.

Now, we have that a given EP can be split in several sub-processes, and that each of the available components can possibly support several sub-processes. To account for this situation, we proceed as follows.

Let $SPC(EP)$ be the set of sub-processes of a given EP that are supported by components (clearly, $SPC(EP) \subseteq Subp(EP)$).

When defining the component-aware size of a given EP, we want to subtract from its standard size $S(EP)$ the fraction that is provided by the available components. To this end, we define $S_{CA}(EP)$ as follows:

$$S_{CA}(EP) = S(EP)(1 - \frac{\sum_{i \in SPC(EP)} S(spi)}{\sum_{j \in Subp(EP)} S(spj)}) \qquad (1)$$

So, with reference to the previous example, the several cases are possible, including the following ones:

- If all sub-processes are supported by components, $SPC(EP) = Subp(EP)$, thus $S_{CA}(EP) = S(EP)(1 - \frac{3+4+5}{3+4+5}) = 0\ FP$. This is consistent with the fact that no code has to be written.
- If only sp1 is supported, $S_{CA}(EP) = S(EP)$ $(1 - \frac{3}{3+4+5}) = 7\frac{9}{12} = 5.25\ FP$. The fact that the component-aware size is less than the standard size accounts for the fact that you do not have to implement sp1, while sp2 and sp3 have to be coded.
- If only sp1 is not supported, while sp2 and sp3 are supported by available components $S_{CA}(EP) = S(EP)(1 - \frac{4+5}{3+4+5}) = 7\frac{3}{12} = 1.75\ FP$. The fact that the component-aware size is much less than the standard size accounts for the fact that you do not have to implement most sub-processes (sp2 and sp3), while only sp1 has to be coded.

So, it seems that the proposed measurement practice can effectively account for the usage of COTS components.

VI. Validation

When COTS components are used, developers do not need to develop the functionalities already provided by components. Accordingly, interesting characteristics of the development process –like the effort needed or the number of injected defects– are expected to depend mainly on the amount of software to be developed, rather than on the amount of software reused.

Therefore, we defined a method to adapt Function Point Analysis to measure the amount of software to be devopeded when components are reused. To get some initial evidence of the actual applicability and effectiveness of the proposed method, we applied it to the measurement of a real-life Web application. To support an effective comparison, two parts of the application were developed, with and without the usage of components.

A. The case study

We implemented a Web application, first from scratch, and then replacing two of the core features by means of existing COTS components.

The validation project has been developed in the context of the "Software Factory" program at the University of Bolzano-Bozen (http://www.newsoftwarefactory.org/).

The project's purpose was the construction of a Web application named 'Jack my tour' (Figure 3) with the purpose of supporting travelers in searching for available destinations (event sites, restaurants, bars, etc.) and optimizing travelling time based on the distance from locations and constraints such as the opening hours and the weather. To this end, the application creates an agenda for travellers and populates it with informatiion on the selected destinations and how to reach them.

The main features of 'Jack my tour' application include:

- Search events in a given zone within a given timeframe.

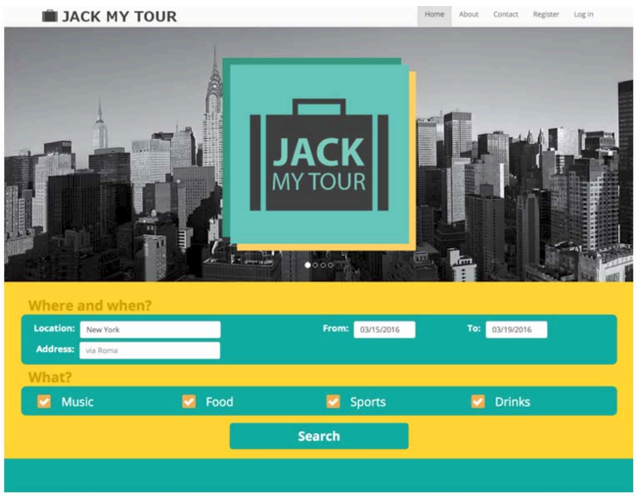

Figure 3. Home page of 'Jack my tour' Web application.

- Show the available events, in the categories "Music", "Food", "Sports" and "Drinks".
- Let the user select a set of events, then organize them in an agenda, accounting for existing constraints.
- Draw a map, showing the route to reach each location.
- Update the route on the map when the user changes the events in the agenda.

The search process performs a query on different online search engines, so as to retrieve a comprehensive list of events, merge them in a common format and remove duplicates. The result of the search process is shown when the results of the search are visualized, so that the user can select destinations (Figure 4).

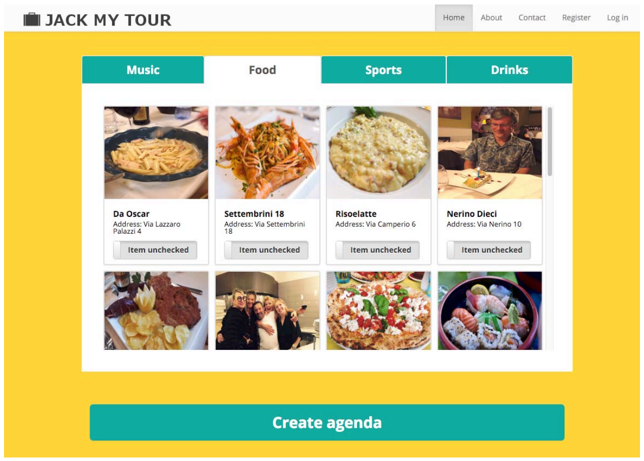

Figure 4. The destination selection page.

The agenda function (Figure 5) supports the graphical presentation of selected events to the user, allowing users to perform several operations, like moving an event to a different timeframe –automatically updating the route on the

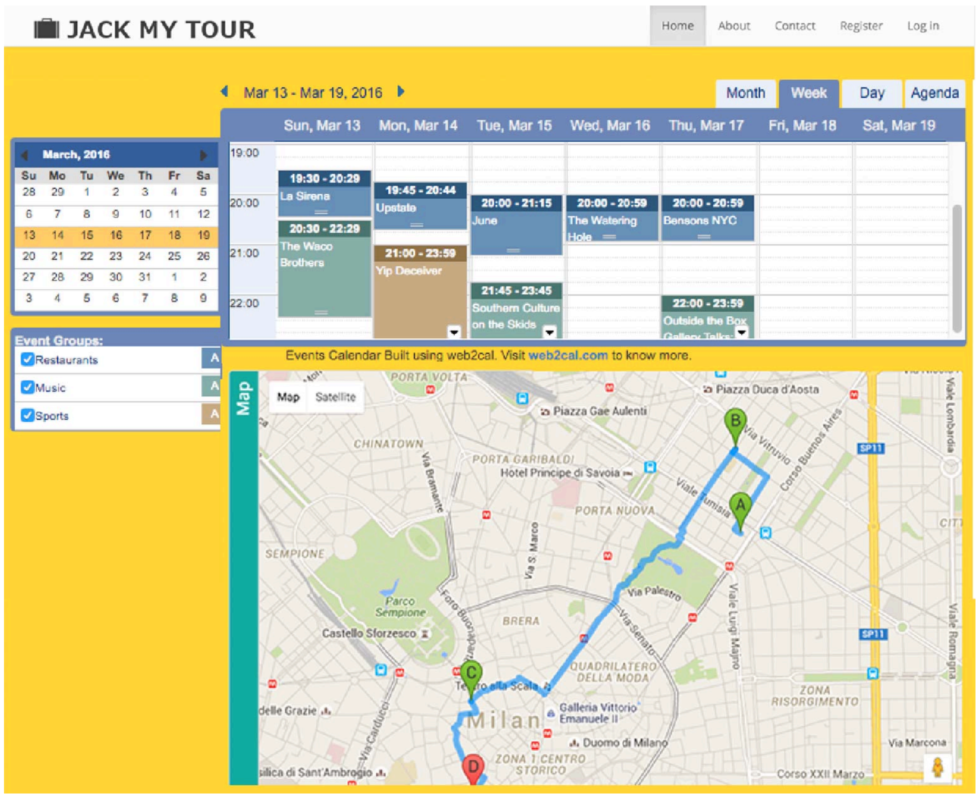

Figure 5. A view of the agenda.

map accordingly–, showing event details and deleting events.

B. Description of the Elementary Processes supported by Components

Here we describe the portion of our application supported by components and its functional measurement via standard IFPUG FP.

1) Agenda: The agenda (Figure 5) is responsible for a set of elementary processes:

- Visualize the selected events, properly representing them in the calendar, according to different views (month, week, etc.). This involves showing the route suggested to reach events' sites.
- Change starting event time and date. This change involves updating the route to reach events' sites.
- Change event duration. This change involves updating the route to reach events' sites.
- Retrieve and show event details.
- Select events that must be visualized in the calendar.

The size of the mentioned EP's is given in Table I.

2) Search Process: The search process retrieves event data form a set of event servers and populates the local event list with those data.

The search process is modelled as a data flow diagram in Figure 6.

Table I
MEASURES OF THE AGENDA'S EP'S.

Elementary Process	**Type**	**S**
Visualize selected events	EQ	3 FP
Change event time and date	EO	4 FP
Change event duration	EO	4 FP
Retrieve and show event details	EQ	3 FP
Select events to be visualized	EI	3 FP
Total	—	**17 FP**

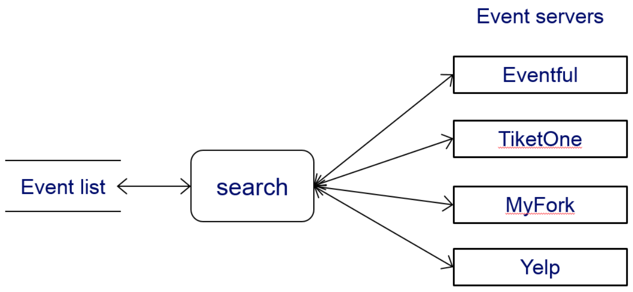

Figure 6. The search process.

The search process is an External Input, since its main intent is feeding the Event list. The analysis of the FTR and I/O DET leads to classify the EI as complex, hence its size is 6 FP.

C. *Component-aware sizing*

1) Component-aware sizing of the Agenda: The processes of the agenda were initially implemented from scratch, and then re-implemented with the Web2Cal component (http://www.web2cal.com/).

Some of the elementary processes are supported by the Web2Cal component, some are not, as specified in Table II by column CS (Component-Supported). The component-aware size of every EP is given in column S_{CA}.

Table II
MEASURES OF THE AGENDA'S EP'S.

Elementary Process	**Type**	S_{CA}	**CS**
Visualize selected events	EQ	0 FP	Fully
Change event time and date	EO	1.7 FP	Partly
Change event duration	EO	1.7 FP	Partly
Retrieve and show event details	EQ	1.3 FP	Partly
Select events to be visualized	EI	0 FP	Fully
Total	—	**4.7 FP**	—

As reported in Table II, two EP are fully supported by the component, therefore their component-aware functional size is zero. Three EP's are partly supported and here we calculate their component-aware functional size.

The EP "Change event time and date" is decomposed in sub-processes as shown in Figure 7.

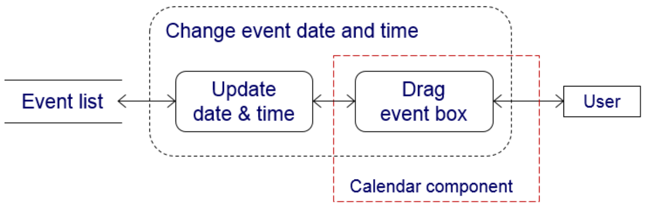

Figure 7. Change starting event time and date EP.

Sub-process "Drag event box" is responsible for managing the interaction with the user. It is a low complexity EO, thus its size is 4 FP.

Sub-process "Update date & time" is responsible for converting user actions into updates of events' dates and starting and finishing times. It is a low complexity EI, thus its size is 3 FP.

Sub-process "Drag event box" is supported by the calendar component; sub-process "Update date & time" is implemented from scratch. Accordingly, $S_{CA}(Change\_event\_date\_and\_time) = 4 \frac{3}{3+4} = 1.7\ FP$.

The EP "Change event duration" is decomposed in sub-processes as shown in Figure 8.

Sub-process "Modify event box" is responsible for managing the interaction with the user. It is a low complexity EO, thus its size is 4 FP.

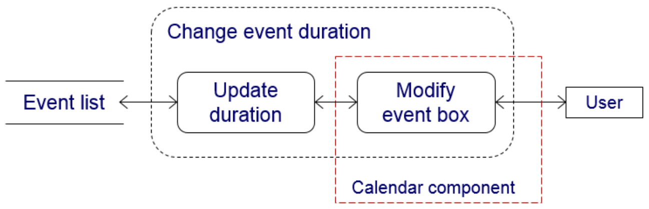

Figure 8. Change Event Duration EP.

Sub-process "Update date & time" is responsible for converting user actions into updates of events' durations. It is a low complexity EI, thus its size is 3 FP.

Sub-process "Modify event box" is supported by the calendar component; sub-process "Update date & time" is implemented from scratch. Accordingly, $S_{CA}(Change\_event\_duration) = 4 \frac{3}{3+4} = 1.7\ FP$.

The EP "Retrieve and show events details" is decomposed in sub-processes as shown in Figure 9.

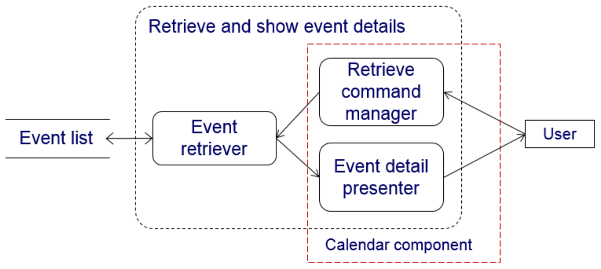

Figure 9. Retrieve and show event details on the calendar EP.

Sub-process "Retrieve command manager" is responsible for managing the commands from the user. It is a low complexity EO, thus its size is 4 FP.

Sub-process "Event retriever" is responsible for retrieving event details according to user commands and provide proper instructions to the "Event detail presenter", which is in charge of displaying event detailed information. It is a low complexity EO, thus its size is 4 FP.

Sub-process "Event detail presenter" is a low complexity EO, thus its size is 4 FP.

Sub-processes "Retrieve command manager" and "Event detail presenter" are supported by the calendar component; sub-process "Event retriever" is implemented from scratch. Accordingly, $S_{CA}(Event\_detail\_retrieval) = 4 \frac{4}{4+4+4} = 1.3\ FP$.

In conclusion, the EP's of the Agenda have a total component-aware size of 4.7 FP, while their total standard size is 17 FP. in practice, considering the contribution of components causes a quite relevant reduction of the EP's size measure.

2) Component-aware sizing of the Search page: The Search process is composed of two sub-processes: the Event Retrieval component, which queries different external sources, and the Event Manager, which feeds the Event List ILF, consolidating the obtained results in a standard format. The decomposition of the process into sub-processes is shown in Figure 10.

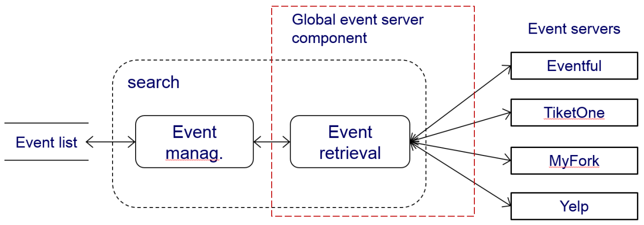

Figure 10. The search process.

The Event Retrieval sub-process in a complex External Query, whose size is 6 FP while the Event Manager is a simple External Input, whose size is 3 FP. The Event Retrieval sub-process is fully supported by the component, while the Event Manager was implemented from scratch.

The component-aware size of the Search EP is $S_{CA}(Search) = 6\ \frac{3}{3+7} = 1.8\ FP$.

D. Component-aware Effort Estimation

During the development of the Jack my tour application, we recorded the effort spent for developing both the component-free and the component-based versions.

Concerning the Agenda, we have a total (component-unaware) size of 17 FP. When considering components (see Table II) the size is 4.7 FP.

The agenda was developed by last-year master students, who worked for seven months. The development of the Agenda from scratch took 160 hours, while the development with the Web2cal component took 80 hours.

Developers took 152 hours to develop the search-component from scratch: 64 hours were spent implementing the Event Manager process and the remaining 88 hours implementing the Event Retrieval process.

The component-based implementation has been estimated in a total of 112 hours.

The situation is summarized in Table III. It should be noted that it was possible to observe the development effort dedicated to page implementations, while it was not possible to single out the effort dedicated to individual EP's, since most activities actually affected multiple EP's.

Table III
SIZE AND EFFORT DATA

	No components		With components	
EP (group)	Size	Effort	Size	Effort
Agenda	17 FB	160 PH	4.7 FP	80 PH
Search	6 FP	152 PH	1.8 FP	112 PH

As expected, with the usage of components both the size measure and the development effort decrease. However development effort decreases less than the size measure. Actually, the decrease in effort is approximately half the decrease in size. This suggests that deriving a model that supports the estimation of component-based development on the basis of component-aware functional size measures is possible. However, in order to be reliable and generally applicable, such model needs to be based on a large dataset and will possibly require that additional parameters are taken into account.

In any case, as an initial result, being able to show that a reduction of size measures is accompanied by a reduction of effort encourages to further refine the proposed measurement method.

VII. Discussion

A. Achieved goals

We can confidently state that the initial goal of defining a functional size measure that is both compatible with IFPUG FP and is able to account for the usage of COTS components has been achieved. In fact, the proposed component-aware measures take properly into account the usage of components by applying Function Point Analysis rules as far as possible.

Concerning the ability to estimate the savings in development effort deriving from component reuse, there seems to be a relationship between the decrease in size measures and the decrease in implementation effort. Nevertheless, the available data –concerning only two modules of the considered application– do not support any conclusion that can be reliably generalized.

B. Limitations of the proposed approach

It should be noted that the proposed approach considers the problem of defining component-aware functional size measurement of software only from the point of view of accounting for the reuse of *available* components. Under this respect, the proposed approach can be useful in avoiding discussions and litigations between developers (who would like to charge the development of the software sized according to the FUR of the whole application) and the payor (who would like to pay only for the actual development effort, hence excluding the effort saved because existing components have been reused).

However, another quite relevant situation occurs when reusable components are developed. In this case, both the cost of development and the value of the delivered software increase. The cost increases because the newly built component has to be packaged and tested as a stand-alone reusable product: this is an additional activity that increases the cost of development. The value increases because in addition to the software application, a component that can be reused (with relevant advantage) in the future is also released.

Dealing with this type of situations is beyond the scope of this paper, but is for sure an interesting topic for further investigations and proposals.

C. How to account for the usage of components

There is little doubt that we need to account for the usage of components when estimating the amount of effort needed to develop a software application, because the usage of ready-to-use components affects the amount of effort required for development. However, the effect of components on development effort can be taken into account in different manners, of which the main ones are:

- The size of the application to be developed is "corrected" to account for the usage of components. This is the approach used in this paper. The result is a 'component-aware' functional size measure.
- Another possible approach consists in measuring the software application via the usual standard IFPUG process, and then introduce a parameter that accounts for component usage in the effort estimation model. In this case, we get an effort estimation model *EstimatedEffort = f(S, Proc, Prod, ReuseFactor)* where S is the functional size measured in IFPUG FP, $Proc$ is a set of parameters accounting for the characteristics of the development process (e.g., experience of developers, tools used, process maturity, etc.), $Prod$ is a set of parameters that account for the characteristics of the software product (e.g., complexity, required reliability, etc.) and $ReuseFactor$ is a parameter that accounts for component reuse. As an effort model we could use COCOMO II: in such case, the AAF parameter would play the role of $ReuseFactor$.

In general, there can be reasons to prefer either approach over the other, depending on specific circumstances. We suggest a couple of reasons in favour of the approach presented here. A first consideration is that our approach is expected to be more accurate, since it considers the contributions of components to each elementary process individually, while the alternative approach applies a single parameter ($CompReuseFactor$) to the whole application. A second consideration is that in contract management, it could be easier to let the software provider and the payor agree on the 'component-aware' sizing method than letting them agree on the value of $CompReuseFactor$, which is more debatable. In fact, deciding to what extent a set of components contributes to the implementation of a complete application is quite hard, while observing if a specific sub-process is supported or not by a component is straightforward.

D. Dealing with data processing

Size measures provided by FPA do not account for the amount of data processing required to provide end users with the functionality described in the FUR [12]. In practice, FPA may assign the same size to elementary processes that involve much different amounts of data elaboration. This is a problem, since more data elaboration usually require more code to be written, hence a greater development effort.

This type of observation applies to components as well: a component may provide great data elaboration capabilities, thus saving a great deal of development effort, while another may incorporate little data elaboration, thus letting us save little development effort. With our approach the savings are evaluated on the bases of the sub-processes that are supported, not on the amount of data elaboration. In other words, our approach suffers from the same limitations as the standard IFPUG measurement.

VIII. Conclusions

In this paper we described some initial proposals and considerations regarding the need to account for the usage of reusable components in software functional measurement via Function Point measures.

Being able to provide a 'component-aware' functional size measure is important, because functional size measures are used for effort estimation, and development effort is affected to a possibly large extent by component reuse.

The proposed approach involves correcting the size of each elementary process based on the fraction of its functionality that is provided by components. An elementary process that is fully supported by components has null size. On the contrary, an elementary process that is not supported at all by any component has the same size yielded by IFPUG measurement; in this sense, our method complies with the IFPUG method.

A first application to a software application confirmed the applicability of the method, and its ability to account for the usage of components. However, the real utility of the proposed approach is in correcting effort estimates to account for the savings caused by component reuse. In our case study we were able to observe that when components are used, both the size measures and the development effort decrease. However, further experimentation is needed to draw generally valid conclusions.

Future work includes the following activities:

- As already mentioned, this paper aims at providing some initial ideas and thoughts about component-aware FSM. The proposed approach needs further experimentation and refinement before it can be reliably adopted in real-life software measurement and estimation and in project planning.
- Experimentation with different types of applications and components is necessary, to evaluate to what extent the proposed approach is applicable, and what refinements and calibrations are needed.
- To define effort estimation models that are 'component-aware', we need a sufficiently numerous dataset, so that statistically significant analysis can be carried out. To

this end, we need to collect measures from multiple projects.

- As discussed in Section VII-C, there can be different ways of accounting for the usage of components in software development. Therefore, it would be interesting to compare the performance of the proposed method to other possible methods, to identify what method is preferable in the given circumstances.
- Finally, we should observe that FPA is not the only widely used FSM method. So, we should consider adapting the proposed approach to other FSM methods, in particular to the COSMIC method [9], [13], which is by far the most used after IFPUG FPA.

IX. Acknowledgments

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission and the "Fondo di Ricerca d'Ateneo" of the Università degli Studi dell'Insubria.

References

[1] A. J. Albrecht, "Measuring application development productivity," in *Proceedings of the joint SHARE/GUIDE/IBM application development symposium*, vol. 10, 1979, pp. 83–92.

[2] International Function Point Users Group, *Function Point Counting Practices Manual - Release 4.3.1*. IFPUG, January 2010.

[3] ISO, *ISO/IEC 20926: 2003, Software engineering– IFPUG 4.1 Unadjusted functional size measurement method– Counting Practices Manual*. ISO/IEC, 2003.

[4] D. Garmus, J. Mayes, and M. Chemuturi, *The IFPUG Guide to IT and Software Measurement*. Auerbach Publications, 2012.

[5] COSMIC, "Measurement practice committee, method update bulletin n.11," Tech. Rep., 2013.

[6] D. Maschino and O. Makar-Limanov, "Wave of the Future Function Point Sizing and COTS Support," in *2006 International Software Measurement and Analysis Conference*, 2006.

[7] I. Brown, "The "Zero Function Point" Problem," in *2009 International Software Measurement and Analysis Conference*, 2009.

[8] S. Trudel and A. Abran, "Measurement of business rules specified as reusable components: Exploratory study of its impact on the functional size of software projects," in *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*. IEEE, 2013, pp. 44–48.

[9] COSMIC, "The COSMIC Functional Size Measurement Method–Version 4.0.1–Measurement Manual (The COSMIC Implementation Guide for ISO/IEC 19761: 2011)," Tech. Rep., April 2009.

[10] L. Lavazza, "Lessons learned on software maintenance: any relief at horizon?" Panel at ICSEA 2014, 2014.

[11] B. W. Boehm, R. Madachy, B. Steece *et al.*, *Software cost estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[12] L. Lavazza, S. Morasca, and D. Tosi, "A Study on the Difficulty of Accounting for Data Elaboration in Functional Size Measures," *Journal on Advances in Software*, vol. 8, no. 1-2, 2015.

[13] International Standardization Organization (ISO), "ISO/IEC 19761:2011 Software engineering – COSMIC: a functional size measurement method," 2011.