Exploring quality-aware architectural transformations at run-time: the ENIA case

Javier Criado[1], Silverio Martínez-Fernández[2],
David Ameller[2], Luis Iribarne[1] and Nicolás Padilla[1]

[1] Applied Computing Group, University of Almería, Spain
{javi.criado, luis.iribarne, npadilla}@ual.es
[2] GESSI Research Group, Universitat Politècnica de Catalunya, Barcelona, Spain
{smartinez, dameller}@essi.upc.edu

Abstract. Adapting software systems at run-time is a key issue, especially when these systems consist of components used as intermediary for human-computer interaction. In this sense, model transformation techniques have a widespread acceptance as a mechanism for adapting and evolving the software architecture of such systems. However, existing model transformations often focus on functional requirements, and quality attributes are only manually considered after the transformations are done. This paper aims to improve the quality of adaptations and evolutions in component-based software systems by taking into account quality attributes within the model transformation process. To this end, we present a quality-aware transformation process using software architecture metrics to select among many alternative model transformations. Such metrics evaluate the quality attributes of an architecture. We validate the presented quality-aware transformation process in ENIA, a geographic information system whose user interfaces are based on coarse-grained components and need to be adapted at run-time.

Keywords: quality-driven model transformation, component-based software, architecture configuration, architecture metrics.

1 Introduction

Many today's software systems need to be adapted at run-time. Well-known examples are component-based software systems for human-computer interaction, whose User Interfaces (UI) need to be modified or reconfigured at run-time depending on user preferences, interactions, system requirements, or other evolution needs. For instance, this type of software is offered by dashboard UIs, such as Netvibes[3], Geckoboard[4], or Cyfe[5] applications.

Previous studies have shown that model transformation is a good approach to adapt the component-based architectures [10]. Existing transformation processes

[3]Netvibes – https://www.netvibes.com/
[4]Geckoboard – https://www.geckoboard.com/
[5]Cyfe – http://www.cyfe.com/

focus on the functionalities of systems, giving less importance to the Quality Attributes (QA). However, adapting the software architecture of a system without considering the QAs at run-time can negatively affect its quality. For instance, if one UI is adapted by only considering its functionalities, such UI may have a less flexible interaction (*e.g.*, complex UI with a greater number of components) or worse maintainability (*e.g.*, costly evolution from introducing unnecessary dependencies among components) than if we consider QAs at run-time. Actually, some QAs (*e.g.*, availability or performance) can only be measured only at run-time since off-line circumstances provide an estimation and not a real value.

The goal of this paper is to study whether model transformations can be improved by considering QAs at run-time. To this end, we present a QA-aware transformation approach to adapt component-based software systems by measuring the quality of different transformation alternatives. Then, we validate the suitability of such QA-aware transformation approach in two scenarios for the ENIA (ENviromental Information Agent) software. ENIA is a GIS (Geographic Information System) whose UIs are based on coarse-grained components and adapted at run-time depending on user preferences, interactions, system requirements, or other evolution needs [1]. Nevertheless, the approach can be applied to other applications offering their functionality as a mashup or a dashboard.

To accomplish this goal, we formulated the following research question: "*Do model transformations improve their quality by considering QAs at run-time in the ENIA case?*". To measure QAs at run-time, we propose a set of software architecture metrics. We use these metrics to evaluate various alternative architectures (each one obtained by executing a different transformation). As a result, we decide which is the best transformation based on the considered QAs.

The paper is structured as follows. Section 2 introduces a background about adapting component-based systems by model transformation. Section 3 presents a QA-aware model transformation approach. Section 4 exemplifies the approach in the ENIA case. Finally, Section 5 ends up with conclusions and future work.

2 Background

In this section we include the required background for contextualizing the presented approach of quality-aware architectural transformations at run-time.

Adapting Component-Based Software Systems. Component-based systems are a type of software which facilitates the execution of adaptation and evolution operations. In this sense, well-known mechanisms of *Component-Based Software Engineering* (CBSE), such as modularization, encapsulation and reuse, favor the development of self-adaptive systems [17].

This software paradigm allows us to manage the components as black-boxes by describing their syntax, semantic, and properties through formal specifications, as in the case of COTS components [6]. Thus, a component can be replaced by other element that matches its specification. Consequently, an architecture can be modified by replacing the parts which need to be adapted.

Model Transformation and Software Architecture Evolution. Model transformation is a common approach to adapt the component-based architecture of software systems [10]. In this context, *Model-Based Engineering* (MBE) techniques facilitate the development of software architectures, defining them (including the structure, components' specifications, and run-time interaction) by models. Moreover, manipulating architecture models at run-time allows us to generate different alternatives based on the same definition [4].

Depending on the model transformation nature (*e.g.*, vertical, horizontal, endogenous, and exogenous) and within the context of software architectures, it is possible to develop refactoring transformations for obtaining different software alternatives. Our goal is to modify the transformation schema proposed in [10] for generating more than one alternative and consider quality information to select the best transformation.

Quality Attributes in Model Transformation. Existing model transformation processes focus on the functionalities of systems, giving less importance to the QA, also known as non-functional requirements or *-ilities* [3]. A notable exception are the guidelines for quality-driven model transformations [12], in which quality is introduced early on the design of the transformation process, avoiding quality evaluation as a separate activity once a model has been transformed. A more recent work presented a model transformation framework designed to automate the selection and composition of competing architectural model transformations [14]. However, up to our knowledge, there is a lack of support to select among alternative architectural transformations considering software architecture metrics at run-time.

Some approaches enable the annotation of model transformations [9] and can be applied for describing QAs in transformation rules. Other proposals extend existing languages with the aim of expressing alternatives and their impact to quality properties at design time [18]. Furthermore, not all QAs share the same importance while adapting or evolving software systems. A recent literature review shows that self-adaptation is primarily used to improve performance, reliability, and flexibility [20]. In this context, an important challenge is to find software architecture metrics that measure quality attributes. The awareness of this problem by the software engineering community is increasing and even dedicated events have been organized [16]. For instance, *dependency structure matrix* metric has been used to measure maintainability [7, 15]. Another examples are the number of components, connections, symbols, and interfaces to measure architectural understandability [19, 22]. In the next section, we use these metrics and propose others to measure the relevant QAs in adaptive systems.

3 QA-aware Model Transformation Approach

This section presents a QA-aware transformation approach to adapt and evolve software systems by measuring the quality of different transformation alternatives. Such QA-aware transformation approach consists of three steps:

(1) Asking the relevant QAs and constraints to developers, architects, and experts in the application domain.
(2) Measuring QAs and constraints at run-time.
(3) Ranking iso-functional alternative software architectures of the model transformation by considering the relevant QAs and constraints at run-time. With this ranking, the software architecture with the best values in architecture metrics is selected.

Next subsections describe the aforementioned steps respectively, which are also depicted in Figure 1. Once the last step is carried out and the transformation alternatives have been ranked, the transformation with the best value is executed for adapting the software architecture.

3.1 Step 1: Relevant QAs and Constraints

Depending on a system's targeted goals and architecturally-significant QAs (*e.g.*, improve its flexibility, maximize the modifiability, minimize the cost, or optimize the execution performance), architectural design decisions can be oriented in different ways. Therefore, decisions about the construction of software architectures, such as component selection, may differ from each other by considering them. For this reason, the first step of the approach is to gather the architecturally-significant QAs as part of the rationale to make such decisions.

This step requires two inputs: stakeholders who know the system's targeted QAs, and a quality model to help the stakeholders to reason about QAs (*e.g.*, ISO/IEC 25010 standard [13]). To gather the architecturally-significant QAs, stakeholders can either conduct a focus group, or directly set them in the settings of an admin interface of the model transformation process. The output of this step is the set of relevant QAs and constraints for the adaptive software system.

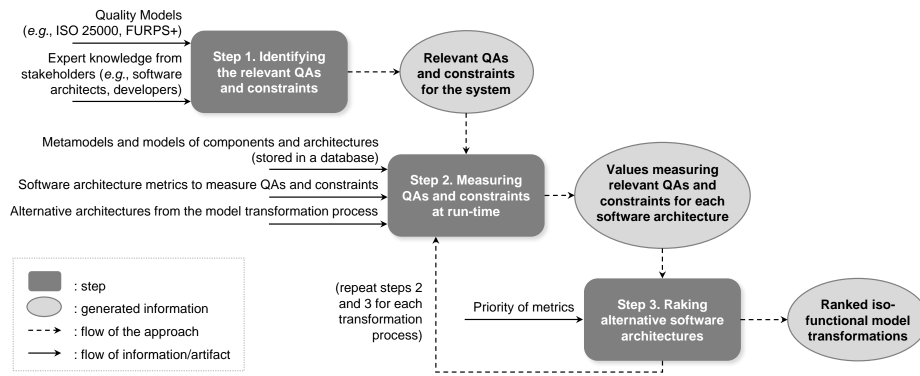

Fig. 1. Steps of the QA-based Trasformation Approach

3.2 Step 2: Measuring QAs and Constraints at Run-Time

Once the set of relevant QAs and constraints are elicited, the goal of the second step is to find specific software architecture metrics to measure QAs. This enables to quantitatively evaluate several alternative software architectures, since the QA satisfaction of these alternative architectures is measured at run-time. The metrics presented here are focused on our particular domain of component-based systems, but they can be adapted according to the needs.

This step requires three inputs: the set of alternative software architectures (generated by the default model transformation process), a set of metrics to measure QAs and constraints (see Table 1 for QAs and Table 2 for constraints), and the specification of the components to feed the metrics at run-time (stored in the metamodel [8, 10]). It is important to emphasize that Tables 1 and 2 show metrics elicited and validated by an expert in the domain architecture of component-based systems, but they may be adapted to the needs of other domains. Table 1 shows simple metrics for important QAs in adaptive systems: performance, reliability, flexibility [20], and a few more among other important metrics that we used in the ENIA study (modifiability, testability and consumed resource). These QAs are meaningful at run-time in our case study as it is described in Section 4. Simple and realistic metrics allow easier adoption in industry [15]. Also, the proposed metrics are just an indicator of a QA, and their improvement must not be seen as a complete satisfaction of any QA. The output of this step is a set of quantitative values measuring the targeted QAs and constraints supporting the selection of the best transformation.

QA	Metric	Description	Derived	Expression
m	c	Number of components	n	—
	pro	Number of provided interfaces	n	—
	req	Number of required interfaces	n	—
	$\overline{pro}$	Average number of provided interfaces	y	pro/c
	σ^2_{pro}	Variance of provided interfaces	y	$\sum_{i=1}^{c}(pro_i - \overline{pro})^2/c$
	$hpro$	Homogenization of provided interfaces	y	$1 - \sigma^2_{pro}$
	$mdep$	Number of mandatory dependencies	n	—
	$odep$	Number of optional dependencies	n	—
	dep	Total number of dependencies	y	$mdep + odep$
	$rmdep$	Ratio of mandatory dependencies	y	$mdep/dep$
	$rodep$	Ratio of optional dependencies	y	$odep/dep$
	dsm	Dependency structure matrix	y	(described in [7])
	pc	Propagation cost	y	(described in [7])
f	r	Number of resizable components	n	—
	m	The highest c from all alternatives	y	$max(c_1, \ldots, c_n)$
	rc	Ratio of components according to m	y	$rc = c/m$
	rr	Ratio of resizable components	y	$rr = r/c$
r/a	er	Error rate (and type of error)	n	—
	ec	Error cost	n	—
p	$extm$	Execution time of a component	n	—
	$rextm$	Ratio of execution time of all components	y	$\sum(extm_i)/c$
t	$ndiag$	Num. of ops. (in pro) intended for diagnostics	n	—
	$ntest$	Num. of ops. (in pro) intended for tests	n	—
cr	$tsize$	Total size of components (in KB)	n	—
	$avgsize$	Ratio of components' sizes (in KB)	y	$tsize/c$

QAs: m: modifiability – f: flexibility – r/a: reliability/analizability
p: performance – t: testability – cr: consumed resources

Table 1. Example of software architecture metrics to measure QAs

Metric	Description	Derived	Expression
c	Number of components	n	—
pro	Number of provided interfaces	n	—
req	Number of required interfaces	n	—
$tsize$	Total size of components (in KB)	n	—
$avgsize$	Ratio of components' sizes (in KB)	y	$tsize/c$
t	Component technology	n	—
st	No. of components sharing the same technology	n	—
ht	Homogenization among components' technologies	y	$max(st/c)$
p	Component provider	n	—
sp	No. of components sharing the same provider	n	—
hp	Homogenization among components' providers	y	$max(sp/c)$
$type$	Component type	n	—
$stype$	No. of components sharing the same type	n	—
$htype$	Homogenization among components' types	y	$max(stype/c)$

Table 2. Example of metrics to measure constraints

3.3 Step 3: Ordering Alternative Software Architectures

In our approach, we first generate the various possible architectures by applying alternative transformation processes, and then we assess the quality of each architecture. After computing at run-time the corresponding metrics to measure the QAs and constraints, it becomes necessary to rank the alternative software architectures considering the relevant QAs and constraints. Thus, the goal of the third step is to select the "best" software architecture. Consequently, the operation responsible for obtaining this architecture, *i.e.*, the corresponding model transformation, is selected as the best alternative.

This step requires one input: the priority of the architecturally-significant QAs and constraints. This order of importance can be established by system's developers or by users for describing their own priorities. In all cases, it must be specified before the adaptation process starts and could be subsequently modified at run-time to vary this priority. The output is a ranked list of iso-functional model transformation. Finally, the model transformation with the best software architecture is performed. The second and third steps of the approach can be performed at run-time if the number of relevant QAs and alternative architectures to be analyzed is delimited in order to guarantee a proper execution.

4 Application of the QA-aware Transformation Approach

In order to demonstrate the feasibility of the QA-aware transformation process, this section applies it to a particular case of software architectures: mashup UIs [11, 21]. Next subsection respectively show the context of ENIA (mashup UIs), two scenarios at ENIA in which the approach was used, and discussions about how the approach improved the model transformation process.

4.1 The Model Transformation Adaptive Context of ENIA

We addressed this research work focusing on component-based software systems for human-computer interaction. More specifically, we validated the approach

presented in Section 3 by using the scenario of ENIA UIs, which are used for managing a GIS through coarse-grained components implemented as widgets. Some examples of these components are maps for showing geographic layers, visual charts for representing datasets, or social widgets for enabling the communication with other GIS entities and the community.

ENIA UIs development highlighted the different alternatives that exist when a new architecture is constructed, whether it is determined at design time or it is generated dynamically at run-time. Moreover, such alternatives may be equally valid depending on the quality factors that are taken into account for its construction. For this reason, a quality-aware transformation approach is addressed. Hence, ENIA has been chosen as our test scenario, since the UIs offered by this system are represented and managed as architectures of coarse-grained components. Each component in ENIA architectures contains the required functionality to perform a task by itself or using its dependencies with other components (*e.g.*, a geographical information map, an e-mail manager, a report generator, or a visual chart of datasets). Furthermore, UIs in ENIA are reconfigured at run-time with the aim of adapting their structure to the user interactions, profile preferences, context changes and pro-active system decisions. Since UIs are represented by models, this adaptation process is based on model-to-model (M2M) transformations of component-based architectures (see [10] for further details).

Model transformations in charge of adapting ENIA's UIs are not preset. On the contrary, these operations are dynamically built depending on the initial UI, context information and available transformation rules. In this sense, it is possible to build different transformation operations for the same inputs. In our previous work [10] we proposed a transformation scoring mechanism that allows us to generate only one possible transformation which ensures the functional resolution of the architecture, taking also into account some extra-functional restrictions. In the present paper, we analyze the possibility of incorporating additional quality information into the adaptation process to improve its behavior.

As the Step 1 of the approach indicates, the relevant QAs that should be considered for constructing and adapting the UIs have been discussed for the ENIA scenario. This work is focused on the highest priorities of ENIA, the *modifiability* attribute of the *product quality model* and the *flexibility* attribute of the *quality in use model* of the ISO/IEC 25010 standard, the *total size* of the UI, the homogenization of the *components' providers*, and the homogenization of the *components' types*. These QAs and constraints have been selected because, from the stakeholders' perspective, ENIA UIs must be reconfigurable and provide a friendly interaction by accomplishing the following objectives:

— UI components must encapsulate the required functionality to resolve a domain task but their size must match the coarse-grained concept.
— UIs with a greater number of simple components are preferred over UIs with a lower number of complex components gathering more functionality.
— UIs must be elastic and flexible with the aim of allowing the modification of their structure and visual representation.

— Users can reconfigure and customize their interfaces, e.g., resizing the component displayed in the interface, changing its position, adding new components, and removing existing ones.
— The time for loading the UIs must be the minimum possible.
— UIs with an homogeneous representation are better managed and more understandable by users, generating less confusion for the interaction.

Next, we present two scenarios in the context of ENIA. In the first scenario, we analyzed four metrics related to modifiability and flexibility QAs. Metrics to measure modifiability, flexibility, analyzability, performance, testability and consumed resources are shown in Table 1. In the second scenario, we applied three constraints related to the aforementioned objectives. Metrics to measure constraints are shown in Table 2. Both scenarios share components related to the GIS domain: a map (M) for displaying geographical information layers, a component showing the messages of a social network account (T), a layer list for selecting the information to be displayed on the map (LL), and a legend for visualizing the correspondence of the displayed layers (L). Furthermore, the number of the component name represents different alternatives of the same component type. For example, $M1$, $M2$ and $M3$ are different alternatives of the map component type (M). It is important to remark that we are not representing the identifier of the instance and therefore an architecture can contain elements with the same component name, but these are different component instances.

4.2 Scenario 1: Considering Modifiability and Flexibility

This scenario focuses on two QAs from Step 1, flexibility and modifiability, and following Step 2 applies four corresponding metrics from Table 1: (*rc*, *rr*, *hpro* and *pc*). These metrics will help to select the best transformation process in charge of adapting a UI from ENIA system based on those QAs. Assuming an initial UI as the one shown in Figure 2 with four components (map, twitter, legend and layer list), a transformation process for incorporating a session component must be performed. Furthermore, the twitter component must be replaced by a new element of the same type which must be connected to the session component for using its information in the social network account.

We address the use of the following four metrics as worthy information to determine the quality of the resulting architecture and consequently the quality of the transformation process in charge of adapting the UI:

(a) *rc* for flexibility – The number of components must be maximized because the more pieces constitute a UI, the more reconfiguration and modification operations can be performed on its architecture.
(b) *rr* for flexibility – The number of resizable components must be maximized since this property favors the flexibility of a UI due to it allows the modification of the components' sizes.
(c) *hpro* for modifiability – The homogenization of the distribution of provided interfaces must be maximized because this property avoids the imbalance

in the functionality which is offered by each component and fosters the modifiability of the architecture.

(d) *pc* for modifiability – The propagation cost must be minimized due to indirect dependencies between components of an architecture affect the modifiability in a negative manner.

Figure 3 shows the four transformation alternatives that can be obtained from the initial UI represented by $A11$. The architecture $A21$ incorporates a session component $S1$, replaces the previous twitter by $T2$ and connects both elements. $A22$ is similar to $A21$ but the session component $S2$ has an optional required interface for querying geo-localization information. The alternative $A23$ gathers a session component $S3$ and a geo-localization component $LC1$ in a container $C2$. In the case of the last alternative, the component $S4$ in $A24$ provides some geo-localization and weather information apart from the session functional interface.

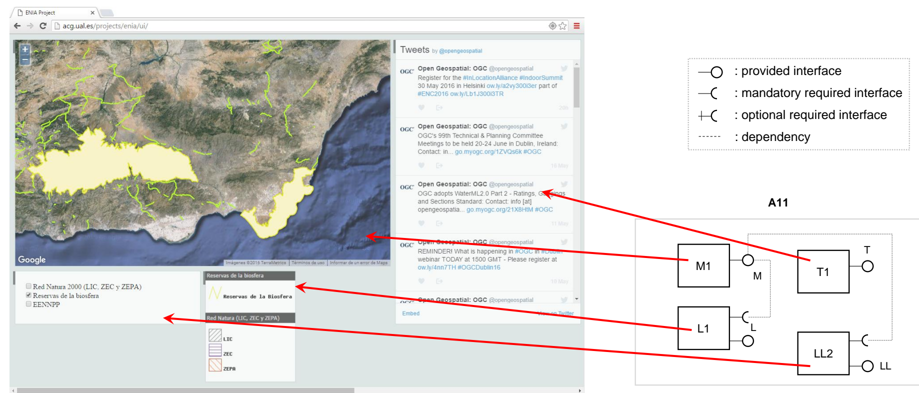

Fig. 2. Initial UI and its architecture

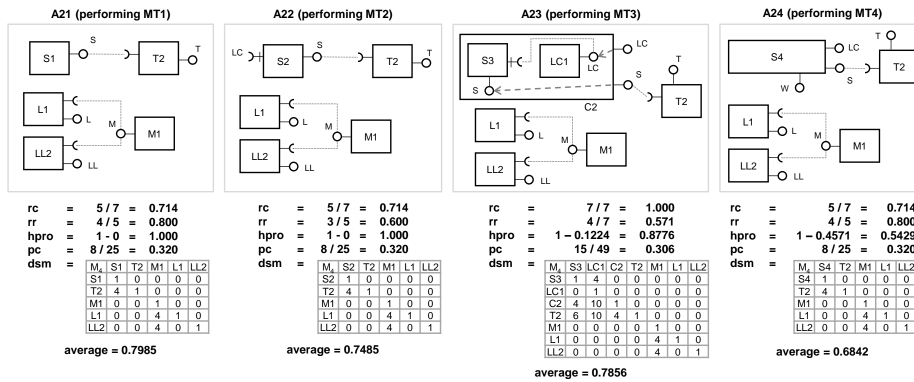

Fig. 3. Transformation alternatives using four example QAs' metrics

In order to select the best model transformation alternative among $MT1$, $MT2$, $MT3$ and $MT4$, the bottom of Figure 3 depicts the values of the metrics calculated for $A21$, $A22$, $A23$ and $A24$. Note that components $M1$, $T2$, $L1$, $C2$, $S1$ and $S4$ are resizable, and components $LL2$, $S2$, $S3$, and $LC1$ cannot be resized. Moreover, delegation of interfaces is considered as a dependency, similar to the connections between required and provided interfaces.

Focusing only on rc, the architecture $A23$ has the best value because it owns the maximum number of components among all the alternatives. In the case of rr, architectures $A21$ and $A24$ are the best alternatives since both gathers four resizable components among the five possible. Architectures $A21$ and $A22$ have the best value for the $hpro$ metric because each component provides one functional interface. On the contrary, the distribution of provided interfaces in $A24$ is the worst possible alternative. With regard to the propagation cost, pc, the architecture $A23$ is the best alternative, as shown in the values obtained from dependency structure matrices (DSMs). We normalized pc value with respect to the rest of metrics with the expression $npc = 1 - pc$.

Finally, we applied Step 3 to select the best alternative by calculating the average of the four metrics. Therefore, the resulting values for $A21$, $A22$, $A23$ and $A24$ are 0.7985, 0.7485, 0.7856 and 0.6842, respectively. Consequently, we can follow this strategy to select $T1$ as the best transformation process that can be performed to (1) adapt the UI and also (2) get the best value for QAs considered in the transformation. Figure 4 shows the graphical representation of the UI described by the architecture $A21$.

4.3 Scenario 2: Considering Components' Size, Provider, and Type

Continuing with the UI represented by $A21$ in the scenario 1, the next transformation process is intended to incorporate a new map into the workspace. Since

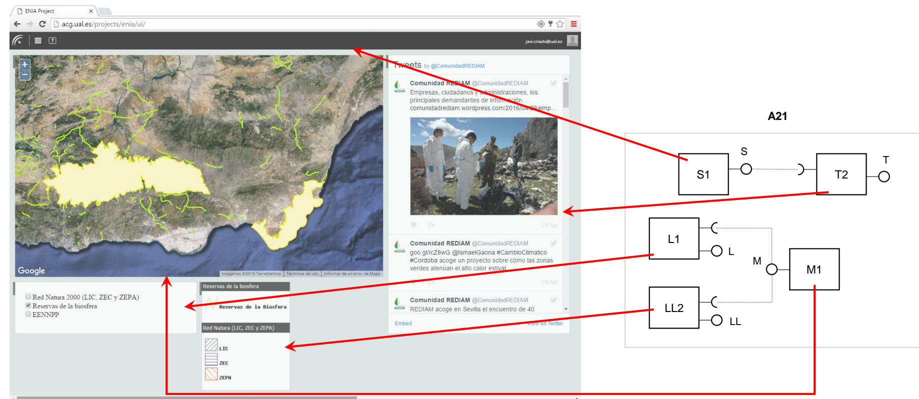

Fig. 4. Transformed UI in scenario 1 and its architecture

the presence of the new map may generate confusion about what is the relationship between the layer list, the legend and the two maps, the components in the initial UI must be restructured accordingly. Figure 5 shows the three alternatives that can be reached from $A21$. The architecture $A31$ replaces the previous map $M1$ by $M2$ and uses $C3$ for containing it. In addition, $C3$ also contains previous $LL2$ and $L2$ components. The new map is resolved with an $M1$ component. The second alternative, $A32$, replaces the initial map $M1$ by $M3$, a map which includes the layer list and legend functionality. The new map is $M1$ type. The alternative $A33$ includes the same replacement of $A31$ but, in this case, the new map is $M2$ type and it is contained in a $C3$ component.

In this transformation process, we want to show that the approach could be used not only for QAs, but also for constraints. In this case, we consider three constraints from Step 1 as drivers to chose a valid alternative for the stakeholders. As a part of Step 2, the metrics used to measure the constraints are:

(a) *tsize* for components' size constraints – The total size of components must be minimized and architectures with a value over 5MB will be rejected. Thus, we try to improve the response time of the browser by reducing the payload of the web components that must be initialized in the UI.
(b) *hp* for components' provider constraints – The homogenization among components' providers must be maximized because, in this scenario, UIs with similar representation are preferred over components with heterogeneous representations. The use of the same provider does not guarantee the pursued homogenization, but the possibilities are greater if the entity providing the components is the same.
(c) *htype* for components' type constraints – The homogenization among components' types must be maximized because it is important to offer the maximum degree of consistency in the structure and representation of the UI's components. Therefore, components of the same type offer their functionality in the same manner.

Regarding the alternatives of Figure 5, each architecture accomplishes the best value for a different metric. In the case of *tsize*, the value of 925 MB from $A32$ is the best alternative. Focusing on *hp*, the best alternative is the architecture $A31$ because it gathers four components ($M1$, $M2$, $L2$ and $LL2$) from the same provider among the total of seven. With respect to *htype*, the best alternative is $A33$, because it contains four components ($M2$ * 2 and $C3$ * 2) having elements of the same type in the architecture. Therefore model transformation $MT4$ is chosen in the case of prioritize *hp*, whereas $MT5$ and $MT6$ are selected if *tsize* and *htype* are prioritized, respectively.

4.4 Discussion about Using the Approach in ENIA

Answering the research question stated in Section 1, we can say that considering QAs at run-time has improved the modifiability and flexibility of generated architectures by model transformations in the ENIA case.

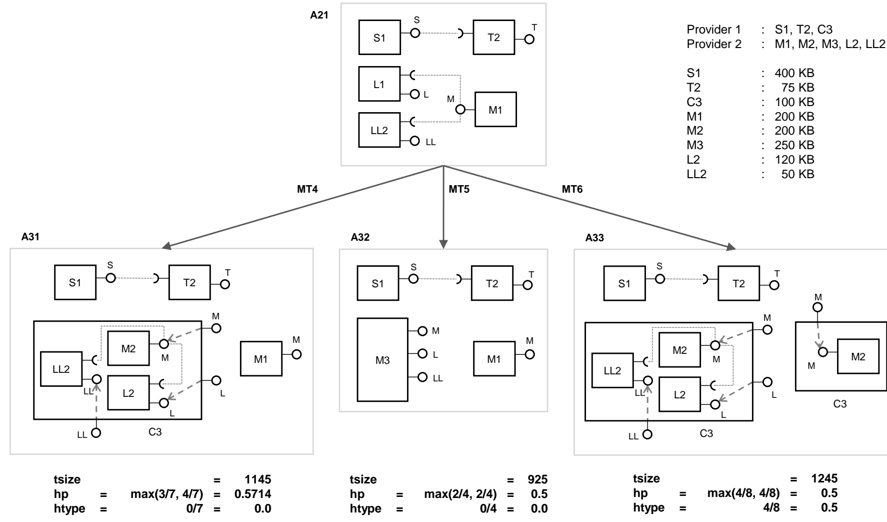

Fig. 5. Transformation alternatives using three example constraints

The main advantage of using metrics related to QAs and constraints in ENIA is the incorporation of quality information in the process of selecting the best transformation operation that can be applied in UI adaptation. This allows us to use additional information (to functional interfaces) for solving the transformation process. In this sense, if these metrics are not applied, the transformation can generate architectures which may result in some drawbacks for the present use or future modifications.

For example, looking at the first scenario (see Figure 3), it is possible to obtain $A22$ as a solution instead of $A21$. In this case, we are 'loosing' the capability of having a session component which can be resized. On the contrary, using our approach we are able to offer 'resizability' of the components through the maximization of the rr metric. If we do not give the maximum priority to rr but we take it into account in the adaptation, at least the transformation at run-time will be enriched to improve the flexibility of generated UIs.

With regard to the future modifications, let us suppose that in the scenario 2 none of the metrics are applied and consequently, the generated transformation is equivalent to $T5$ and the resulting architecture is $A32$. In this case, if the next adaptation step is aimed to remove the capability of selecting the layers to be displayed on the map (*i.e.*, LL provided interface), we faced two options: (1) the component $M3$ must be modified for hiding this interface and disabling its functionality, or (2) the component $M3$ must be replaced by other map which does not include this functionality, such as $M1$ or $M2$. In both options, we have to perform additional operations compared to those required in the case of starting the adaptation from the architectures $A31$ or $A33$, scenario in which we only should remove the component $LL2$.

Apart from these advantages, nothing is free in software engineering, and the performance of the QA-aware model transformation approach is an important trade-off that must be noted. Performance is related to the computation time necessary to (a) build each transformation alternative, (b) execute them obtaining the resulting architecture, and (c) measure each architecture to decide which transformation alternative is the best in terms of the quality information. The cost of these three execution times must be incorporated to the evaluation of the adaptation process described in [10] and, consequently, may not be possible to evaluate a large number of alternatives at run-time, having to limit the number of architectures evaluated to satisfy performance requirements.

5 Conclusions and Future Work

It is well accepted in the software architecture community that QAs are the most important drivers of architecture design [2]. Therefore, QAs should guide the selection of alternative software architectures from a model transformation process, considering the synergies and conflicts among them [5].

This work has analyzed how considering QAs at run-time can improve model transformation processes. Results in the ENIA case, a dashboard UI, show that using a quality-aware architectural transformation at run-time can improve architectural-significant QAs such as modifiability and flexibility. The main contribution of this paper is a quality-aware transformation approach at run-time, which consists of three steps: identifying relevant QAs and constraints, measuring them at run-time, and selecting the best alternative model transformation.

Future work spreads in several directions. First, once we analyzed in the ENIA case that quality-aware transformations can improve significant requirements in adaptive dashboards UIs, the presented set of metrics can be refined. Thus, we plan to work further in a reference set of QAs and their corresponding metrics for adaptive dashboard UIs out of practice, and then provide guidelines for using those metrics (*e.g.*, combination of metrics). Second, more experimentations and reports can be done in other adaptive domains besides dashboard UIs. Third, we will study the possibility of handling the QAs during the generation of the alternative architectures to reduce the number of variants. Finally, a formal validation process in terms of execution times and model checking of the generated architectures could improve the proposed approach.

Acknowledgments. This work was funded by the Spanish MINECO and the Andalusian Government under TIN2013-41576-R and P10-TIC-6114 projects.

References

1. ACG. ENIA Poject – Development of an intelligent web agent of environmental information, `http://acg.ual.es/projects/enia/`
2. Ameller, D., Ayala, C., Cabot, J., Franch X.: Non-functional requirements in architectural decision making. IEEE Software. 30(2), 61–67 (2013)

3. Ameller, D., Franch, X., Cabot, J.: Dealing with non-functional requirements in model-driven development. In: RE'2010, pp. 189–198. IEEE (2010)
4. Bencomo, N., Blair, G.: Using architecture models to support the generation and operation of component-based adaptive systems. In: Software Engineering for Self-Adaptive Systems, pp. 183–200. Springer Berlin Heidelberg (2009)
5. Boehm, B.: Architecture-based quality attribute synergies and conflicts. In: SAM'2015, pp. 29–34. IEEE Press (2015)
6. Carney, D. Leng, F.: What do you mean by COTS? Finally, a useful answer. IEEE Software. 17(2), 83–86 (2000)
7. Carriere, J., Kazman, R., Ozkaya, I.: A cost-benefit framework for making architectural decisions in a business context. In: ICSE'2010, pp. 149–157. IEEE (2010)
8. Criado, J., Iribarne, L., Padilla, N., Ayala, R.: Semantic matching of components at run-time in distributed environments. In: OTM 2015 Workshops, LNCS 9416, pp. 431–441. Springer International Publishing (2015)
9. Criado, J., Martínez, S., Iribarne, L., Cabot, J.: Enabling the reuse of stored model transformations through annotations. In: ICMT'2015, LNCS 9152, pp. 43–58. Springer International Publishing (2015)
10. Criado, J., Rodríguez-Gracia, D., Iribarne, L., Padilla, N.: Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces. Software: Practice and Experience. 45(12), 1677–1718 (2015)
11. Daniel, F., Matera, M.: Mashups – Concepts, Models and Architectures. Springer-Verlag Berlin Heidelberg (2014)
12. Insfran, E., Gonzalez-Huerta, J., Abrahão, S.: Design guidelines for the development of quality-driven model transformations. In MODELS'10, LNCS 6395, pp. 288–302. Springer Berlin Heidelberg (2010)
13. ISO/IEC. ISO/IEC 25000. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE (2014)
14. Loniewsli, G., Borde, E., Blouin, D., Insfran, E.: An Automated Approach for Architectural Model Transformations. In: Information System Development: Improving Enterprise Comm., pp. 295–306. Springer International Publishing (2014)
15. Martínez-Fernández, S., Ayala, C.P., Franch, X., Marques, H.M.: REARM: A Reuse-Based Economic Model for Software Reference Architectures. In: ICSR'2013, LNCS 7925, pp. 97–112. Springer Berlin Heidelberg (2013)
16. Ozkaya, I., Nord, R.L., Koziolek, H., Avgeriou, P.: Second Int. Workshop on Software. Architecture and Metrics. In: ICSE'2015, pp. 999–1000. IEEE Press (2015)
17. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 14:1–14:42 (2009)
18. Solberg, A., Oldevik, J., Aagedal, J.Ø.: A framework for qos-aware model transformation, using a pattern-based approach. In: CoopIS, DOA, and ODBASE, pp. 1190–1207. Springer Berlin Heidelberg (2004)
19. Stevanetic, S., Javed, M.A., Zdun, U.: Empirical evaluation of the understandability of architectural component diagrams. In: WICSA'2014 Companion, pp. 4:1–4:8. ACM New York (2014)
20. Weyns, D., Ahmad, T.: Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review. In: ECSA'2013, LNCS 7957, pp. 249–265. Springer Berlin Heidelberg (2013)
21. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Computing. 12(5), 44–52 (2008)
22. Zimmermann, O.: Metrics for architectural synthesis and evaluation: Requirements and compilation by viewpoint: An industrial experience report. In: SAM'2015, pp. 8–14. IEEE Press (2015)