Automated end user-centred adaptation of web components through automated description logic-based reasoning

David Lizcano [a,*], Fernando Alonso [b], Javier Soriano [b], Genoveva López [b]

[a] Open University of Madrid, UDIMA, School of Computer Science, Collado Villalba 28400, Madrid, Spain
[b] Universidad Politécnica de Madrid, School of Computer Engineering, Campus de Montegancedo, s/n, Boadilla del Monte 28660, Madrid, Spain

A B S T R A C T

Context: This paper addresses one of the major end-user development (EUD) challenges, namely, how to pack today's EUD support tools with composable elements. This would give end users better access to more components which they can use to build a solution tailored to their own needs. The success of later end-user software engineering (EUSE) activities largely depends on how many components each tool has and how adaptable components are to multiple problem domains.
Objective: A system for automatically adapting heterogeneous components to a common development environment would offer a sizeable saving of time and resources within the EUD support tool construc-tion process. This paper presents an automated adaptation system for transforming EUD components to a standard format.
Method: This system is based on the use of description logic. Based on a generic UML2 data model, this description logic is able to check whether an end-user component can be transformed to this modelling language through subsumption or as an instance of the UML2 model. Besides it automatically finds a con-sistent, non-ambiguous and finite set of XSLT mappings to automatically prepare data in order to leverage the component as part of a tool that conforms to the target UML2 component model.
Results: The proposed system has been successfully applied to components from four prominent EUD tools. These components were automatically converted to a standard format. In order to validate the pro-posed system, rich internet applications (RIA) used as an operational support system for operators at a large services company were developed using automatically adapted standard format components. These RIAs would be impossible to develop using each EUD tool separately.
Conclusion: The positive results of applying our system for automatically adapting components from cur-rent tool catalogues are indicative of the system's effectiveness. Use of this system could foster the growth of web EUD component catalogues, leveraging a vast ecosystem of user-centred SaaS to further current EUSE trends.

1. Introduction

Interest and investment in end-user development (EUD) are mounting all the time, and the impact of EUD [17] looks set to outstrip forecasts made by Christopher Scaffidi, Brad Myers and Mary Shaw back in 2005 [33], predicting that over 55 million end users (users without programming skills) would be developing their own applications by the end of 2013. There are many EUD tools for this purpose, such as spreadsheets, web-based mashup development environments, e-mail or database data filtering tools, which help millions of end users to personally develop software solutions to meet their own particular needs.

Within the web environment, many software suppliers, including Microsoft, Apple, IBM, Yahoo!, Oracle, etc., have developed their own support tools to help end users develop applications, particularly rich internet applications (RIAs) built from software components, offering DIY (do-it-yourself) guidance on evolving end-user developments to meet end-user demands and requirements [20], like [11] (now part of Chrome Productivity Tools), Yahoo! Pipes and Dapper [44,43], Microsoft Popfly [29], Open [16], JackBe [14], AMICO [2], Marmite [28] or EzWeb [7].

These solutions are based on the visual connection of components of different levels of abstraction within a graphical user interface (GUI). End users can use these components to access

* Corresponding author. Tel.: +34 918 56 16 99; fax: +34 918 56 16 97.
E-mail address: : david.lizcano@udima.es (D. Lizcano).

and leverage all sorts of services and resources to meet their needs. Although these tools have major weaknesses and do not always fit the EUD bill [20,17], they do offer a number of EUD functionalities that are useful for building end-user solutions. Apart from the visual development interface that they offer end users, their other key distinctive feature is the type of components that they each provide. The components that they each offer are useful for building web solutions with very limited functionality. For example, iGoogle is useful for creating a personal web portal, Yahoo! Pipes and Dapper are useful for mixing heterogeneous plain or rich text sources to create a new RSS feed, and Open Kapow is useful for creating a web portal built with screen scrapings from heterogeneous sites. The tool component catalogues limit the type of problems that they can address. Combining their catalogues would constitute a major qualitative leap in terms of the type and complexity of the solutions that could be produced. But to date each tool uses a different component model which rules out their joint use.

Then again, a new composite development model for EUD tools [19] is emerging. However, it has not been specified or structured yet. As far as we know, no detailed study has been conducted of tool components and success factors and generated products. The pressure to compete in an ever more globalized EUD solution ecosystem has driven developers (Google, Yahoo!, Microsoft, Amazon, Apple, Sun, IBM, etc.) to develop and optimize their own tools without specifying an underlying common component model. The result is that each tool offers its own component or component catalogue, and the components used in some tools are not usable in others.

Taking into account that EUD development is a visual component-based composition process as part of which data flows are created among components, two fundamental conclusions have emerged over the last few years in the field of web EUD. First, there must be a common web component model for all tools and, second, EUD success is closely related to the number and variety of web components available to users [20]. Now, most end users do not know how to program, and the only option that they have for developing a solution to their problem is to use components available in their work tool catalogue. But end users will not be able do this if their catalogue does not contain a component capable of performing the computation or accessing the data or operation/function that they require to solve the problem at hand. The above tools have actually been more or less successful depending on the wealth, variety and number of available components in their catalogues. We often find that one particular catalogue contains some components but not others, which are, on the other hand, available in other tools.

The challenge, then, is to come up with a system that is capable of adapting the components of these tools to a universal web component catalogue including parts of as many tools as possible and giving users access to a comprehensive component ecosystem [27]. As far as we know, there is no such system to date. On this ground, we have defined and specified a EUD web component model. Based on this model, we propose an automated system for translating web components existing in current tool catalogues and defined in XML to this model. This system is able to feed a catalogue that conforms to this model by automatically adapting heterogeneous components. This system is a new and original contribution to the EUD field.

The underlying component model was presented in [22]. The proposed component model is grounded on the success factors of other analysed EUD tools, as described in [23]. Our previous statistical studies with end users have demonstrated its effectiveness and usefulness [24]. We have also built an EUD environment that instantiates this component model, called EzWeb [8]. This environment was first reported in [20]. This tool was built as part of the Networked European Software and Service Initiative (NESSI) strategic research project. EzWeb is now being used as part of two European 7th Framework Programmes that we are partnering: as part of the Mashup-as-a-Service solution of 4CaaSt (building the Platform as-a-Service of the Future) [1], and as part of the applications and services ecosystem and delivery framework generic enablers for end users to build application mashups of FI-WARE (building the Future Internet Core Platform) [9]. Thanks to the experience that we have gathered over the last five years, part of which is accessible at the web site [26], we have found that a good end user-centred development environment and a good component model is insufficient for non-programmer users to efficiently build their own composite applications to meet particular requirements. In order to achieve this goal, it is necessary, on the one hand, to provide a wizard to translate requirements to interconnectable components, as reported in [25], and, on the other, to heavily populate a component catalogue. This research is part of this line, which we have not addressed so far. A system for automatically adapting heterogeneous components of several EUD tools would make it possible to create an enormous ecosystem of components that are compatible with each other and with a consistent EUD development environment.

This article presents a system capable of automatically translating and adapting heterogeneous components with XML templates, created by manufacturers like Yahoo!, Google or Open Kapow, to the common components model that we propose. Thus, components by any manufacturer can be imported to the common standard component catalogue, which component model compliant development tools, like EzWeb, can use. This is a key step towards standardizing web components for web application EUD.

The proposed automated adaptation system is a method of adapting EUD web components (respectively denoted by their manufacturers as pipes, operators, gadgets, widgets, etc.) with the potential for adapting external components to any EUD tool. Besides, the system is potentially applicable in other computing fields requiring the adaptation of an XML file to another schema or template, defined according to its XSD.

The research questions addressed in this paper are:

- *RQ1*: Is the proposed system for automatically adapting EUD components capable of adapting any XML EUD component to a standard format?
- *RQ2*: Are the automatically adapted components as efficient as the original components?
- *RQ3*: What does component adaptation time depend on?
- *RQ4*: How efficient is the automatic EUD adaptation system in terms of time and resources taken to adapt each component compared with manual component adaptation?

The remainder of the article is structured as follows. Section 2 details work related to this article. First, we explain what components are like today, what format they use and why heterogeneity is a handicap. Then we present earlier research concerned with automated reasoning on software component models and UML to clarify the elements that have been used in this paper and why none of the existing proposals is valid for achieving our aim. Section 3 details the designed automated adaptation model, its architecture, how it works, the description logic-based mathematical groundwork used to support the mapping and an example of the automated adaptation of a component to the standard format. Section 4 reports an analysis of the automated component adaptation system applied to different EUD tool components and responds to each of the stated research questions. Section 5 reports an experiment as part of a project partnered by one of the world's largest companies. This scenario was useful for examining the potential of the proposed system and specifying the real saving in end-user time and effort. Section 6 discusses the threats to validity of the

analysis and the experiment reported. Finally, Section 7 outlines some conclusions, highlighting the contributions of this research and how the technology can be transferred to other branches of knowledge. Additionally, it illustrates future research lines.

2. Related work

This section details work related to our proposal. The idea behind the proposal is to apply existing notions of automated reasoning on UML models researched in diverse fields of knowledge engineering to automatically adapt components to a target metamodel (and its component format). The adapted components would be added to a universal components catalogue for use by EUD tools that conform to the metamodel.

2.1. EUD components: format, heterogeneity and importance

EUD web tools offer end users the opportunity to develop composite web applications by visually composing components linked by data flows and events. For these tools to be successful, they need to cater for end user needs. For example, they need to match the level of abstraction at which end users devise and analyse solutions by breaking down problems into easier-to-solve parts, something which very few tools manage to do [32]. Development will be unsuccessful, however, if the tool is missing any component required to perform the target functionality at any stage of the development of the solution that they have devised. Therefore, a well-fed catalogue of components is a necessary, albeit not sufficient, condition for achieving success among end users.

All EUD tool manufacturers, including Yahoo!, Google, Microsoft, Apple, Open Kapow, Presto, have made it their business to build a host of components for their tools. They provide the best possible support (in the shape of public and documented APIs, component development environments, well-defined templates, etc.) to enable users with some programming knowledge to build their own components. However, providers have not yet negotiated a standard component format, metadata and wrappings required for component integration with tool APIs and runtime engines. All providers have adopted their own formats and component models. They make sure that their components are used for their own platform only (often because they have expended substantial resources on their construction) because it is the number of available components that will set their platform apart from others. The race to build a catalogue containing more components than competitor tools has precluded the search for solutions for integrating components of one platform into another. Although components are public and can be replicated, adapted or even used commercially under licence (as some portals like Programmable Web actually do), providers do not encourage such activities, nor do they publish enough documentation, IDEs or APIs for managing the components outside the context of the tool for which they were built. This does not really benefit end users, whose main concern is to gain access to the largest possible ecosystem of components for developing their solution, irrespective of who supplied each component.

Even so, all the tools use the XML language to define components, their operation and their visual appearance, integrated with tool-dependent HTML, JSP, SOAP, JavaScript, scripting PHP elements, etc. Additionally, they define an XSD/XML template for components that are valid for their tool. Throughout the article, we will give examples of tool components and their templates to illustrate these formats.

Some initiatives, such as UWA (Universal Widget API) led by Netvibes [30], or attempts by WAI (Web Accessibility Initiative), the Web API Working Group and WAF (Web Application Format) set up within W3C [41,40,39], aimed to create a common web component implementation format. They have all failed so far because manufacturers are reluctant to negotiate and agree on a common component model. Manufacturers are more concerned about expanding their own catalogue than the potential benefits of a common model and format for the EUD field.

As manufacturers are extremely unlikely to commit to a common specification of components and their formats, our proposal is to automatically map components to a common model that is accessible for existing tools. These components could be used as part of a common catalogue irrespective of their source. All tools that conform to the common component model, like EzWeb, could use this universal catalogue. In order to put together such a catalogue we have built an automated component transformation system, which extracts the XSD (equivalent to a UML2 model) from the component XML and then uses description logic to map the original UML to the target common model. There are similar proposals in other fields, which are reviewed and documented in Section 2.2.

2.2. Automated reasoning and mapping on UML models

There are many proposals and papers based on the application of description logic and inference systems for automated reasoning on data and component models. One of the earliest proposals in this field was reported in [5] and put forward a system of automated reasoning on UML based on description logics. The aim of the system was to provide computer-aided support during the application design phase in order to automatically detect relevant properties, such as inconsistencies and redundancies in UML diagrams. This proposal maps the UML model to basic DLR-type description logic. Although unable to infer changes to such models, it is capable of reasoning and discovering if any data model includes any omissible loops, interdependencies or relationships. This reasoning system was applied to a great many data models during software maturity improvement processes at Telecom Italia in order to support company data warehousing. The proposal has been further evolved to increase its potential, as described in [4].

Automated reasoning concepts were applied to structurally complex datasets and components for the first time in [36]. They introduced data metamodelling to automate data changes in order to homogenize the data structure. In this case, the data all came from the same problem domains, but it was the first time this type of reasoning was used to make changes to data schemas and syntax, although the proposal was confined to changes to their physical format in a database. Haarslev and Möller were the first to document the benefits of a DL Reasoner (RACER) as a tool for this type of automated reasoning [37].

Sattler [35] suggested how to describe terminological knowledge based on description logic in order to transform entity-relationship models built in the software engineering design phases to refined models based on design patterns. This was later used in several domains for mapping general-purpose UML models to other models with different features [15,38]. In both cases, UML models and metamodels were transformed as part of model-driven software engineering (model-driven approaches or MDA). Transformations like these were also applied in [10], where the MOFLON tool was used to propose, by means of inference rules, changes to the software models for developing goal-driven applications. These proposals are the seed for the automated adaptation proposed in this research, although none are directly applicable to the problem, as EUD components and their XML templates are unlike the basic UML models on which the above proposals operate.

With the Internet and XML boom, a host of proposals emerged, such as [6], which paved the way for applying all possible automated transformations and checks on UML enabled by description logic to data sources modelled by their XSD/XML schema.

Most description logic applications in software engineering have targeted the UML-based analysis and design phases. However, very little research has focused on expediting or supporting software development as such. There are, however, several model reasoning and mapping systems applicable to the automated adaptation of heterogeneous data sources, which could be suited for component and component template adaptation. One example is [42]. They proposed a DLR_{DM} description logic for automated reasoning and mapping of metadata accompanying the data mining-based KDD process, whose expressive richness is very similar to the logic proposed in this paper. To do this, they used a reasoning model based on AR-DMM (Automatic Reasoning in Data Mining Modeling) to automate data pre-processing and check data consistency. Noteworthy in the field of heterogeneous data source adaptation is research by Vavliakis et al. [18] proposing a framework for unifying and transforming heterogeneous data sources for query, processing and reasoning in a relational database. They proposed a description logic for transforming relational databases into ontologies for semantic reasoning, which, however, is unable to homogenize the data syntax. Hence it is not applicable to the adaptation of software components. Another noteworthy proposal is [31], proposing plausibility description logic (DLP) for handling information sources with heterogeneous data representation formats. They proposed a system that generates a common data model from different data sources, all of which it subsumes. This is useful for semantic web modelling, but is not valid if the generated final model is to be used to perform automated tasks, like adapting syntax to a known API, as its syntax and structure is completely unknown.

The use of description logics for mapping data sources in order to homogenize their structure and syntax is the key to understanding our proposal, which aims to leverage the success of such proposals in the field of knowledge engineering in order to adapt software components to support EUSE.

As far as we know, there are no proposals exploiting the proven advantages and applications of description logic for mapping UML or XSD models to other compatible models. These transformations are able to map heterogeneous software components from different domains and adapt them to a predefined universal schema. This shift can be used to adapt a software component to more than one end-user software development environment. This is the aim of the research reported here.

3. System for automated EUD component adaptation based on the use of description logic

This article reports an automated adaptation system (AAS) which we built from scratch. The AAS uses description logic to adapt heterogeneous web components to a known template. Based on a generic UML2 data model that we propose in this paper, it is able to check whether an XML EUD component is a non-ambiguous instance consistent with the above model and automatically finds a finite set of XSLT mappings to adapt the original component to the target model. This section describes our contributions to this problem domain: the underlying generic component model, the architecture on which our automated component adaptation proposal is based, the mathematical description logic-based specification of the generic UML2 component model. It concludes with an example illustrating the automatic adaptation of a component to standard format.

3.1. Component model

The first step in the application of the proposed automated adaptation system is to define a EUD universal component conceptual model in UML2. This model specifies what a EUD component is generally like. This model was published in [25] but, for clarity's sake, is briefly described here. The proposed model (Fig. 1) subsumes existing EUD tools and is able to describe any end-user web component.

We employ a UML2 class diagram, following the UML2 superstructure specification defined in ISO/IEC DIS 19505-2, in order to specify the component model. To complete this diagram, we use MOF (meta-object facility) [OMG, 06]. MOF is a facility defined and used in ISO/IEC 19502:2005. ISO/IEC 19502:2005 describes its importance and applicability in model-driven engineering, enabling the creation of a strict level-3 meta-modelling schema [34], and offering the possibility of running or checking schema instances or subsumptions in UML notation (descending to modelling level 2).

As shown in Fig. 1, the design element is the basic component of the component model. This element is composed of a user-centred visual interface for accessing a wrapped resource. Any component will be linked in the final solution with other components through pre- and post-conditions based on facts that guide the dataflow, where a fact is an information item composed of a datum and its associated lightweight semantics. The development environment suggests components and compositions to users at design time based on their current dataflow and lightweight semantic annotations by other users. The most abstract elements will be the full EUD solutions, previously generated by another end user; this solution is composed of a mashup of several design elements, and has several workspaces. Workspaces are visual spaces all displayed at the same time by a composite interface that aims to tackle part of the problem. These workspaces include several interconnected gadgets, where a gadget is a visual element that manages user interaction with a particular remote resource. This gadget may render a single view or a screen flow (such as a survey composed of several forms) for the user to interact with the remote resource or resources associated with the gadget. Each of these visual interaction items is termed resource representation. A resource representation is composed of the view and the back-end resource. The back-end resource is composed of operators and service wrappings.

The end-user components will be published in a business marketplace-style collaborative and federated catalogue. It is this catalogue that the proposed AAS should automatically populate. Any user will be able to search the catalogue for new components and compose solutions sourced from other user recommendations about the data managed by the partially designed solution, etc.

So, our goal is to offer a system capable of assembling components from multiple catalogues to form a larger and stratified component catalogue in order to help end users find the component that best fits their problem. On this ground, there is a full-blown hierarchy of design elements devised to fit the level of abstraction required by users for different development process workflows. These levels of abstraction include anything from full solutions to back-end resources (simple data operators, like filters, concatenators, etc., or wrapped services). Each element in this hierarchy is adapted to a different level of abstraction, so that users can find and focus on the detail level they would like to find and with which they feel confident: the full solution (or RIA) fits the systemic view that the user envisages for tackling the problem.

The proposed model also enables end users to establish a dataflow among visual elements where a new data item in one component leads all the collaborative interfaces to take a computational step. This is a spreadsheet-like approach, save that each element displays a richer visual interface and invokes particular remote services, resources or distributed data as wrapped services.

Service wrappings are the atomic design elements of our component model; they are the smallest pieces that an end user can

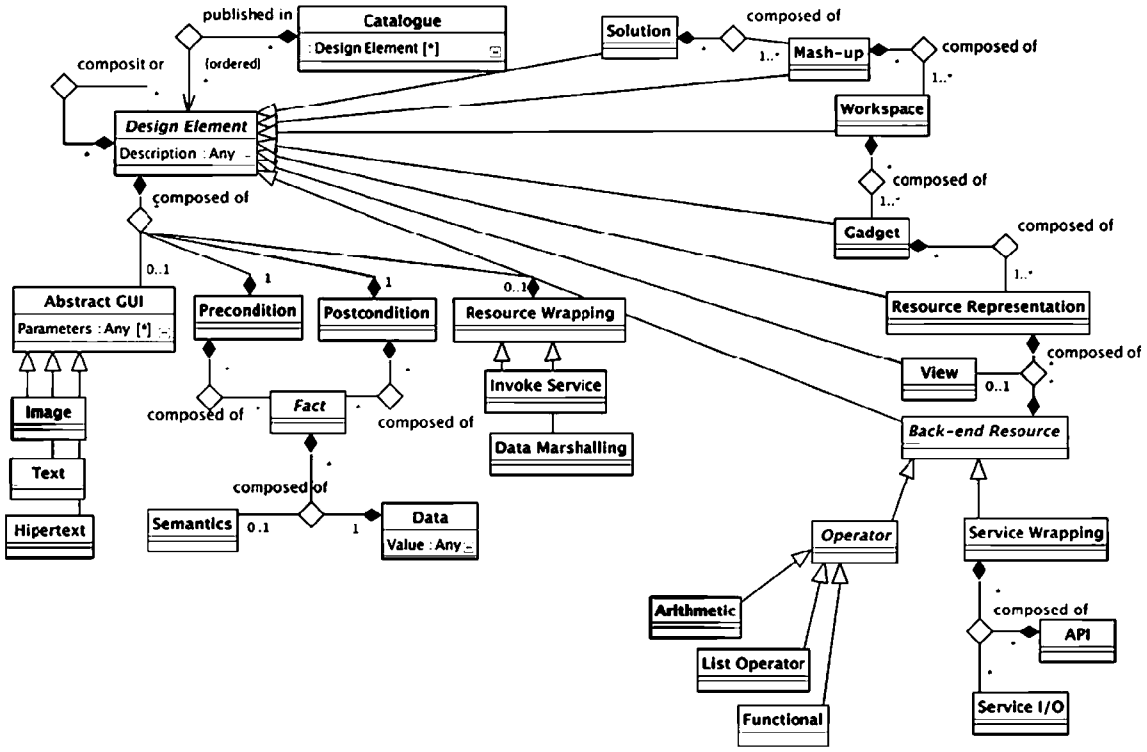

Fig. 1. UML EUD component model.

handle and understand. These elements, composed of an API and some inputs and outputs, are especially abundant on the Internet as part of web services ecosystems, as web services are really easy to transform into wrapped service components. It is these web services that implement the business logic of each component. Because the component is really a more or less complex wrapping that ultimately invokes a remote web service and the back-end is not really executed on the component EUD platform, a tool component can be translated to the syntax of another tool wrapping component without the executable source code or alike having to be transformed. Ultimately, most components serve to wrap SOAP messages, and it is their templates that do the wrapping and generate data flows. Converting one tool component into a valid component for another tool is just a matter of substituting one template for another.

3.2. Proposed architecture for automating component adaptation

The next step is to design an architecture for automatically transforming the components of any manufacturer to a known format with which the catalogue and EUD runtime platforms can operate. The framework architecture that we created is based on two existing external tools, HermiT and MOFLON. The architecture uses the component model that we proposed in Section 3.1, encoded in the description logic that we devised, which is described in Section 3.3. Fig. 2 illustrates the architecture supporting the AAS.

Fig. 2 shows that the proposed UML2 model (a) is mapped to SROIQ description logic (b) by means of a series of mappings that are described in Section 3.3. We use a tool called MOFLON to automatically build a rule box called ABox, "Assertion Component" (c) from the output description logic. The description logic is also used to build a terms box, called TBox, "Terminological Component" (d), which contains a description of the terms used (*solution*, *mashup*, *gadget*, *precondition*, *postcondition*, etc.). These two components are offered as inputs for HermiT (e), an existing reasoning tool, which has another two inputs: a particular EUD component, formatted in XML (f) and its respective XSD schema (g). HermiT outputs two Boolean values:*Subsumption* and *Instance*, and a new ABox, ABox'. *Subsumption* indicates whether the input component

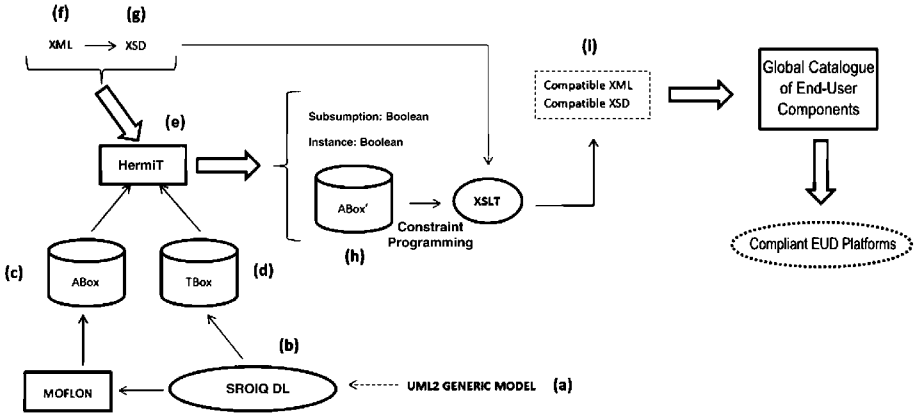

Fig. 2. Proposed system architecture.

model is a subsumption of the generic model; *instance* indicates whether the component syntax is an instance of the generic model. The ABox' component, annotated with constraints, is a new constraint-based rule box that can be translated to a series of XSLT mappings using constraint programming (h). As they are applied to the initial XML data of the component and its XSD schema, the XSLT mappings transform the component into another equivalent component structured to conform to the proposed generic UML2 model (i). This transformed component and its automatically generated schema can be stored in the universal catalogue of end-user components and is ready for use by any tool that conforms to the component model, like EzWeb. Note that the step from (f) to (g) is instantaneous thanks to the many open source tools that can infer an XSD from a component's XML data.

Sections 3.3 and 3.4 explain how to map our generic UML2 model to SROIQ description logic, and we present the potential automated reasoning services offered by the automated generation of the ABox' component (using the MOFLON tool and HermiT reasoner) composed of a finite set of XSLT mappings that are useful for converting the EUD tool components into other equivalent components that can used on a universal target EUD platform.

3.3. Mathematical formulation of the UML2 model of EUD components through description logic

As discussed above, the HermiT tool, used in conjunction with MOFLON, is able to validate the UML models, and their respective XSD schema, used by different component providers [12]. This validation is useful for checking whether the components of one provider would be valid for the proposed component model and is also able identify the XSLT mappings necessary for adapting such components to a catalogue and runtime platform that conforms to the above model. HermiT works with mathematical descriptions based on description logic, which it uses to operate on the UML2 metamodel, performing subsumption and instance checking operations on new components and their schema [4]. The kernel theory of our research is the mathematical formulation presented in this section, which we built from scratch in order to transform EUD components from a source to a target format. Table 1 lists the concept and role constructors used to develop the proposed mapping in order to define a sufficiently expressive type of description logic. Additionally, it indicates the computational complexity associated with the two fundamental operations targeted by mathematical reasoning: subsumption ($\models C \sqsubseteq D$) and instance checking ($\models C(i)$). As many model mappings as necessary will be implemented until the UML model converges on a valid instance or subsumption, identifying breakpoints detected for isolation during the component adaptation process.

Table 1 shows that, according to the naming scheme defined by Baader et al. [3], we use SROIQ description logic, an extension of the description logic underlying OWL-DL, SHOIN, with a number of expressive resources that are useful for our purpose [13]. SROIQ uses expressive resources that were suggested by ontology developers as useful additions to OWL-DL, and which, additionally, do not affect DL decidability and practicability. SROIQ uses complex role inclusion axioms of the form $R \circ S \subseteq R$ or $S \circ R \subseteq R$ to express propagation of one property along another one, which has proven useful in UML terminologies. Furthermore, it includes reflexive, antisymmetric, and irreflexive roles, disjoint roles, a universal role, and $\exists R.Self$ constructs, which are useful for defining concepts such as component bindings to input/output messages. Finally, it uses negated role assertions in ABoxes and qualified number restrictions.

SROIQ is a trade-off between the expressiveness of the language used to construct the terminological information and the complexity associated with the reasoning processes on both terminological and assertive model information [3,4]. In this respect, the trade-off for using other types of logic, such as the family of description logics derived from DLR logic that remove the binary role constraint and introduce n-ary role constructors, which would have enabled a more straightforward mapping than proposed in this paper, would be a much greater computational cost. Note that tools like HermiT have been unable to classify a UML2 model expressed in DLRreg, that is, the DLR logic extension with union, composition and transitive closure of binary role constructors as a projection of n-ary roles on two of its components.

Apart from the traditional conceptual subsumption ($C \sqsubseteq D$) and equivalence ($C \equiv D$) axioms, constrained in the sense that only D can be an expression of concept (and therefore C must be an atomic concept), the subsumption axiom has also been used in order to create role hierarchies in relationships among web components,

Table 1
Constructors of concepts and roles used in the proposed mapping.

Constructor	Syntax	Semantics	Logic type	Compl. $\models C \sqsubseteq D$	Compl.$\models C(i)$
Atomic concept	A	$A^{\mathcal{J}} \subseteq \Delta^{\mathcal{J}}$	$\mathcal{FL}_0$	P	P
Domain	T	$\Delta^{\mathcal{J}}$			
Empty	$\bot$	$\emptyset$			
Conjunction	$C \sqcap D$	$C^{\mathcal{J}} \bigcap D^{\mathcal{J}}$			
Universal	$\forall R.C$	$\{x \mid \forall y\colon R^{\mathcal{J}}(x,y) \rightarrow C^{\mathcal{J}}(y)\}$			
Existential	$\exists R.T$	$\{x \mid \exists y\colon R^{\mathcal{J}}(x,y)\}$	$\mathcal{FL}^-$	P	P
Atomic negation	$\neg A$	$\Delta^{\mathcal{J}} \setminus A^{\mathcal{J}}$	$\mathcal{AL}$	P	P
Qualified existential	$\exists R.C$	$\{x \mid \exists y\colon R^{\mathcal{J}}(x,y) \wedge C^{\mathcal{J}}(y)\}$	$\mathcal{E}$	NP	PSPACE
Negation	$\neg C$	$\Delta^{\mathcal{J}} \setminus C^{\mathcal{J}}$	$\mathcal{C}$	PSPACE	PSPACE
Enumeration	$a_1 \ldots a_n$	$a_1^{\mathcal{J}} \ldots a_n^{\mathcal{J}}$	$\mathcal{C}$	PSPACE	PSPACE
Disjunction	$C \cup D$	$C^{\mathcal{J}} \cup D^{\mathcal{J}}$	$\mathcal{U}$		
Cardinality	$\geqslant nR$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y)\} \geqslant n\}$	$\mathcal{N}$		
	$\leqslant nR$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y)\} \leqslant n\}$			
	$=nR$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y)\} = n\}$			
Qualified cardinality	$\geqslant nR.C$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y) \wedge C^{\mathcal{J}}(y)\} \geqslant n\}$	$\mathcal{Q}$	EXP	EXP
	$\leqslant nR.C$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y) \wedge C^{\mathcal{J}}(y)\} \leqslant n\}$			
	$=nR.C$	$\{x \mid \#\{y \mid R^{\mathcal{J}}(x,y) \wedge C^{\mathcal{J}}(y)\} = n\}$			
Selection	f: C	$\{x \in Dom(f^{\mathcal{J}}) \mid C^{\mathcal{J}}(f^{\mathcal{J}}(x))\}$	R_F		
Transit. roles	R^+	$\bigcup_{n \geqslant 1} (R^{\mathcal{J}})^n$	$()^+$		
Inverse roles	R^-	$\{(y,x) \in \Delta^{\mathcal{J}} \times \Delta^{\mathcal{J}} \mid R^{\mathcal{J}}(a,b)\}$	$()_{R^-}$		
Role composition	$R \circ S$	$R^{\mathcal{J}} \circ S^{\mathcal{J}} = \{(x,z) \mid \exists y \in \Delta^{\mathcal{J}}$ $R^{\mathcal{J}}(x,y) \wedge S^{\mathcal{J}}(y,z)\}$	$()_{R^\circ}$		

hence subindex H. The aim of the construction- and axiom-level SROIQ mapping is to specify how this description logic captures the semantics of mapped design elements.

The formal semantics of the UML2 model mapped to SROIQ is based on a model theory where $I = \langle D, \cdot I \rangle$ is an interpretation, and:

- $D \neq \emptyset$ represents a domain or universe of discourse such that $D = \sum \cup \Upsilon$ with $\Upsilon = \bigcup_{i=1}^{n} \Upsilon_{Di}$, $\Upsilon_{Di} \bigcap \Upsilon_{Dj} = \emptyset$ and $\sum \bigcap \Upsilon = \emptyset$, and $\sum$ is the domain of managed components and Υ_{Di} is the set of values associated with each basic data type D_i supported by UML2 (highlighting the free type *any*, although other types, such as integer, string, etc. can be specified)
- $\cdot^I$ is the projected interpretation function:
  - $D_i^I = \Upsilon_{Di}$.
  - $C_i^I \subseteq \sum$.
  - $A_i^I \subseteq \sum x \Upsilon$.
  - $R_i^I \subseteq \sum x \ldots x \sum \equiv \sum^n$.

To illustrate the proposed mapping, Table 2 describes the process of translating the UML2 component model to the proposed description logic.

For a more detailed description of the proposed mapping, see [21], which confirms that all the EUD web component tools conform to the specified EUD component model. In this research the mapping is the starting point for automatically adapting multiplatform components.

3.4. XSLT mappings on heterogeneous components for adaptation to the UML2 model

The UML2-SROIQ mapping is a semantic specification of our UML2 component modelling conceptualizations for running automated reasoning services. Such services include checking that a component XSD schema or template is an instance or subsumption of the proposed universal EUD component model or deciding which linear XSLT mapping should be applied to the XSD template to match its syntax to the proposed model. The knowledge base semantics is equivalent to a set of first-order predicate logic axioms. Like any other set of axioms, it contains implicit knowledge that can be specified through logical inference. The fundamental inference service is consistency checking for assertion knowledge bases (ABox). This service discovers the mapping rules that should be applied recursively to originate that consistency (ABox'). The mapping of the UML2 model to a terminological knowledge base (TBox) leads to the construction of a terminology T. As we build the UML2 model, we have to check whether each new class makes sense. If it contradicts the remainder of the model, it will never be able to be instantiated consistently. From the logical viewpoint, a new concept C makes sense if there is at least one interpretation I that satisfies the axioms of T and for which the concept denotes a non-empty set. This interpretation is called model and is written $T \vDash C$. This property C with respect to T is called satisfiability. The reasoning services offered by the description logic subsystem of any modelling tool (e.g., HermiT) can be applied to the UML2 model designed in description logic to verify that an XML component conforms to the expected syntax of the proposed EUD component model or transform this component to match the syntax. All the reasoning services are based on the following prototype services:

- **Satisfiability:** A concept C is satisfiable with respect to a terminology T if there is a model I of T such that $C^I \neq \emptyset$. It is written $T \vDash C$. The satisfiability of T is expressed as $T\vDash$.
- **Subsumption:** A concept C is subsumed by a concept D with respect to T if $C^I \subseteq D^I$ for any model I of T. It is written $T \vDash C \sqsubseteq D$. Subsumption can be expressed in terms of satisfiability as $T \vDash C \sqsubseteq D \Leftrightarrow T \nvDash C \sqcap \neg D$. Similarly, satisfiability can be expressed in terms of subsumption as $T \nvDash C \Leftrightarrow T \vDash C \sqsubseteq \bot$.
- **Equivalence:** A concept C is equivalent to a concept D with respect to T if $C^I = D^I$ for any model I of T. It is written $T \vDash C$ *equiv* D. Equivalence can be expressed in terms of satisfiability as $T \vDash C \equiv D \Leftrightarrow T \nvDash C \sqcap \neg D$ and $T \nvDash \neg C \sqcap D$ and in terms of subsumption as $T \vDash C \equiv D \Leftrightarrow T \vDash C \sqsubseteq D$ and $T \vDash D \sqsubseteq D$.

Table 2
UML2 to SROIQ Mapping.

UML2 Element	New concepts and roles	New DL axioms
Class C	Concept C	
Attribute a of C with type T	Binary role a	$C \sqsubseteq \langle =1a \rangle \sqcap \exists$ a.T
Attribute a of C acting as keyword	Binary role a	$C \sqsubseteq \langle =1A \rangle \sqcap \exists$ A.D $\sqcap \langle \leqslant 1A^- \rangle$
Attribute a of C with type T[]	Binary role a	$C \sqsubseteq \langle \geqslant 1a \rangle \sqcap \forall$a.T
Attribute a with associated card. (ni..nj)	Binary role a	$C \sqsubseteq \langle \geqslant n_i a \rangle \sqcap \langle \leqslant n_j a \rangle \sqcap \forall$a.T
Dependency A	Binary role A roles R1 and R2	$T \sqsubseteq \forall A.C_2 \sqcap \forall A^-.C_1$ $C_1 \sqsubseteq \forall A.C_2 \sqcap [\geqslant n_i A] \sqcap [\leqslant n_j A]$ $C_2 \sqsubseteq \forall A^-.C_1 \sqcap [\geqslant m_i A^-] \sqcap [\leqslant m_j A^-]$ $A \sqsubseteq R_1, R_1 \sqsubseteq A, A^- \sqsubseteq R_2, R_2 \sqsubseteq A^-$
n-Ary association with multiplicity	Concept A roles A_r, R1…Rn	$A \sqsubseteq \exists R_1.C_1 \sqcap \ldots \sqcap \exists R_n.C_n \sqcap$ $\langle \leqslant 1R_1 \rangle \sqcap \ldots \sqcap \langle \leqslant 1R_n \rangle$ $C_i \sqsubseteq \forall R_i^-.A \sqcap \langle \geqslant n_i R_i^- \rangle \sqcap \langle \leqslant n_j R_i^- \rangle$ i = 1,…,n
Binary association with multiplicity	Concept A roles A_r, R1…Rn	$A \sqsubseteq \exists R_1.C_1 \sqcap \exists R_2.C_2$ $\sqcap \langle \leqslant 1R_1 \rangle \sqcap \langle \leqslant 1R_2 \rangle$ $C_1 \sqsubseteq \forall R_1^-.A \sqcap \langle \geqslant m_i R_1^- \rangle \sqcap \langle \leqslant m_j R_1^- \rangle$ $C_2 \sqsubseteq \forall R_2^-.A \sqcap \langle \geqslant n_i R_2^- \rangle \sqcap \langle \leqslant n_j R_2^- \rangle$ $A_r \equiv R_1^- \circ R_2$ $C_1 \sqsubseteq \forall A_r.C_2 \sqcap \langle \geqslant m_i A_r \rangle \sqcap \langle \leqslant m_j A_r \rangle$ $C_2 \sqsubseteq \forall A_r^-.C_2 \sqcap \langle \geqslant m_i A_r^- \rangle \sqcap \langle \leqslant m_j A_r \rangle$
Non-disjoint partial inheritance relation		$C_i \sqsubseteq C$, i = 1,…,n
Disjoint partial inheritance relation		$C_i \sqsubseteq C$, i = 1,…,n $C_i \sqsubseteq \neg C$, for all $i \neq j$
Non-disjoint total inheritance relation		$C_i \sqsubseteq C$, i = 1,…,n $C \sqsubseteq \bigcup_{i=1}^{n} C_i$
Disjoint total inheritance relation		$C_i \sqsubseteq C$, i = 1,…,n $C_i \sqsubseteq \neg C$, for all $i \neq j$ $C \sqsubseteq \bigcup_{i=1}^{n} C_i$

- **Disjointness:** Two concepts C and D are disjoint with respect to T if $C^I \cap D^I = \emptyset$ for any model I of T. Disjointness can be expressed in terms of satisfiability as $T \not\models C \sqcap D$ and in terms of subsumption as $T \models C \sqcap D \sqsubseteq \bot$.

There follows a representative snippet of our UML2-SROIQ mapping. The UML2-SROIQ mapping is the core of the proposed system, forms the reasoning TBox and is the input used by the MOFLON tool to generate the ABox which is used in the HermiT tool:

DesignElement⊑Class⊓∃Any.Description⊓⟨⩽1Description⟩⊓
∀RoleComposition.DesignElement⊓
∀group_Composition-.DesignElement⊓
∀part_Composition-.DesignElement⊓
∀publishedin_Part-.publishedin⊓
∀publishedinRole-.Catalogue⊓
∀composedof_Group-.composedof⊓
∃composedofRole.AbstractGUIDE⊓
⟨⩽1composedofRole.AbstractGUIDE⟩⊓
∃composedofRole.ResourceWrapper⊓
⟨⩽1composedofRole.ResourceWrapper⟩⊓
∃composedofRole.Precondition⊓⟨=1composedofRole.Precondition⟩⊓
∃composedofRole.Postcondition⊓⟨=1composedofRole.Postcondition⟩

composition⊑Aggregation⊓∃group_Composition.DesignElement⊓
∃part_Composition.DesignElement⊓
⟨⩽1group_Composition⟩⊓⟨⩽1part_Composition⟩
roleComposition≡group_Composition-∘part_Composition

publishedin⊑Aggregation⊓∃publishedin_Group.Catalogue⊓
⟨⩽1publishedin_Group⟩⊓
⟨⩽1publishedin_Part⟩⊓
∃publishedin_Part.DesignElement
publishedinRole≡publishedin_Group-∘publishedin_Part

composedof⊑Aggregation⊓∃composedof_Group.DesignElement⊓
⟨⩽1composedof_Group⟩⊓⟨⩽1composedof_Part⟩⊓
∃composedof_Part.Precondition⊓⟨ =1composedof_Part.Precondition⟩⊓
∃composedof_Part.Postcondition⊓⟨ =1composedof_Part.Postcondition⟩⊓
∃composedof_Part.AbstractGUIDE⊓⟨⩽1composedof_Part.AbstractGUIDE⟩⊓
∃composedof_Part.ResourceWrapper⊓
⟨⩽1composedof_Part.ResourceWrapper⟩

composedofRole≡composedof_Group-∘composedof_Part
Catalogue⊑Class⊓∃ ⟨DesignElement1,DesignElementn⟩⊓
∀publishedinRole.DesignElement ⊓
∀publishedin_Group-.publishedin
[...]

Thanks to this mapping and to the reasoning services it is possible to adapt a component described in XML, find the XSD template or schema used by its source EUD tool, check whether the schema is an instance of the general XSD schema of the proposed EUD component model, and find a set of mappings to translate the component using a generic and uniform syntax for publishing and using the component in the proposed EUD component model. ABox' contains constraints that can be mapped into XSL annotations using a supporting ontology and thesaurus of EUD components [21] that we developed to operate on the constraint-based programming framework of HermiT, as illustrated in the example given in Section 3.5.

3.5. Example of automated adaptation of a component to the standard format

We selected the Yahoo! Pipes component as an example to illustrate adaptation because it had an especially complex structure. Fig. 3 illustrates a source code snippet of a Yahoo! Pipes component designed to display a table with FlickR web service images based on a string entered as input data.

These data conform to an XML format shared by all EUD tools and obey a specialized Yahoo! Pipes syntax. Accordingly, special XML labels are used to write component inputs, calls to external web services and outputs, etc. Yahoo! Pipes components are generally represented by flat text, although they can be exported and automatically converted to XML files.

The XSD schema of the components of a particular EUD tool can be automatically identified from the component XML thanks to programs such as XSD Generator. XSD Generator instantiates the schema, also known as template, to which the components of this tool should conform. Providers sometimes publish these templates for use by programmers/users to craft their own components that conform to the syntax required by the tool. On other occasions, the provider may even omit the template altogether in order to protect

```
<?xml version="1.0" encoding="UTF-8" ?>
    // https="true" ensures that only HTTPs connections are allowed
    <table xmlns="http://query.yahooapis.com/v1/schema/table.xsd" https="true">
      <meta>
        <sampleQuery> select * from {table}</sampleQuery>
      </meta>
      <bindings>
        <select itemPath="rsp" produces="XML">
          <urls>
            <url>http://api.flickr.com/services/rest/</url>
          </urls>
          <inputs>
            <key id='method' type='xs:string' paramType='variable' const="true" default="flickr.auth.getFrob" />
            <key id='api_key' type='xs:string' paramType='variable' required="true" />
            <key id='secret' type='xs:string' paramType='variable' required="true" />
          </inputs>
          <execute><![CDATA[
                y.include("http://www.yqlblog.net/samples/flickr.js");
                // GET the flickr result using a signed url
               var fs = new flickrSigner(api_key,secret);
                response.object = y.rest(fs.createUrl({method:method, format:""})).get().response();]]>
          </execute>
        </select>
      </bindings>
    </table>
```

Fig. 3. Source code snippet of a Yahoo! Pipes component.

the intellectual property of their own components. Our system is geared up for this possibility, as it is able to deduce the XSD schema used as a template of a subset of tool components that conform to the schema. Fig. 4 shows the schema for Yahoo! Pipes components.

If the input for our automated components adaptation system were a Yahoo!Pipes component in XML format, with the respective component XSD schema, the system would perform a finite number of mappings to adapt the component to the proposed EUD component model with the valid syntax for compliant tools, such as EzWeb. As a result, it would be possible to publish the component in a universal catalogue, receive updates of the respective component published by the original supplier or use the EUD component on platforms other than Yahoo! Pipes. Fig. 5 illustrates a snippet of the XLST mappings required to adapt the source file of any Yahoo! Pipes component compliant with the schema illustrated in Fig. 4. The mappings are inferred by HermiT from the ABox generated for this component type.

4. Analysis of the automated component adaptation system applied to the different components of EUD tools

In order to analyse the proposed automated component adaptation system, we applied the system to heterogeneous components from four completely different EUD tools, namely, Yahoo! Pipes, Open Kapow, JackBe, and Marmite. We transformed the components

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xsd:annotation>
                <xsd:documentation xml:lang="en">Yahoo!Pipes Schema</xsd:documentation>
        </xsd:annotation>
        <xsd:element name="YahooPiping">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="mashup" type="Mashup" minOccurs="0"
maxOccurs="unbounded" />
                        </xsd:sequence>
                </xsd:complexType>
                        <xsd:unique name="mashupIdKey">
                                <xsd:selector xpath="mashupId" />
                                <xsd:filed xpath="mashupId" />
                        </xsd:unique>
                        ...
        </xsd:element>
        <xsd:complexType name="Mashup" stereotype="RIA">
                <xsd:sequence>
                        <xsd:element name="Pipe" type="xsd:pipe" />
                        <xsd:element name="date" type="xsd:date" />
                        <xsd:element name="AuthorName" type="xsd:string" />
                        <xsd:element name="AuthorRank" type="xsd:unsignedInt" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Pipe" stereotype="RIA">
                <xsd:sequence>
                        <xsd:element name="Module" type="xsd:module" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Module" stereotype="RIA">
                <xsd:sequence>
                        <xsd:element name="Layout" type="xsd:image" />
                        <xsd:element name="backend" type="WS*" minOccurs="0" maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Backend" stereotype="RIA">
                <xsd:sequence>
                        <xsd:element name="Operator" type="xsd:L-FOLD" />
                        <xsd:element name="RSS_ticker" type="feed_rss" url="xsd:url"
data="xsd:string" />
                </xsd:sequence>
        </xsd:complexType>
</xs:schema>
```

Fig. 4. XSD schema for Yahoo! Pipes components.

```
<?xml version="1.0" standalone="yes" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<KDD_Prepared_Data>
        <RIA>
                <Key> YahooPiping </Key>
                <Description>
                        <Mashup>
                                <Key> mashupIdKey </Key>
                                <Value> <xsl:value-of select="YahooPiping/Mashup/mashupIdKey"/> </
Value>
                                <Key> AuthorName </Key>
                                <Value> <xsl:value-of select="YahooPiping/Mashup/AuthorName"/> </Value>
                        </Mashup>
                        </WorkSpace>
                                <Key> Date </Key>
                                <Value> <xsl:value-of select="YahooPiping/Mashup/date"/> </Value>
                                <Key> AuthorRank </Key>
                                <Value> <xsl:value-of select="YahooPiping/Mashup/AuthorRank"/> </Value>
                        </WorkSpace>
                        ...
                        <Gadget>
                                <Key> Pipe </Key>
                                <Description>
                                        <Key> Description </Key>
                                        <Value> <xsl:value-of select="YahooPiping/Mashup/Pipe/
Description"/> </Value>
                                </Description>
                                <Key> Module </Key>
                                <Description>
                                        <Key> Layout </Key>
                                        <Value> <xsl:value-of select="YahooPiping/Mashup/Pipe/Module/
Layout"/> </Value>
                                        <Image>
                                                <Key> Layout </Key>
                                                <Type>
                                                        <Key> Image </Key>
                                                        <Value> <xsl:value-of select="YahooPiping/
Mashup/Pipe/Module/Layout/src"/> </Value>
                                                </Type>
                                        </Image>
                                        <Key> BackEnd</Stab-TS>
                                        <Operator>
                                                <Key> List_Operator </Key>
                                                <Operator> <xsl:value-of select="YahooPiping/Mashup/
Pipe/Module/Operator/L-FOLD"/> </Operator>
                                        </Operator>
                                        <Wrapped_Service>
                                                <mode> REST architecture </mode>
                                                <url> <xsl:value-of select="YahooPiping/Mashup/Pipe/
Module/BackEnd/RSS_ticker/url"/> </url>
                                                <Data> <xsl:value-of select="YahooPiping/Mashup/Pipe/
Module/BackEnd/RSS_ticker/data"/> </data>
                                                                                        ...
                                        </Wrapped_Service>
                                </Description>
                        </Gadget>
                </Description>
        </RIA>
</Generic_EUD_Prepared_Component>
</xsl:template>
</xsl:stylesheet>
```

Fig. 5. XSLT mapping schema generated automatically by the automated component adaptation system for Yahoo! Pipes.

of the above tools and checked that the structure transformed to the proposed standard model operated in exactly the same way as the original structure in their native environment. The component templates of the EUD tools are completely different, their XMLs adhere to heterogeneous rules, as well as syntax and complex structures that contain single-valued data and complex operating functions with lists and operators. Therefore, it would suffice to test the effectiveness of the proposed system. It would require a huge workload, and programming knowledge, to adapt these components to a specific EUD tool, as each particular component would have to be individually crafted to adapt their source features to the target platform conditions.

Yahoo! Pipes, Open Kapow, JackBe and Marmite are very successful tools today and all have a similar architecture. The RIAs that they are used to create have one or more mashups of visual elements of different types. Each mashup is further composed of one or more dashboards, whose mission is to offer users different runtime workspaces in the shape of separate sets of widgets. Each dashboard is composed of one or more interlinkable widgets, although widgets from different dashboards cannot be interconnected. A widget is the basic visual element of which the final solutions built using Open Kapow will be composed, and no knowledge of web technologies is required for management. Each widget is implemented by one or more robots cooperating with each other, and the management, operation and configuration of such robots does require some web components programming knowledge. An Open Kapow robot is a functional part that performs a specific computational task. It is composed of an interface (which may be hidden and not be displayed at runtime) and by back-end resources that implement the workflows of the final solution without the user perceiving all the implementation details. These resources can perform arithmetic, functional or list handling operations on data or invoke remote web services through their specialized API. A robot can only invoke a service if it has been implemented to include specific details about how to manage the service API (the syntax and technology used) and how to manage sent and received data. Therefore, a user will never be able to modify the robot to manage any new web services required. Our

automated adaptation system generates XSLT mappings from these components. These XSLT transformations convert dashboards into workspaces, widgets into gadgets and robots into wrapped services invoked using REST or SOAP syntax depending on whether the components are from Open Kapow, JackBe (which use REST or POX-RPC calls with URLs and string data types) or Marmite (which uses a SOAP envelope syntax and integer or string data types).

The first research question to be considered is:

RQ1: Is the proposed system for automatically adapting EUD components capable of adapting any XML EUD component to a standard format?

Table 3 summarizes the adaptations of the components of the four analysed EUD tools. A total of 740 components were automatically adapted, using approximately 185 components from each analysed EUD tool. A specialized XSLT mapping schema had to be built for each of the four EUD tools. We converted different component types for each tool in order to analyse whether component class had any impact on the results of the automated adaptation process.

After automatically adapting the tool components, we checked whether the resulting components were valid and operated properly on EzWeb, which is different from the original tool. White-box and black-box tests were conducted on the adapted components within the EzWeb platform, and integration tests based on input and output data were run to test any components that had been successfully integrated into complex RIAs [26]. The system correctly adapted 730 out of the 740 components, failing to generate XML files adapted to the generic schema for 10 components. We considered an adaptation to be satisfactory if, after white- and black-box testing with the component in its source environment and after adaptation to EzWeb, we found that: (a) the component conformed to the necessary template for execution in EzWeb; (b) the component correctly generated the target outputs in response to input data using random test values; (c) the structure and internal architecture was the same (which was tested automatically using the ALTOVA UModel). The error rate for EUD tool component adaptation ranged from 0.52% to 2.15%. The component adaptation success rate was around 98%, which is rather good for an automated end-user software component adaptation method.

We analysed the components that were not successfully adapted for use in EzWeb and found that at the root of the problem were missing values within the XML source files for certain fields of the component provider templates, causing errors in their three structures. The component source code contained XML errors with respect to the domain XSD schema data. A possible future line of research would be to build mechanisms capable of detecting and processing such missing values into our automated adaptation system in order to adapt the component source code to the target DOM structure (Document Object Model, an API that provides a standard set of objects for representing HTML documents and XML), even if the XML file does not include all the leaves or branches.

The second research question to be addressed is:

RQ2: Are the automatically adapted components as efficient as the original components?

We randomly selected a significant sample of 40 out of the 730 correctly adapted components, 10 for each of the different EUD source tools, and checked that the adapted components run on EzWeb were equally efficient as their respective source components run on the tool for which they were developed. Table 4 shows the selected tool components, the number of machine code instructions (in millions of machine instructions, mi) for the component running on the original tool and on EzWeb, and execution time (in milliseconds, ms) of the original and adapted components. Execution times exclude the time difference between client and server request and response for each EUD tool.

We found that the number of instructions that the server will have to execute using the EUD tool is more or less equal before and after mapping, and again there are no major differences with respect to execution time. Besides, the small time variations are due to the HTTP/XML and/or SOAP transfer of networked requests among servers, which are unrelated to the actual component and depend on network traffic, server overload, etc.

Furthermore, we published and used the automatically adapted components with the EzWeb tool to check that the components work properly. The EzWeb tool used has been reported elsewhere [20,22,23]. Whereas earlier studies examined components created ad hoc for EzWeb, this research looks at the automated adaptation of external components for use with this tool without the need for expert intervention.

The third research question to be answered is:

RQ3: What does component adaptation time depend on?

Table 5 shows how long it took the system to extract and apply the XSLT file for each of the EUD tools. It specifies the mean size in bytes for each component type, the mean number of lines of XML in the component source files, the number of lines of XSD in the schema for the original EUD tool, the time taken to extract the XSLT for the original EUD tool (an operation run just once at the start of the preparation process), mean time to apply the XSLT to each specific tool and, finally, the total adaptation time for all components of each EUD tool, which covers both the extraction of the XSLT and its application to all the analysed components. As Table 5 shows, we found that the time taken to create and correctly apply the necessary XSLT mappings increases in proportion to component complexity (indicated by the component bytes, lines of XML and lines of XSD).

We analysed these results to ascertain whether there was any correlation between either the time taken in the different phases of the automated component adaptation system with either the component size in terms of bytes or lines of XML, or the number

Table 3
Results of automated EUD component adaptation system.

Source EUD tool	N samples	Poorly adapted components	Adaptation error rate (%)
Yahoo! Pipes	190	1	0.52
Open Kapow	180	2	1.11
JackBe	186	4	2.15
Marmite	184	3	1.63

Table 4
Automated component adaptation system performance.

Data domain	N samples	Machine code instructions for source tool	Machine code instructions for EzWeb	Mean execution time for standard data on source tool	Mean execution time for standard data on EzWeb
Yahoo! Pipes	10	15,421 mi	15,430 mi	1247.35 ms	1249.67 ms
Open Kapow	10	56,325 mi	56,312 mi	5813.12 ms	5624.12 ms
JackBe	10	39,599 mi	39,472 mi	4921.87 ms	5000.18 ms
Marmite	10	21,492 mi	21,530 mi	3031.69 ms	3030.78 ms

Table 5
Automated component adaptation system performance.

Data domain	N samples	Mean component size (bytes)	Mean number of XML lines per component	Lines of XSD	Time to extract XSLT (mseg)	Mean time to apply XSLT (mseg/component)	Total adaptation time (mseg)
Yahoo! Pipes	190	42487.00	68910.50	1728	34.56	5.76	1128.96
Open Kapow	180	9924.07	17227.96	412	1.95	1.47	266.55
JackBe	186	15381.61	31305.91	942	10.21	2.67	506.83
Marmite	184	12395.67	29530.11	536	3.30	2.52	466.98

of lines of the XSD modelling the XML file of the respective component. This yields an objective measure of the structural component complexity beyond the size of their source code. Table 6 shows the correlations of times with respect to these explanatory variables.

We found that the performance of the automated adaptation system is not directly correlated with the size of the source files of the components measured in bytes (values are not close to 1). However, Table 6 highlights two logically very strong correlations, close to the maximum value equal to 1. There is statistical evidence that the time taken to extract the XSLT depends directly on the number of lines of the respective EUD tool XSD (that is, template complexity), whereas the time to apply XSLT depends not on the XSD schema, but on the number of lines of XML in the component source code.

Table 6
Correlations between times taken and data features.

Correlation matrix	Time taken to extract XSLT	Mean time taken to apply XSLT per component
Mean component size (bytes)	0.2123	0.3211
Mean number of lines of XML per component	0.478	0.997
Lines of tool XSD	0.996	0.562

We analysed how the XSLT extraction time varied depending on the number of lines of the EUD tool XSD schema (template complexity) and how the XSLT application time evolved depending on the number of lines of the XML source files. Fig. 6 shows these variations.

The time taken to extract the XSLT file of an EUD tool varies quadratically with respect to tool template complexity (the number of lines of their XSD), whereas XSLT application varies linearly depending on the number of lines of component XML source code. We confirmed these results statistically fitting an ANCOVA (analysis of covariance) linear and nonlinear regression model to both time variables under study. Tables 7 and 8 shows the model fits for the XSLT extraction time and XML-dependent application time, respectively.

The values R^2 and adjusted R^2, which are equal to 1, indicate that the explanatory variables used (number of lines of the XSD template and of XML, respectively) fully explain the observed time variables, which statistically corroborates the identified relationships between the variables. The equations governing the two models are as follows:

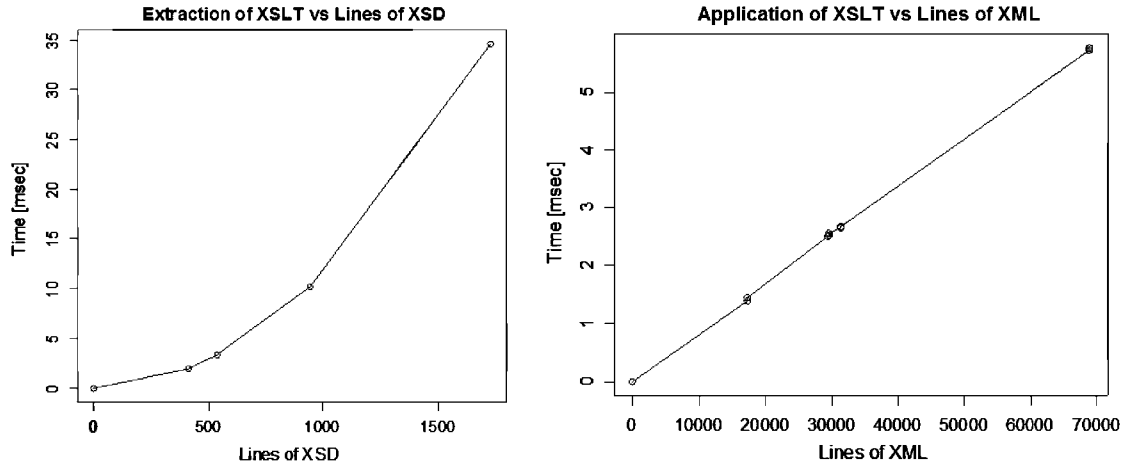

Fig. 6. Performance data depending on component features.

Table 7
Nonlinear adjustment of XSLT extraction time.

Coefficients of adjustment of XSLT application time with respect to lines of XSD	
R (correlation coefficient)	1.000
R^2 (coefficient of determination)	1.000
SCR	0.000

Observations	Weights	X	Y	Y (Model)	Residues	Standardized residues
Predictions and residues						
Obs1	1	1728.000	34.560	34.560	0.000	0.038
Obs3	1	412.000	1.950	1.953	−0.003	−0.547
Obs5	1	942.000	10.210	10.211	−0.001	−0.280
Obs7	1	536.000	3.300	3.296	0.004	0.788

Table 8
Linear adjustment of XSLT application time.

Coefficients of adjustment of XSLT application time with respect to lines of XML					
R (correlation coefficient)					1.000
R^2 (coefficient of determination)					1.000
R^2aj. (adjusted coefficient of determination)					1.000
SCR					0.002
Source	GDL	Sum of squares	Mean square	Fisher's F	Pr > F
Evaluation of the value of the information originated by the variables (H0 = Y = Moy (Y))					
Model	1	20.750	20.750	59529.845	< 0.0001
Residues	7	0.002	0.000		
Total	8	20.752			

$$T_{extraction\_XSLT} = 7.3066E-02 - 2.5742E-04 * N_{lines\_XSD} + 1.1698E-05 * N^2_{lines\_XSD} \quad (1)$$

$$T_{application\_XSLT} = 6.6139E-02 + 8.2612E-05 * N_{lines\_XML} \quad (2)$$

The computational complexity of the automated adaptation algorithm is a square function of the respective tool template complexity for the operation of extracting the XSLT mapping from any EUD tool, but linear for the operation of applying the mappings to the actual components. The biggest cost in terms of time is when the framework is applied to a new EUD tool. However, this operation is only performed once for each tool (and there are fewer than twenty really relevant tools on the market), and, at less than 35 ms for all the analysed tools, this is an acceptable cost (see Table 4). Finally, the computational cost of applying the automated adaptation system to any component obeys the following equation:

$$T_{preparation\_components} = T_{extraction\_XSLT} + N_{components} * T_{application\_XSLT} \quad (3)$$

The processing time taken to apply the AAS to a single component or sole source file will largely depend on the template complexity of the EUD tool for which it was programmed, whereas the impact of the size of the XML will be negligible.

The fourth research question to be answered is:

RQ4: How efficient is the automatic EUD adaptation system in terms of time and resources taken to adapt each component compared with manual component adaptation?

Although the data indicate that the AAS adapts a set of components in hardly any time at all, the system's real merit is only fully appreciable considering just how time and resource consuming it is to adapt these components by hand. This is illustrated in the experiment reported in Section 5. Based on the experience gathered as part of 7th European Union Framework Programme research and the development projects, like EzWeb and FAST (targeting the construction of a development environment for new EUD components by end users that require such components) and other similar projects [1,9], we estimate that the effort required to populate a component catalogue of a new EUD tool containing from 80 to 100 general-purpose components (operators, data source managers, BPM abstractions, data visualizers, etc.) by manually adapting components from other commercial components usually accounts for 50% of the total programming effort for setting up the EUD support environment of the new tool. The time taken to adapt these components using the AAS system (of the order of seconds) would drastically reduce the total programming workload for building the above EUD environment.

Because, as shown in the experiment described below, end user success at developing applications hinges on the size of the component catalogue, we consider that an automated adaptation system for heterogeneous web components, like AAS, which is able to adapt existing components automatically and save time and resources, is a major advance in the EUD field.

5. Experiment: AAS efficiency compared to manual component adaptation

It now remains to test how much time and resources the system saves compared with manual component adaptation. To do this, following the guidelines for software engineering experiments and case studies published by Runeson and Höst [45], AAS was applied at a corporation using an EUD tool as a development environment for users without programming skills. To do this, we conducted an experiment in partnership with Telefónica I + D, a subsidiary of a Spanish broadband and telecommunications provider which is the fifth largest mobile network provider in the world.

The experiment was based on the design of a web portal. The web portal was to function as an operational support system (OSS) for Telefónica employees with different roles, traits, knowledge and needs and operate as a dispatching and trouble-ticketing system to support their routine work. Such systems are the key component of software systems for processing user queries and claims at Telefónica. They are essentially general-purpose systems, and are therefore applicable to a broad-spectrum of possible scenarios, ranging from electronic dispatching of administrative records within the public administrations to the management of incidents and claims by business service end users.

An OSS has to be configured and personalized by end users within the company to adapt the web portal to their personal needs. A commercial agent has need of tools for consulting the contact information of customers in an area, the availability of special offers, a log of communications with customers, former customers and potential new users, etc. A technical assistance agent has need of tools specifying the state of the telephone network in an area, the location of the nearest optical multiplexor to a customer's home, the simple network management protocol (SNMP) elements for network resources, etc. Therefore, an OSS is a perfect scenario for EUD: it would be great to have a EUD tool that had a catalogue of components for this problem domain large enough for all end users to each be able compose a RIA operating as their personal OSS by interconnecting these components. We found, as noted below, that none of the existing EUD tools has all the components that would be needed to create an OSS of this sort.

In a pilot study, we split 28 end users with different positions and corporate profiles at Telefónica, none of whom had programming skills, into four groups. We supervised these work groups along with a company computing engineer. Each group of seven users analysed a tool: Yahoo! Pipes and Dapper, Open Kapow, JackBe and Marmite. They reported that none of the tools were of any use for building a multifunctional OSS adaptable to more than one business role, because components were missing. Some tools provided support for one task type but were missing components for others:

- The Yahoo tools are unable to manage synchronous/asynchronous communication systems between operator/customer or among internal system users. Additionally, they cannot invoke back-end services based on SOAP or POX-RPC. But they do have access to components for managing geolocation tasks, finding contact details and building to-do lists.
- Open Kapow is useful for building small visual interfaces based on pre-existing web portals, but offers no mechanisms for creating visual components that are able to find addresses, invoke back-end Telefónica services or create automated interactions

among different components (such as select a customer from the database and display a situation map or log of messages sent/received by the user).

- Marmite is able to manage communication, messaging and geolocation systems, as well as invoking services using HTTP and XML, but cannot administer complex data sources, manage to-do lists, set up front-ends to manage network services or complex protocols like SOAP, SNMP, etc.

The results of this pilot study suggested that the set problem would not be able to be solved without components from more than one tool, and a qualitative leap in the type of solutions that end users would be able to build could be achieved using more than one catalogue and approach.

Therefore, the construction of a multifunctional OSS, similar to the example illustrated in Fig. 7, would require a catalogue containing a set of components offered by all the analysed tools. According to the pilot study conducted by the above workgroups, the catalogue would have to be populated by 80 a priori heterogeneous components from different catalogues and thus conform to different coding standards.

The zoomed screenshot in Fig. 7 illustrates a simple scenario extracted from a Telefónica core OSS environment created using EzWeb and a universal catalogue containing elements from different EUD tools, built as explained later using our AAS system. The zoomed section, which is part of a more general OSS now deployed at Telefónica as a fully operational environment, connects four components: a to-do list listing customer complaints, a customer data viewer, a Google map and a network status map. None of the analysed tools (Yahoo!, Open Kapow, JackBe or Marmite) has all of these components, although they each have some. This fully functional environment was built by visually linking components from a universal catalogue to each other and to the enterprise back-end: a user selects a given task from the to-do list, the directory gadget will display customer details and have a customer/task selection option, the network map will represent the selected customer's network status and the Google map gadget will display the selected customer's address on a map. None of the existing EUD tools have all the components that end users need to build this application. The required components, totalling, as mentioned above, 80 were identified by the work groups and are defined in Table 9.

Table 9 shows how many and what type of components were identified as necessary for building the OSS. Column N shows the number of components that the four work groups participating in the pilot study identified as necessary, whereas the other columns show how many such components, which could be used as-is for the specified purpose if transformed to the standard format, the respective EUD tools provide. Clearly, existing tools have some but not all of the necessary components. Together, however, the tools provide 100% of the necessary components. Some components by different tools serve the same purpose, in which case the components with the smallest template were selected for mapping, as such components are easier to adapt. In order to provide end users with a wider range of components from which to select whichever better fits their needs, a universal catalogue may contain several equivalent components, which differ merely as to their visual appearance. For this experiment, however, we selected the simplest components to adapt the minimum necessary set.

Because none of the catalogues include all the components (the most useful catalogue would be Yahoo containing 22 out of the 80 necessary components, less than 28%) and since the respective

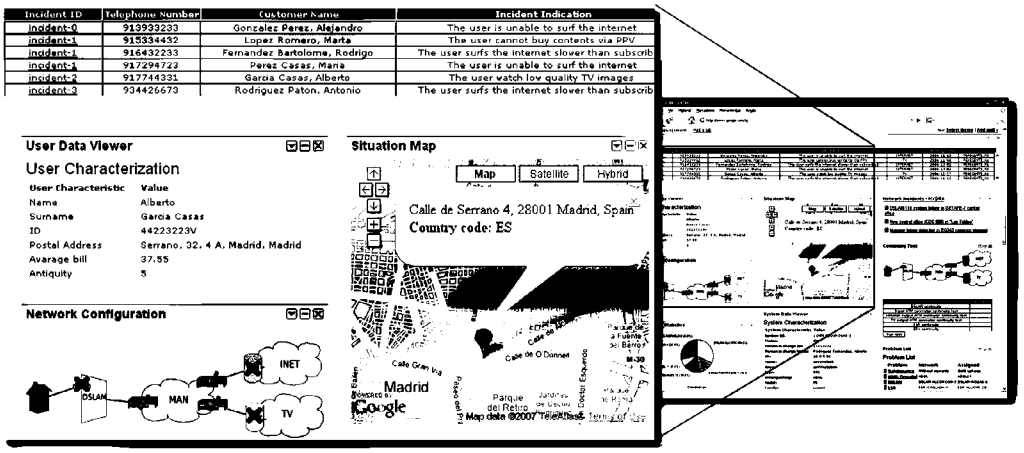

Fig. 7. Example of an OSS built using a EUD tool that has a universal component catalogue.

Table 9
Components identified as necessary for building a valid catalogue for building the OSS by EUD at Telefónica I + D.

Necessary Components	N	Yahoo! Pipes and Dapper	Open Kapow	JackBe	Marmite	Components available across all tools
RSS data handling components	7	5	0	1	1	100%
SNMP management components	5	1	1	3	0	100%
Geolocation components	4	1	1	1	1	100%
User-system communication components	14	0	2	0	12	100%
BPM management components	11	1	0	6	4	100%
Data viewer components	12	1	7	2	2	100%
Ticketing components	8	3	2	2	1	100%
Basic data operators	9	4	3	2	0	100%
Component connectors	10	6	0	4	0	100%
Total	80	22	16	21	21	80

Table 10
Description of the workload for building a universal catalogue.

Work group	Imported components	Created components	Technology training time	Mean adaptation time per component	Mean creation time per component	Total time taken per person	Total time taken by project group
A	75	5	2 h	2.810 h	4.600 h	23.57 h	235.75 h
B	80	0	4 h	0.022 h	–	5.76 h	5.76 h

tools are proprietary software for which new components cannot be easily created, it was decided to create a new universal catalogue that would contain the 80 necessary components for an open software tool like EzWeb.

Two work groups were set up: (a) Group A was composed of 10 web programmers from Telefónica I + D, acquainted with the necessary technologies (like PHP, Python, XHTML, JavaScript, etc.) and assigned the task of creating a catalogue of the components necessary for the OSS application and for use by the EzWeb open source tool, by either adapting or creating components; (b) Group B, composed of a single external programmer, instructed in the use of our automated adaptation system, who received the same assignment, save that he was not to program or adapt the components by hand but using the AAS system reported in this paper. The goal was to build a universal catalogue containing the 80 identified components in the shortest possible time.

The ten programmers in Group A opted to reuse 75 components provided by existing tools. They built five for EzWeb from scratch, one of which was a component for graphically displaying numerical data and the other four for processing RSS data produced by invoking SOAP-based back-end services, necessary to perform BPM (Business Process Management) tasks. Group A decided to build these five components from scratch because they thought that this would be easier and faster than adapting their very complex Yahoo! Pipes components (as indicated by the number of lines of XSD).

The lone programmer in Group B used the AAS automated system to correctly transform all 80 components from the catalogues of the other tools. The time taken (in hours) to complete the above assignment is shown in Table 10.

These data provide a better picture of the real benefit of the proposed system within a real-world business environment in a scenario requiring a specialized catalogue of components designed for a particular problem domain. It took a coordinated group of 10 programmers three full work days to adapt the components provided by the other tools to a common standard catalogue. It took the whole work group almost half a work day to complete each component. It would have taken a single programmer around a month to complete the assignment, whereas the lone programmer using the AAS automated adaptor did the job in just 5 h and 45 min, plus another four hours that were spent teaching the person to use the HermiT reasoner, extract the XML-format components, automate their XSD extraction and publish the result in the shared catalogue. It took the programmer just over 1 min and 19 s to translate each component to the proposed standard language and publish it in the universal EzWeb catalogue.

The catalogues created by the two groups are absolutely equivalent and were used to build the OSS required by the Telefónica I + D company that directed the experiment. How end users used the EzWeb EUD tool to build the OSS program based on this universal catalogue is beyond the scope of this paper but has been published elsewhere [24,25].

6. Discussion on threats to validity

This discussion on threats to the validity of the above analysis and experiment addresses to four aspects of validity, which can be summarized as follows:

- *Construct validity:* This aspect of validity refers to the extent to which the analysed operational measures really represent what the researcher has in mind and what is being investigated according to the research questions:
  - RQ1: According to the statistics posted on their web portals, this paper has examined the four tools with the largest number of users. All 730 correctly adapted components have been measured through white- and black-box tests run on 185 components sampled from their component catalogues to check the result of the adaptation process. This threat could have been reduced by choosing more tools and more components of each tool. But we consider that the analysed study adequately analyses this research question, and demonstrates the utility of the proposal.
  - RQ2: In order to check the effectiveness of the adapted with respect to the original components, we analysed two measures before and after the conversion: machine code instructions and mean execution time on the same hardware. These labour-intensive tests were conducted for a total sample of 40 components, 10 for each of the different EUD source tools, picked at random from the 730 correctly adapted components. We used common measures for this type of research question. Although we might have analysed more components, the similarity of the resulting data suggests that this threat is not relevant to either the number of instructions generated by the adapted component or the execution time with respect to the original component.
  - RQ3: The component adaptation time (function of the number of XML lines of the component and number of XSD lines of the schema) does not pose relevant threats, as the component is only adapted once. In response to this research question, we studied correlations and analysed covariances between the explained variable "adaptation time" and all the possible descriptive variables of the 730 analysed components: provider, host type, processed data type, URLs, component internal architecture, lines of source code, lines of component template, etc. This study (based on linear and non-linear regression models) is often used to address this type of questions.
  - RQ4: The study of AAS efficiency in terms of the time taken and resources consumed in component adaptation compared with manual adaptation was conducted on a real scenario using an explanatory, quantitative and controlled experiment. The objective quantitative measure used is the total project development time, but it would be worthwhile conducting other studies (like case studies in real companies) that can help to better quantify the benefits of using the system. The experiment clearly illustrates that system use really does generate a tangible and significant benefit, and there is no threat in this respect.
- *Internal validity:* This aspect of validity is of concern when causal relations are examined. When the researcher is investigating whether one factor affects an investigated factor there is a risk that the investigated factor is also affected by a third factor. If the researcher is not aware of the third factor and/or does not know to what extent it affects the investigated factor, there is a threat to the internal validity. These threats only apply to research questions 2, 3 and 4:

- In RQ2, the adapted component execution environment compared to its unadapted source environment is the only factor that has a causal relation in the study. Analysing performance quantitatively without establishing causal relations in both cases should eliminate this threat.
- In RQ3, we established that the number of lines of XML and the number of lines of the XSD schema variables affect the time taken by the system to adapt the component. We used regression models to analyse many other qualitative and quantitative variables that describe each component, and no other correlations were found.
- In RQ4, two teams A and B (one composed of 10 web programmers who imported components or built non-existent components and another composed of a single external programmer who used the AAS system) performed a controlled and predesigned experiment in order to check the results of using and not using the adaptation system. In this case, the threat is that another team A might have been more efficient than the team A participating in the experiment. However, as the programmers were selected at random and there is an impressive difference between the total time taken by teams A and B to develop the project, this is not a feasible threat.

– *External validity:* This aspect of validity is concerned with the extent to which it is possible to generalize the findings and the findings are of interest to other people outside the investigated case.
In the conducted study, the findings for the studied tools, which are the most successful on the market today, can be generalized to the subset of EUD tools, whose components use XML and XSD as serialization and internal codification instruments. This adds value to the research conducted in the field of WEUD (web end user development). However, the results are not applicable to other EUD tools that are of no use for building web applications (such as spreadsheets or desktop visual coding programs). We are working on applying the proposed system to other totally different domains, like automatic XML data source preprocessing for enacting KDD processes with excellent preliminary results. This gives a flavour of the multidisciplinary potential of the proposal in other domains.
– *Reliability:* This aspect is concerned with the extent to which the data and the analysis are dependent on specific researchers. Hypothetically, if another researcher conducted the same study later on, the result should be the same.
- Save the experiment conducted, where there is a dependency of the results on the selected sample of users, the analyses and measurements conducted in remainder of the research were automated and are completely repeatable by external researchers. The experiment quantifies the benefit of applying against not applying the proposed system, but the time difference measured in each case should be considered as indicative and can vary in each new conducted experiment.

7. Conclusions and Future Work

We believe that the development of an automated component adaptation system such as the proposed AAS, which processes and converts any web component to a standard format provided that the component conforms to a standard template, is a major advance in the EUD field. Our proposal is based on the use of description logic. Based on a generic UML2 component model, it is able to check whether a particular component in XML can be unambiguously and consistently mapped to this model. It is able to automatically find a finite set of XSLT mappings to adapt the component so that it can be used by a different EUD tool to the one for which it was developed.

Our automated web components adaptation system has been tested on components from four very different existing EUD tools with satisfactory results. In all cases, the proposed notation was applied to structurally complex components which we managed to adapt to a general-purpose EUD tool, governed by a UML2 component model that we defined here. The sound results of applying this proposal on components from several such complex and different EUD tools confirm its merit. This system enables end users to develop complex applications, sharply increasing the potential of EUD development and improving the EUD paradigm's options of evolving successfully. On other hand, it offers a sizeable saving of time and resources within the component cataloguing process.

The proposed system is easily generalizable for automatically adapting any XML-based file to a target XSD schema in such a manner that the structure but not the content of the information is modified, irrespective of whether this information is a data source or the source code of a web component or resource. Additionally, the proposed mapping can be applied with slight changes to any other UML2. This means that our research is usable in any branch of knowledge where XML data have to be prepared and adapted to a XSD schema other than the one to which they already conform, such as the adaptation of heterogeneous data sources, and the preparation of complex data for knowledge discovery processes (KDD).

Regarding the limitations of the research, the system has to extract the XSD schema modelling the original data source and is prone to error if any values of those data are missing. Additionally, the system is only applicable to information in XML format and is not applicable to formats like JSON or unlabelled plain text files.

The next logical step in this line of research is to develop the automated component adaptation system to process possible missing values in components. Another research line would be to apply the proposed system to other software engineering fields, such as the automated preprocessing of XML-based structurally complex data, the automated adaptation of components in component-based software engineering and generally any field where it is necessary to process, fit or map heterogeneous sources of labelled data in order to apply inference or automated processing techniques.

References

[1] 4CAAST PROJECT, Official Web Site, 2012. <http://4caast.morfeo-project.org>.
[2] AMICO, AMICO Web Platform, 2012. <http://amico.sourceforge.net>.
[3] F. Baader, D. Calvanese, D. Mcguinness, D. Nardi, P. Patel-schneider, The Description Logic Handbook: Theory, Implementation and Applications, University Press, Cambridge, UK, 2003.
[4] Andrea Calì, Diego Calvanese, Giuseppe DeGiacomo, Maurizio Lenzerini, A formal framework for reasoning on UML class diagrams, Foundations of Intelligent Systems, Lecture Notes in Computer Science, vol. 2366, Springer, Berlin/Heidelberg, 2002, pp. 423–427.
[5] Daniela Berardi, Diego Calvanese, Reasoning on UML class diagrams using description logic based systems, in: Workshop on Applications of Description Logics, 2001.
[6] Diego Calvanese, Maurizio Lenzerini, Representing and reasoning on XML documents: a description logic approach, J. Logic Comput. (1999) 295–318.
[7] EZWEB, EzWeb Web Platform, 2012. <http://conwet.fi.upm.es/morfeo-project/ezweb_blog/?lng=en>.
[8] EZWEB PROJECT, Official Demo Web Site, 2011. <http://demo.ezweb.morfeo-project.org>.
[9] FI-WARE PROJECT, Official Web Site, 2012. <http://www.fi-ware.eu>.
[10] Franklin Ramalho, Jacques Robin, Mapping UML class diagrams to object-oriented logic programs for formal model-driven development, in: 3rd UML Workshop in Software Model Engineering, 2004.
[11] GOOGLE, iGoogle Web Platform, 2012. <http://www.google.com/ig>.
[12] V. Haarslev, R. Möller, Racer System Description, Springer-Verlag, Germany, 2001. pp. 701–705.
[13] I. Horrocks, O. Kutz, U. Sattler, The Even More Irresistible SROIQ, American Association for Artificial Intelligence, 2006.
[14] JACKBE, JackBe Presto Cloud Web Platform, 2012. <http://prestocloud.jackbe.com/>.
[15] Jocelyn Simmonds, Cecilia M. Bastarrica, A tool for automatic UML model consistency checking, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), ACM, New York, NY, USA, 2005, pp. 431–432.

[16] KAPOW SOFTWARE, Open Kapow and Kapow Katalyst Web Platform, 2012. <www.kapowsoftware.com>.
[17] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M.B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, The state of the art in end-user software engineering, J. ACM Comput. Surv. 43 (3) (2011) (Article 21).
[18] Konstantinos N. Vavliakis, Andreas L. Symeonidis, Georgios T. Karagiannis, Pericles A. Mitkas, An integrated framework for enhancing the semantic transformation, editing and querying of relational databases, Exp. Syst. Appl. 38 (4) (2011) 3844–3856. http://dx.doi.org/10.1016/j.eswa.2010.09.045. ISSN 0957-4174.
[19] H. Lieberman, F. Paterno, V. Wulf, End-User Development, Kluwer Springer Academic Publishers, Dordrecht, The Netherlands, 2006.
[20] D. Lizcano, F. Alonso, J. Soriano, G. Lopez, A user-centric approach for developing and deploying service front-ends in the future internet of services, Int. J. Web Grid Serv. 5 (2) (2009) 155–191.
[21] D. Lizcano, Formalization of the Emerging End-User Programming Paradigm, Ph. D, Thesis Published by Universidad Politécnica de Madrid, Fundación General UPM Publishers, Spain, December 2010.
[22] D. Lizcano, F. Alonso, J. Soriano, G. Lopez, A new end-user composition model to empower knowledge workers to develop rich Internet applications, J. Web Eng. 3 (10) (2011) 197–233.
[23] D. Lizcano, F. Alonso, J. Soriano, G. Lopez, End-user development success factors and their application to composite web development environments, in: Proceedings of the Sixth International Conference on Systems, ICONS, vol. 11, 2011, pp. 99-108.
[24] D. Lizcano, F. Alonso, J. Soriano, G. Lopez, Supporting end-user development through a new composition model: an empirical study, J. Univ. Comput. Sci. 18 (2) (2012) 143–176.
[25] D. Lizcano, F. Alonso, J. Soriano, G. Lopez, A web-centered approach to end-user software engineering, ACM Trans. Softw. Eng. Methodol. 22 (4) (2013) 29, http://dx.doi.org/10.1145/2522920.2522929.
[26] D. Lizcano, End-User Development Web Site, Statistical Survey of the End-User Development Paradigm, 2013. <http://apolo.ls.fi.upm.es/eud/>.
[27] J.A. Macías, F. Paternò, Customization of web applications through an intelligent environment exploiting logical interface descriptions, J. Interact. Comput. 20 (1) (2008) 29–47.
[28] MARMITE, Marmite Web Platform, 2012. <http://www.cs.cmu.edu/~jasonh/projects/marmite/>.
[29] MICROSOFT, Microsoft Popfly Web Platform, 2012. <http://www.popfly.com>.
[30] Netvibes UWA Specification. <http://dev.netvibes.com/doc/uwa/documentation> (accessed 24.09.07).
[31] Luigi Palopoli, Giorgio Terracina, Domenico Ursino, A plausibility description logic for handling information sources with heterogeneous data representation formats, Ann. Math. Artifi. Intell., vol. 39, Springer, Netherlands, 2003, pp. 385–430.
[32] J. Rode, Y. Bhardwaj, M.A. Perez-quinones, M.B. Rosson, J. Howarth, As easy as "Click": end-user web engineering, in: Proceedings of the 2005 International Conference on Web Engineering, 2005, pp. 478–488.
[33] C. Scaffidi, M. Shaw, B. Myers, The "55M End User Programmers" Estimate Revisited, Technical Report CMU-ISRI-05-100, Carnegie Mellon University, 2005.
[34] R. Sobek, Official MOF Specification from OMG, Technical Report, Object Management Group, Inc., USA, 2005.
[35] U. Sattler, Terminological Knowledge Representation Systems in a Process Engineering Application, PhD Thesis, LuFG Theoretical Computer Science, RWTHAachen,1998.
[36] V. Haarslev, R. Möller, RACER system description and its applications, in: Proc. of IJCAR, 2001.
[37] V. Haarslev, R. Möller, High Performance Reasoning with Very Large Knowledge, 2001.
[38] Ragnhild Van Der Straeten, Tom Simmonds Mens, Jonckers Jocelyn, Stevens Viviane, Whittle Perdita, Using description logic to maintain consistency between UML models, in: The Unified Modeling Language. Modeling Languages and Applications, Lecture Notes in Computer Science, vol. 286, Springer, Berlin/Heidelberg, 2003, pp. 326–340.
[39] W3C Web Accessibility Initiative, 2012. <http://www.w3.org/WAI/>.
[40] W3C Web API Working Group, 2012. <http://www.w3.org/2008/webapps/>.
[41] W3C Web Application Formats, 2008. <http://www.w3.org/2006/appformats/>.
[42] Xiaodong Zhu, Hengshan Wang, Hongcheng Gan, Chunchang Gao, Construction and management of automatical reasoning supported data mining metadata, in: The International Conference on Business Management and Electronic Information (BMEI), 2011, pp. 205–210.
[43] YAHOO!, Yahoo! Pipes Web Platform, 2012b. <http://pipes.yahoo.com/>.
[44] YAHOO!, Yahoo! Dapper Web Platform, 2012a. <http://open.dapper.net>.
[45] Per Runeson, Martin Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Eng. 14 (2) (2009) 131–164, http://dx.doi.org/10.1007/s10664-008-9102-8.