Using Declarative Workflow Languages to Develop Process-Centric Web Applications

Mario L. Bernardi∗, Marta Cimitile†, Giuseppe Di Lucca∗, and Fabrizio M. Maggi‡
∗Department of Engineering-RCOST, University of Sannio, Italy
†Faculty of Jurisprudence, Unitelma Sapienza University, Italy
‡Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
{mlbernar, dilucca}@unisannio.it, marta.cimitile@unitelma.it, f.m.maggi@tue.nl

Abstract—Nowadays, process-centric Web Applications (WAs) are extensively used in contexts where multi-user, coordinated work is required. Recently, Model Driven Engineering (MDE) techniques have been investigated for the development of this kind of applications. However, there are still some open issues. First, a complete roundtrip engineering support is still missing. Second, the lack of a proper integration of the existing MDE methodologies with workflow modeling techniques does not allow the developers to model the components of a WA in a compact and easy way. Third, the existing MDE approaches are based on procedural languages and not on declarative languages, even if these are more suitable to describe processes characterized by high variability. To address these issues, in this paper, we adopt a conservative approach to roundtrip engineering for the development of process-centric WAs and propose the integration of three MDE metamodels used to represent the main components of a WA with the metamodel of Declare, a declarative language to represent business processes. The proposed approach has been used to develop an online system for visit reservation in a hospital in order to validate its feasibility, correctness and effectiveness.

Keywords: Model Driven Engineering, Declarative Processes, Domain Specific Languages, Code Generation, Web Applications Development

I. INTRODUCTION

Nowadays, business processes are becoming more and more important in Web Applications (WAs) and play an important role in all contexts where multi-user, coordinated work is required [13]. The widespread affirmation of process-centric WAs can be linked to the increasing need for supporting and guiding users in the process execution while preserving the hypertext flexibility and the WAs advantages. With regard to this, several traditional WAs development methods have already evolved to include an explicit process view [12] and also the more recent Model Driven Engineering (MDE) approaches are becoming more “process oriented”. However, with respect to these latter approaches, some issues still need to be addressed [18].

The main drawback is related to the lack of support for conservative round-trip engineering. This makes the generation of the business process management components more difficult due to the several manual customizations that are often not trivial and that force the developers to execute several iterations of the generation steps.

Another drawback is that, in traditional MDE approaches, the process metamodel is not well integrated with the metamodels describing the other components of the WA. A consequence of this is that the developers cannot model the components of the application in an integrated way. Therefore, it is not possible to eliminate any duplicate information and it is necessary to trace elements belonging to different and not compatible metamodels.

Finally, traditional MDE approaches for the development of process-centric WAs, are based on procedural processes modeling languages instead of declarative ones that are suitable to support flexibility in the process definition [24]. A declarative language allows to compactly represent the processes underlying WAs that work in unstable environments and involve multiple execution paths.

To address these issues, in this paper, we propose an approach to integrate and extend the existing MDE methods used to develop WAs with new mechanisms to correctly merge and weave, in the regeneration of the process-related components, possible incoming changes in the process structure. In particular, we integrate three MDE metamodels representing the structure of information, service and presentation layers of a WA with the metamodel of Declare [24], a declarative language to represent business processes. This approach enables the automatic generation of a fully working prototype of a process-centric WA. The use of MDE technologies in conjunction with a declarative process modeling language allows the developers to represent the behavior of a WA with a high level of flexibility as a compact set of constraints. Furthermore, this approach allows the developers to work on the process only at the model level thus reducing the effort for customizing and maintaining the generated code. When changing the process at the model level, the changes are automatically translated and forwarded to both the process engine and the UI layer that, at run-time, will behave according to them. In this way, the developers are not required to customize the generated code when the underlying process is changed (e.g., due to modifications in the business process requirements).

The paper is structured as follows. Section 2 discusses related work. Section 3 and Section 4 describe the proposed approach and the tool developed to support it respectively. Section 5 reports the application of the approach to a case study. Some conclusive remarks are in Section 6.
II. RELATED WORK

The topic of the integration of process modeling techniques with MDE approaches for the development of WAs is not new in literature. In [16], the authors use a workflow conceptual model supporting the interaction between the hypertext and an underlying workflow management system. In [12], an approach is proposed based on the integration of the Object-Oriented Hypermedia method (OO-H) and the UML-based Web Engineering (UWE). In [20], the content and the navigation models are extended with activity entities and activity nodes respectively, represented through UML primitives. In [22], Business Process Management (BPM) [27] and Object Oriented Web Solution (OWS) [8] are combined. In particular, in this work, model-to-model transformations are used to generate a navigational model from the BPM definition and model-to-text transformations are used to produce an executable process definition in BPEL. Recently, in [3], the authors have presented an integrated approach and a toolsuite supporting the specification, the design and the implementation of complex, multi-party business processes. It is based on BPEL for the process representation, on the MDE methodology and on code generation techniques to produce WAs. Finally, in [4], BPMN is used to describe process actors, tasks and business constraints. These are transformed into a Web user interface and into a Web service orchestration model encoded in a Domain Specific Language (DSL) called WebML.

Differently from these methodologies, our approach is based on a code generator that uses a flexible template language for an effective round trip engineering support. Moreover, in other approaches, the integration of MDE with workflow modeling languages is always based on procedural notations, whereas, in our approach, we use a declarative language. Procedural notations such as BPMN [2], Petri nets [17], UML ADs [19], EPCs [23] and BPEL [5] explicitly describe the steps of a process and can be used to force the process participants to follow a strict procedure. In contrast, declarative languages [6], [9], [25] are able to offer multiple alternatives during the process execution and can be used when participants are more autonomous in the process execution. In [30], [7], the authors discuss the advantages of declarative languages (flexibility, explicit specification of business rules and a logic-based verification support) and their disadvantages (they cannot be used to support and guide the participants in the process execution). To the best of our knowledge, in literature, there are not approaches that integrate declarative modeling languages and MDE for WA development. In this paper, we use Declare1, a declarative process modeling language proposed by Pesic and van der Aalst in [24]. Finally, no one of the aforementioned approaches provide a tight integration of a process metamodel with the metamodels representing the other components of a WA. Our approach makes the development of a WA easier because it allows the developers to represent the WA more compactly, thus allowing them to reduce redundancies in the models’ descriptions.

III. PROPOSED APPROACH

In this paper, we propose a Platform Independent Model (PIM) [1] to represent a WA by means of four linked metamodels. Each metamodel is associated to a corresponding DSL and describes a particular aspect of the WA (i.e., information layer, service layer, presentation layer and a process layer based on the Declare language).

The information layer is modeled by a common set of modeling concepts that allow us to define sound and rich information models. To define the necessary constructs and modeling primitives, we use semantics and notation provided by the UML class diagrams. An excerpt of this model is shown in Figure 1-(a). Here, the main concept is the Classifier meta-class that models a generic domain object specializing as an Entity (representing the super-class of unique and persistent objects in the WA domain model). Relationships are represented by means of the Association metaclass. An Association specifies minimum and maximum cardinality of relationship roles, caching attributes, mappings to the relational model and other properties useful to support the generation of the source code and of the other artifacts needed for the WA development. The ValueObject metaclass can represent a TransferObject (needed to handle values), a PrimitiveType (mapped to a language type) or a UserDefinedType that refers to types that need special treatment (e.g., binary objects, images, internationalized text). The Operation represents behaviors that refer to the elements of the information layer meta-model.

Figure 1-(b) shows the metamodel used to represent the service layer of the WA. The main metaclass is ServiceDefinition: it represents a set of behaviors adhering to a well known external interface. The operations of the interface are specified through the ServiceOperation metaclass. Each ServiceOperation can specify a behavior directly or indirectly by delegating it to one or more operations of a DataAccessObject (DAO) which is responsible for finally managing data for an Entity.

The presentation layer metamodel, shown in Figure 1-(c), describes the UI layer. The UIRoot is the main metaclass of this layer aggregating (indirectly by means of UIBundles) the Activity concept regarded as either a user activity (requiring human intervention) or a batch activity. The Activity metaclass is the root of a hierarchy (not explicitly showed in figure due to space constraints) of several kinds of predefined activities (such as ListActivity, TableActivity or GridActivity just to name few). These activities take data from the instances of the information model classes and organize them using different kinds of visual layouts (i.e., lists, tables and views). Each Activity has a set of properties (instances of the metaclass ActivityProperty) pointing to external data sources or directly to attributes of the information model. The Presentation layer metamodel also takes care of the navigability information: each Activity

1http://www.win.tue.nl/declare
can specify transitions towards other activities and the post-
conditions that must be satisfied on the output data. This is
particularly important to ensure the information exchanges
among instances of activities according to what specified by
the process definition.

The processes underlying a process-centric WA are de-
scribed through Declare. Declare constraints have a formal
semantics based on LTL and the language is provided
with facilities to execute and verify process models [28].
In this way, it is possible to easily check for possible
inconsistencies in the model and to understand whether the
modeled process behaves as expected. Moreover, Declare
provides a simple interface that is easily understandable for
users with different backgrounds. We do not describe the
Declare language in detail due to space constraints; the main
concepts are reported in the metamodel reported in Figure 1-
(d). In this metamodel, a Process is composed of a set of
possible events and constraints on events. The Constraint
metaclass is the abstract root of a rich set of constraints (e.g.,
Precedence, NotCoexistence) each one having a well defined
LTL semantics and a graphical representation as specified
by the Declare language definition. An Event puts together a
ProcessActivity instance with an ActivityState representing a
valid state in which that Activity can be at a given time. The
ProcessActivity is linked to the information layer metamodel
mapping attributes to the input/output parameters of the
ServiceOperation. It specifies the behavior associated to the
ProcessActivity itself. Each ProcessActivity is also linked to
the presentation layer being associated to a single UIRoot
containing several inner activities. These activities can be
created by using the Attribute class thus allowing the user
to interact with the process.

A. Code Generation

The proposed approach adopts MDE technologies for the
automatic generation of a fully working workflow-enabled
WA starting from its design models. At the heart of our
solution there is a generative approach centered on the four
linked metamodels just illustrated. These metamodels are
provided as input to a code generator that is able to generate
the implementation for a target platform adopting different
well-known architectural patterns. In our case, the target
platform is a J2EE spring-based one; it exploits a Command-
based Model-View-Controller (MVC) design to build the UI,
an aspect-orientation approach to improve the modularity of
persistence and logging concerns, and dependency injection
to increase flexibility and reuse.

The generation of the application takes all four models
as input and, by executing a set of model-to-text transfor-
mations, automatically generates the source code and all
operations. In addition, a Service template to generate the code for the information layer operation and the associated DAO and Service classes. The to generate the Java classes implementing an Entity, its associated domain classes is omitted.

To illustrate the generation approach with more details, an excerpt of an Xpand template, reported in Figure 2, is used to generate the UI views and the navigational bars on the entity instances that allows for the execution of the other CRUD operations. A navigational bar with links to those pages is generated by another template and injected into each view of the application, allowing the user to quickly reach and manage the contents associated with every Entity in the model. Furthermore, a PendingTasks page is generated, in which all the process instances (grouped by process) are showed (using a TableActivity). On every instance, a contextual menu is built. This menu mixes a static part (containing process management actions such as start, stop or remove activities) and a dynamic one obtained by querying at runtime the Declare Engine component. This dynamic portion shows only the valid steps that can be executed, given the actual state of the process instance and the set of constraints in the reference Declare process model. The proposed approach allows for a conservative approach to both forward and reverse round trip engineering (RTE). Changes to the generated WA can be made by modifying the models or directly the generated artifacts. When changing any model, the code generator will rebuild every artifact from scratch according to the changes. However, the developers can easily customize a generated artifact by removing the annotation “auto-generated” added by the code generator to each artifact when it is created for the first time. The generator always preserve the customized artifacts in the successive generations and always rebuilds the auto-generated ones from scratch. Our approach also defines annotations on artifacts used to reconcile the models from which they were generated. This is not yet implemented in the current version of our supporting tool but future work will be devoted also to this.

IV. THE M3D SUPPORTING TOOL

A tool called Model Driven Development with Declare (M3D) was developed to support the proposed approach. Figure 3 depicts the M3D architecture of both the generation platform (on the left side) and the generated application (on the other software resources needed for the execution. The generation process of the J2EE based executable prototype produces a rich set of resources (source code, HTML and CSS resources, Platform-specific Descriptors and metadata, DBMS scripts) including the integration code responsible to query the Declare Engine about the next possible activities to be executed by the users (given the current process instance and the set of constraints).

The model-to-text transformations are implemented as a hierarchy of templates written in the Xpand language. To illustrate the generation approach with more details, an excerpt of an Xpand template, reported in Figure 2, is discussed in the following. This Xpand template is used to generate the Java classes implementing an Entity, its Operation and the associated DAO and Service classes. The template to generate the code for the information layer transformation defines, for each Entity, the specialization of a generic DAO providing CRUD operations and useful persistence methods (by means of the CRUD_Operations and DB_Operations expansions). In addition, a Service class is generated for each ServiceDefinition specified in the model. In turn, a Service class contains an external operation for each ServiceOperation specified in the model. Each ServiceOperation is generated as an empty stub in which, by adopting the Generation Gap pattern [14], the developers can customize operation behavior. The delegation is implemented by means of an Aspect Oriented Programming (AOP) approach by generating an aspect that takes care of managing the DAO instances and of injecting such instances into the delegating services. Several templates are used to generate the UI views and the navigational bars that allow users to move through the application contents. In particular, for each Entity defined in the information model, a page containing a CRUDActivity is generated. This CRUDActivity is built by composing a ListActivity (to implement the retrieve operation) and a contextual menu on the entity instances that allows for the execution of the other CRUD operations. A navigational bar with links to those pages is generated by another template and injected into each view of the application, allowing the user to quickly reach and manage the contents associated with every Entity in the model. Furthermore, a PendingTasks page is generated, in which all the process instances (grouped by process) are showed (using a TableActivity). On every instance, a contextual menu is built. This menu mixes a static part (containing process management actions such as start, stop or remove activities) and a dynamic one obtained by querying at runtime the Declare Engine component. This dynamic portion shows only the valid steps that can be executed, given the actual state of the process instance and the set of constraints in the reference Declare process model. The proposed approach allows for a conservative approach to both forward and reverse round trip engineering (RTE). Changes to the generated WA can be made by modifying the models or directly the generated artifacts. When changing any model, the code generator will rebuild every artifact from scratch according to the changes. However, the developers can easily customize a generated artifact by removing the annotation “auto-generated” added by the code generator to each artifact when it is created for the first time. The generator always preserve the customized artifacts in the successive generations and always rebuilds the auto-generated ones from scratch. Our approach also defines annotations on artifacts used to reconcile the models from which they were generated. This is not yet implemented in the current version of our supporting tool but future work will be devoted also to this.

IV. THE M3D SUPPORTING TOOL

A tool called Model Driven Development with Declare (M3D) was developed to support the proposed approach. Figure 3 depicts the M3D architecture of both the generation platform (on the left side) and the generated application (on

Figure 2: An excerpt of the Xpand rule defined to translate an Entity into a DAO java class (the part generating the associated domain classes is omitted)
the right side). The core layer of our generation environment is based on the Eclipse Modeling Platform including the Xtext framework, needed to develop the DSL Editors, and the Xpand framework, used to drive the generation tasks. On top of it, the code generation layer contains the Xpand Template manager component. This component is responsible of applying templates on the user models in order to generate the needed resources. During the generation step several resource-specific generators are used (HTML, JAVA, XML, SQL). The IDE integration layer is comprised of four DSL editors (one for each metamodel). They are built using the Xtext framework and integrated into the tool, thus allowing the developer to model the application and to start the generation process.

The developer, after having defined all four models, interacts with the IDE layer to start the generation process. This process is driven by the Xpand Workflow included in the Xpand Generator that acquires the four models and performs the generation by using the services of the middle layer. The tool-chain is implemented as a set of Eclipse Modeling plugins, while the EMF/MOF metamodel is used to represent and manage the four metamodels. In this way, we are able to support the entire generation process (from modeling to code generation and customization) within a single development environment. As transformation language, we use Xpand mainly for its excellent support of EMF metamodels. Nevertheless, other features of this language (aspect orientation, polymorphism, extensibility, type system abstraction and validation) are very useful for generating the implementation of crosscutting features such as persistence, logging and especially to allow for a flexible customization of the generated code. Starting from a model instance, a textual output can be generated by defining and expanding polymorphic macros based on the model elements and their attributes and properties. The elements of the model can be referenced and accessed by the Xpand transformation expressions specified in the templates. The templates can import other templates and model instances and, in addition, also functional extensions can be implemented using Java. In our approach, a Java component (called Xpand Design Java Extension) implements several handlers and element mappers to drive the process by generating all the needed resources. Each element of the model contributes to all transformations. For instance, an Entity of the information model is translated into a Java class and is linked to the ServiceDefinition classes in which it is involved. Each ServiceDefinition is, in turn, associated to the DAO accessing the DBMS layer defined by a set of SQL scripts. Similarly, each activity in the process is generated as a Java class linked to the Activity class as specified in the process definition. A generated process manager interacts with the Declare Engine updating (when an activity is executed) the state of the entire process, thus enabling and disabling (according to the process constraints) the activities that can be executed. When an activity is executed, its linked root Activity is instantiated by the process manager and its UI part is rendered allowing the user to specify input data if needed. The code generator is responsible for the generation of the ProcessManager code that dynamically injects ActivityTransition instances into ProcessActivity objects by querying the process engine.

The right side of Figure 3 shows the interaction at the process level between the UI manager, that is responsible for handling the user interaction and the Declare Engine by acting on the basis of the modeled process. The UI manager, at each interaction event issues queries to the Declare Engine and enables the right ProcessActivity asking the user how to proceed. When the user chooses to execute an activity, the ProcessManager collects the referenced input instances and process variables by wrapping them into a context object, and exposes them to the target activity.

V. A CASE STUDY: A WEB APPLICATION TO RESERVE HOSPITAL VISITS

We evaluated the feasibility, correctness, and effectiveness of our approach through the development of a WA. In order to assess the effectiveness of the approach for developers, we compared the effort needed to develop the WA by adopting the proposed MDE approach with respect the effort required to develop it by using a traditional approach. In both cases, the application was developed using J2EE; the traditional approach uses Drools as workflow engine, whereas the proposed approach is based on Declare. The developed application, called eHospital, is used for the management
of a Dutch hospital [15].

A. Context definition

Here, for the sake of brevity, we just focus on the portion of the eHospital WA that manages the visit reservations from external patients. A patient can ask for a visit through the hospital system and her/his referring doctor can add (or update) her/his case history in the database. These two activities could be executed in any order and the system must be able to match the visit request and the case history through the patient sanitary code. After that, a nurse checks the visit request and could ask the referring doctor to add further details to the patient case history, if needed. When all the required information is available, the nurse approves the visit request sending it to an internal doctor responsible to plan the visit. The doctor is also in charge of choosing the tests needed for the visit (e.g., a lab test, a pre-assessment, X-ray). On the basis of the tests selected by the doctor and considering the availability of the involved resources, the nurse schedules the visit and fixes a possible appointment. The patient can confirm the appointment or can ask to modify it. In this latter case, the resources availability is checked again and a new appointment is scheduled.

B. Declare Process

Figure 4 shows the Declare process model underlying the WA to be developed. In this model, activities are represented as rectangles (e.g., *Request Visit*) and constraints are represented as connectors between activities (e.g., *precedence*). There are dozens of different Declare constraints possible. In the presented model, we use four types of constraints: *response, precedence, succession, absence*. Consider, for instance, the *response* constraint between *Confirm Request* and *Schedule Visit*. This constraint means that “whenever Confirm Request is executed, Schedule Visit is eventually executed”.

Every constraint graphically represented in Declare has a formal back-end grounded in LTL. For instance, the considered *response* constraint can be written in LTL as $\Box (\text{ConfirmRequest} \rightarrow \Diamond \text{ScheduleVisit})$. In a formula like this, it is possible to find traditional logical operators (e.g., implication $\rightarrow$), but also temporal operators characteristic of LTL (e.g., always $\Box$ and eventually $\Diamond$). In general, by using the LTL language it is possible to express constraints relating activities (atoms) through logical operators or temporal operators. The logical operators are: implication ($\rightarrow$), conjunction ($\land$), disjunction ($\lor$) and negation ($\neg$). The main temporal operators are: *always* ($\Box p$, in every future state $p$ holds), *eventually* ($\Diamond p$, in some future state $p$ holds), *next* ($\circ p$, in the next state $p$ holds) and *until* ($p \uhr q$, $p$ holds until $q$ holds).

The *precedence* constraint between *Check Request* and *Confirm Request* means that “*Check Request* can be executed, only if *Confirm Request* has already been executed”. This constraint can be written in LTL as $(\neg \text{ConfirmRequest} \sqcup \Box \text{CheckRequest}) \lor \Box (\neg \text{ConfirmRequest})$. A *succession* constraint is the conjunction of the *precedence* constraint and the *response* constraint. For instance, the *succession* constraint between *Request Visit* and *Check request* means that “whenever *Request Visit* is executed, *Check Request* is eventually executed” and “*Check Request* can be executed, only if *Request visit* has already been executed”. The *absence*
constraint associated to Request visit (labeled with 0..1) means that Request Visit can be executed at most once in a process instance. The LTL semantics for this constraint is  \( \neg \diamond (\text{RequestVisit} \land \Box (\text{RequestVisit})) \).

Each activity in the Declare model is associated to a specific role. In our example, Request visit and Confirm Appointment are associated to Patient, Update Case History is associated to Referring Doctor, Check Request, Ask for Medical Info, Confirm Request, Check Resources Availability, Schedule Visit and Modify Appointment are associated to Nurse, and Choose Tests is associated to Internal Doctor.

C. Web Application Development with M3D

Figure 5 shows an excerpt of the DSL instance describing the information model of the case study.

Figure 6: An excerpt of the DSL instance describing the service model of the case study

Figure 7: An excerpt of the DSL Declare model

given types. These collections are used to express an Association having roles involving multiple objects (such as the aggregation between CaseHistory and MedicalInformation or the composition of the person and Address).

The service model, reported in Figure 6, imports the information model of Figure 5 and defines the operations needed to implement the business logic of our example. For instance, the Visit Entity is defined by specifying its Attribute and Operation classes modeling data and behaviors of the visit conceptual domain (such as the list of visits and the method to compute the resources needed for a visit).

The model defines a DAO for each Entity in the model and introduces a mapping between Service and DAO classes. Each DAO in the model introduces CRUD operations for Entities (for the sake of brevity, the figure only shows retrieving operations) and each ServiceOperation specifies dependencies on them. The model excerpt also shows the definition of three services (VisitService, PatientService and RequestService) used to manage respectively visits, patient information and request management. The VisitService
builds upon the VisitDAO by specifying at modeling time the delegations to the required operation. Both the Service-Operation specifications scheduleVisit and cancelVisit in the model delegate to the updateStatus method of the VisitDAO that allows the operations to change the state of a particular visit. This delegation is taken into account by the generator by using dependency injection of DAO instances.

Figure 7 shows a small excerpt of the Declare DSL model for our case study. The model shows the integration of the Declare DSL model and the information model. Each Event defines its inputs and outputs as scoped variables (that are bounded to the current instance of the Process or the ProcessActivity) or as attributes (of instances of persistent entities accessed through DAOs).

Figure 8 is a screenshot of the resulting eHospital WA. The upper part of the figure shows the navigation bar that is automatically generated from the information model (with a button for each Entity of the model) and injected into each UI activity to allow for navigation: in this case, the application administrator is able to execute CRUD operations on each Entity. The Pending Request Visits button (in the navigation bar) lists the pending Request Visit tasks. The contextual menu shows the activities that the Nurse (the logged in user) is allowed to execute.

D. Effort Comparison with respect to traditional development

To perform a first comparison of the effectiveness of the approach the eHospital, application requirements were given to two developer teams developing the application using:

- **Generic UML modelling** and manual coding and exploiting commercial generic UML tools from industry;
- **M3D approach**, exploiting the proposed approach and our supporting tool.

The teams had the same structure (one designer and two developers) and a comparable level of expertise on the target technologies. To perform the effort comparison we chose a set of metrics adequate to quantify productivity and quality in industrial MDE projects as studied in several empirical studies regarding the impact of MDE on reducing effort and increasing efficiency ([21], [26], [29]). In Table I, the set of metrics we chose to perform the effort and efficiency evaluation is reported.

The data collection has been carried out by monitoring the developers during the development period and on the produced artifacts (models and code). Table II reports the obtained results. The values reported in the table lend themselves to several interesting considerations. First of all, we can discuss model size and complexity. The model size for the traditional scenario is much higher because the designers have to produce more artifacts at different levels (both high and low level). In the MDE case this is not needed, since our approach takes as input a set of PIM models. The level of abstraction of such platform independent UML models (PIMs) is higher and the models are, therefore, simpler and easier to understand and to write. This is also shown by the model complexity metric that is lower for the MDE case.

Still, with respect to the size metrics, using a standard UML tool allows for very few code generation facilities and hence a manual coding is required to produce almost all the source code (98.5% including application metadata). The adoption of the M3D reduces the code to be written to 11.30% of the total, letting the developers to focus on the process logic and the behavior to be customized.

Another main benefit regards the documentation generation. In the traditional approach, often the developers can “skip” the models in order to speed up the coding (leaving the application without up to date documentation).
### Table I: The set of metrics taken into account

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size and Dimension</strong></td>
<td></td>
</tr>
<tr>
<td>Process Model Size (PMS)</td>
<td>Size of the process models as defined in [10]</td>
</tr>
<tr>
<td>Class Model Size (CMS)</td>
<td>Size of the class diagram as defined in [10]</td>
</tr>
<tr>
<td>Number of Diagrams (ND)</td>
<td></td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>Defined excluding white lines but including comments</td>
</tr>
<tr>
<td>Manual Lines of Code (MLOC)</td>
<td>Lines of Code written by developers</td>
</tr>
<tr>
<td>Generated Lines Of Code (GLOC)</td>
<td>Lines of Code emitted by code generators</td>
</tr>
<tr>
<td><strong>Model and Code Complexity</strong></td>
<td></td>
</tr>
<tr>
<td>Model Complexity (MC)</td>
<td>Complexity of the input models [10]</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>Complexity of software classes</td>
</tr>
<tr>
<td>Lack Of Cohesion in Methods (LCOM)</td>
<td>Variant of LCOM defined by Hitz and Montazeri in [11]</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td></td>
</tr>
<tr>
<td>Time to Model (TM)</td>
<td>Time needed to build the design model of the application</td>
</tr>
<tr>
<td>Time to Code (TC)</td>
<td>Time needed to implement the design model of the application</td>
</tr>
</tbody>
</table>

In contrast, the models become the real "source" of the application and the developers are interested to keep them up to date in order to perform the code generation from them. As a consequence, the models always accurately reflect the current implementation.

Note that there is still a part of the application (in our case 11.3% of the total, less than 2.1KLOC) that is written by hand and needs to be understood by the developers. However, since this part is strictly related to the underlying architecture of the target platform, the developers should be aware of how it is structured and of its behavior. This could represent a factor limiting the adoption of our approach since this requires an initially steep learning curve. This issue can be addressed by producing high quality documentation of the reference architecture and example applications to help the developers to be quickly productive.

As far as the complexity of the source code is concerned, the values in Table I (CC,CE and LCOM-HS) suggest that the code generator produces code that is slightly more complex with respect to the one written by hand. This is true for both the max and the average values and is due to the fact that the generation templates are not smart enough to generate code that is as efficient, easy to understand and reduced in size as the one written by hand. On the other hand, we can see a significative difference in the time needed to perform modeling and development phases in the two scenarios. The manual development scenario required almost ten times to perform the coding and four times to perform the application modeling with respect to the MDE approach. Similar considerations about the effort are valid also when maintaining/evolving the WA. We evaluated the impact of changes of some constraints in the process and compared the effort needed to modify the WA with the two different approaches. Due to the paper’s space constraints, here we just report that by using the M3D approach the average effort was about ten times lesser than using the General UML modelling one. The proposed PIMs play a central role here shifting the effort from the developers towards the generator engine.

### Table II: Metric evaluation in the two development scenarios

<table>
<thead>
<tr>
<th>Metric</th>
<th>Traditional Development</th>
<th>M3D Development</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size and Dimension</td>
<td></td>
</tr>
<tr>
<td>ND</td>
<td>1 class diagram</td>
<td>1 entity diagram (class)</td>
</tr>
<tr>
<td></td>
<td>3 activity diagram</td>
<td>1 activity diagram</td>
</tr>
<tr>
<td></td>
<td>6 sequence diagram</td>
<td>1 sequence diagram</td>
</tr>
<tr>
<td>PMS</td>
<td>136</td>
<td>142</td>
</tr>
<tr>
<td>CMS</td>
<td>979</td>
<td>405</td>
</tr>
<tr>
<td>LOC</td>
<td>149</td>
<td>124</td>
</tr>
<tr>
<td>MLOC</td>
<td>13.5%</td>
<td>14.2%</td>
</tr>
<tr>
<td>GLOC</td>
<td>1.2%</td>
<td>68.70%</td>
</tr>
<tr>
<td>Model and Code Complexity (package level, min/max/average)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td>105</td>
<td>228</td>
</tr>
<tr>
<td>CC</td>
<td>1 - 31.13</td>
<td>0.62 - 12.4</td>
</tr>
<tr>
<td>CE</td>
<td>28 - 36</td>
<td>0.05 - 0.59</td>
</tr>
<tr>
<td>LCOM-HS</td>
<td>0.02 - 12.05</td>
<td>0 - 18.42</td>
</tr>
<tr>
<td></td>
<td>Effort (time, hours)</td>
<td></td>
</tr>
<tr>
<td>TM</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>TC</td>
<td>80</td>
<td>30</td>
</tr>
</tbody>
</table>

**VI. CONCLUSION**

The integration of MDE techniques with a workflow language allows us to overrun some typical drawbacks in the development of WAs mainly for WAs used in contexts with high variability. In this paper, we have presented an approach based on the integration of four MDE metamodels representing the main components of a WA and the declarative process modeling language Declare. The integration of different metamodels supports a clear separation of concerns and enables both the reuse of the analysis models and the reuse of the design models in different contexts. Furthermore, exploiting a declarative process modeling language, such as Declare, allows us to represent in a compact way workflows with several alternative paths. A Declare representation allows the workflow designer to describe in a simple and compact way processes where several activities can be performed in different orders. The case study carried out showed the applicability of the approach and underlined some advantages deriving by its adoption. Indeed, the set of the defined metamodels was very effective to model a process-centric WA: only a small ratio of LOC had to be manually customized. This is a valuable advantage for the developers in the implementation and the maintenance of a
REFERENCES


