The Journal of Systems and Software 131 (2017) 505–527

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

CollabRDL: A language to coordinate collaborative reuse

Edson M. Lucas[a,b,*], Toacy C. Oliveira[a,d], Kleinner Farias[c], Paulo S.C. Alencar[d]

[a] PESC/COPPE, Federal University of Rio de Janeiro, Brazil
[b] IPRJ/UERJ, Polytechnic Institute, State University of Rio de Janeiro, Brazil
[c] PIPCA, University of Vale do Rio dos Sinos (Unisinos), Brazil
[d] David Cheriton School of Computer Science, University of Waterloo, Canada

ARTICLE INFO

Article history:
Received 19 June 2015
Revised 17 January 2017
Accepted 31 January 2017
Available online 2 February 2017

Keywords:
Software reuse
Collaboration
Framework
Language
Reuse process

ABSTRACT

Coordinating software reuse activities is a complex problem when considering collaborative software development. This is mainly motivated due to the difficulty in specifying how the artifacts and the knowledge produced in previous projects can be applied in future ones. In addition, modern software systems are developed in group working in separate geographical locations. Therefore, techniques to enrich collaboration on software development are important to improve quality and reduce costs. Unfortunately, the current literature fails to address this problem by overlooking existing reuse techniques. There are many reuse approaches proposed in academia and industry, including Framework Instantiation, Software Product Line, Transformation Chains, and Staged Configuration. But, the current approaches do not support the representation and implementation of collaborative instantiations that involve individual and group roles, the simultaneous performance of multiple activities, restrictions related to concurrency and synchronization of activities, and allocation of activities to reuse actors as a coordination mechanism. These limitations are the main reasons why the Reuse Description Language (RDL) is unable to promote collaborative reuse, i.e., those related to reuse activities in collaborative software development. To overcome these shortcomings, this work, therefore, proposes CollabRDL, a language to coordinate collaborative reuse by providing essential concepts and constructs for allowing group-based reuse activities. For this purpose, we extend RDL by introducing three new commands, including *role, parallel*, and *doparallel*. To evaluate CollabRDL we have conducted a case study in which developer groups performed reuse activities collaboratively to instantiate a mainstream Java framework. The results indicated that CollabRDL was able to represent critical workflow patterns, including parallel split pattern, synchronization pattern, multiple-choice pattern, role-based distribution pattern, and multiple instances with decision at runtime. Overall, we believe that the provision of a new language that supports group-based activities in framework instantiation can help enable software organizations to document their coordinated efforts and achieve the benefits of software mass customization with significantly less development time and effort.

1. Introduction

Modern software systems are developed by people working together, since the complexity of these systems requires knowledge related to numerous fields, including programming languages, human-machine interfaces and databases, which goes beyond the knowledge associated with a single system application domain. In this context, collaboration among people emerges as an important factor for the success of a software project development, and, therefore, tools to support collaborative work are crucially needed (Barthelmess and Anderson, 2002).

Another important aspect pertaining the construction of software systems is Software Reuse (Frakes and Kang, 2005). This concept involves, in part, reusing the knowledge acquired in previous projects during the development of a current project, and can result in higher-quality outcomes and resource savings. For this, some reuse techniques have proposed in the last decades, such as the RDL, a Reuse Description Language (Oliveira et al., 2007). In this scenario, Collaborative Software Reuse (Mendonça et al., 2008) (Noor et al., 2007) combines concepts of collaborative work and software development with those related to reusable artifacts, so that the development process can progress harmoniously (Mohagheghi and Conradi, 2007).

* Corresponding author.
E-mail addresses: edmlucas@cos.ufrj.br, emlucas@iprj.uerj.br (E.M. Lucas), toacy@cos.ufrj.br (T.C. Oliveira), kleinnerfarias@unisinos.br (K. Farias), palencar@uwaterloo.ca (P.S.C. Alencar).

http://dx.doi.org/10.1016/j.jss.2017.01.031
0164-1212/© 2017 Elsevier Inc. All rights reserved.

In order to achieve the full potential of reusing software artifacts, one must embrace Systematic Reuse, which advocates the need of repeatable and (semi-)formal reuse processes (Rothenberger et al., 2003), where reusable artifacts and their associated constraints are known upfront Furthermore, reuse should be planned and coordinated (Malone and Crowston, 1994), and developers must perform the reuse activities configuring reusable artifacts in a collaborative and coordinated way to avoid errors and rework. A key aspect when practicing Systematic Reuse is documentation, and the documentation of a typical collaborative software reuse process needs to describe activities that can be interactive. Thus, multiple executions of the same process can produce different software behavior, as they are consequences of choices and responses resulting from interactive activities. Therefore, the documentation can support building software with different characteristics for the same domain, e.g., when a team decides to reuse the framework Portlet to build Web-based systems (Bellas, 2004) (Hepper, 2008).

In addition, Oliveira et al. propose a Reuse Description Language (RDL) for describing reuse processes and minimizing the problems associated with the instantiation of object-oriented frameworks (Oliveira et al., 2007). RDL is a textual and executable language, allowing the representation of reuse activities organized as a reuse process. RDL is also an interactive language. As a result, the RDL runtime environment prompts reusers during the framework instantiation process, to gather application-specific information (Oliveira et al., 2007, 2011).

Although RDL is effective for representing reuse activities, it falters when a collaborative reuse process is needed, a common scenario in software development projects. Today, a program in RDL expresses a sequential reuse process typically representing a single reuser, and is therefore unsuitable for complex reuse situations (Oliveira et al., 2007, 2011). The key problems are that RDL is (1) imprecise for specifying the interplay between the reuse activities, (2) inefficient for allowing developers to create working groups based on the available critical skills and responsibilities, and (3) ineffective for specifying how different working groups should perform distinct reuse activities collaboratively and in parallel. Hence, developers end up being unable to use the RDL constructs to support a systematic collaborative reuse process, especially when parallel activities performed by specific working groups need to be represented.

This paper, therefore, extends RDL towards supporting collaborative reuse activities. The extension, which leads to a new language called CollabRDL, involves the definition of three commands: (1) *Role* allows assigning reuse activities to working groups; (2) *Parallel* allows modularizing a set of commands that can be simultaneously performed; and (3) *Doparallel* allows performing blocks of commands concurrently. These commands were selected for our collaborative reuse extension for three reasons.

First, to promote a systematic, collaborative reuse in a coordinated way in RDL, developers must be able to allocate reuse activities to development team members considering their skills and responsibilities. Moreover, the current literature (e.g., (De Paoli and Tisato, 1994) (OASIS, 2006) (BPMN, 2011) (Cortes and Mishra, 1996) (Li and Muntz, 2000) (Briggs et al., 2003) (Fuks et al., 2007)) highlights that collaborative languages must allow associating activities to working groups; otherwise, the collaboration in development teams can be compromised. In fact, the Communication, Coordination and Cooperation (3C) model, proposed in (Ellis et al., 1991), refers to the activity-aware software development as a way to generate context for the execution of activities based on the understanding of activities performed by other developers.

Second, an ever-present need in collaborative software development is the concurrent execution of activities. For this, developers need to carefully define upfront which activities may be performed in parallel. Unfortunately, these definitions are usually done based on several mentally held indicators (a.k.a. experience) of developers, or even by personal communication (i.e., informally). This may transform the modularization of commands that can be run together into an informal but error-prone task.

Third, the current version of the RDL cannot determine how blocks of activities must be performed in parallel, despite its capacity to represent the sequential behavior, for example, using a traditional Loop command. Today, pivotal concepts (e.g., coordination) for supporting the simultaneity and synchronization of activities are still lacking. Hence, the RDL fails to reach the required coordination of modularized blocks of activities so as to enable them to work together effectively. Therefore, we argue that these commands are necessary and sufficient to support the broader collaboration issues found in RDL. We make also no claims about the generality of the proposed commands beyond collaboration in the context of RDL programs.

In addition, these new commands overcome critical problems, which include the inability of specifying how the produced artifacts and the acquired knowledge should be reused and the lack of constructs for describing how the reuse activities should be performed by groups of developers asynchronously or in parallel. These commands are fully supported by a runtime environment based on the workflow and Business Processes Manager (BPM), which provides facilities to load, start and run processes, besides allowing workflow functionalities. We chose Business Process Model and Notation (BPMN) to CollabRDL environment because BPMN is a pattern in the context of BPM that has been maintained by the Object Management Group (OMG). Furthermore, there are many environments offering support to BPMN (BPMN, 2016). Our initial evaluation has shown that the proposed commands are effective by using them to represent a set of well-established workflow patterns and performing a realistic case study in which two working groups collaboratively performed reuse activities for instantiating a mainstream Java framework, OpenSwing (OpenSwing, 2015). In total, the reuse process led to the creation of 49 attributes and the redefinition of 90 methods.

The remainder of this paper is organized as follows. Section 2 briefly introduces the concepts of collaboration, reuse process and describes RDL. Section 3 presents CollabRDL as an extension of RDL and describes its commands. Section 4 presents an environment for running reuse processes expressed in CollabRDL. Section 5 presents our evaluation of CollabRDL. Section 6 reviews related work and, lastly, Section 7 concludes our paper with a final discussion and a brief description of future work.

2. Background

CollabRDL aims at defining a collaborative reuse process. As a result, the following subsections briefly introduce the main concepts used in this work, such as Collaboration, Reuse Processes and RDL.

2.1. Collaboration

Collaboration is a kind of cooperation where interactions between people must take place in an organized and planned manner to achieve a common goal. Fuks et al. (2007) explore the 3C model that was originally presented by Ellis et al. (1991). The 3C model is composed of Communication, Coordination and Cooperation. Coordination makes the link between communication and cooperation in order to promote collaboration. Cooperation is a set of operations during a session in a shared working environment in the context of groupware (Ellis et al., 1991). In this sense, the 3C model emphasizes the importance of awareness, defined by

Dourish and Bellotti (1992) as *an understanding of the activities of others, which provides a context for your own activity*.

2.2. Software processes and reuse

Fuggetta (2000) defines software process as a coherent set of policies, organizational structures, technologies, procedures and artifacts that are needed to design, develop, deploy and keep up software. Therefore, there is a need to define mechanisms for people to collaborate harmoniously to achieve the goals of the software construction process. For example, in a development project, several activities need to be delegated to professionals with specific skills, such as database administrator, programmers and analysts, called roles. This approach does not change when the development is through the reuse of already developed software to be reused, which is the case of frameworks.

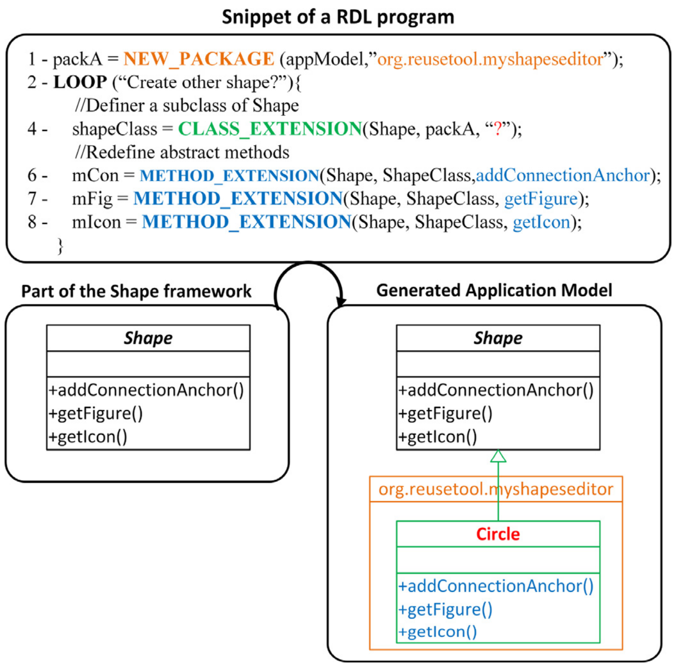
Software Reuse is an important practice in the software development process and refers to the use of existing reliable software artifacts or software knowledge to build new software systems (Frakes and Kang, 2005). Experimental studies indicate that software reuse helps to improve software quality and increasing of productivity. To illustrate, in (Lim, 1994), two case studies were conducted: the first reduced by 51% defects (quality improvement) and increased productivity by 57%, and the second got 24% in reducing defects and 40% increasing productivity. However, other studies were not so conclusive (Frakes and Succi, 2001).

Framework instantiation is a type of software reuse. A framework is a comprehensive application that can be reused to produce custom applications (Mohamed and Schmidt, 1997). In the early stages of research on frameworks, this paradigm was based on object-oriented with Smalltalk language, Lisp and C++ and led to a wide range of applications, including some involving graphical user interfaces and compilers (Johnson and Foote, 1988). Historically, the reuse process of Model-View-Controller framework is described through CookBook (Krasner and Pope, 1988), using natural language to document the key steps for its instantiation. The notion of hook was defined by Froehlich et al. (1997) and can be understood as an evolution of CookBook, having a structured description such as a name that identifies the hook, type, and condition, while still keeping the natural language descriptions.

The Software Reuse Process is presented with two goals, developing for reuse and development with reuse (De Almeida et al., 2005). The first goal focuses on the development of reusable assets (Arango and Prieto-Diaz, 1991) and the second on the development of applications using reusable assets, also known as application engineering. Initially and until 1998, reuse was focused on domain engineering (De Almeida et al., 2005). After this period, the focus shifted to Software Product Lines, SPLs. An SPL is a set of systems that share common and manageable features that meet the needs of a particular market segment or mission and are developed from a shared set of core assets in a predetermined way (Northrop, 2002).

2.3. RDL

RDL, which is the acronym for Reuse Description Language, is a language used to represent the process of object-oriented framework instantiation and support software reuse, which aims at explicitly representing the processes that specify reuse activities (Oliveira et al., 2007). RDL is also an interactive language in which the reuser, who performs the instantiation of a framework, needs to interact with the system by answering questions related to the reuse activities. For example, the execution of an RDL command could prompt the question, “What is the class name?”, and the reuser can answer this question by providing a class name:



```
1 - packA = NEW_PACKAGE (appModel,"org.reusetool.myshapeseditor");
2 - LOOP ("Create other shape?"){
       //Definer a subclass of Shape
4 -    shapeClass = CLASS_EXTENSION(Shape, packA, "?");
       //Redefine abstract methods
6 -    mCon = METHOD_EXTENSION(Shape, ShapeClass,addConnectionAnchor);
7 -    mFig = METHOD_EXTENSION(Shape, ShapeClass, getFigure);
8 -    mIcon = METHOD_EXTENSION(Shape, ShapeClass, getIcon);
    }
```

Fig. 1. Extension to reuse an object-oriented framework modeled in UML.

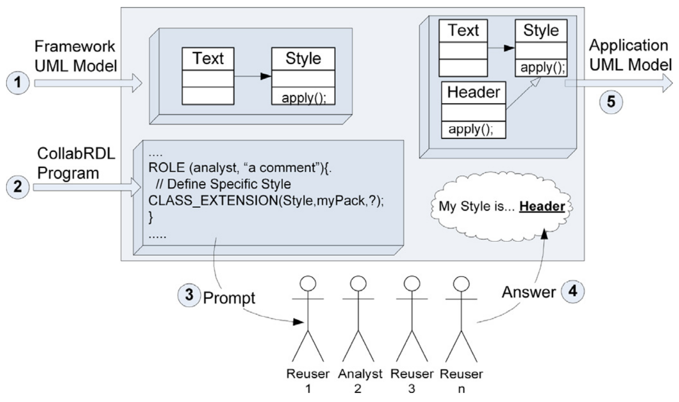
“Person”. The RDL programs can be executed in an environment so-called ReuseTool (Oliveira et al., 2011).

Fig. 1 shows a reuse extension of an object-oriented framework modeled in Unified Modeling Language (UML). This example is based on the framework Shape, which is described in detail in (Oliveira et al., 2011). The result of this reuse extension is *Circle* class, which expands the *Shape* class by redefining its methods—*addConnectionAnchor, getFigure,* and *getIcon—and leads to* a new package called *org.reusetool.myshapeseditor*.

Fig. 1 also shows a snippet of a program in RDL that guides this specific reuse process. For example, line 1 describes the creation of a package, *packA*, and line 2 indicates the beginning of a block of activities that can be repeated at runtime as many times the reuser needs. Thus, when reusers execute this line, they will be prompted the question: Create another shape? If the question is answered, the program will advance to the LOOP command block. Next, line 4 describes the creation of a new class that extends *Shape*. For this purpose, the superclass, Shape, is specified, as well as the package, *packA*, into which it will be inserted. Moreover, the parameter “?” indicates that, at runtime, the reuser will be prompted a question and needs to provide the name of the new class. In this example, the reuser chose the class name *Circle*. Finally, lines 6, 7 and 8 redefine the *addConnectionAnchor, getFigure, getIcon* methods of the parent class Shape in the class pointed to by the variable *shapeClass*.

It is noteworthy that many executions of the snippet of the program in RDL in Fig. 1 can generate several different models because there are two descriptions that may vary from execution to execution, thus giving rise to different results. The first is related to the number of classes that can be generated from the extension of Shape, and this is due to the LOOP command, as it allows the reuser, at runtime, to define the number of classes that extend the Shape class. Moreover, the second description involves choosing the names of the new classes that inherit from Shape.

RDL has two well-defined phases, a Development Phase that has an RDL program as a product, and an Execution Phase, where the products are application models in UML. The RDL program can be seen as an execution plan for a Software Product Line whose domain is the framework, and the products are applications in the



```
....
ROLE (analyst, "a comment"){.
  // Define Specific Style
CLASS_EXTENSION(Style,myPack,?);
}
.....
```

Fig. 2. CollabRDL reuse process overview, adapted of Oliveira et al. (2011).

domain. In the Development Phase, expert reusers of the framework, using documentation and examples of applications, write the RDL program to guide reuse process. A program expressed in RDL defines the extension points of the framework, requesting, when necessary, user decisions on questions about optional and alternative extensions involving situations based on a feature model (Filho et al., 2004).

The Execution Phase guides step-by-step activities of a specific framework reuse through an RDL program to generate an application. Reusers do not need to be as knowledgeable about the framework as those involved in the development of an RDL, but they need to have some knowledge about the framework, and access to its documentation and to the specification of the application to be generated. As described in (Mendonça et al., 2008) (Noor et al., 2007) the reuse process can be performed by several people aiming to build software with better quality and shorter duration. However, the current version of RDL describes reuse as a single-user form, not allowing multiple reusers to execute snippets of RDL in the same framework reuse. In other words, there are no specification mechanisms to express tasks to be performed in parallel, and connect people with specific skills with certain tasks. In this context, the definition of an extension of RDL, so-called CollabRDL, is highly needed because it will support pivotal concepts and mechanisms related to collaboration and execution of tasks in parallel.

3. CollabRDL

This Section presents an overview of CollabRDL and focuses on three coordination-oriented commands, namely *role, parallel,* and *doparallel*.

3.1. CollabRDL overview

As previously mentioned, CollabRDL extends RDL to support collaborative reuse activities. Fig. 2 shows an overview illustrating how the proposed extension changes the traditional reuse process in RDL by introducing, in contrast with RDL, which assumes a single reuser, a set of reusers who will execute group-related activities. Moreover, Fig. 2 shows how the reusers are engaged in a five-step collaborative reuse process.

First, the reusers provide UML models of a particular framework to be instantiated. Typically, these input models have some classes, along with their methods, that represent the main concepts and behavior that will be reused, e.g., the classes *Text* and *Style*, and their methods *apply()*.

Then, as a second step, a CollabRDL program is provided, which involves both the traditional RDL commands and new commands, including ROLE, PARALLEL, and DOPARALLEL. Next, the reusers interact with each other to provide a particular instance of the input framework models. It is important to highlight that this collaboration cannot be undertaken by using the existing RDL language. Finally, an output application is generated.

To illustrate how an RDL Script can be created, Fig. 3 shows the steps that a CollabRDL developer can follow to create a CollabRDL program. The first step is to “identify reuse activities,” in which reusers consult the model of the framework that will be instantiated and its documentation and examples, as well as identify the reuse activities by using the strategy of delimitation of the framework regions (e.g., the Model, View and Control pattern) or features (i.e., the description of use cases). The second step is to “identify the activities that can be performed in parallel,” taking into account their interdependencies.

The main guideline is to create a table to help in the organization of parallelism among activities. In this example, the activities 1, 2, and 3 can be executed in parallel, just after the activities 9 and 10 are executed in parallel with activities 11 and 12. The third step is to "define reuse groups" by taking into account the required skills and responsibilities to carry out the activities identified in the first step. The fourth step, "Assign activities to groups", is achieved when all activities identified in the first step are associated with a group, which was defined in the third step. Finally, the fifth step, "Express in CollabRDL", involves describing the identified activities, considering their parallel executions and their associations with groups using legacy RDL commands and the new CollabRDL commands.

3.2. CollabRDL commands

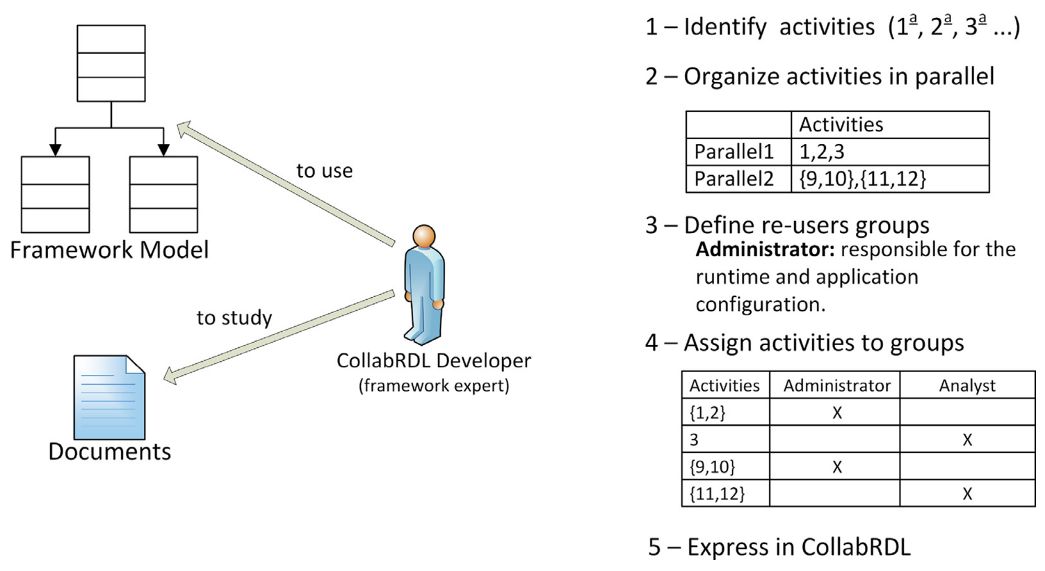
CollabRDL must support reusers in multiple groups who work in parallel throughout the reuse process. The underlying assumption is that the language should be able to support coordination since individual efforts need to be organized and coordinated to produce a result. Coordination theory states that coordination is a set of principles about how activities can be jointly organized and implemented through multiple group efforts (Malone and Crowston, 1990). In this context, CollabRDL must allow developers to assign tasks to groups and represent parallel activities.

The new proposed commands must be consistent with the existing RDL structure. RDL follows the paradigm of imperative programming languages (Jouault and Kurtev, 2006). Code 1 shows a snippet of a typical RDL program. In line 1, the NEW_PACKAGE command precedes CLASS_EXTENSION command in line 2, and is represented textually in the line ending with “;”. Thus, it is guaranteed that the CLASS_EXTENSION command will be initiated only after the completion of NEW_PACKAGE command. The CLASS_EXTENSION command uses the result of NEW_PACKAGE, a reference to the created package (*packA*). In this case, the NEW_PACKAGE command is a pre-requisite for CLASS_EXTENSION.

The existing set of RDL commands (Oliveira et al., 2007) is complemented with the *role, parallel*, and *doparallel* commands in order to define the first version of CollabRDL, which allows developers to represent parallel activities and assign them to groups of reusers. On the other hand, at runtime, it helps reusers to coordinate reuse activities in teams. The new CollabRDL commands will be introduced in order, from general to specific, in the following sections by: a) explaining the command goals; b) present an example to illustrate use scenarios for the commands; and, in Appendix A, providing the Backus Normal Form (BNF) representation that defines the commands.

3.2.1. ROLE command

CollabRDL uses the concept of roles to associate activities to groups. Role is a human-oriented construct; it relies on the fact that in general a person can interact in different ways in


Framework Model
to use
to study
CollabRDL Developer
(framework expert)
Documents
1 – Identify activities (1ª, 2ª, 3ª ...)
2 – Organize activities in parallel
3 – Define re-users groups
Administrator: responsible for the runtime and application configuration.
4 – Assign activities to groups
5 – Express in CollabRDL

	Activities
Parallel1	1,2,3
Parallel2	{9,10},{11,12}

Activities	Administrator	Analyst
{1,2}	X	
3		X
{9,10}	X	
{11,12}		X

Fig. 3. Steps to create a CollabRDL program.

```
packA = NEW_PACKAGE(appModel, "my.domain.pachName");
shapeClass = CLASS_EXTENSION (Shape, packA,"?");
METHOD_EXTENSION (Shape,shapeClass, addConnectionAnchor);
IF ("Add Feature - Restrict Connections?") THEN {
    m = METHOD_EXTENSION(Shape,shapeClass,canConnect);
}
```

Code 1. The snippet of an RDL program. The Shape framework was described in detail in Oliveira et al. (2011).

```
ROLE (analyst, "Only two-dimensional figures are allowed!"){
  shapeClass = CLASS_EXTENSION(Shape, packA,"?");
  m =  METHOD_EXTENSION(Shape,shapeClass,addConnectionAnchor);
  ADD_CODE(shapeClass,m,"return new ChopboxAnchor(iFigure);");
  m = METHOD_EXTENSION(Shape,shapeClass,getFigure);
  ADD_CODE(shapeClass,m,"return new IFIGURE_SUBCLASS);");
  m = METHOD_EXTENSION(Shape,shapeClass,getIcon);
  ADD_CODE(shapeClass,m,"return createImage(\"NEW_SHAPE.gif\");");
     ...
}
```

Code 2. ROLE command.

the society (Smith et al., 1998). Role is also defined as how an entity participates in a relationship (Uschold et al., 1998). Further, languages that can be used to represent collaboration mechanisms often implement roles (De Paoli and Tisato, 1994) (OASIS, 2006) (BPMN, 2011) (Cortes and Mishra, 1996) (Li and Muntz, 2000) (Briggs et al., 2003).

Code 2 shows the syntax of the new ROLE command in CollabRDL. This command is used to assign activities to groups. Thus, at runtime, all activities that need user intervention and are in the block bounded by braces shall be delegated to reusers in the *analyst* group shown as the first parameter, and the information " Only two-dimensional figures are allowed!" is provided as an output. In this way, a member of the analysis group can be assigned to execute the next activity.

3.2.2. PARALLEL command

Code 3 shows the PARALLEL command used to group blocks of commands that can be executed in parallel. This command implements a set of restrictions related to concurrency, synchronization of activities, and allocates activities to actors as a coordination element. The first FLOW block indicates that the reuser (coordinator) will delegate interactive activities, which are shown in braces

```
PARALLEL {
    FLOW (analyst, “A comment!”){
        A;
    }
    FLOW (designer, “Other comment!”){
        B;
    }
}
```

Code 3. PARALLEL command.

```
LOOP (“Repeat?”){
    ROLE (analyst, “A comment”){
        A;
...
    }
}
```

Code 4. The combination of the LOOP command with ROLE command.

("{" and "}") to the *analyst* group, and that a message with specific instructions is provided through the second parameter, such as in the case of the ROLE command.

Moreover, without waiting for someone else in the *analyst* group to perform the activities in the first FLOW block, the reuser (coordinator) may delegate activities in the second FLOW block to a group of designers. The right-hand braces that close the PARALLEL command ("}") indicate that the reuser wait for the execution of all activities in both FLOW blocks to finish in order to proceed with the reuse process. The PARALLEL command allows two or more FLOW blocks.

Code 4 shows the combination of the LOOP command with new ROLE CollabRDL command. This combination will generate ROLE blocks for sequential execution. First, a reuser must answer *yes* to the question “Repeat?” and wait for someone from the *analyst* group to perform all interactive activities that are in the ROLE block. Only after the completion of these activities, the question “Repeat?” will be issued again. If the answer is positive, the ROLE block will be again delegated to the *analyst* group and, as in the first iteration, this group is expected to carry out their activities so that the question in the LOOP is presented once again.

3.2.3. DOPARALLEL command

In other situations, in contrast with the sequential behavior represented by LOOP command, there is a need to execute blocks

```
DOPARALLEL{
    ROLE (analyst, "A comment."){
        A;
    }
}WHILE ("Run the block again?");
```

Code 5. DOPARALLEL command.

in parallel. The DOPARALLEL command represents this behavior by implementing restrictions on the simultaneity and synchronization of activities as a coordination element.

Code 5 expresses that, at runtime, an activity *A* will be assigned to an *analyst* group and without waiting for its end, the question "Run the block again?" will be displayed. If the answer is positive, another instance of activity *A* will be assigned to the *analyst* group; otherwise, the flow will move to the end of DOPARALLEL, the semicolon (";"), which will wait for the completion of all instances of activity included in the DOPARALEL command block which, in this example, are instances of the activity *A*.

4. The CollabRDL runtime environment

The runtime environment for process reuse expressed in CollabRDL should provide facilities to load, start and run processes, besides supporting basic workflow functionality (WFMC, 1999). In the context of BPMN, there are more than 70 environments offering these features (BPMN, 2016). Therefore, if we convert a CollabRDL program to BPMN, we will take advantage of these features. Our choice was the *Activiti* environment (Activiti, 2015) because it is free and open source, and also uses, as in the case of the RDL core, Java technology.

The runtime *CollabRDL-Activiti-Explorer* is based on the workflow and *Activiti* business processes manager (Activiti, 2015). It uses the Activiti environment and functionality to manipulate the RDL core artifacts in UML. To use this execution environment, we need to convert the process described in CollabRDL to XML format with Activiti-BPMN markings.

Fig. 4 shows how to use the *CollabRDL-Activiti-Explorer* environment to run a CollabRDL program. The first step converts the program written in CollabRDL to BPMN-Activiti using the *RDL.BPMN-Activiti* package. The interactive activities are converted into *UserTask* type activities and the non-interactive into *serviceTask*. *UserTask* type activities are those that will be defined by reusers (people) and *serviceTasks* are performed automatically at runtime (BPMN 2011). The *UserTasks* need the classes that implement the *TaskListener* interface to delegate the execution of activities to be reused to the *RDL-core* and *RDL.Artifact-UML* packages, since the *serviceTasks* need classes that implement *JavaDelegate*.

Step 2 loads the BPMN-Activiti program converted into Activiti-Explorer, including the *RDL-core* and *RDL.Artifact-UML* packages. The Activiti-Explorer is a system developed for Web platforms that allows creation of users and groups using their Manager tab, with start and end processes through the Process tab. The activities delegated to CollabRDL by ROLE or FLOW at runtime will appear in a separate group, the *Queued* menu item, and the activities to be performed by who that started the process will appear in the *Involved* item.

5. Evaluation

RDL has been evaluated with illustrative examples to instantiate real object-oriented frameworks: Shape (Oliveira et al., 2011); DTFrame (Oliveira et al., 2007); REMF (Mendonça et al., 2005); HotDraw (Oliveira et al., 2004); and GEF (Kovalski, 2005). These examples show how RDL programs can systematically describe how to tailor a specific software design based on a framework. For example, in Oliveira et al. (2011) the *ShapesProducts* RDL program when executed builds a new software design using the Shape framework. In addition, an Exploratory Case Study (Runeson, 2009) evaluated the ReuseTool, an environment to run RDL programs (Oliveira et al., 2011). The research question was: "Is the ReuseTool approach capable of representing and executing the instantiation process (RDL programs) for diverse frameworks?". The Study involved one researcher and 3 subjects with good object-oriented programming, JAVA and modeling skills and a mix of academic and industry backgrounds. Each subject chose one or more object-oriented frameworks from literature or self-made and created RDL programs. The result was five RDL programs for five frameworks, and the ReuseTool created successful frameworks instances for all five frameworks: Shapes (GEF, 2015); JHotDraw (HotDraw, 2015); Ecommerce Product Line (Gomaa, 2004); Arcade Product Line (APL, 2015); and REMF (Oliveira et al., 2007).

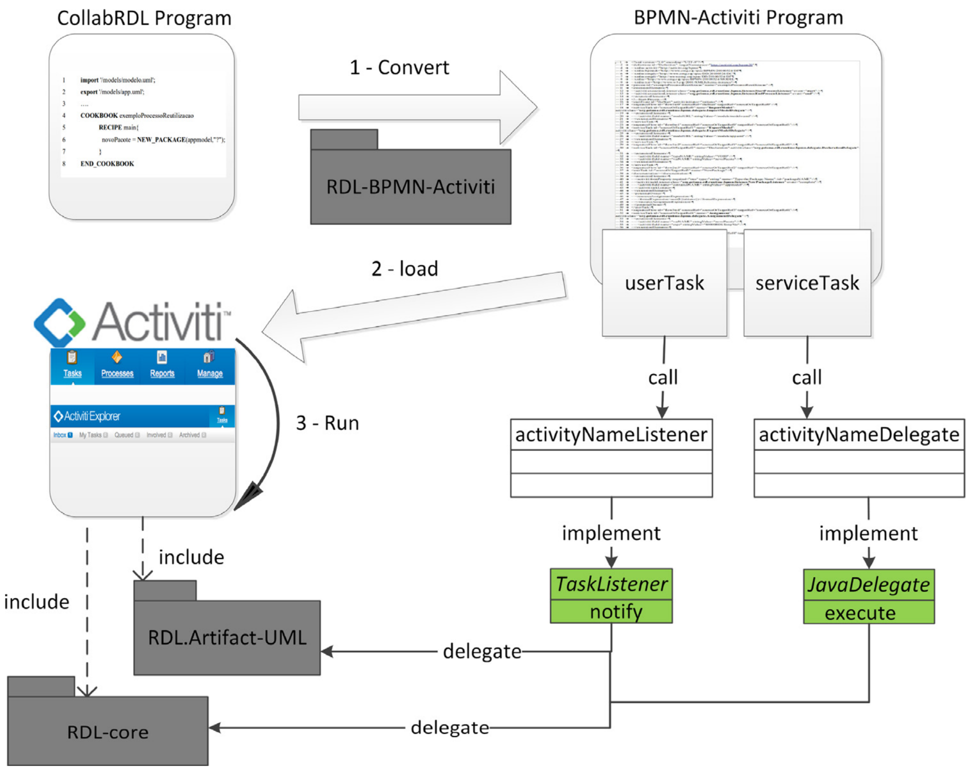
CollabRDL has been evaluated by assessing its ability to represent specific workflow patterns. Workflow systems can be seen as systems that help organizations specify, execute, monitor, and coordinate the flow of work cases within a distributed office environment (Bull Corporation, 1992). A workflow typically has two stages. The first stage focuses on representing the work process as well as its modeling and highlighting the workflow activities. The second stage is the implementation, where the represented process, once executed, generates the work that will result in a product or service. Although most processes and activities can be modeled, some activities need to involve human interaction due to their nature. Thus, as a result, these interactive activities have to be performed outside the computerized environment.

In general, workflow patterns were created so that the fundamental requirements that arise during business process modeling on a recurring basis could be captured, and these requirements could be described in an imperative form (Van Der Aalst et al., 2003). Van Der Aalst et al. proposed 20 workflow patterns, and a few years later, Russell et al. (2006) reviewed this work. These patterns are used to compare workflow tools, indicating whether or not the tool implements certain standards (Wohed et al., 2009; EWP, 2015) (CPE, 2015) (OSPE, 2015). For this evaluation, we will focus on seven patterns, which relate to coordination and teamwork, and present their representation in CollabRDL.

We have assessed CollabRDL with respect to the following coordination-oriented patterns: Parallel Split Pattern (WCP2), Synchronization Pattern (WCP3), Exclusive Choice Pattern (WCP4), Simple Merge Pattern (WCP5), Multi-Choice Pattern (WCP6), Role-based Distribution Pattern (WRP2), and Multiple instances with decision at runtime (WCP14). In Appendix B, we show how these patterns are represented in CollabRDL. Next, CollabRDL was evaluated by conducting a case study in which developer groups performed reuse activities collaboratively to instantiate a mainstream Java framework.

5.1. Evaluating CollabRDL

This section presents a realistic case study conducted in order to evaluate CollabRDL in which two working groups collaboratively performed reuse activities for instantiating a mainstream Java framework, OpenSwing (OpenSwing, 2015). In this context the commands ROLE, PARALLEL and DOPARALLEL are evaluated with respect to their functionality in order to realize the reuse activities in team and in parallel when possible. The questions for this objective are 1 – Can the ROLE command offer a block of activities to a group? 2 – Can the PARALLEL command create instances of activities block for running in parallel? 3 – Can the DOPARALLEL command create threads of the same block of activities? The following metrics are used to answer these questions: 1 – Number



```
import '/models/modelo.uml';
export '/models/app.uml';
....
COOKBOOK exemploProcessoReutilizacao
          RECIPE main{
           novoPacote = NEW_PACKAGE(appmodel,"?");
          }
END_COOKBOOK
```

Fig. 4. CollabRDL programs run in the CollabRDL-Activiti-Explorer environment.

of activities contained in activities block; 2 – Number of roles; 3 – Number of flows in parallel.

This study uses the OpenSwing framework (Section 5.1.1) and can be described in 3 steps: 1- A model was created to represent a target application (Section 5.1.2); 2 - A program was created to generate CollabRDL applications for this domain (Section 5.1.3); 3 - A trial was conducted using the execution environment CollabRDL-Activiti-Explorer with two physically distant working groups to generate the target application, described in detail in Section 5.1.4. Section 5.1.5 presents the results and the threats to validity are reported in Section 5.1.6.

This case study was designed this way because the new CollabRDL commands are of the type workflow. Moreover, applications in OpenSwing involve the participation of people with different skills and responsibilities and, therefore, these applications are suitable for evaluating aspects related to collaboration.

The next section shows the use of these commands in the instantiation of an object-oriented framework, thus showing that they are needed in a specific reuse case. However, we believe that this type of structure is also useful in describing the reuse of generic artifacts.

5.1.1. *The openSwing framework*

Swing is a library of graphical components for Java applications (Swing, 2016). OpenSwing (OpenSwing, 2015) in turn offers new components based on Swing and is also a framework that supports the development of customized applications using the MVC (Model, View and Control) paradigm. Using OpenSwing is possible to develop desktop-like applications for the Internet in three layers, that is, besides Swing, (1) Java, (2) Hypertext Transfer Protocol (HTTP) Servlets and (3) databases, or distributed applications in three layers, that is, besides Swing, (1) Remote Method Invocation (RMI), (2) Java Beans and (3) databases.

OpenSwing supports MDI (Multiple Document Interface) that can contain several other screen structures. The MDI interface allows the customization of many features, such as title, menu language selection, authentication, and windows menu. Our case study is based on the OpenSwing Demo 10 (Tutorial OpenSwing, 2016), which describes the creation of an MDI application server client. This example implements the functionality of registration department, employees and tasks.

The OpenSwing framework allows the creation of complete applications based on the MVC architecture, and is organized to facilitate the distribution of reuse activities between teams with different profiles/responsibilities such as *analyst, designer, database administrator* so on. As a result, block independent activities can be identified so that they are executed in parallel, with the goal of reducing the total time to create the application. Therefore, this framework is suitable for use in a scenario of collaborative reuse.

Fig. 5 shows the class model in UML for the OpenSwing Demo10 (Tutorial OpenSwing, 2016). The classes are marked with "from demo10". We omitted the attributes and methods of the OpenSwing classes and interfaces, and attributes of Demo10 to facilitate the visualization of the model. The model shows the interface implementations, inheritance and dependencies between classes. As an example, the *ClientApplication* class implements *LoginController* interface to add authentication and authorization mechanisms to the application, and the *MDIController* to add *MDI features*.

The *DemoClientFacade* class is instantiated by *ClientApplication* and through their methods the control classes are instantiated. Control classes must inherit from *GridController* and implement the *GridDataLocator* interface to control the actions in the windows, linking the data layer to the view layer. All classes that inherit from *InternalFrame* do it to get the behavior of windows to interact with the user. The classes that are mapped to the database need to inherit from *ValueObjectImpl* directly, as in the case of *GridEmpVO*, or indirectly, as in the case of *EmpVO*.

Table 1 shows the extension points of Demo10. The *GridController* class is the most complex extension. It has two attributes

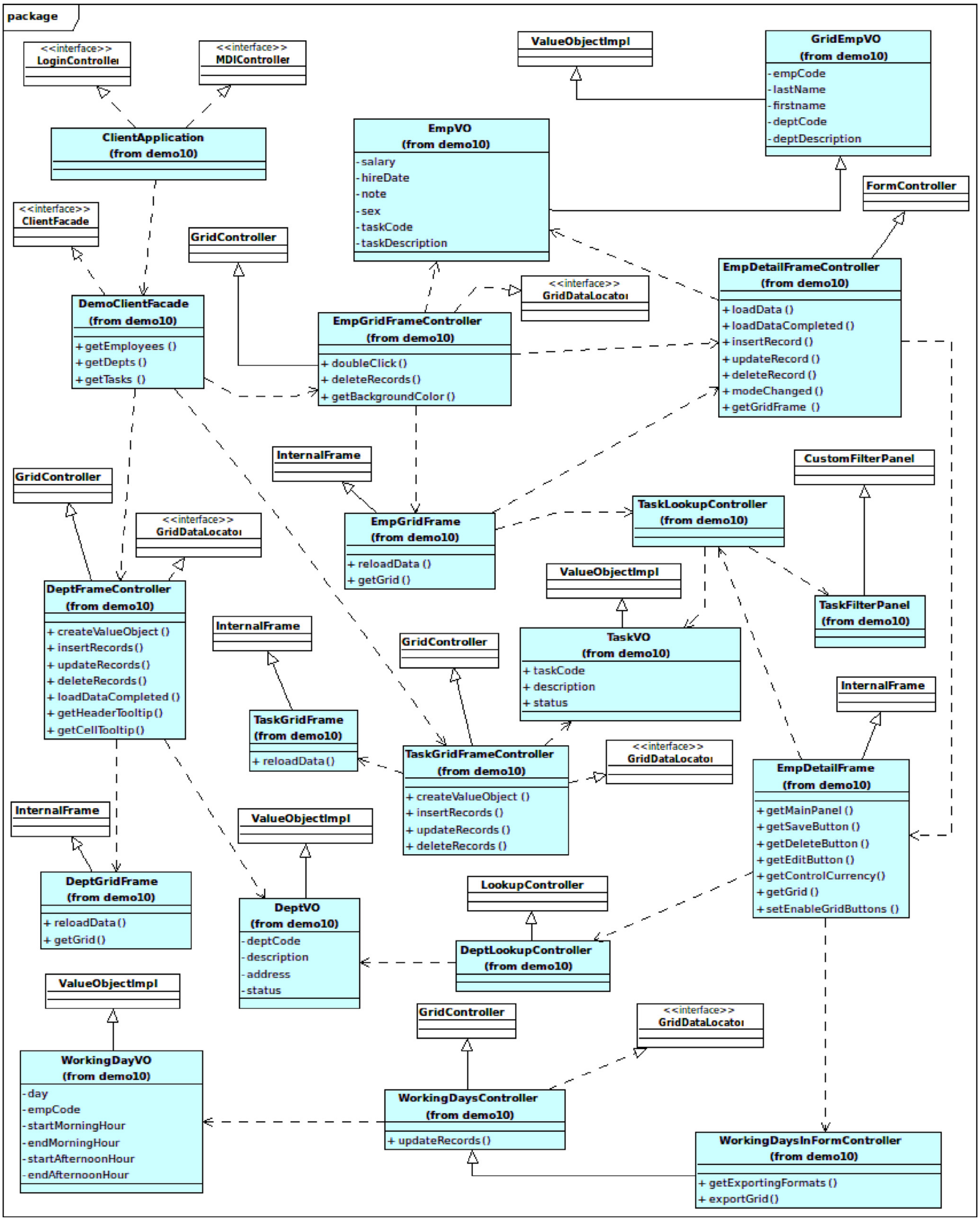


Fig. 5. The UML Class Model of the Openswing Demo10 framework.

Table 1
Extension points of Demo10.

Extension Points	Attributes	Methods*
ClientFacade	0	0
CustomFilterPanel	0	7
GridController	2	48
GridDataLocator	0	1
InternalFrame	9	2
LoginController	0	4
LookupController	38	11
MDIController	0	16
ValueObjectImpl	0	1
Total	**49**	**90**

* Methods that can be redefined.

and 48 methods, which can be redefined. The *MDIController* interface presents the need to implement 16 methods. *ClientFacade* interface is created only to serve as a super type, so it does not define services. The total of all the attributes of extension points is 49 and there are 90 methods that can be redefined. These numbers do not include inherited attributes and methods. They indicate the complexity of reuse, which involves attributes are associated with data handling and methods of application behavior. Moreover, in general an application to implement specific requirements usually does it through various control, model and view classes.

5.1.2. Instantiating a new application

The new application will have all Demo10 classes, except the *WorkingDayVO, WorkingDaysControlller, WorkingDaysInFormController,* and *TaskFilterPanel* classes. The construction of this application model will be guided by a CollabRDL program that takes in to account group development and explores as much as possible the performance of activities in parallel. Fig. 6 shows the UML model checking classes that are intended to be built by the reusers group. The *ClientApplication* and *DemoClientFacade* classes, labeled with the letter I on the left, should be built by who initiates the instantiation process, a person who needs to be knowledgeable about the framework and can delegate tasks to reusers. The *EmpGridFrame, TaskGridFrame, DeptGridFrame* and *EmpDetailFrame* classes, labeled with the letter D on the left, should be offered to the *designer* group. The control classes labeled by the letter A are associated with the *analyst* group.

Note that Demo10 is organized according to the MVC (Model, View and Control classes) paradigm. As a result, the new application also will follow this organization. In this way, specific tasks may be re-arranged according to the level of expertise of the groups in these areas of the framework. This helps the framework to be instantiated by groups in a collaborative way.

5.1.3. The CollabRDL instantiation program

A CollabRDL program was written to guide the process of collectively creating new applications for Demo10 making use of the new ROLE, PARALLEL and DOPARALLEL commands (CollabRDL Page,2015). This program does not include the creation of *WorkingDayVO, WorkingDaysControlller, WorkingDaysInFormController* and *TaskFilterPanel* classes because they are not required to generate the application of this case study. Table 2 shows the CollabRDL program created for this case study in numbers. We considered as activities all commands that add some information to the application model including, for example, NEW_REALIZATION indicating that a class will implement an interface, and interactive activities that need the user to enter some information that should be added to the model, as in the case of the CLASS_EXTENSION command. Thus, in order to have a meaningful comparison, we do not count the IF statements as an interactive activity, since they only call for a yes or no response from the user, but, because of their importance, we created a column to explain this type of interaction. Moreover, the FLOW command that is part of the PARALLEL command produces the same result as the result of the ROLE command, delegating activities to groups. However, we did not count them separately.

Table 2 shows that the greatest effort to instantiate applications based on the Demo10 domain is concentrated in the execution of block activities from the DOPARALLEL command. These amounts to 70 activities, with 5 being interactive and contain 53 IFs, with 20 of these nested, therefore requiring much reuser intervention in the decision whether or not to add items to the application model. Note that the count of the IFs in ROLE command is the same for DOPARALLEL command because the ROLE is contained in its block.

5.1.4. Running a reuse process in CollabRDL

The CollabRDL program was converted to BPMN-Activiti and was loaded into CollabRDL-Activiti-Explorer execution environment. In this environment two users and three groups were registered: the *analyst, designer* and *databaseAdministrator groups*. A user was associated to the *analyst* group and the other with all others groups.

The goal is to reproduce the target model (Fig. 6) by groups using four roles. A reuser (coordinator) initiates the process execution and runs the first stretch of activities as shown in Code 6. In the first command, the execution environment asks for the name of the package where the model of the new application will be generated. Then, the execution environment asks for the name of the client application class, where the coordinator should answer *ClienteApplication*, as seen in Fig. 6. Next, without need for coordinator intervention, the environment creates two associations for the created class, both indicating the implementation of *MDIController* and *LoginController* interfaces. Then, it prompts so that the name of facade class is entered, which must be *DemoClientFacade*, and associates the new class to the *ClientFacade* interface, always according to the model in Fig. 6, the target model.

Code 7 shows the reduced code snippet for DOPARALLEL command. The full stretch (CollabRDL Page, 2015) expresses the activities needed to create all control and windows classes, which are the classes whose names end with *FrameController* or *frame*, respectively. The first command in the DOPARALLEL block asks for the initiator of the proceedings, i.e., the name of new facade method referenced by *clientFacadeClass*, class created in the Stretch I, Code 6. The answer should be *getEmployees, getDepts* or *getTasks* in this example, which indicates the importance of following the dependencies in the model as seen in Fig. 6.

The activities that create classes named with the suffix *LookupController* and *VO*, respectively, are independent of each other. Thus, as seen in Code 8, a block of FLOW was written to generate the classes of type *LookupController* classes and another to generate the classes of type *VO*, all of them offered by the *analyst* group. The first FLOW is a LOOP to create *LookupController* classes depending on the response of the *analyst*, including or not the *validateCode, loadData* and *getTreeModel* methods in the application model. The second FLOW presents a LOOP for creating the classes of type *VO* and other to create methods in *VO* classes

Code 9 is the fourth and the last stretch. It adds to the list of *databaseAdministrator* group activity to create a database, and it is external because it does not handle the application model in UML but is rather a task that is executed in another execution environment.

Figs. 7 and 8 are snapshots of CollabRDL-Activiti-Explorer running this case study. The users were working in separate geographical locations and used Skype to communicate to each other throughout the process of creating the application model. The execution was performed during a work session and lasted approximately 1 h and 30 minutes. The user who performed the activities

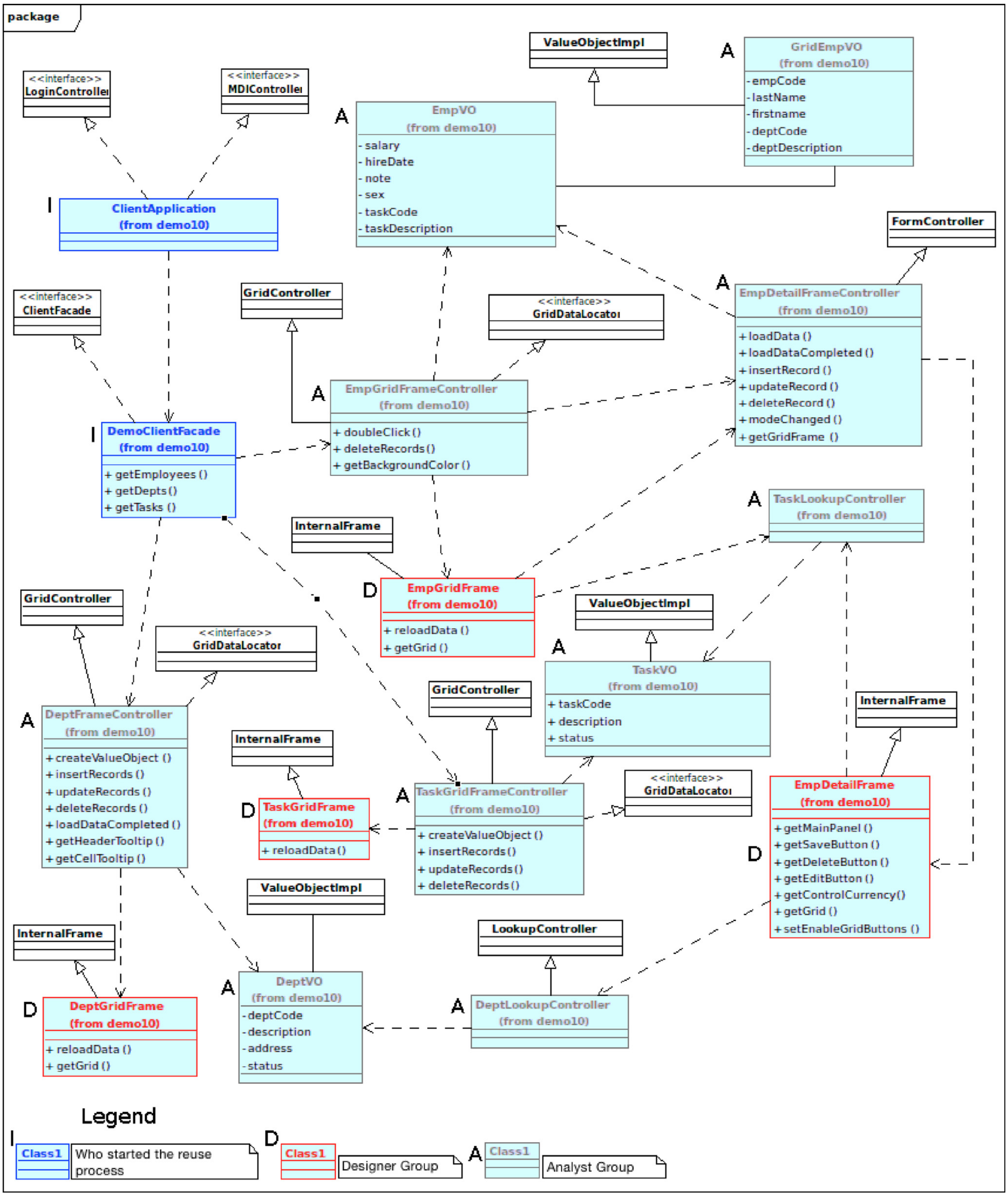

package
ValueObjectImpl
GridEmpVO (from demo10) -empCode -lastName -firstname -deptCode -deptDescription
EmpVO (from demo10) - salary - hireDate - note - sex - taskCode - taskDescription
<<interface>> LoginController
<<interface>> MDIController
ClientApplication (from demo10)
FormController
EmpDetailFrameController (from demo10) +loadData () +loadDataCompleted () +insertRecord () +updateRecord () +deleteRecord () +modeChanged () +getGridFrame ()
<<interface>> ClientFacade
GridController
<<interface>> GridDataLocator
EmpGridFrameController (from demo10) + doubleClick () + deleteRecords () + getBackgroundColor ()
DemoClientFacade (from demo10) + getEmployees () + getDepts () + getTasks ()
TaskLookupController (from demo10)
InternalFrame
EmpGridFrame (from demo10) + reloadData () + getGrid ()
TaskVO (from demo10) + taskCode + description + status
DeptFrameController (from demo10) +createValueObject () +insertRecords () +updateRecords () +deleteRecords () +loadDataCompleted () +getHeaderTooltip () +getCellTooltip ()
TaskGridFrame (from demo10) + reloadData ()
TaskGridFrameController (from demo10) + createValueObject () + insertRecords () + updateRecords () + deleteRecords ()
EmpDetailFrame (from demo10) +getMainPanel () +getSaveButton () +getDeleteButton () +getEditButton () +getControlCurrency() +getGrid () +setEnableGridButtons ()
LookupController
DeptGridFrame (from demo10) + reloadData () + getGrid ()
DeptVO (from demo10) -deptCode -description -address -status
DeptLookupController (from demo10)
Legend
I Class1 Who started the reuse process
D Class1 Designer Group
A Class1 Analyst Group

Fig. 6. The UML Class Model of the new application based on the Demo10 application domain.

```
appPack = NEW_PACKAGE(appmodel,"?");
clientApplicationClass = NEW_CLASS(appPack, "?");
NEW_REALIZATION(clientApplicationClass, org.openswing.swing.mdi.client.MDIController);
NEW_REALIZATION(clientApplicationClass, org.openswing.swing.permissions.client.LoginController);
clientFacadeClass = NEW_CLASS(appPack, "?");
NEW_REALIZATION(clientFacadeClass, org.openswing.swing.mdi.client.ClientFacade);
```

Code 6. Stretch I to generate a new application based on Demo10.

Table 2
The CollabRDL program in numbers (Demo 10-based applications).

	Quantity	Activities (Interactive)	FLOW	IF	LOOP
DOPARALLEL	1	70 (5)	–	53 (20 nested)	0
PARALLEL	1	6 (2)	2	3	3 (1 nested)
ROLE	4 (2 nested)	– (6)	–	53 (20 nested)	0

```
DOPARALLEL {
   newFunction = NEW_METHOD(clientFacadeClass, "?");
   ROLE (analyst, "Create frameControllerClass,frameClass and detailFrameControllerClass."){
      frameControllerClass = CLASS_EXTENSION(org.openswing.swing.table.client.GridController, appPack, "?");
      NEW_REALIZATION(frameControllerClass, org.openswing.swing.table.java.GridDataLocator);
      IF ("Do you want redefine enterButton method?") THEN {
        m =
METHOD_EXTENSION(org.openswing.swing.table.client.GridController,frameControllerClass,enterButton);
     }
     IF ("Do you want redefine doubleClick method?") THEN {
       m =
METHOD_EXTENSION(org.openswing.swing.table.client.GridController,frameControllerClass,doubleClick);
     }
     Suppressed code
     ROLE (designer, "Use visual plugin for Eclipse or Netbeans!"){
       frameClass = CLASS_EXTENSION(org.openswing.swing.mdi.client.InternalFrame, appPack, "?");
       EXTERNAL_TASK("Edit frameClass using visual plugin for Eclipse or Netbeans.");
     }
     Suppressed code
  }//End ROLE (analyst....
} WHILE("Create another function?"); //End DOPARALLEL
```

Code 7. Stretch II to generate a new application based on Demo10.

```
PARALLEL {
    FLOW (analyst, ""){
    LOOP ("Create another LookupController?"){
    lookupControllerClass = CLASS_EXTENSION(org.openswing.swing.lookup.client.LookupController,
                                                                     appPack, "?");
    IF ("Redefine validateCode method?") THEN {
           m = METHOD_EXTENSION(org.openswing.swing.lookup.client.LookupController,
                                                         lookupControllerClass,validateCode);
    }
    IF ("Redefine loadData method?") THEN {
           m = METHOD_EXTENSION(org.openswing.swing.lookup.client.LookupController,
                                                         lookupControllerClass,loadData);
    }
    IF ("Redefine getTreeModel method?") THEN {
           m = METHOD_EXTENSION(org.openswing.swing.lookup.client.LookupController,
                                                    lookupControllerClass,getTreeModel);
    }
    }//End LOOP
    }//End FLOW
    FLOW (analyst, ""){
    LOOP ("Create another VOClass?"){
          VOClass = CLASS_EXTENSION(org.openswing.swing.message.receive.java.ValueObjectImpl,
                                                                     appPack, "?");
          LOOP ("Create another attribute in VOClass?"){
                   NEW_ATTRIBUTE (VOClass,"?", int);
          }
     }//End LOOP
     }//End FLOW
}// End PARALLEL
```

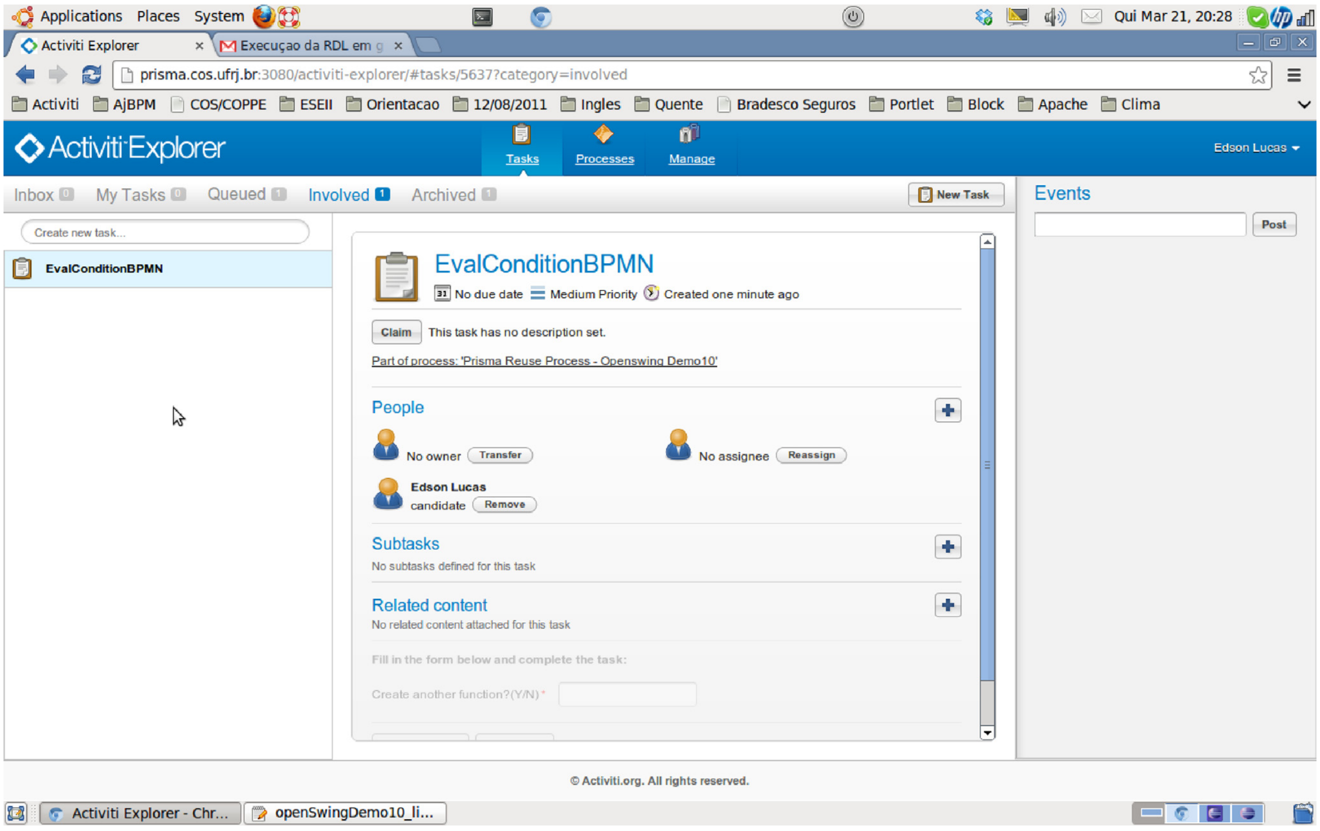
Code 8. Stretch III to generate a new application based on Demo10.

```
ROLE (databaseAdministrator, "See VOClass
          created."){
          EXTERNAL_TASK("Create Database.");
}
```

Code 9. Stretch IV to generate a new application based on Demo10.

in the block of the DOPARALLEL command could receive help at runtime. Thus, the application model was generated in a collaborative way, according to the 3C model introduced by Fuks et al. (2007). This is the case because the coordination was expressed in CollabRDL, there was communication via Skype and text through the execution environment, and there was cooperation as the two parts of the model were generated by different roles, which work together to produce a complete model for the intended application.

Table 3 compares the target application model with the application model generated in this case study. With the exception of the *EmpDetailFrameController* class, all other classes are present in the generated model. As a result, the generated model had 24 elements more and 13 less when compared to the target application model in Fig. 6. For the purpose of counting, we considered the



Applications Places System
Qui Mar 21, 20:28
Activiti Explorer
Execuçao da RDL em g
prisma.cos.ufrj.br:3080/activiti-explorer/#tasks/5637?category=involved
Activiti AjBPM COS/COPPE ESEII Orientacao 12/08/2011 Ingles Quente Bradesco Seguros Portlet Block Apache Clima
Activiti Explorer
Tasks Processes Manage
Edson Lucas
Inbox 0 My Tasks 0 Queued 0 Involved 1 Archived 0
New Task
Events
Post
Create new task...
EvalConditionBPMN
Claim
Transfer
Reassign
Remove
Activiti Explorer - Chr...
openSwingDemo10_li...

EvalConditionBPMN

No due date Medium Priority Created one minute ago

This task has no description set.

Part of process: 'Prisma Reuse Process - Openswing Demo10'

People

No owner No assignee

Edson Lucas candidate

Subtasks

No subtasks defined for this task

Related content

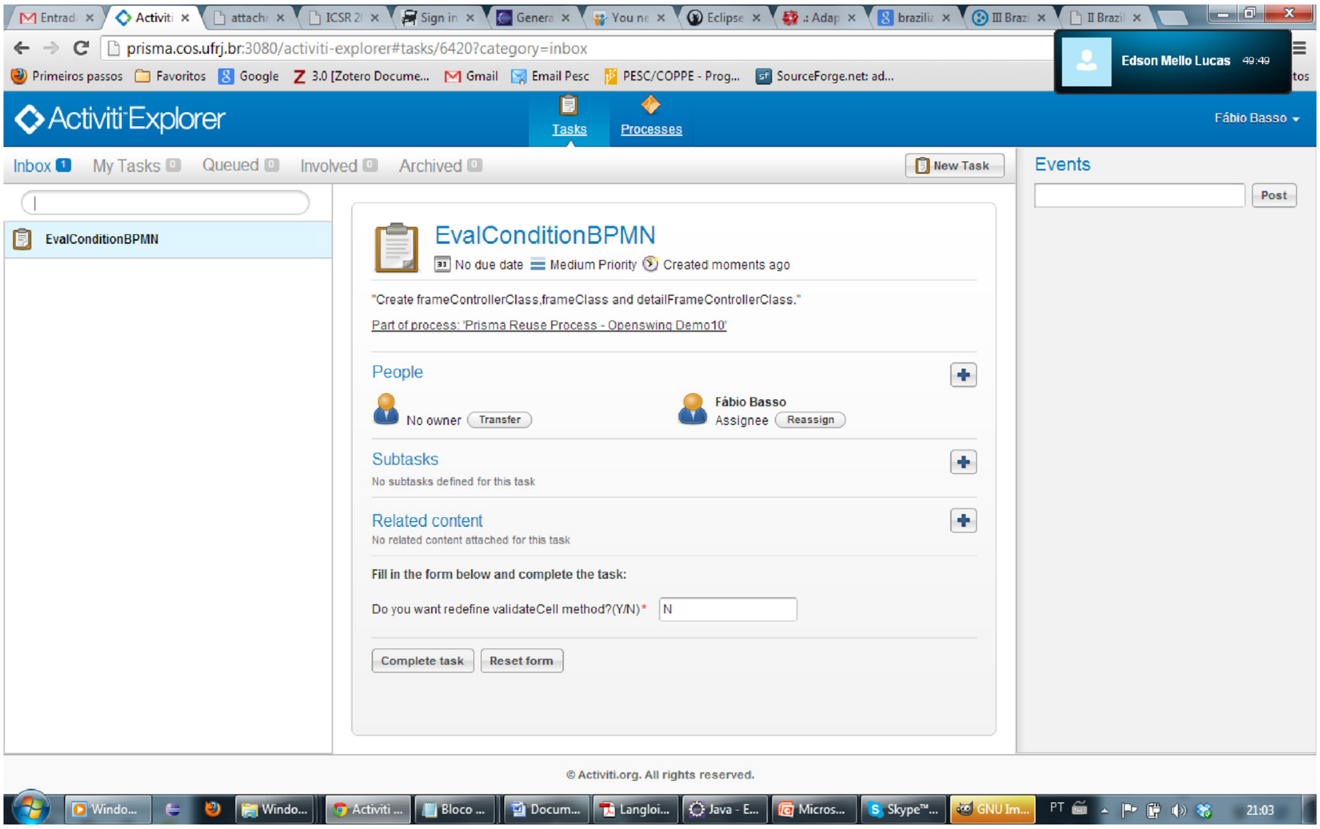
No related content attached for this task

Fill in the form below and complete the task:

Create another function?(Y/N)*

© Activiti.org. All rights reserved.

Fig. 7. Snapshot of CollabRDL-Activiti-Explorer for the first reuser.



Entrad Activiti attach ICSR 2 Sign in Genera You ne Eclipse :: Adap brazilia III Brazi II Brazi
prisma.cos.ufrj.br:3080/activiti-explorer#tasks/6420?category=inbox
Primeiros passos Favoritos Google 3.0 [Zotero Docume... Gmail Email Pesc PESC/COPPE - Prog... SourceForge.net: ad...
Edson Mello Lucas 49:49
Activiti Explorer
Tasks Processes
Fábio Basso
Inbox 1 My Tasks 0 Queued 0 Involved 0 Archived 0
New Task
Events
Post
EvalConditionBPMN
Transfer
Reassign
Complete task Reset form
Windo... Windo... Activiti ... Bloco ... Docum... Langloi... Java - E... Micros... Skype™... GNU Im... PT 21:03

EvalConditionBPMN

No due date Medium Priority Created moments ago

"Create frameControllerClass,frameClass and detailFrameControllerClass."

Part of process: 'Prisma Reuse Process - Openswing Demo10'

People

No owner Fábio Basso Assignee

Subtasks

No subtasks defined for this task

Related content

No related content attached for this task

Fill in the form below and complete the task:

Do you want redefine validateCell method?(Y/N)* N

© Activiti.org. All rights reserved.

Fig. 8. Snapshot of CollabRDL-Activiti-Explorer for the second reuser.

Table 3
Comparing the generated application model to the target model.

Class	Target Model	Generated Model	Differences	Causes
ClientApplication	3	3	0	
DemoClientFacade	5	5	0	

EmpGridFrameController	6	13	▸ 8 elements more;
			▸ one less;
				▸ Awareness
				▸ Not expressed in CollabRDL;

EmpDetailFrameController	9	0	9	Unknown
DeptFrameController	10	18	8 elements more	Awareness
TaskGridFrameController	7	15	8 elements more	Awareness
EmpGridFrame	4	2	2 less	External Activity
EmpDetailFrame	9	2	7 less	External Activity
DeptGridFrame	4	2	2 less	External Activity
TaskGridFrame	3	2	1 less	External Activity
DeptLookupController	2	2	0	
TaskLookupController	1	2	1 more	Error on Target Model
EmpVO	8	8	an incorrect inheritance	Expressed wrong in CollabRDL;
GridEmpVO	7	7	0	
DeptVO	6	6	0	
TaskVO	5	5	0	
Total	**87**	**92**	**24 elements more and 13 less. And one error.**	

following elements in the model: attributes; operations; interface implementation; description of inheritance; and class presence itself. In the second column of Table 3, we present the number of expected elements per class. The third column presents the number of elements created by implementing the reuse activities per class described in the CollabRDL Demo 10. Moreover, the last column identifies some of the causes that led to the differences.

The *ClientApplication, DemoClientFacade, DeptLookupController, GridEmpVO, DeptVO* and *TaskVO* classes were created as they were specified in the target model. The *EmpGridFrameController, DeptFrameController* and *TaskGridFrameController* control classes are generated by the activities of iterations in the DOPARALLEL command block and therefore have the same number of elements. Code 10 shows the snippet of CollabRDL that produced these elements, and it demonstrates that reuser answered *NO* to the question "Drag event is enabled, do you want to set to false?" for all these control classes. This mistake happened because the reuser only consulted the target model, which was depicted in Fig. 6, and he has not looked at the CollabRDL program. This leads us to consider awareness as a cause of problems in the execution environment.

The absence of the *getBackgroundColor* method in the *EmpGrid-FrameController* class is because the activity to create it was not described in CollabRDL program. In addition, the absence of the *EmpDetailFrameController* class was not due to a negative answer to the question "Create Detail Frame?", because *EmpDetailFrame* class was created and is part of the block of this condition.

Classes ending in Frame are created by the designer group and are expressed in CollabRDL as an external activity of the execution environment, and for this reason their methods do not appear in the target model. This type of activity is very well done with visual editors for building windows, which are common in IDE tools such as Eclipse and Netbeans.

The *TaskLookupController* class has one element more than the target model, since it presents inheritance from *LookupController* in the generated model. This activity was described in CollabRDL and is correct, but the target model omitted this inheritance. The *EmpVO* class inherits from *GridEmpVO* but CollabRDL provided an inheritance from *ValueObjectImpl*.

Finally, we consider a programming mistake in CollabRDL, i.e., the CollabRDL programmer did not represent correctly a reuse activity that was responsible for two differences between the models. Concerning the question regarding external activities, the target model identifies the elements needed to build the application windows. However, the CollabRDL program represented this

```
IF ("Drag event is enabled, do you want set to false?") THEN {
        dragEnabledMethod = METHOD_EXTENSION(org.openswing.swing.
                                    table.client.GridController,
                                    frameControllerClass,dragEnabled);
        ADD_CODE(frameControllerClass,dragEnabledMethod,
                                                         "return false;");
} ELSE {
        m = METHOD_EXTENSION(org.openswing.swing.
                                    table.client.GridController,
                                    frameControllerClass,dragEnter);
        m = METHOD_EXTENSION(org.openswing.swing.
                                    table.client.GridController,
                                    frameControllerClass,dragExit);
        m = METHOD_EXTENSION(org.openswing.swing.
                                    table.client.GridController,
                                    frameControllerClass,dragOver);
        m = METHOD_EXTENSION(org.openswing.swing.
                              table.client.GridController,
                              frameControllerClass,dropActionChanged);
        m = METHOD_EXTENSION(org.openswing.swing.
                              table.client.GridController,
                              frameControllerClass,dragDropEnd);
        m = METHOD_EXTENSION(org.openswing.swing.
                              table.client.GridController,
                              frameControllerClass,dropEnter);
        m = METHOD_EXTENSION(org.openswing.swing.
                              table.client.GridController,
                              frameControllerClass,dropExit);
        m = METHOD_EXTENSION(org.openswing.swing.
                              table.client.GridController,
                              frameControllerClass,dropOver);
}
```

Code 10. Code snippet used to generate more elements than expected. .

type of activity using visual programming software, which generates windows, and the execution environment did not build the resulting model generated from this activity. There were 12 differences similar to this one. We consider the specification error in the target model as an error in the project implementation project, and this was the cause of the difference. Overall, nine differences have unknown causes. Furthermore, 24 differences were caused by the reuser (lack of) information related to the environment that needed to be taken into account in the reuse activities (i.e., the so-called awareness). The high number of differences related to awareness is justified because the execution environment tool is based on business processes that have implementation limitations when it comes to multi-user environment features.

5.1.5. Summary of the evaluation results

CollabRDL has been evaluated by assessing its ability to represent specific workflow patterns. For this purpose, we have shown in Appendix how the following coordination-oriented workflow patterns are represented in CollabRDL: Parallel Split Pattern (WCP2), Synchronization Pattern (WCP3), Exclusive Choice Pattern (WCP4), Simple Merge Pattern (WCP5), Multi-Choice Pattern (WCP6), Role-based Distribution Pattern (WRP2), and Multiple instances with decision at runtime (WCP14). CollabRDL has also been evaluated by a case study in which developer groups performed reuse activities collaboratively to instantiate a mainstream Java framework.

In the case study reuse activities were performed when possible in parallel. This study was essentially planned to answer three questions: 1-"Can the ROLE command offer a block of activities to a group? "; 2- "Can the PARALLEL command create instances of activities block for running in parallel?"; and 3 - "Can the DOPARALLEL command create threads of the same block of activities?". In context of the first question, this study used 4 ROLE commands with 6 interactive activities and 53 responses to IF commands executed by the groups that have been declared in ROLE commands (*analyst, designer* and *databaseAdministrator* groups). The answer of the question 2 is affirmative because a PARALLEL command created two instances of same block in parallel. The command was declared with two interactive activities, two flows in parallel, containing 3 IF commands and 3 LOOP commands. And the answer for the question 3 is also affirmative because a PARALLEL command executed more than once its block declared with 5 interactive activities and 53 IF commands.

5.1.6. Threats to validity

In this study case, validity threats were based on a checklist to analyze threats according to construct validity, internal validity, external validity, and reliability, as suggested by Runeson (2009).

5.1.6.1. Construct validity. The errors on CollabRDL Program were cause of 24% of differences between the target model and the generated model. The causes were generated either the absence activity to add a method was not expressed in CollabRDL or an incorrect inheritance was expressed in CollabRDL. One error, 3% of differences, was caused by error on Target Model. In addition, the execution environment of CollabRDL makes calls to reuse core functions of the RDL core, which is designed for to single reusers. Also in this core, a process needs to be started and completed in a single work session.

The lack of awareness is a problem considering the execution environment tools. The 65% of differences between the target model and the generated model were caused by lack of awareness. Another limitation related to awareness happens when you need to identify which iteration an activity belongs. For example, the command "IF (" Do you want resets doubleClick method? ") THEN" appears three times in the activities queue for the *analyst* group, one for each iteration. However, the Activiti-Explorer tool offers us such implementation features that enable the creation of a single queue for each group, but this is not suitable for the DOPARALLEL command.

5.1.6.2. Internal validity. The study had only a trial with one CollabRDL program applied in only one framework by two subjects. Then the result was a limited sample size. One of subjects is also one of the co-authors of this work. Both of them were exposed to the CollabRDL execution environment before the case study took place. Regarding knowledge about the OpenSwing framework, one of the subjects had previously used this framework in software development projects and the other had not.

As it is defined, in terms of its function, the DOPARALLEL command allows the initiator of the process to delegate ROLE block activities to the analyst group, answer yes for the question “Create another function?” and create a new method again by using the first DOPARALLEL command block without waiting for the implementation of the first ROLE block instance. However, in this case, errors can occur due to a simultaneous access to a critical region. This happens because the reference to *frameControllerClass* will be change for all instances of the ROLE block. To avoid simultaneous access to the critical regions, the initiator can create a method in the facade class, wait for the fulfillment of all the other activities of the DOPARALLEL block, and only after it gets an answer “yes” the question “Create another function?” is asked.

5.1.6.3. External validity. This study was made with one coordinator and two reuse groups. A subject performed activities of coordination, and the other subject performed activities associates to groups. However, other studies can be created without constraints of groups and subjects. In the same way, the CollabRDL program had one DOPARALLEL command, one PARALLEL command, and four ROLE commands, but there are not constraints about a number of commands on CollabRDL programs.

5.1.6.4. Reliability. This study case demonstrated that CollabRDL commands add support to collaboration in RDL. The ROLE command delegated reuse activities to groups. The PARALLEL command created two flows of work and synchronized them. Finally, the DOPARALLEL command built many instances of the same block to be executed by groups. It is expected that the result should be similar to other CollabRDL programs.

5.2. Discussion

Today, software development is conducted in increasingly turbulent business environments with globally distributed teams. Although the development teams have indeed succeeded in elaborating practices to manually instantiate frameworks, these practices are based on the experience of the team members (i.e., individually), rather than a method carefully defined and widely grasped by all members (i.e., formal and globally used). In other words, the manual instantiation predominantly aims at the individual level, whereas the CollabRDL raises the instantiation process to a team-class level by determining a new form to instantiate and reuse the descriptions about how frameworks can be instantiated.

Even though developers can be accustomed to use frameworks in a manual way, this form of use can be seen am error-prone and time-consuming task. First, the whole process is often driven by trial and error so as to achieve a correct instantiation. Second, it is very difficult, if not impossible, to automatically identify instantiation problems, or even detect problems without investing much effort, given the problem hand. Third, the manual instantiation relies on an understanding of what the key elements of framework mean, and such semantic information is typically not included in any formal way. Fourth, the manual instantiation process becomes tiresome mainly for novices, who usually have little understanding about the built-in business rules of large frameworks.

It is worth using CollabRDL, as opposed to the manual framework reuse, when there is one (or more) development team that needs to carefully determine the interplay between the reuse activities, and form working groups taking into account their skills and responsibilities. Another scenario would be when development team members, e.g., software architect, need to detail how to form blocks of commands (related to reuse activities) to be run by different working groups collaboratively and in parallel. Perhaps it is not worth to be CollabRDL program in the context of small development teams, formed for two or three developers, because the benefits of the CollabRDL are not widely perceived.

We also highlight that the experience of developers in reusing the framework matters in both approaches, either manual or based on the CollabRDL program. On the other hand, the lower the level of experience of the development team members, the greater the benefits of the CollabRDL can be perceived, because the reasoning developed to instantiate the frameworks can be reused by other developers. In contrast, the manual approach requires that all developers have similar reasoning to instantiate the frameworks, which increases the likely of instantiation problems, cause by divergent reasoning developed.

It is also important to expose CollabRDL programs should be created by experienced developers to properly address the required hotspots and also promote teamwork. To mitigate creating RDL programs from scratch, Gomes et al. (2014) describes how to generate RDL programs from repository of applications that reuse frameworks using process mining algorithms. Creating CollabRDL programs automatically using the same process mining approach seems feasible and is subject to future work.

6. Related work

This session presents some of work related to this article identified in the literature on Software Reuse. This related work is organized in three main areas: Framework Instantiation, Software Product Line (SPL) and Chain Transformation (MDD). Next, Section 6.4 presents a comparison with this related work.

6.1. Framework instantiation

Initially, the process of reusing a Model-View-Controller framework was described through CookBook using natural language and documenting the steps for its instantiation (Krasner & Pope, 1988). The notion of Hook was defined by Froehlich et al. (1997), which can be seen as an evolution of a CookBook, and had a structured description involving a name that identifies the hook, the type of the hook, relevant conditions, etc., while still keeping the natural language in the descriptions.

RDL (Reuse Description Language) is a language for describing the reuse process, and provides a systematic way to instantiate object-oriented frameworks, but does not address collaborative aspects (Oliveira et al., 2007). Cechticky et al. (2003) describe an environment for instantiating frameworks that uses a transformation language to generate code from the formal specifications of the framework components. They make use of processes but do not address the issue of teamwork.

Cechticky et al. (2003) approach provides a component-based environment assumes as input Visual Proxy Beans and the specification of the application. The Beans, implemented as JavaBeans, are generated from the components of the framework described in XML, and the transformation language is Extensible Stylesheet Language Transformations (XSLT). The output is a formal description of the XML application configuration that is processed by the XSLT program to produce the application code.

Holmes and Walker (2013) described an approach for planning and performing reuse in a scenario where the source code was not properly designed to be reused. The plan is based on a model of a pragmatic reuse process with six tasks, whereas a tool suite called Gilligan supports the Locate dependencies, Triage dependencies and Enact plan tasks. The Locate dependency task involves only static structural relationships, such as Inheritance relationships, method call relationships, field references, Has-Type (only for fields), and containment relationships. The Triage dependencies task offers a structural element six triage decisions: Accept (an element that reuser should reuse); Reject (an element will not be reused); Remap (an element will be changed); Provided (e.g., source and target use same library); Extract (e.g., class field); and Inject (inject any arbitrary fragment of code into a class being reused). In terms of the Enact plan task, Gilligan supports three decisions: Extraction (which source code should be reused); Integration (Managing source code additions, Managing dangling references and Managing unnecessary code); and Finalizing Source Code Modifications.

6.2. Software product lines

The practice of reusing suggests a collaborative solution. In Mendonça et al. (2007), the authors show an example of this approach, which allows users to collaboratively set up a new product by a process within a Software Product Line. Next, Mendonça et al. (2008) have added improvements to this process called Collaborative Product Configuration (CPC) with a detailed presentation of algorithms. A tool based on this approach was developed, the SPLOT - Software Product Lines Online Tools, which is available for free download, including the source code (SPLOT, 2016).

CPC gets as input a feature model (Kang et al., 1990). This model is labeled in configuration regions associated with groups taking into account the knowledge in the field (e.g., database administrator, security expert) and decision on the project (e.g., managers, supervisors). The model is then converted to BPMN and includes the execution plan for a CPC. The execution can result in decision conflicts. For example, a participant may select a feature that depends on another that was not selected in one area of responsibility by other participants. Then, to solve these types of inconsistencies specific algorithms were created to support the validation of the execution plan (Mendonça et al., 2008).

CPC allows you to associate tasks to groups of people that perform tasks in parallel, synchronize them during the setup process of the new product, and produce an executable plan. However, this approach does not describe the process of choosing the characteristics of the new product, leaving that responsibility to the group responsible for a specific configuration region. However, in case of a region with many configuration reuse rules, this solution can be problematic.

Rabiser et al. (2007) present an approach to support the configuration of a new product by extending and adapting the model variability (i.e., features) in order to eliminate or add features to the model according to the information about a specific project, that is, about the product to be created. They also present a metamodel with new elements that are not part of the variability model, but are important in the product configuration process. The metamodel represents a product that has one or more properties, and each property is related to the decision whether or not to add a variability artifact in the product model. A role is associated with one or more decisions and, finally, and guidance related to one or more decisions can be provided to assist with tips and recommendations those involved in the new product configuration. However, this approach does not make explicit that activities can occur in parallel and under specific synchronization constraints.

Noor et al. (2007) present a collaborative approach to express a process in the scope of Product Line using thinklets, but does not explore the possibility of activities in parallel, a limitation that is similar to the one related to Rabiser et al. (2007). A thinklet is defined as the smallest unit of intellectual capital required for creating a repeatable, predictable pattern of collaboration among people working to achieve a goal (Briggs et al., 2003). They assume that the engineering of collaboration is the development of repeated collaborative processes conducted by the practitioners themselves.

The Noor et al. (2007) is organized into three layers: the process layer, the patterns layer, and the thinklets layer. In the process layer, the objectives and the tasks relevant for achieving them are defined, including the identification and selection of participants. The pattern layer makes use of collaboration engineering patterns discussed in (Briggs et al., 2003), and includes patterns such as Divergent (e.g., brainstorming), Convergent (adding or eliminating concepts), Organization, Evaluation, and Consensus. The tasks are mapped to these patterns with the goal of realizing the application. Finally, in the thinklets layer the thinklets that can be executed are defined. Noor et al. (2007) collaborative approach supports representing tasks and their order to define the process of creating a product group, but does not explore the possibility of parallel activities.

Hubaux et al. (2009) defined Feature Configuration Workflows (FCW) that use the concept of workflow to describe the process of configuring a new product from a feature model, allowing the distribution of activities between people and the performance of activities in parallel. In Abbasi et al. (2011) a tool for FCW is presented. FCW supports the association of part of workflow activities of the features model with the group of people that will perform them. This can be seen as a form of delegation. In that way, when the workflow is executed, people in specific groups approve or disapprove features of the new product and the model is synchronized, respecting change permissions of the groups. In addition, the workflow can be built with the purpose of analyzing conflicting decisions between groups about the application features.

6.3. Transformation chains

Model Driven Architecture (MDA), specified by OMG, has become a reference in model driven development. This architecture consists of models in three different levels of abstraction, with mappings from more abstract to more specific levels until the concrete level of program code is reached. CIM (Computation Independent Model) are models that belong to the most abstract level, and refer to abstractions that are easier for humans to understand, bing independent of any implementation technology. PIM (Platform Independent Model) are models to specify the structure and functionality of the system without referencing technical details, and add more information to the CIM models without relying on platform-specific information. In contrast, in the PSM (Platform Dependent Model) layer, the application components and their interfaces are described in a more specific way. Indeed, in PSM the models are built for a specific platform (e.g., Corba, Java, Dot Net), based on the transformation of PIMs.

In addition, in terms of transformations, the top-level, more abstract Computation Independent Models (CIMs) can be transformed into PIMs using T1 transformations. Then, the PIMs can be transformed into one or more Platform Specific Models (PSMs) using transformations T2 to TN. For example, a transformation T2 can lead to a model consistent with the Java 2 Enterprise Edition platform. Finally, code generation is performed using the platform-specific models, that is, the PSMs are transformed into code in different programming languages.

Transformations are based on mappings between different levels of model abstractions. A mapping is defined as a set of rules and techniques used to modify a model aiming to create another (more concrete) model. PIM to PIM transformations are performed to improve, filter out or specialize models without the necessity to embed components that depend on a specific platform. The PIM to PSM transformations happen when PIMs are refined and can be designed for an execution platform. These transformations are based on the characteristics of the target platform, and therefore dependent on it, making use of its concepts and components. In contrast, the transformations of type PSM to PSM are used for component deployment. An example is the configuration and deployment of a component to a platform to meet particular system specifications. Even though the PSM to PIM transformations are difficult to automate, they can be supported by tools to generate abstract models from specific platform models. Although the MDA approach implicitly supports collaborative work, especially in the case of transformations, it does not exploit this issue in an explicit, representation-oriented way.

Constraint-Driven Development (CDD) refers to an model driven development approach in UML (Lano, 2008). It defines use cases, class diagrams, state machines, and constraints to describe the functionality of the system at its PIM abstract layer. From these descriptions, Platform Specific Models (PSMs) are generated systematically with the help of automated tool processes. Finally, the code is generated from PSMs in a way that is similar to the MDA approach.

UML-RSDS (Reactive System Design Support) is an extension of UML to support the CDD. It redefines classes and state diagrams, making use of a simplified code in Object Constraint Language (OCL) to express the constraints. CDD as well as MDA does not address issues related to collaboration. An example of a development that uses this approach can be described in five steps: 1 - Construction of the use cases, class diagrams and state diagrams for the PIM level using a UML-RSDS tool; 2 - Analysis of consistency and completeness of the PIM models; 3 - Transformation of models to perform semantic analysis; 4 - Transformation models to improve the quality or to suit a platform (PSM). This involves transforming the PIMs to PSMs that rely on a Java platform; 5 - Generate Java code from the PSM Java specification.

In a more general sense, there are other approaches related to this work reported in the literature: Maia et al. (2007) present Oddyssey-MDA, that supports the transformation of PIMs to PSMs using Meta-Object Facility (MOF, 2002), XML Metadata Interchange (XMI, 2002), and Java Metadata Interface (JMI, 2002); ATLAS Transformation Language (ATL), which is a transformation language for defining transformations of declarative and imperative forms (Jouault and Kurtev, 2006); and (Di Ruscio et al, 2012), who have proposed a preliminary classification of transformation approaches and languages.

6.4. Discussion

Table 4 presents a comparison with related work. The area of Framework Instantiation, which relates to items 1, 3 and 4, probably started with Krasner and Pope (1988) using a natural language, but these authors did not address collaboration. Oliveira et al. (2007) present a new language to describe reuse using imperative form, but they do not use collaboration constructs. The same applies to Cechticky et al. (2003), who used an approach based on a transformation language. Holmes and Walker (2013), referred to in item 3, built a tool to support the reuse process of object-oriented source files by a single user. In terms of Software Product Lines, which relate to items 5–8, Mendonça et al. (2008) present algorithms to support group work, but they did not describe the process of choosing the characteristics of the new product. Rabiser et al. (2007) work on the expansion and adaptation of the feature model, but they do not explicitly support the representation of activities in parallel. Noor et al. (2007) make use of thinklets, but they also do not explore activities in parallel. Hubaux et al. (2009), in the area of Software Product Line, make use of workflows to describe the process of configuring a new product collaboratively. In the area of Transformation Chains, which relates to item 9 in Table 4, Lano (2008) presents an extension of UML to support the Development Guided by constraints using OCL, but they do not address the issues of collaboration, synchronization or teamwork-based concepts. Finally, in the area of Staged Configuration in Multi-level

Table 4
Comparison of related work.

	Reference	Area	Manner	Collaboration
1	Krasner & Pope, 1988	Framework Instantiation	CookBook with natural language	They do not address collaboration
2	Holmes and Walker, 2013	Object-oriented source	Tailored process supported by tool.	They do not address collaboration
3	Oliveira et al., 2007	Framework Instantiation	Imperative Language	They do not make framework instantiation collaboratively
4	Cechticky et al., 2003	Framework Instantiation	Transformation Language	They do not make framework instantiation collaboratively
5	Mendonça et al., 2008	Software Product Line	They presented algorithms.	They not describe the process of choosing the application characteristics
6	Rabiser et al., 2007	Software Product Line	They worked on the expansion and adaptation of the feature model	They do not make explicit the activities in parallel
7	Noor et al., 2007	Software Product Line	They use thinklets	They do not explore activities in parallel
8	Hubaux et al., 2009	Software Product Line	They use workflow to describe the process of configuring a new product	The approach is collaborative, but does not support for asynchronous configuration.
9	Lano, 2008	Transformation Chains	He presented an extension of UML and use OCL.	The author did not address the issues of collaboration
10	Czarnecki et al., 2005	Staged Configuration	Multi-level Staged Configuration, mentioned the participation of people with different roles	The approach is collaborative, but sequential.

Staged Configuration, which relates to item 10, Czarnecki et al. (2005) mention the participation of people with different roles, but the proposed approach is sequential.

CollabRDL extends RDL with concepts related to collaboration in the imperative form and follows the structure of RDL. Moreover, as in the case of Hubaux et al. (2009), CollabRDL makes use of workflow environments. In relation to other related works, some do not support the execution of activities in parallel, while others do not address the issue of collaboration at all. Therefore, we conclude that CollabRDL advances the state of the art in software mass customization since current approaches do not support the representation and implementation of collaborative instantiations that involve individual and group roles, the simultaneous performance of multiple activities, restrictions related to concurrency and synchronization of activities, and allocation of activities to reuse actors as a coordination mechanism.

7. Conclusions and future work

In this paper, we have presented CollabRDL, an extension of RDL that focuses on collaborative group activities. In the presentation of CollabRDL, we aimed at achieving the following objectives: (1) to allow the definition of groups of reusers; (2) to support the delegation of reuse activities to specific groups of people; (3) to represent parallel reuse activities; and (4) to present an evaluation based on the representation of some coordination-oriented workflow patterns and based on a case study involving framework instantiation.

The first objective was achieved using the *Activiti* execution environment (Activiti, 2015). The second one was achieved using the ROLE and FLOW commands running in the *CollabRDL-Activiti-Explorer* environment. The third objective was achieved using the PARALLEL and DOPARALLEL commands, and the configuration of the Activiti execution environment, the so-called *CollabRDL-Activiti-Explorer*. Finally, the evaluation, the fourth and last specific objective, was achieved by representing the specific workflow patterns in CollabRDL and by conducting the case study.

CollabRDL was assessed with respect to the following workflow patterns: Parallel Split Pattern (WCP2), Synchronization Pattern (WCP3), Exclusive Choice Pattern (WCP4), Simple Merge Pattern (WCP5), Multi-Choice Pattern (WCP6), Role-based Distribution Pattern (WRP2), and Multiple instances with decision at runtime (WCP14). The case study was conducted by two developers working in groups that collaboratively performed reuse activities for instantiating a mainstream Java framework (OpenSwing, 2015). The ROLE, PARALLEL and DOPARALLEL commands were evaluated with respect to their functionality in order to realize the reuse activities in teams and, when possible, in parallel. The questions for this objective are: 1 – Can the ROLE command assign a block of activities for a group? 2 – Can the PARALLEL command create streams of execution of blocks of activities in parallel? 3 – Can the DOPARALLEL command create flows of the same block of activities?

As a result, the case study shows that three CollabRDL commands were effective. The ROLE command coordinated six reuse activities and fifty three questions among the *analyst, designer* and *databaseAdministrador* groups. The PARALLEL command has managed to create two flows of activities in parallel. Moreover, the DOPARALLEL performed more than once specific block activities.

As future work, we seek to improve CollabRDL by (1) supporting the concept of environment awareness in collaborative software reuse, and by (2) implementing an appropriate treatment of critical regions. These two elements are necessary as a basis to conduct further studies to explore all the desirable features of the new commands with the involvement of more individuals and groups. The first item, awareness, can be potentially dealt with using visualization techniques (Botterweck et al., 2008), and the second involves the use of concurrent languages (Scott, 2009).

Moreover, empirical studies will be performed to better understand how the effectiveness and usage of the CollabRDL can be affected by different collaboration scenarios. In particular, we are concerned on exploring scenarios found in: (1) *large organizations*, where the collaboration is more organized and practitioner has fixed role in the team; and (2) *open source projects*, in which the collaboration is based on voluntary work of the participants, developed tasks can be added to and removed from the project at any time, and developers can change their roles or even have multiple roles.

Finally, we hope that the proposed extension and the issues outlined throughout the paper may encourage other researchers and developers to use and evaluate the CollabRDL in the future under different circumstances. This work can be seen as a first

```
1- //Tokens
2 - BEGIN_ROLE = 'ROLE';
// ------
3 - role_statement
4 -  BEGIN_ROLE LEFT_PARENTHESIS role
COMMA comment
5 -  RIGHT_PARENTHESIS {
codeGen.addBeginRoleBlock ($role.text,
$comment.text); }
6 -  statement_block { codeGen.addEndRoleBlock();
}
7 -;
```

Code 11. Grammar of ROLE command.

step in a more ambitious agenda on how to better support collaboration in systematic reuse.

Acknowledgements

This work was partially supported by the Brazilian funding agencies CAPES and CNPq and Natural Sciences and Engineering Research Council of Canada (NSERC).

Appendix A

This appendix shows the CollabRDL commands in the Backus Normal Form (BNF) representation.

A.1. ROLE command

In Code 11, the lines 3–7 show the BNF that identifies the ROLE command. This command has two parameters represented by the non-terminal symbols *role* and *comment*. The parameter *role* identifies the group, and the parameter *comment* refers to the comment communicated to this group. Besides the BNF, Code 11 shows how the BPMN code generator handles this command. In line 5, there is an expression, *codeGen.addBeginRoleBlock ($ role.text, $ Comment.Text)*, that represents a method call *addBeginRoleBlock* with parameters $*role.text* and $*comment.text*. This method belongs to the *BPMNCodeGenerator* class of the RDL-Activity-BPMN package, which is responsible for generating the BPMN code in the Extensible Markup Language (XML) format for all commands.

A.2. PARALLEL command

In Code 12, lines 3–8 show BNF that identifies PARALLEL command. This command has a block with two or more *flow_statement* statements. Line 5 shows the beginning of the block with the method call, *addFork*, which is responsible for creating threads in parallel for *flow_statement* statements. In addition, line 7 declares a method call, *addJoin*, at the end of the block, to join these threads of execution. The *flow_* statement, in lines 9–13, which is similar to ROLE with respect to its parameters, declares a block of activities that can be executed in parallel by the PARALLEL command. Again, the *addFork* and *addJoin* methods belong to the class *BPMNCodeGenerator*.

A.3. DOPARALLEL command

In Code 13, lines 4–9 show the BNF of the DOPARALLEL command. Line 5 starts with a call to *addBeginDoparallelBlock* method. Line 6 declares that the DOPARALLEL command contains a *statement_block*, which is a generic block to package reuse commands and language structures. Lines 7 and 8 indicate iteration, and the expression *addEndDoparallelBlock ($ condition.text)* refers to a call to *addEndDoparallelBlock* method with parameter *$condition.text*. As it happens in the case of all commands, the *addBeginDoparallelBlock* and *addEndDoparallelBlock* methods belong to the *BPMNCodeGenerator* class.

Appendix B

In order to show its expressiveness, in this appendix CollabRDL is used to represent seven coordination-oriented workflow patterns.

B.1. The Parallel Split Pattern (WCP2)

The Parallel Split Pattern is defined as the division of a branch in one or more branches that can be executed in parallel. It is also known as AND-split, parallel routing, parallel split or fork. Fig. 9 shows this pattern in BPMN 2.0.

Code 14 shows the WCP2 pattern in CollabRDL. The PARALLEL command is the implementation of this pattern combined with the Role-based Distribution (WRP2) pattern presented in Section B.6. The *A* activity in BPMN is mapped to *activityA* in CollabRDL. The division of branches is indicated by PARALLEL in CollabRDL and by the square symbol with a circle and a cross in BPMN. The branches of the activities *B* and *C* in BMPN have a direct correspondence with the FLOW blocks in CollabRDL.

B.2. The Synchronization Pattern (WCP3)

This pattern defines the convergence of two or more branches into a single next branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Its synonyms are AND-join, rendezvous, and synchronizer. Fig. 10 shows this pattern in BPMN 2.0 and Code 15 shows the representation of this pattern in CollabRDL.

The *activityC* in CollabRDL has a direct correspondence with the *C* activity in BPMN. They are performed only after the execution of the FLOW blocks in CollabRDL and the activities *A* and *B* in BPMN, respectively. Thus, we conclude that CollabRDL implements the Synchronization Pattern (WCP3), AND-join.

B.3. The Exclusive Choice Pattern (WCP4)

The Exclusive Choice Pattern defines the division of a branch into two or more branches. When the input field is enabled, the thread of control is immediately passed to an output branch based on the result of a logical expression associated with the branch. Its synonyms are XOR-split, exclusive OR-split, conditional routing, switch, decision, or case statement. Fig. 11 illustrates this pattern in BPMN 2.0 and Code 16 shows the representation of this pattern in CollabRDL. The IF-ELSE command is part of the RDL and was inherited by CollabRDL without amendment.

B.4. The Simple Merge Pattern (WCP5)

Fig. 12 shows this pattern in BPMN 2.0 and Code 17 shows the representation of this pattern in CollabRDL. The Simple Merge Pattern defines the convergence of two or more branches into a single subsequent branch. The outputs of the branches are inputs to the control segment of the subsequent branch. The next activity will use only one of the outputs of previous branches. Its synonyms are XOR-join, exclusive OR-join, asynchronous join, and merge. This code does not represent the case when activities A and B start, and the flow goes to C when at least one of these two activities ends. However, there must be more evidence that this type of control is needed in a collaborative reuse before adding it into CollabRDL. Thus, we conclude that CollabRDL still does not fully implement the Simple Merge pattern (WCP5).

```
//Tokens
PARALLEL = 'PARALLEL';
parallel_statement
:     PARALLEL
      LEFT_CURLY { codeGen.addFork(); }
        flow_statement flow_statement (flow_statement)*
       { codeGen.addJoin(); } RIGHT_CURLY
;
flow_statement
: BEGIN_FLOW LEFT_PARENTHESIS role COMMA comment
  RIGHT_PARENTHESIS { codeGen.addBeginFlowBlock ($role.text, $comment.text); }
  statement_block { codeGen.addEndFlowBlock(); }
;
```

Code 12. Grammar of PARALLEL command.

```
//Tokens
BEGIN_DOPARALLEL = 'DOPARALLEL';
END_DOPARALLEL = 'WHILE';
// ------
doparallel_statement
:BEGIN_DOPARALLEL
                    { codeGen.addBeginDoparallelBlock(); }
      statement_block
END_DOPARALLEL LEFT_PARENTHESIS condition RIGHT_PARENTHESIS
SEMICOL { codeGen.addEndDoparallelBlock($condition.text);}
;
```

Code 13. Grammar of DOPARALLEL command.

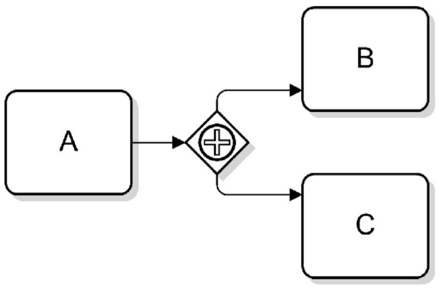


Fig. 9. Parallel Split Pattern (AND-split) in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
activityA;
PARALLEL{
    FLOW (...){
        activityB;
    }
    FLOW (...){
        activityC;
    }
};
```

Code 14. The Parallel Split Pattern (AND-split) represented in CollabRDL.

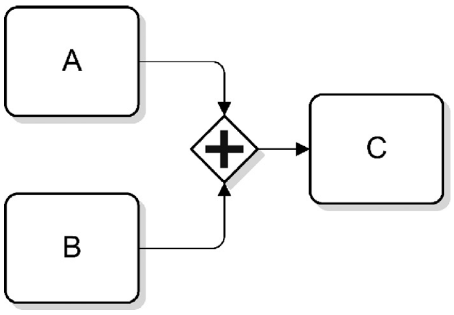


Fig. 10. Synchronization Pattern (AND-join) in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
PARALLEL{
    FLOW (...){
        activityA;
    }
    FLOW (...){
        activityB;
    }
}
activityC;
```

Code 15. The Synchronization Pattern (AND-join) represented in CollabRDL.

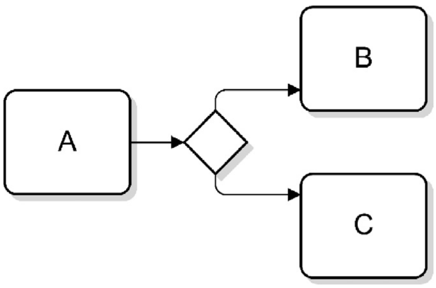


Fig. 11. Exclusive Choice Pattern (XOR-split) in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
activityA;
IF ("Do you want perform activityB?") THEN{
   activityB;
}ELSE{
   activityC;
};
```

Code 16. The Exclusive Choice Pattern (XOR-split) represented in CollabRDL.

B.5. The Multi-Choice Pattern (WCP6)

The Multi-Choice Pattern is defined as a branch splitting into two or more branches. When the input branch is enabled, the thread of control is immediately passed to one or more branches of the output based on the result of a logical expression associated with the branch. It is also known as Conditional routing, selection, OR-split, or multiple choices. Fig. 13 shows this pattern in BPMN 2.0 and Code 18 shows the representation of this pattern in CollabRDL.

Code 18 indicates that FLOW blocks will only run if they get a positive answer in the evaluation of the IF command. This combination with the IF statement transforms the PARALLEL command, which was originally an AND-split, into an OR-split. Thus, we conclude that CollabRDL implements WCP6.

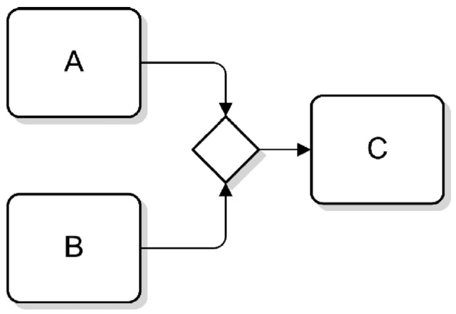


Fig. 12. Simple Merge Pattern (XOR-join) in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
IF ("Do you want perform activityA?") THEN{
   activityA;
}ELSE{
   activityB;
}
activityC;
```

Code 17. The Simple Merge Pattern (XOR-join) represented in CollabRDL.

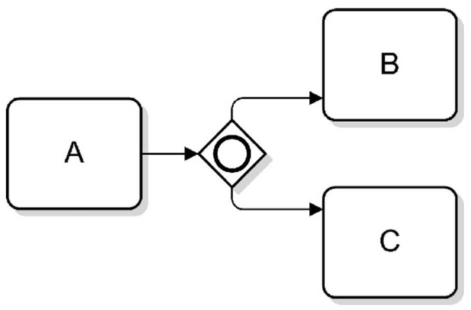


Fig. 13. Multi-Choice Pattern (OR-split) in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
activityA;
PARALLEL{
   FLOW (...){
      IF ("Do you want perform activityB?") THEN{
         activityB;
      }
   }
   FLOW (...){
      IF ("Do you want perform activityC?") THEN{
         activityC;
      }
   }
};
```

Code 18. The Multi-Choice Pattern (OR-split) represented in CollabRDL.

B.6. The Role-based Distribution Pattern (WRP2)

This pattern represents the roles defined for some activities at model time will be distributed to groups at runtime. Roles are used to gather people into groups that have common characteristics and skills. Fig. 14 shows this pattern in BPMN 2.0 using the *analyst* lane to mark the *analyst* role. Code 19 shows the ROLE command that was created to represent the WRP2 pattern. The first parameter of ROLE indicates that the *analyst* group is assigned to perform activities in the block, in this example, activity *activityA* only. As a result, we can conclude that the WRP2 pattern is implemented by the ROLE command in CollabRDL.

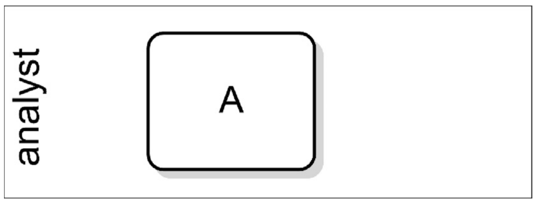


Fig. 14. Role-based Distribution Pattern in BPMN 2.0, adapted of Russell et al. (2006) and White (2004).

```
ROLE (analyst, “Comment.”){
   activityA;
};
```

Code 19. The Role-based Distribution Pattern represented in CollabRDL.

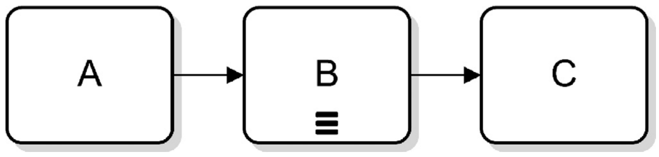


Fig. 15. Multiple instances of B in serial.

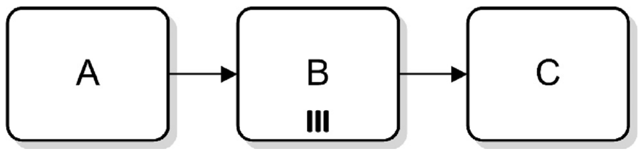


Fig. 16. Multiple instances of B in parallel.

```
DOPARALLEL{
   ROLE (analyst, "Comment."){
      activityA;
   }
}WHILE (“Run the block again?”);
```

Code 20. Multiple instances in parallel with decision at runtime represented in CollabRDL.

B.7. The Multiple instances with decision at runtime (WCP14)

This pattern describes the creation of multiple instances of an activity in a process. They are independent of each other, so they can be executed in parallel, and the number of instances will be decided at runtime. It is necessary to synchronize the instances at the end of their execution before passing the flow to the next activities. Figs. 15 and 16 show variants of this pattern, i.e., the serial and parallel of this pattern in BPMN 2.0 notation.

The DOPARALLEL command expresses blocks to be executed in parallel at runtime. Code 20 shows that *activityA* will be offered to the *analyst* group, and without waiting for its completion, the question "Run the block again?" will be issued. Another instance of *activityA* will be offered to the *analyst* group if the reuser answers yes. Another case, if the reuser answers no, the flow will go to the end of DOPARALLEL, marked by a semicolon (";"), which will wait for the completion of all instances of *activityA* that were created.

In other situations, which represent a different parallel behavior obtained with the DOPARALLEL command, there is a need to repeat a sequence of activities for a number of times defined at runtime. One way to express this is through the LOOP command. Code 21 shows this situation, in which the question "Run the block?" is asked at runtime and, if the answer is positive, the

```
LOOP ("Run the block?"){
    ROLE (...){
        activityA;
    }
}
```

Code 21. Multiple instances in serial with decision at runtime represented in CollabRDL.

activityA is performed, and, only after its execution, the question LOOP will be reissued. This behavior indicates that the LOOP command produces the execution of n blocks of code serially and the variable n is set at runtime.

We conclude that CollabRDL implements the Multiple Instances with a decision at runtime pattern. In the parallel case the commands DOPARALLEL and FLOW can be used, and in the serial case the commands LOOP and ROLE are sufficient to represent the pattern.

References

Abbasi, E.K., Hubaux, A., Heymans, P., 2011. A toolset for feature-based configuration workflows. In: *Software Product Line Conference* (SPLC), 2011 15th International, pp. 65–69. vol., no.22-26.

Activiti, http://www.activiti.org/, accessed in May 2015.

APL Documentation at http://www.sei.cmu.edu/productlines/ppl/, accessed in May 2015.

Arango, G..E, Prieto-Diaz, R., 1991. Introduction and overview: domain analysis concepts and research directions. Domain Analysis and Software Systems Modeling. IEEE Press.

Barthelmess, P., Anderson, K.M., 2002. A view of software development environments based on activity theory. Comput. Supported Cooperative Work 11 (1-2), 13–37.

Botterweck, G., Thiel, S., Nestor, D., Abid, S.B., Cawley, C., 2008. Visual tool support for configuring and understanding software product lines. In: Proceedings - 12th International Software Product Line Conference, SPLC 2008, p. 77.

Bellas, F., 2004. Standards for second-generation portals. IEEE Internet Comput. 8 (2), 54–60.

BPMN, http://www.bpmn.org/, accessed January 2016.

BPMN, Object Management Group. Business process modeling and notation, Version 2.0, formal/2011-01-03 January 2011.

Briggs, R.O., De Vreede, G., Nunamaker Jr., J.F., 2003. Collaboration engineering with thinklets to pursue sustained success with group support systems. J. Manage. Inf. Syst. 19 (4), 31–64.

Bull Corporation, FlowPath functional specification. Bull S. A., Paris, France, September 1992.

Cechticky, V., Chevalley, P., Pasetti, A., Schaufelberger, W., 2003. A generative approach to framework instantiation. In: Lecture Notes in *Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2830, pp. 267–286.

CollabRDL Page, 2015, http://prisma.cos.ufrj.br/collabrdl/, accessed June 2015.

Cortes, M., Mishra, P., 1996. DCWPL: A programming language for describing collaborative work. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work. Boston, Massachusetts, United States, pp. 21–29. November 1996.

CPE, Commercial Product Evaluation, http://www.workflowpatterns.com/evaluations/commercial/index.php, accessed in April 2015.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2005. "Staged configuration through specialization and multi-level configuration of feature models. Softw. Process 10 (2), 143–169.

De Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V.C., De Lemos Meira, S.R., 2005. A survey on software reuse processes. Information Reuse and Integration, Conf, 2005. IRI -2005 IEEE International Conference on. vol., no., pp.66,71, 15-17 Aug. 2005.

De Paoli, F., Tisato, F., 1994. CSDL: a language for cooperative systems design. IEEE Trans. Softw. Eng. 20 (8), 606–616.

Di Ruscio, D., Eramo, R., Pierantonio, A, 2012. "Model transformations. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7320 LNCS. Bertinoro Itária, pp. 91–136.

Dourish, P, Bellotti, V, 1992. Awareness sand coordination in shared workspaces. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW). Toronto, pp. 107–114.

Ellis, C.A., Gibbs, S.J., Rein, G.L., 1991. "Group-ware: some issues and experiences. Commun. ACM 34 (1), 38–58.

EWP. Evaluating of workflow products, http://www.workflowpatterns.com/vendors/index.php, accessed in April 2015.

Filho, I.M., De Oliveira, T.C., De Lucena, C.J.P., 2004. A framework instantiation approach based on the features model. J. Syst. Softw. 73 (2), 333–349.

Frakes, W.B., Succi, G., 2001. An industrial study of reuse, quality, and productivity. J. Syst. Softw. 57 (2), 99–106.

Frakes, W.B., Kang, K., 2005. Software reuse research: status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536.

Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.G., 1997. Hooking into object-oriented application frameworks. In: Proceedings of the 19th International Conference on Software Engineering. Boston, pp. 491–501. May.

Fuggetta, A., 2000. Software process: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, pp. 25–34.

Fuks, H., Raposo, A., Gerosa, M.A., Pimentel, M., Lucena, C.J.P., 2007. The 3C collaboration model. In: The Encyclopedia of E-Collaboration, Ned Kock (org), pp. 637–644.

GEF Documentation at http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html accessed October 2015.

Gomaa, H., 2004. Designing Software Product Lines with UML. Addison Wesley.

Gomes, T.L., Oliveira, T.C., Cowan, D., Alencar, P., 2014. Mining reuse processes. In: CIBSE 2014: *Proceedings of the 17th Ibero-American Conference Software Engineering*, p. 179.

Hepper, S., 2008, Java portlet specification version 2.0 ,jsr-286.

Holmes, R., Walker, R.J., 2013. Systematizing pragmatic software reuse. ACM Trans. Softw. Eng. Methodol. 21 (4), 44 Article 20 (February 2013).

HotDraw Documentation at www.jhotdraw.org/, accessed October 2015.

Hubaux, A., Classen, A.,E, Heymans, P., 2009. Formal modelling of feature configuration workflows. In: Proceedings of the 13th International Software Product Line Conference (SPLC '09). Pittsburgh, PA, USA. Carnegie Mellon University, pp. 221–230.

JMI. Java metadata interface specification JSR 040 Java Community Process. http://www.jcp.org/.

Johnson, R.E., Foote, B., 1988. Designing reusable classes. In: Journal of Object-Oriented Programming, 1, pp. 22–30. 35.

Jouault, F., Kurtev, I., 2006. "Transforming models with ATL. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3844 LNCS, pp. 128–138.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., 1990. "Feature-oriented domain analysis (FODA) feasibility study", SEI, CMU, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-21.

Kovalski, F., 2005. SPEMTool. Portuguese, Undergrad Final Assessment at Faculty of Informatics.

Krasner, G.E., Pope, S.T., 1988. A "cookbook for using the model-view-controller user interface paradigm in smalltalk-80". J. Object-Oriented Program. 1 (3).

Lano, K., 2008. Constraint-driven development. Inf. Softw. Technol. 50 (5), 406–423.

Li, D., Muntz, R.R., 2000. A collaboration specification language. SIGPLAN Notices (ACM Spec. Interest Group Program. Lang.) 35 (1), 149–162.

Lim, W.C, 1994. Effects of reuse on quality, productivity, and economics. IEEE Softw. 11 (5), 23–30.

Maia, N., Bacelo, A.P.T., Werner, C.M.L., 2007. Odyssey-MDA: a transformational approach to component models. In: International Conference on Software Engineering and Knowledge Engineering (SEKE'2007). Boston, USA, pp. 9–14. July.

Malone, T.W., Crowston, K., 1990. What is coordination theory and how can it help design cooperative work systems? In: Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work (CSCW '90). New York, NY, USA. ACM, pp. 357–370.

Malone, T.W., Crowston, K., 1994. Interdisciplinary study of coordination. ACM Comput. Surv. 26 (1), 87–119.

Mendonça, M., Oliveira, T.C.; Lucena, Carlos J.P.; Alencar, Paulo; Cowan, Donald D. Assisting Framework Instantiation: Enhancements to Process-Language-based Approaches Technical Report School of Computer Science at University of Waterloo, CS-2005-25, 2005, http://www.cs.uwaterloo.ca/research/tr/2005/25/CS-2005-25.pdf.

Mendonça, M, Cowan, D, Oliveira, T, 2007. A process-centric approach for coordinating product configuration decisions. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07). IEEE Computer Society.

Mendonça, M., Cowan, D., Malyk, W., Oliveira, T., 2008. Collaborative product configuration: formalization and efficient algorithms for dependency-analysis. J. Softw. 3 (2), 69–82.

MOF (2002) Meta Object Facility (MOF) specification, version 1.4, OMG.

Mohagheghi, P.E, Conradi, R., 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Softw. Eng. 12 (5), 471–516.

Mohamed, F., Schmidt, D.C., 1997. Object-oriented application frameworks. Commun. ACM 40 (10), 32–38 (October 1997).

Noor, M.A., Grünbacher, P., Briggs, R.O., 2007. A collaborative approach for product line scoping: a case study in collaboration engineering. In: *Proceedings of the IASTED International Conference on Software Engineering, SE 2007*, p. 216.

Northrop, L.M., 2002. SEI's software product line tenets. IEEE Softw. 19 (4), 37–40.

OASIS, ebXML Business Process Specification Schema, Technical Specification v2.0.4, December 2006.

Oliveira, T.C., Alencar, P., Cowan, D., 2011. ReuseTool - An extensible tool support for object-oriented framework reuse. J. Syst. Softw. 84 (12), 2234–2252.

Oliveira, T.C., Alencar, P., Cowan, D., Filho, I.M., Lucena, C.J.P., 2004. Software process representation and analysis of framework instantiation. IEEE-Trans. Softw. Eng. p145–p159 March.

Oliveira, T.C., Alencar, P.S.C., De Lucena, C.J.P., Cowan, D.D., 2007. RDL: a language for framework instantiation representation. J. Syst. Softw. 80 (11), 1902–1929.

Openswing, 2015, http://oswing.sourceforge.net/.

OSPE, Open Source Product Evaluation, http://www.workflowpatterns.com/evaluations/opensource/index.php, accessed in April 2015.

Rabiser, R., Grünbacher, P., Dhungana, D., 2007. Supporting product derivation by adapting and augmenting variability models. In: Proceedings - 11th International Software Product Line Conference, SPLC 2007. Kyoto, Japan, p. 141.

Rothenberger, M.A., Dooley, K.J., Kulkarni, U.R., Nada, N., 2003. Strategies for software reuse: a principal component analysis of reuse practices. IEEE Trans. Softw. Eng. 29 (9), 825–837.

Runeson, P., Host, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Eng. 14 (2), 131–164 April 2009.

Russell, N., Ter Hofstede, A.H.M., Van Der Aalst, W.M.P. and Mulyar, N., "Workflow control-flow patterns: a revised view". BPM Center Report BPM-06-22, BPMcenter.org, 2006.

Scott, M.L., 2009. Programming Language Pragmatics, 3 ed. Morgan Kaufmann, New York.

Smith, R.B., Hixon, R.E, Horan, B., 1998. Supporting flexible roles in a shared space. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work. New York, NY, USA, pp. 197–206.

SPLOT - Software Product Lines Online Tools http://www.splot-research.org, accessed in May 2016.

Swing, http://docs.oracle.com/javase/tutorial/uiswing/start/about.html, accessed in May 2016.

Tutorial Openswing, http://oswing.sourceforge.net/tutorial.html, accessed in May 2016.

Uschold, M., King, M., Moralee, S., Zorgios, Y., 1998. The enterprise ontology. Knowl. Eng. Rev. 13 (1), 31–89.

Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B.E, Barros, A.P., 2003. Workflow patterns. Distrib. Parallel Databases 14 (1), 5–51.

WFMC, Workflow Management Coalition, 1999, *Workflow management coalition terminology & glossary*. Document Number WFMC-TC-1011, Issue 3.0.

White, S.A., 2004. Process Modeling Notations and Workflow Patterns. IBM Corp., United States http://www.workflowpatterns.com/vendors/documentation/BPMN_wfh.pdf.

Wohed, P., Russell, N., Ter Hofstede, A.H.M., Andersson, B., Van Der Aalst, W.M.P., 2009. Patterns-based evaluation of open source BPM systems: the cases of jBPM, OpenWFE, and Enhydra Shark. Inf. Softw. Technol. 51 (8), 1187–1216.

XMI (2002) XML Metadata Interchange (XMI) specification, v1.2, OMG.

Edson Mello Lucas is currently a PhD student at Federal University of Rio de Janeiro (COPPE/UFRJ), Brazil. He received his Master in Systems Engineering and Computer Science from COPPE/UFRJ in 2013. He is also a systems analyst at Polytechnic Institute, State University of Rio de Janeiro, Brazil (IPRJ/UERJ). Edson has experience in computer science, with emphasis on Software Engineering, acting on the following topics: software reuse; frameworks; systems analysis, design and programming object-oriented; collaborative processes and content management systems.

Dr. Toacy Oliveira is an Assistant Professor at Federal University of Rio de Janeiro (COPPE/UFRJ), Brazil. He is also Adjunct Professor with the David R. Cheriton School of Computer Science at the University of Waterloo, Canada. He received his education at Pontifical Catholic University of Rio de Janeiro, Brazil (Electrical Engineering - 1992, MSc-1997, PhD - 2001) and spent 3 years at University of Waterloo as a posdoc. His current research interests are under the software engineering umbrella, including software processes and software reuse. Toacy focuses on the use notations, models, processes and tools to improve the way software systems are developed. He has published over 40 refereed publications, and has been a member of program committees of numerous highly-regarded conferences and workshops. He has been a leading investigator in national projects supported by CAPES and CNPq. Toacy also has strong attachments to the local software industry that lead to the creation of a software development company in 1998, the OWSE Informatica that currently employes more than 50 software engineers.

Dr. Kleinner is an Assistant Professor in the Interdisciplinary Postgraduate Program in Applied Computing (PIPCA) at the University of Vale do Rio dos Sinos (Unisinos). Farias received his Ph.D. in Software Engineering in 2012 from the Department of Informatics at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He also received an M.S. in Computer Science in 2008 from the Pontifical Catholic University of Rio Grande do Sul (PUC-RS). As a researcher, his work focuses on software modeling, domain-specific modeling language, empirical software engineering, aspect-oriented software development, software architecture, neuroscience applied to software engineering, and model-driven software development. Today, Farias investigates how software developers invest effort to compose design models created in parallel by different development teams. He has published his research results in premier software engineering conferences, including ICSE, MODELS, AOSD, and journals, including JUCS and SoSym. Farias is a member of the OPUS Research Group and MobiLab.

Dr. Paulo Alencar is a Research Professor with the David R. Cheriton School of Computer Science and Associate Director of the Computer Systems Group. He has more than 20 years of experience in the software engineering and formal methods, and has made significant contributions to many areas, including software design and architectures, user interfaces, (mobile) web-based systems, open and big data applications, software agents, and event and context-oriented applications. Dr. Alencar has received international research awards from organizations such as Compaq (Compaq Award for best research paper in 1995 for his work on a software evolution theory) and IBM (IBM Innovation Award, 2003). He has published over 180 refereed publications, and has been a member of program committees of numerous highly-regarded conferences and workshops. He has been a leading investigator in national projects supported by NSERC, Sybase, IBM, CITO, CSER, SAP and Bell, and has engaged in many collaborative international projects involving Germany, Argentina, Canada, USA and Brazil.