On the Use of Model Transformations for the Automation of the 4SRS Transition Method

Sofia Azevedo[1], Ricardo J. Machado[1], and Rita Suzana Pitangueira Maciel[2]

[1] Universidade do Minho, Portugal
{sofia.azevedo,rmac}@dsi.uminho.pt
[2] Universidade Federal da Bahia, Brazil
ritasuzana@dcc.ufba.br

Abstract. Automation is the essence of MDD (Model-Driven Development). Transforming models into models following a set of rules is at the core of automation. It allows using tools to enliven processes that have been defined. Transition methods are most likely the most important player in the engineering of software. The 4SRS (Four Step Rule Set) is a transition method we adopt in this paper to focus the discussion on the transition from the analysis to the design of software. It has been formalized as a small software development process that can be plugged into larger software development processes. That formalization was conducted with the SPEM (Software & Systems Process Engineering Metamodel), which is a process modeling language for the domain of software and systems. This paper exemplifies how a transition method like the 4SRS can be modeled with the SPEM as a way to study the benefits of the automatic or semiautomatic execution of a transition method as a small dedicated software development process.

Keywords: automation, software process modeling, transition method, software development process, software process modeling language.

1 Introduction

The automation of software processes may be facilitated by process modeling. Method modeling is also essential when it comes to process modeling. The same method can be used by many different processes and more than once in the same process. A method can be defined as a general description of how to develop software and systems. This description shall be the basis for defining or formalizing software and systems development processes. Software and systems development processes can be described as sequences of phases and milestones. The sequence of phases and milestones represents the development lifecycle of the product, so processes may represent product development lifecycles. In this context, methods are only contextualized in a development lifecycle when positioned within a process. Processes can be modeled with process modeling languages like the SPEM (Software & Systems Process Engineering Metamodel) [1]. A process modeling language can be defined as the instrument to express software development processes through a process model [2].

M. Bajec and J. Eder (Eds.): CAiSE 2012 Workshops, LNBIP 112, pp. 249–264, 2012.

The 4SRS (Four Step Rule Set) [3-5] method allows the iterative and incremental model-based transition from user functional requirements (represented as use cases) to system functional requirements (or logical architectures represented as component models). In other words the transformation of use cases (which are dealt with during the analysis of software development) into logical architectures (which are dealt with during the design of software development) is conducted with a method specifically conceived for the purpose. The main concern of the method is: (1) not to lose any user requirements when moving from the analysis to the design; and (2) to assure that no user requirements that have not been elicited with the customer are considered.

A logical software architecture can be faced as a view of a system composed of a set of problem-specific abstractions composed from the system's functional requirements and it is represented as objects or object classes [6]. Another definition of logical software architecture is a module view representing the static structure of the software system (the system's functional blocks, including traceability back to the use cases that express the system's functional requirements) [7]. In the context of this paper, a logical software architecture (represented as a component model) is a design artifact representing a functionality-based structure of the system being designed.

The SPEM (2.0) is a software and systems development process modeling language. We used the SPEM to model the 4SRS as a process. We showed how to model transition methods as processes with a process modeling language (the SPEM) and we used the 4SRS as the example of a transition method to illustrate our approach. In the context of this paper, a transition method is a method that describes how to transform analysis artifacts (use case diagrams) into design artifacts (component models) to develop software. Some other transition methods may for instance describe how to transform design artifacts into implementation artifacts, or how to generate test artifacts, or how to transform business modeling artifacts. Being the 4SRS a method, we formalized it as a small dedicated (at transitioning from analysis to design) software development process that can be plugged into larger software development processes.

The problem this paper addresses is the automation of transition methods, particularly those modeled with the SPEM. The 4SRS was modeled with the SPEM in order to formalize it as a software process. It had to be automated so that it could be enlivened by means of a tool. The SPEM was chosen because it is standard, therefore it would be possible to benefit from the advantages of using a standard that is available to every professional of process modeling. Assuming that disciplines are sets of tasks that can be grouped according to their particular relevance in specific phase(s) from large software development processes into which small dedicated software development processes can be plugged into, transition methods are methods that describe how to transform artifacts from one discipline of a large software development process into artifacts from another discipline of such a process. Transition methods have particularities regarding other methods. They realize a change in the perspective on the system, consequently in the artifacts that represent the system from different perspectives, as well as they mark a change in the phase of the large software development process. In this paper, we use the 4SRS as the example of a transition method modeled with the SPEM and we illustrate the

automation of transition methods modeled with the SPEM, in this case a transition method that transforms analysis artifacts into design artifacts. The goal of the automation of transition methods modeled with the SPEM is the automatic or semiautomatic execution of those methods as small dedicated software development processes, in this case the 4SRS. The intent is to decrease the cost of introducing the method into large software development processes, facilitating its use. The (semi)automatic execution of the 4SRS transition method was based on the Moderne [8], which is a tool developed in the Federal University of Bahia (Brazil). The Moderne is a model-driven tool of process modeling and execution. The Moderne tool allows the execution of the 4SRS in an explicit model-driven approach, which implies generating logical architectures through model transformations using a model-to-model transformation language [9, 10].

The paper is structured as follows. Section 2 exposes basics concepts to understand software process modeling, automation, execution and enactment. Section 3 is on work related to process execution through tools, on the SPEM and on process architecture. Section 4 elaborates on a vision over the SPEM and presents the 4SRS transition method. Section 5 shows the preparation necessary for the automation of transition methods modeled with the SPEM, particularly the work undertaken to prepare the automation of the 4SRS. Section 6 presents the case study. Finally Section 7 provides some concluding remarks.

2 Basic Concepts

The goal of processes is to assure the quality of products and the productivity in developing them [11]. Process comprehension and process communication may be negatively affected by the lack of a standard and unified terminology [12]. In such conditions process enactment is far away from process definition, thus the quality of products and the productivity in developing them may be compromised and the goal of processes may not be achieved. Process modeling using a standard and unified terminology suits some purposes like process understanding, process design, process training, process simulation and process support, among others [13]. A short definition of process is the way activities are organized to reach a goal [14]. In the case of software development, a process (software process) can be defined as the set of activities (analysis, design, implementation, testing, among others) organized to deliver a software product (goal). A process model is an artifact that expresses a process with the intention of (among other things) automating that process. A tool supports the execution of the process to consistently reach the goal (delivering a software product).

In 1995 Conradi and Liu [15] says that enactable process models are low-level process models in terms of abstraction. Process modeling languages suit the purpose of detailing process models to make them enactable. According to Henderson-Sellers [16], a process enactment is an instance of a process in a particular project with actual people playing roles, deadlines having real dates and so on. Different enactments have different people playing the same role and different dates for the same deadline.

Bendraou, *et al.* [17] considers that process enactment shall contemplate support for automatic task assignments to roles, automatic routing of artifacts, automatic control on work product states, among others. In what process enactment is concerned, Feiler and Humphrey [18] define an enactable process as an instance of a process definition that shall have process inputs, assigned agents (people or machines that interpret the enactable process), an initial state, a final state and an initiation role. A process definition is a set of enactable process steps. Process definitions can be a composition of subprocess definitions as well as process steps can be a composition of process substeps. A process definition only fits for enactment when fully refined, which means that it cannot be more decomposed into subprocess definitions and into process substeps. The act of creating enactable processes from process definitions is defined by Feiler and Humphrey as process instantiation. They define process enactment as the execution of a process by a process agent following a process definition. Bendraou, *et al.* [17] consider that support for the execution of process models helps coordinating participants, routing artifacts, ensuring process constraints and process deadlines, simulating processes and testing processes. The use of machines in process enactment is called process automation and requires for a process definition to be embodied in a process program [18]. Gruhn [19] defines automatic activities as those executed without human interaction. He mentions that the automation of activities is one of the purposes of process modeling. Another purpose is to govern real processes on the basis of the underlying process models.

3 Related Work

Processes can be executed through tools. In [11] Osterweil considers coding software processes (as part of programming them). Process modeling is one of the parts of programming a software process with a model-driven approach. Software process code is in a lower abstraction level when compared to software process models and it can be executed by computers. Software process code specifications require that software process models define (for new software processes) or formalize (for existing software processes) how software artifacts shall be input to or output from software process tools and how those artifacts are to be handled by the right roles at the right time of the process. Software process models can be analyzed to identify process steps that may be automated. The number of processes being followed to develop software is high. Some key software development processes like software requirements specification and software design lack definition (if they're new ones) or formalization (if they're existing ones). Software design for instance is a process that can be modeled and coded. This paper shows an approach to code a previously modeled software development process, which is the 4SRS transition method modeled with the SPEM as a small dedicated software development process. We analyzed software process models to identify process steps that might be automated with the Moderne tool.

A process' capability can be assessed according to three criteria [20]: task customization, project customization and maturity customization. According to

Henderson-Sellers, *et al.* [20], SPEM allows for task customization and for project customization, but not for maturity customization. SPEM allows for task customization because it allows for the selection of techniques for each task according to the organization and the expertise of the professionals in those techniques. For instance various techniques can be used to elaborate a requirements specification depending on the organization and the project: questionnaires, workshops, storyboards, prototypes and others. SPEM allows for project customization since it allows for the selection of activities, tasks and techniques according to the project. The tasks required to be performed and the products to be developed (models and documents) vary from one project to another. Project customization is a matter of selecting or omitting portions of a process. SPEM does not allow for the addition/removal of activities and tasks to/from a process, and consequently work products depending on the capability or maturity level of the organization. Furthermore Henderson-Sellers, *et al.* [20] refers that SPEM allows for the definition of discrete activities and steps, therefore allowing for process fragment selection.

Kruchten [6] defines development architecture in his "4+1" View Model of Software Architecture. The software system is structured in subsystems that shall be developed by one or a small number of developers (a team). That structure of subsystems is the development architecture, which can be used to allocate work to teams. The development architecture is in fact a logical architecture. In a higher level of abstraction a logical architecture may be process-based (according to our designation) or a process architecture (according to Kruchten's designation), consisting of a functionality-based structure of the process being designed. To our concern a product-based logical architecture or product architecture is an architecture that resides in a lower level of abstraction comparatively to a process-based or process architecture and consists of a functionality-based structure of the product being designed. Allocating work to teams developing subsystems (as the development architecture can be used to) presupposes that those subsystems can also represent process architecture components that consist of tasks (ultimately steps) that are performed by some roles to produce some output (work products). In fact process architecture components are activities that compose a process structure. Activities are a (kind of) work breakdown element. We can conclude that the 4SRS is not only a method dedicated at transitioning from analysis to design but can also be a method for defining a development architecture (or process architecture). This paper shows how to automate transition methods, particularly those modeled with the SPEM. We use the 4SRS previously modeled with the SPEM as the example of a transition method modeled with the SPEM and we illustrate the automation of transition methods modeled with the SPEM by means of this transition method that transforms analysis artifacts into design artifacts. Automated transition methods modeled with the SPEM can be automatically or semiautomatically executed as small dedicated software development processes. This paper focuses on transition methods from the product development point of view and not from the process architecture point of view.

4 Synopsis of the 4SRS Transition Method

This paper shows how to automate transition methods, particularly those modeled with the SPEM. We use the 4SRS previously modeled with the SPEM as the example of a transition method and we illustrate the automation of transition methods modeled witht the SPEM by means of this transition method that transforms analysis artifacts into design artifacts. Automated transition methods modeled with the SPEM can be automatically or semiautomatically executed as small dedicated software development processes.

The SPEM is a modeling language that contains the minimal elements to define software and systems development processes. The SPEM distinguishes between the concepts of process and of method content. From the process' perspective, activities represent work that can be assigned to roles that intervene on the process, and require inputs and/or outputs (work products) to be performed. Activities are relevant for modeling phases of a development lifecycle (waterfall, iterative and incremental are three types of development lifecycles). The same role can be used in an early phase of a development lifecycle represented by an activity and in a later phase of the same development lifecycle represented by another activity, which may mean that e.g. that role will handle different work products in those two different phases.

A method content can be considered as a set of concepts (tasks, roles and work products) that allows representing a software and systems development method and its techniques which can be positioned within a specific software and systems development lifecycle (consider that a method is composed of many techniques and that a development lifecycle can be composed of a sequence of phases and milestones for example). Processes shall use method content elements and organize them into sequences. A method content is a stepwise definition of tasks that shall be performed by roles to originate work products. They may consume work products as well. A task from a method content may have its steps, inputs or outputs (work products) changed depending on the development lifecycle it is positioned.

The main difference between a method content and a process is that a method content defines methods and techniques for software and systems development processes, whereas a process defines the positioning of those methods and techniques within a development lifecycle composed of e.g. a sequence of phases and milestones. When a specific composition of tasks, roles and work products (a specific method content) is positioned within a development lifecycle, it means that the method content has been applied to that part of the process where it has been positioned. It also means that the method content was used by that process.

The 4SRS is a method for obtaining system functional requirements from user functional requirements [3]. Use cases model user functional requirements and logical architectures model system functional requirements. Use cases are problem-related, technology-independent and are dealt with during the analysis phase of software development. Logical architectures are solution-related, technology-independent and are dealt with in the beginning of the design phase of software development.

According to Kaindl [21], we can classify 4SRS as a transition method. Kaindl argues that it is difficult to move from the analysis to the design of software. From the

perspective of object-oriented software development, the main reason is that analysis objects and design objects represent different kinds of concepts. Analysis objects are from the problem domain and represent objects from the real world. Design objects are from a solution domain and shall indicate how the system shall be developed. Design objects are abstractions of code or the implementation details needed in order to build a system with that solution to that problem. Design objects are both an abstraction of concepts from the problem domain and of the implementation of the system to be built. An analysis model can become part of a design model by influencing architectural decisions. The 4SRS is a method that allows moving from the analysis to the design of software. In the case of the 4SRS, the analysis model (a UML (Unified Modeling Language) [22] use case diagram) influences architectural decisions that originate the design model (a UML component model).

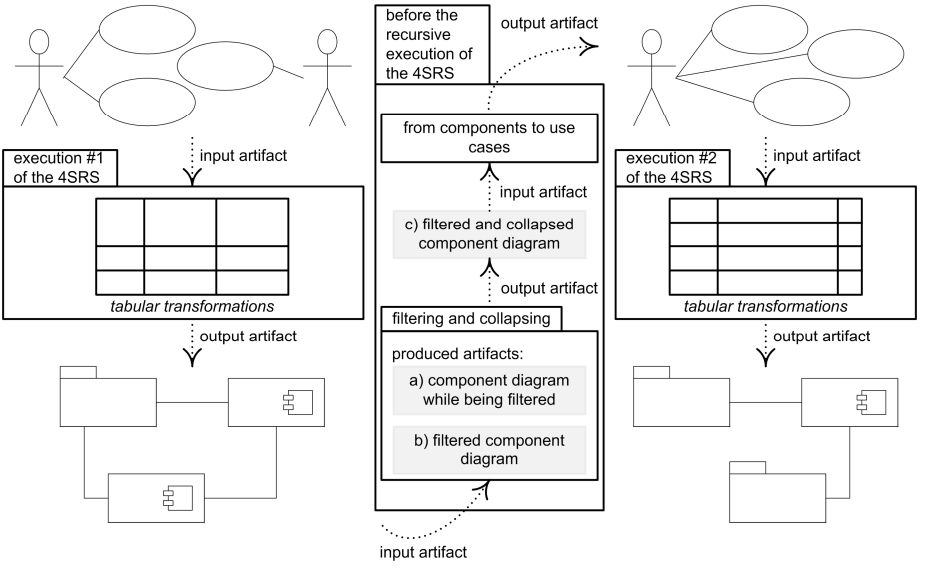

Fig. 1. Schematic representation of the recursive execution of the 4SRS method

Shortly the 4SRS method is composed of the following steps: (1) *Component Creation*, to create three kinds of components for each use case (an interface component, a control component and a data component; other kinds of components could be created, so this is not a limitation of the method, rather an architectural decision); (2) *Component Elimination*, to remove redundant requirements and find missing requirements (this step is vital in order to validate the components blindly created in the previous step and includes eliminating the components whose requirements are already represented by other components; the finding of missing requirements means that components have been inadequately eliminated or use cases are missing); (3) *Component Packaging and Nesting*, to semantically group components in packages; and (4) *Component Association*, to define associations of components with each other in the component model.

Architectural refinement is the approach the 4SRS method takes to increment a primary logical architecture with detail (by primary we mean the architecture that is going to be detailed) [4]. In the context of this paper, recursion is the ability of the method to be executed over parts of its output artifact after transformed into the input artifact for that execution. As depicted in Figure 1, the 4SRS method may be applied recursively, in several *executions*. In the context of each one of those executions, various *iterations* can be performed. Although there is no stopping rule for iterating over the same set of use case diagrams, it shall be performed until the results obtained generate a logical architecture that does not benefit from additional iterations in terms of the elimination of redundant requirements, the finding of missing requirements and the increasing of the logical architecture's cohesion. There cannot be components isolated from the rest of the architecture when the global architecture is composed from the various logical architectures generated by the different executions. In the case of refinement (by recursion), when one of the executions is considered to be finished by the modeler, the output of that execution's last iteration (a component model) is going to originate the input of a subsequent execution's first iteration (a use case diagram). The task flow of the new execution is exactly the same as the task flow of the preceding one. Again in the case of refinement (by recursion), the logical architectures produced by the various executions are situated in lower levels of abstraction and cover less functionality than the logical architectures they refined.

Considering architectural refinement, the sequence of steps for the 4SRS method is the following: (1) *Component Creation*; (2) *Component Elimination*; (3) *Component Packaging and Nesting*; (4) *Component Association*; (4+1) *Filtering and Collapsing*; and (4+2) *From Components to Use Cases*. The first four steps are the original steps and the other ones are what we call the *intermediate steps*, which are performed in between executions of the 4SRS method. The step 2 is composed of seven microsteps. The microstep (2.i) *Use Case Classification* is about determining the kinds of components that will originate from each use case according to the eight possible combinations. According to this classification, the microstep (2.ii) *Local Elimination* is about eliminating the components blindly created in the step 1 by analyzing the textual description of the use cases and deciding on whether those components make sense in the problem domain. The microstep (2.iii) *Component Naming* is about naming the components that have not been eliminated in the previous microstep. The microstep (2.iv) *Component Description* is about textually describing the components named in the previous microstep, based on the textual descriptions of the use cases they originated from, on nonfunctional requirements and on design decisions. The microstep (2.v) *Component Representation* is about determining whether some components represent both their own system requirements and others'. The microstep (2.vi) *Global Elimination* is about eliminating the components whose requirements are already represented by other components (elimination of functional redundancy). Finally the microstep (2.vii) *Component Renaming* is about renaming the components that were not eliminated in the previous microstep and that represent additional components.

The filtering consists of considering some components as the subsystem for refinement and discarding those that are not associated with them [5]. The collapsing

consists of hiding the details of the subsystem whose components are going to be refined. Later on, those components are replaced inside the limits of the subsystem's border by others of lower abstraction level.

The intermediate step (4+2) *From Components to Use Cases* of the 4SRS method is composed of two intermediate substeps: the (4+2.i) *Deriving Use Cases from Components* and the (4+2.ii) *Detailing Use Cases*. The goal of the (4+2.i) *Deriving Use Cases from Components* is to derive the use cases to hand out as input for the succeeding recursive execution of the 4SRS method from the components to refine. The goal of the (4+2.ii) *Detailing Use Cases* is to refine those use cases.

5 Automating the 4SRS Method

The goal of automated transition methods modeled with the SPEM is to automatically or semiautomatically execute them as small dedicated software development processes. That goal was achieved with the support of a tool. That tool is the Moderne, which is a tool developed at the Federal University of Bahia (Brazil). The Moderne is a model-driven tool of process modeling and execution. The Moderne tool allows the execution of the 4SRS in an explicit model-driven approach, which implies generating logical architectures through model transformations using a model-to-model transformation language. These model transformations can be executed with any ATL (Atlas Transformation Language) [9] engine that uses UML, and not only with the Moderne.

This section exposes the way the 4SRS transition method modeled with the SPEM has been automated according to our definition of goal for the automation of transition methods modeled with the SPEM: the automation allows the automatic or semiautomatic execution of these transition methods as small dedicated software development processes. By automatic it is meant that models (the artifacts) are transformed using a transformation language or based on some action the modeler (the tool user) performs with the tool (to which the tool is programmed to respond) or even based on rules the tool has been programmed with to respond to some particular event without any modeler's action. By semiautomatic it is meant that the tool supports decisions the modeler has to make by allowing him to represent them in the diagrams.

The modeling of the 4SRS transition method with the SPEM we performed beforehand (and that is not the focus of this paper) had to be adapted in order for the method to be automatically or semiautomatically executed as a microprocess with the Moderne tool. In the context of this paper, a microprocess is a small dedicated software development process dedicated at transitioning from analysis to design and that can be plugged into larger software development processes. From the perspective of the SPEM, a process can be considered to be at least a method content (or shortly method) positioned within a development lifecycle. In this paper, a method defines tasks (composed of steps), roles and work products, therefore methods are modeled with the following elements: tasks (and steps), roles and work products. We subclassed tasks into transition tasks and intermediate tasks, steps into transition steps and intermediate steps, and finally work products into initial work products, intermediate work products and final work products. Figure 2 illustrates some

examples of these elements. In the case of tasks, steps and work products, the stereotypes respectively indicate the type of task, step or work product according to the subclassing just mentioned.

The model of the 4SRS transition method with the SPEM (elaborated beforehand) was adapted by adding the transformation rule *i-c-dComponentsFromLeafUseCases* as an input to the transition task *ComponentCreation* and by adding the intermediate task *UseCaseDiagramDrawing* to the model. The transformation rule is an ATL [9] rule that defines how to transform the use case diagram (the initial work product *UseCaseDiagram*) into the component diagram (the intermediate work product *i-c-dComponents*). The intermediate task had to be modeled to give the input (initial work product *UseCaseDiagram*) to the task that consumes the only initial work product in the model of the 4SRS transition method with the SPEM, which are the transition task *ComponentCreation* and the initial work product *UseCaseDiagram*. The intermediate task *UseCaseDiagramDrawing* was needed since the Moderne tool does not allow creating an input to a task in the context of the task itself, rather in the context of another task as output of that own task.

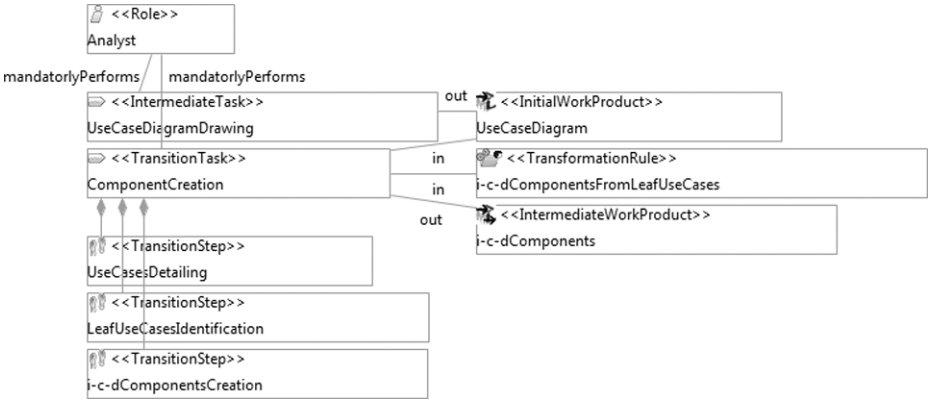

Fig. 2. The 4SRS transition method modeled with the SPEM for automation purposes

We analyzed the model of the 4SRS transition method with the SPEM to identify the steps that could be automated with the Moderne tool. Table 1 shows that analysis. Some of the steps from the 4SRS transition method where concluded to be fully automated with the Moderne tool whereas others where concluded to be semiautomated or not automated at all. The automation capability is the ability of a method's step to be automated with a tool. The ones concluded to be fully automated with the tool were classified as "Automatic" in terms of their automation capability, the ones concluded to be semiautomated with the tool were classified as "Semiautomatic" and the ones concluded to be not automated with the tool where classified as "Not automatic". The automatic steps were automated using ATL model-to-model transformation rules. A semiautomatic step depends on some modeler's action in the models by means of the tool before the ATL model-to-model transformation rules concerning that particular step can be applied. The not automatic

steps comprise actions that are fully performed by the modeler even that they consist of input for the models or for the information attached to the models, in the tool.

6 Case Study

The use case diagram in Figure 3 was used to exemplify the model-to-model transformations the Moderne tool is able to perform in the context of the 4SRS. The diagram is based on the Fraunhofer IESE's GoPhone [23] case study, which presents a series of use cases for a part of a mobile phone product line particularly concerning the interaction between the user and the mobile phone software.

Table 1. Analysis of the automation capability of the steps from the 4SRS

4SRS Step/Microstep	Automation Capability
Step 1: Component Creation	Automatic
Microstep 2.i: Use Case Classification	Not automatic (the modeler shall decide each use case's classification)
Microstep 2.ii: Local Elimination	Semiautomatic (the modeler shall tag in the component diagram the components to eliminate or maintain)
Microstep 2.iii: Component Naming	Not automatic
Microstep 2.iv: Component Description	Not automatic
Microstep 2.v: Component Representation	Not automatic (the modeler explicitly relates components in the diagram through Dependency relationships indicating which component represents others)
Microstep 2.vi: Global Elimination	Automatic (based on the Dependency relationships mentioned above in this table)
Microstep 2.vii: Component Renaming	Not automatic
Step 3: Component Packaging and Nesting	Not automatic
Step 4: Component Association	Semiautomatic (partially based on the rules for associating components and partially based on the modeler's decision)
Intermediate step 4+1: Filtering and Collapsing	Semiautomatic (the collapsing is automatic; the filtering is semiautomatic depending partially on the modeler's decision to perform refinement and with the automatic exclusion of the components not associated with any component from the region to refine determined by the modeler)
Intermediate microstep 4+2.i: Deriving Use Cases from Components	Not automatic
Intermediate microstep 4+2.ii: Detailing Use Cases	Not automatic

The transformation of the use case diagram in Figure 3 into the corresponding component diagram was defined in an ATL rule that mostly determines what leaf use cases are. Leaf use cases are those from which interface components, control components and data components are generated in the step 1 of the 4SRS (Component Creation). The ATL rule defines that leaf use cases are those that are included by at least one use case and that do not include any other use case, and those that are not included by any use case and do not include any use case. The rule also defines some associations between components, and between components and actors (from the use case diagram). This anticipates part of the step 4 (Component Association) of the 4SRS to the step 1 (Component Creation).

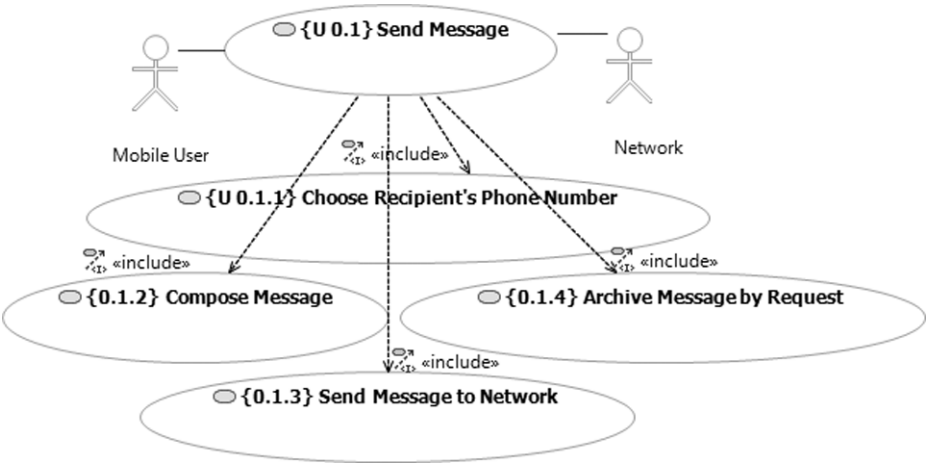

Fig. 3. A use case diagram from the GoPhone

Figure 4 depicts part of the ATL rule that determines the leaf use cases of a use case diagram. The function `srcIncludes()` gets all associations whose source is the element that called the rule. The component diagram generated from the use case diagram in Figure 3 through the ATL rule and the Moderne tool is in Figure 5.

```
helper context UML!UseCase def : isLeaf() : Boolean =
        self.srcIncludes()->size() = 0;
```

Fig. 4. Part of the ATL rule that determines the leaf use cases of a use case diagram

We have defined some well-formedeness rules or constraints in OCL (Object Constraint Language [24]) to do what the following figures illustrate.

A transition task can transform an initial work product into an intermediate work product or an intermediate work product into an intermediate work product or even an intermediate work product into a final work product. The OCL code for these constraints is in Figure 6. Figure 7 shows a validation error signaled with a cross in a transition task that transforms an intermediate work product into an initial work product.

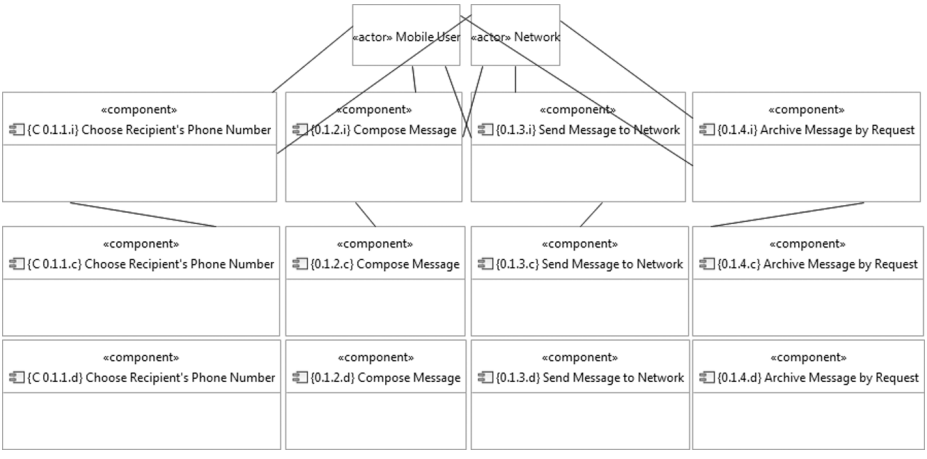

Fig. 5. The component diagram automatically generated from the use case diagram in Figure 3

```
context TransitionTask
inv:
(self.in->forAll(wp | wp.oclIsTypeOf(InitialWorkProduct))
and
self.out->forAll(wp | wp.oclIsTypeOf(IntermediateWorkProduct)))
OR
(self.in->forAll(wp | wp.oclIsTypeOf(IntermediateWorkProduct))
and
self.out->forAll(wp | wp.oclIsTypeOf(IntermediateWorkProduct)))
OR
(self.in->forAll(wp | wp.oclIsTypeOf(IntermediateWorkProduct))
and
self.out->forAll(wp | wp.oclIsTypeOf(FinalWorkProduct)))
```

Fig. 6. The OCL code for constraints on the relation between transition tasks and work products

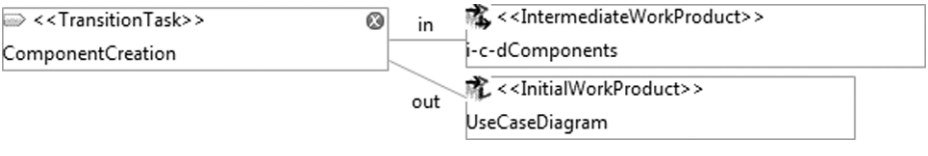

Fig. 7. An example of a validation error in the constraints on the relation between transition tasks and work products

An intermediate task transforms a final work product into an initial work product. The OCL code for this constraint is in Figure 8. Figure 9 illustrates a validation error signaled with a cross in an association between an intermediate task and an initial work product.

```
context IntermediateTask
inv:  self.in->forAll(wp | wp.oclIsTypeOf(FinalWorkProduct));
inv:  self.out->forAll(wp | wp.oclIsTypeOf(InitialWorkProduct));
```

Fig. 8. The OCL code for constraint on the relation between intermediate tasks and work products

Fig. 9. An example of a validation error in the constraint on the relation between intermediate tasks and work products

A transition step can only be contained by a transition task, whereas an intermediate step can only be contained by an intermediate task. The OCL code for these constraints is in Figure 10. Figure 11 depicts that a composition could not be drawn between a transition task and an intermediate step. In the case of the constraints above, a validation error was signaled with a cross in model elements because changing properties (the names) of the associations would eliminate that error. In this case changing properties of the association would not eliminate the error since the association should not exist in the first place to obey the constraint.

```
context IntermediateTask
inv: steps->forAll(step | step.oclIsTypeOf(IntermediateStep))

context TransitionTask
inv: steps->forAll(step | step.oclIsTypeOf(TransitionStep)))
```

Fig. 10. The OCL code for constraints on the relation between tasks and steps

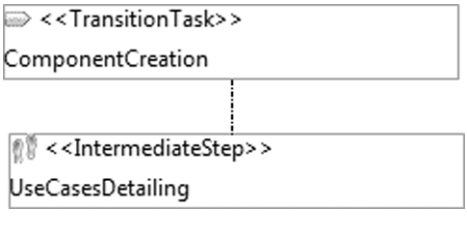

Fig. 11. An example of the impossibility of a composition between a transition task and an intermediate step

7 Conclusions

Transition methods describe how to transform artifacts originally produced within a certain discipline of a large software development process into artifacts from another discipline of such a process. Some transition methods are targeted at moving from the

analysis to the design of software. The 4SRS is a method that allows transforming UML use case diagrams (considered here as the analysis model) into the logical architecture of the system as a UML component model (considered here as the design model, the first technical artifact to initiate the design of the system).

This paper describes the automation of a transition method, using the 4SRS method modeled with the SPEM as the case study. We adopted the Moderne tool to automate the generation of logical architectures through model transformations defined with the ATL language. We have defined some well-formedeness rules or constraints in OCL to validate the modeling of the 4SRS transition method with the SPEM.

The SPEM model of the 4SRS transition method elaborated beforehand was adapted for automation purposes. For instance a transformation rule was added to the model.

The OCL constrains on the modeling of the 4SRS transition method with the SPEM established the well-formedeness of the relations between transition tasks and work products, between intermediate tasks and work products, and between tasks and steps (all of these are elements for modeling methods). The violation of the constraints on those relations was tagged in the model through validation errors.

In terms of future work, we plan to assess the efficiency of our approach by adopting the Moderne tool to apply the 4SRS transition method in a real industrial project.

References

[1] OMG. Software & Systems Process Engineering Meta-Model Specification - version 2.0 (2008), http://www.omg.org

[2] Conradi, R., Jaccheri, M.L.: Process Modelling Languages. In: Derniame, J.-C., Kaba, B.A., Wastell, D., et al. (eds.) Promoter-2 1998. LNCS, vol. 1500, pp. 27–52. Springer, Heidelberg (1999)

[3] Machado, R.J., et al.: Transformation of UML Models for Service-Oriented Software Architectures. Presented at the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2005, Greenbelt, Maryland, USA (2005)

[4] Machado, R.J., Fernandes, J.M., Monteiro, P., Daskalakis, C.: Refinement of Software Architectures by Recursive Model Transformations. In: Münch, J., Vierimaa, M., et al. (eds.) PROFES 2006. LNCS, vol. 4034, pp. 422–428. Springer, Heidelberg (2006)

[5] Fernandes, J.M., et al.: A Demonstration Case on the Transformation of Software Architectures for Service Specification. Presented at the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES 2006, Braga, Portugal (2006)

[6] Kruchten, P.: Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software 12, 42–50 (1995)

[7] Clements, P., et al.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Upper Saddle River (2002)

[8] Gomes, R.A., et al.: Moderne: Model Driven Process Centered Software Engineering Environment. Presented at the 2nd Congresso Brasileiro de Software: Teoria e Prática, CBSoft 2011, São Paulo, Brasil (2011)

[9] The Eclipse Foundation. ATL Project (2010), http://www.eclipse.org/m2m/atl
[10] Brown, A.W., et al.: Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In: Beydeda, S., et al. (eds.) Model-Driven Software Development, pp. 1–16. Springer, Heidelberg (2005)
[11] Osterweil, L.J.: Software Processes Are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9. Presented at the 1997 International Conference on Software Engineering, ICSE 1997, Boston, Massachusetts, USA (1997)
[12] Maciel, R.S.P., et al.: An Integrated Approach for Model Driven Process Modeling and Enactment. Presented at the XXIII Brazilian Symposium on Software Engineering, SBES 2009, Fortaleza, Brazil (2009)
[13] Fuggetta, A.: Software Process: A Roadmap. Presented at the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick, Ireland (2000)
[14] Estublier, J.: Software are Processes Too. In: Li, M., Boehm, B., Osterweil, L.J. (eds.) SPW 2005. LNCS, vol. 3840, pp. 25–34. Springer, Heidelberg (2006)
[15] Conradi, R., Liu, C.: Process Modelling Languages: One or Many? In: Schäfer, W. (ed.) EWSPT 1995. LNCS, vol. 913, pp. 98–118. Springer, Heidelberg (1995)
[16] Henderson-Sellers, B.: Process Metamodelling and Process Construction: Examples Using the OPEN Process Framework (OPF). Annals of Software Engineering 14, 341–362 (2002)
[17] Bendraou, R., et al.: A Comparison of Six UML-Based Languages for Software Process Modeling. IEEE Transactions on Software Engineering 36, 662–675 (2010)
[18] Feiler, P.H., Humphrey, W.S.: Software Process Development and Enactment: Concepts and Definitions. Presented at the 2nd International Conference on the Software Process, ICSP 1993, Berlin, Germany (1993)
[19] Gruhn, V.: Process-Centered Software Engineering Environments, A Brief History and Future Challenges. Annals of Software Engineering 14, 363–382 (2002)
[20] Henderson-Sellers, B., et al.: Process Construction and Customization. Journal of Universal Computer Science 17, 326–358 (2004)
[21] Kaindl, H.: Difficulties in the Transition from OO Analysis to Design. IEEE Software 16, 94–102 (1999)
[22] OMG. Unified Modeling Language: Superstructure - version 2.2 (2009), http://www.omg.org
[23] Muthig, D., et al.: GoPhone - A Software Product Line in the Mobile Phone Domain. Fraunhofer IESE, IESE-Report No. 025.04/E (March 5, 2004)
[24] OMG. Object Constraint Language: Specification - version 2.2 (2010), http://www.omg.org