Reuse of Software Process Fragments Is Reuse of Software Too

Luuk Groenewegen
Leiden University, Dept. of Computer Sc.
P.O. Box 9512, 2300 RA Leiden, The Netherlands
luuk@wi.leidenuniv.nl

Gregor Engels
Leiden University, Dept. of Computer Sc.
P.O. Box 9512, 2300 RA Leiden, The Netherlands
engels@wi.leidenuniv.nl

Abstract

By means of a special software process model component we have been successful in specifying software process model evolution. This component also handles much reuse of software process model fragments. To handle the reuse of the software as it is being produced by a software process, another extra component can be added to the model for that software process. It is argued why this extra component is comparable to the special component for evolution.

1. Introduction

The experience we have with reuse is a rather biased one. It is in the context of evolution of software process models written in Socca, that we reuse many earlier evolution phases as well as many other software process models written in Socca. Nevertheless we think this bias a happy circumstance, or even condition, for enabling us to arrive at our view of facilitating the management and the coordination of software reuse by incorporating a special process fragment for reuse into our software process model.

This special process fragment for reuse is comparable to special process fragments for evolution. (In Socca the special fragment for evolution is Wodan, see [8].) Reuse of software then is viewed as change of the software's so-called intrinsic process. The basic reason for the very bias providing us with an essential view on reuse, is also present in the well known observation "software process models are software too", see [5].

In order to clarify our viewpoint this paper has the following structure. First we discuss our Socca experience with reuse in process evolution. Then we underline what exactly in Socca is of great help in modelling evolution and particularly this form of reuse. After that we argue why managing reuse of software in general can be considered as similar to managing evolution. This argument will add another meaning to "software process models are software too".

2. Evolution in Socca

Socca is a specification formalism together with a method for the analysis and design, and ultimately also the enaction of (software) process models, see e.g. [2]. Socca is related to OMT [6], which it extends considerably by addressing communication more precisely. The name Socca is an abbreviation for Specification Of Coordinated Cooperative Activities.

Like OMT, Socca is an eclectic, see [4], process modelling language. As such it consists of several sublanguages carefully integrated with each other. These sublanguages are Class Diagrams (CDs), State Transition Diagrams (STDs), Paradigm and Object Flow Diagrams (OFDs). The name Paradigm is itself an abbreviation, it means PARallelism its Analysis, Design and Implementation by a General Method.

Globally speaking, building a Socca model consists of several steps. In the first step the static structure of the problem and its solution is specified. Like in OMT this is done by means of a CD. The second step of Socca builds the external behaviour, specified as STD, of each class separately, expressing for that class all possible execution sequences of export operations, as these operations are given in the CD. Socca's third step specifies for each export operation of each class the internal or hidden behaviour as a separate STD. This STD expresses all possible sequences of calls to export operations, as being part of the hidden behaviour.

The execution model for this huge amount of STDs, is that all STDs are simultaneously active, each being executed as it were on its private processor. In step four of Socca the communication between these simultaneously active STDs is specified by means of Paradigm, see e.g. [7]. This gives the precise coordination of the various STDs, the external ones serving as the so-called manager processes of the internal ones. Finally the fifth step of Socca relates the behavioural effects of the STDs to the attributes from the CD by means of OFDs.
In [8] it is shown how evolution of a Socca model can be incorporated into the model, even evolution on the fly. To that aim a special STD is added to the model, called Wodan — abbreviation for What Ought to be Done As Necessary. Wodan actually does two different things. First, at every relevant time instant Wodan actually prescribes the current evolution phase of the whole model, by assigning the current data, behaviour and communication to each object. Second, while continuing the prescription of the current evolution phase, Wodan moreover defines the evolution phase it is going to prescribe next. Then Wodan changes the old prescription into the new prescription.

As it turns out during the preparation and definition of a next evolution phase, there is much reuse of model fragments in this preparation. It is interesting to see how these fragments are cut along border lines from two different modularizations. One modularization is object-oriented, so its units are the objects, where aspects like data, behaviour, communication come together. The other modularization is perspective-oriented or aspect-oriented. Its units are the data model, the behaviour, the communication and the functionality. Within Socca these are the ingredients of the eclecticism: the CD, the STDs, Paradigm and the OFD. So a fragment to be reused may consist of the communication between a project manager and her/his team members. In that case the objects in the fragment are from two classes only, project manager and team member. Moreover, only the communication with these objects remains the same, whereas the other aspects are changed. Both modularizations are well supported by Socca, which is a great advantage of it.

What we learned from the reuse in this particular situation, is first of all the usefulness of the two basic forms of modularization and the numerous ways they can be combined into different model fragments. This then results in a rich(er) form of modularization. In addition, on top of this rich form of modularization different forms of interfaces can be constructed, thereby adding new features to or new views on a particular model fragment. This turns out to be very useful for modelling new forms of communication on top of an existing model fragment, or for extending the external behaviour of one or more classes. Another very useful consequence of this modularization consists of different possibilities of regrouping on the basis of (multiple) inheritance and aggregation. This gives ample as well as structured modelling freedom, which in the case of aggregating or inheriting behaviour, see [1], is related to how design patterns can be constructed as mentioned in [3]. Finally, the special component Wodan further structures this reuse. Wodan does this by first reusing process fragments for the definition of the new evolution phase and for the various intermediate phases. After that, Wodan actually reuses the thus adapted process fragments during the enactment of the subsequent intermediate phases and eventual next phase.

3. A Reuse Component in the Software Process Model

In reusing arbitrary software, not only its data, but also its algorithms, its communication protocols and its functionality are relevant candidates for being reused. This is similar to reuse of software process fragments, and after all, this is not surprising, as software processes are software too, see [5]. Here however we want to stress the similarity even further, in making it more symmetric: software is a process too. The basic argument for this is, software specifies the process consisting of the numerous steps to be taken by a suitable processor. In addition, together with system software such as a compiler or interpreter and an operating system, that process is also enactable or even enacting.

In order to discriminate between the process specified by the software itself on the one hand, and what is usually referred to as the software process on the other hand — being the process (model) of the engineering of the software — we call the former one the intrinsic process (of the software).

Now reuse of software comes down to adapting or changing its intrinsic process. The change itself may suffice for the specification of the new intrinsic process. Or the thus changed — or possibly unchanged — process may be part of a larger intrinsic process, i.e. the software is reused in a larger software system.

As reuse of software actually consists of changing the intrinsic process, and as this changing is part of the engineering of the new software, this changing is also part of the software process, and therefore of the software process model. In particular for a software process where reuse is a kernel activity, as is the case with process support for software product lines, this changing of the intrinsic process of various pieces of software should be recognized as essential for that type of software process support. To that aim such a software process model should then contain a special component for taking care of that kind of change. In the case of Socca such a component can be compared to Wodan, especially the part of Wodan responsible for creating the new definitions. In process support for a software product line a component for changing the intrinsic processes is even more needed, as in general each piece of software is expected to be reused.

Such a component should provide facilities to model the software before and after reuse in a more process-oriented fashion than usual. In this light this component also allows for reversed engineering, to generate some process model for the already existing software if such a model for the
intrinsic process did not yet exist. Note that the process modelling language (PML) for expressing the change of the intrinsic process, is not necessarily the same as the PML for expressing the software process. In principle any suitable PML will do. A criterion could be that the PML for the intrinsic process should fit as close as possible to the specification language that has been and is being used for the design of the software itself.

Summarizing, to raise the reuse-sensitivity of a process model we propose to incorporate into that model a special reuse component, taking care of changing the intrinsic processes of all relevant pieces of software. This is not unlike Wodan, the component added to Socca taking care of the evolution of the software process and its model. In this capacity, Wodan manages quite a lot of reuse of process elements. The new reuse component should exploit the two forms of modularization mentioned, and should integrate them with inheritance and aggregation. As an interesting challenge for the near future we will develop a detailed Socca process model fragment for this very component. The component itself is to use the more customary OMT language for the description of the intrinsic processes.

References


[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (Mass.), 1995.


