4 PSP$_{DC}$: An Adaptation of the PSP to Incorporate Verified Design by Contract

Silvana Moreno, Universidad de la República
Álvaro Tasistro, Universidad ORT Uruguay
Diego Vallespir, Universidad de la República

4.1 Introduction

Personal Software Process (PSP) incorporates process discipline and quantitative management into the software engineer's individual development work. It promotes the exercise of careful procedures during all stages of development with the aim of achieving high quality of the final product and thereby also increasing the individual's actual productivity [Humphrey 2005, 2006].

Formal methods, in turn, use the same methodological strategy, namely emphasizing care in the procedures of development (as opposed to relying on testing and debugging.) In fact they establish a radical requirement in this respect, which consists of mathematically proving that the programs satisfy their functional requirements.

In this paper we investigate how to integrate the use of formal methods into PSP, by formulating a new version of PSP.

Design by Contract (DbC) is a technique devised (and patented) by Bertrand Meyer for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language [Meyer 1992]. This language is seamlessly integrated with the programming language so that the specified conditions can actually be evaluated at run-time, which allows, among other things, the ability to handle violations by means of a system of exceptions. When appropriate techniques and tools are incorporated that allow proving that the components satisfy the established requirements, it indicates the use of a formal method usually called Verified Design by Contract (VDbC.) This is the method we propose to consider for integration with PSP.

Our alternative version of PSP is called PSP$_{DC}$. With respect to ordinary PSP, it incorporates new phases and modifies others as well as adding to the infrastructure new scripts and checklists. Specifically, we add the phases of Formal Specification, Formal Specification Review, and Formal Specification Compile. In addition we modify the Compile phase into a new phase called Compile and Proof. The general idea is to supplement the design with formal specifications of the components (which are produced in the first three new phases listed above) and the code with a formal proof that it matches the formal specifications (which are produced in the Compile and Proof phase.) This proof is to be carried out with the help of a tool akin to a verifying compiler that statically checks the logical correctness of the code in addition to the syntax.

We know of only two works in the literature that propose to combine PSP and formal methods. Babar and Potter combine Abrial's B Method with PSP into B-PSP [Babar 2005]. They add the phases of Specification, Auto Prover, Animation, and Proof. A new set of defect types is added and logs are modified to incorporate data extracted from the B machine's structure. The goal of this work is to provide the individual B developers with a paradigm of measurement and evaluation that promotes reflection on the practice of the B method, inculcating the habit of recognizing causes of defects injected so as to be able to avoid these in the future.

Suzumori, Kaiya, and Kaijiri propose the combination of VDM and PSP [Suzumori 2003]. The Design phase is modified incorporating the formal specification in the VDM-SL language. Besides, the phases of VDM-SL Review, Syntax Check, Type Check, and Validation are added. One result arising from applications is that the use of VDM contributes to eliminate defect injection during design.

4.2 Formal Methods

Formal methods hold fast to the tenet that programs should be proven to satisfy their specifications. Proof is, of course, the mathematical activity of arriving at knowledge deductively (i.e., starting off from postulated, supposed, or self-evident principles and performing successive inferences, each of which extracts a conclusion out of previously arrived at premises).

In the application of this practice to programming we have among the first principles the so-called semantics of programs, which allow us to understand program code and thereby know what each part of the program actually computes. This makes it in principle possible to deductively ascertain that the computations carried out by the program satisfy certain properties. Among these properties are input-output relations or patterns of behavior that constitute a precise formulation of the so-called functional specification of the program or system at hand.

Formal logic, at least in its contemporary mathematical variety, has striven to formulate artificial languages into which it is possible to frame the mathematical activity. There should therefore, according to this aim, be a language for expressing every conceivable mathematical proposition and also a language for expressing proofs, so that a proposition is provable in this language if and only if it is actually true. This latter desirable property of the language is called its correctness. This kind of research began in 1879 with Frege [Frege 1967] with the purpose of making it undisputable whether a proposition was or was not correctly proven. Indeed, the whole point of devising artificial languages was to make it possible to automatically check whether a proposition or a proof was correctly written in the language. That is to say, the proofs accepted were to be so on purely syntactic (i.e. formal) grounds and, given the good property of correctness of the language, that was enough to ensure the truth of the asserted propositions.

Frege’s own language turned out to be not correct and, for that reason mainly, shortly after its failure the whole enterprise of formal logic took a different direction, namely that of studying the artificial languages as mathematical objects to prove their correctness by elementary means. This new course was actually also destined to failure.

The overall outcome is nevertheless very convenient from an engineering viewpoint. We can go back to Frege’s program, and nowadays we have technology that makes it feasible to develop formal proofs semi-automatically. The proof systems (or languages) are still reliable although not complete (i.e., not every true proposition will be provable). But again, there is no harm in the practice, and the systems are perfectly expressive from an engineering perspective. These advances allow us to define formal methods in software engineering as a discipline based on the use of formal languages and related tools for expressing specifications, and carrying out proofs of correctness of programs.

Notice that the semi-automatic process of program correctness proof is of course a static kind of checking. We can think of it as an extension of compilation, which not only checks syntax but also properties of functional behavior. Therefore it is convenient to employ the general idea of a semi-automatic verifying compiler to characterize the functionality of the tools employed within a formal methods framework.

DbC is a methodology for designing software proposed and registered by Bertrand Meyer [Meyer 1992]. It is based on the idea that specifications of software components arise, like business contracts, from agreements between a user and a supplier, which establish the terms of use and performance of the components. That is to say, specifications oblige (and enable) both the user and the supplier to certain conditions of use and a corresponding behavior of the component in question.

In particular, DbC has been proposed in the framework of Object Oriented Design (and specifically in the language Eiffel) and therefore the software components to be considered are classes. The corresponding specifications are pre- and post-conditions to methods, establishing respectively their terms of use and corresponding outcomes, as well as invariants of the class (i.e., conditions to be verified by every visible state of an instance of the class). In the original DbC proposal, all the specifications are written in the language Eiffel itself and are computable (i.e., they are checkable at run-time).

Therefore, DbC within Eiffel provides at least the following:

- A notation for expressing the design that seamlessly integrates with a programming language, making it easy to learn and use.
- Formal specifications, expressed as assertions in Floyd-Hoare style [Hoare 1969].
- Specifications checkable at run-time and whose violations may be handled by a system of exceptions.
- Automatic software documentation.

However, DbC is not by itself an example of a formal method as defined above. When we additionally enforce proving that the software components fit their specifications, we are using VDbC. This can be carried out within several environments, which all share the characteristics mentioned above:

- The Java Modeling Language (JML) implements DbC in Java. VDbC can then be carried out using tools like Extended Static Checking (ESC/Java) [Cok 2005] or TACO [Galeotti 2010].
- Perfect Developer [Crocker 2003] is a specification and modeling language and tool which, together with the Escher C Verifier, allows performing VDbC for C and C++ programs.
- Spec# [Barnett 2004] allows VDbC within the C# framework.
- Modern Eiffel [Eiffel 2012] allows it within the Eiffel framework.

4.3 Adaptation

Figure 17 displays the PSP. We assume the engineer will be using an environment like any of those listed at the end of the previous section. This means that a computerized tool (akin to a verifying compiler) is used for

- Checking syntax of formal assertions. These are written in the language employed in the environment (e.g. as Java Boolean expressions, if we were employing JML) which we shall call the carrier language.
- Computing proof obligations (i.e. given code with assertions, to establish the list of conditions that need to be proven to ascertain the correctness of the program).
- Developing proofs in a semi-automatic way.

We now briefly review Figure 17 summarizing the most relevant novelties of PSP:

- After the Design Review phase, there comes a new phase called Formal Specification. It is in this phase that the design is formalized, in the sense that class invariants and pre- and post-conditions to methods are made explicit and formal (in the carrier language).
- The Formal Specification Review has the purpose of detecting errors injected in the formal specification produced in the previous phase. A review script is used in this activity.
- The Formal Specification Compile phase consists of automatically checking the formal syntax of the specification.
- The phase of Code Compile and Proof comes after production and revision of the code. This is the proper formal verification phase, carried out with help of the verifying compiler.

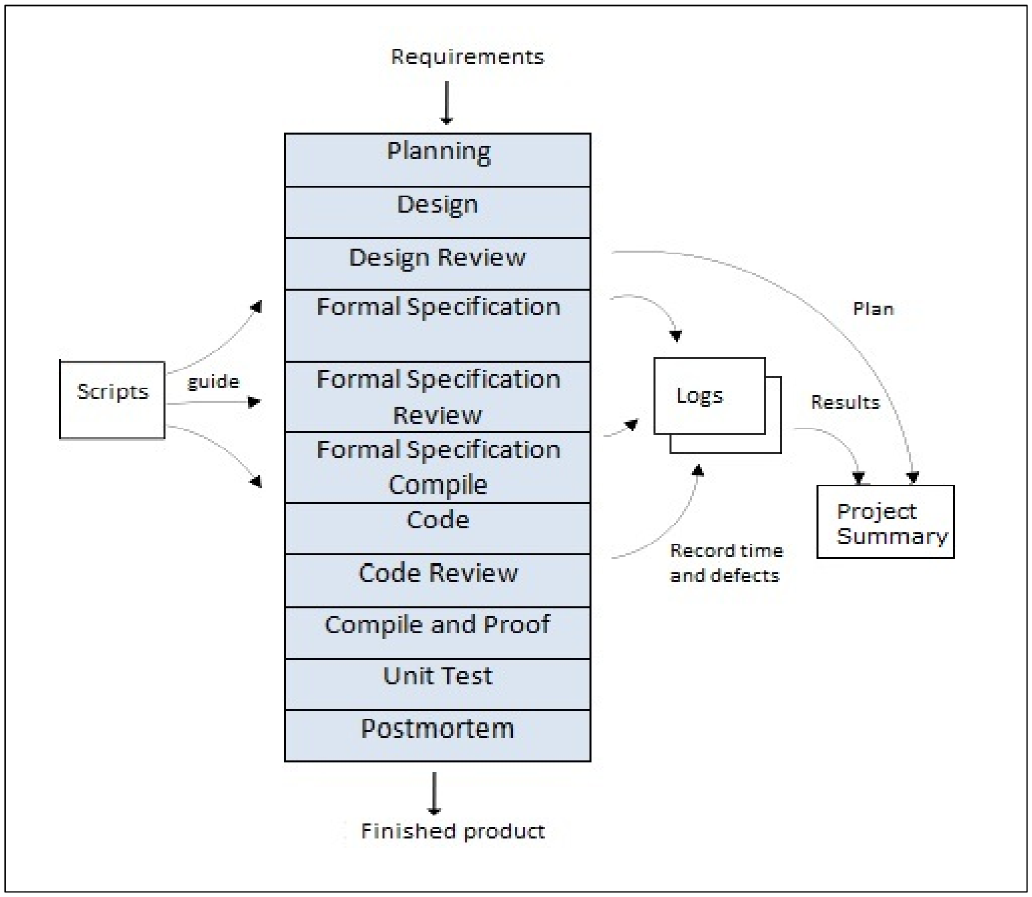

Figure 17: Personal Software Process

In the following subsections we present in detail all the phases of the PSP_{DC}, indicating in each case the activities to be performed and the modifications introduced with respect to the original PSP.

4.4 Planning

The activities of this phase are Program Requirements, Size Estimate, Resource Estimate, Task and Schedule Planning, and Defect Estimate.

Program Requirements is for ensuring a precise understanding of every requirement. This activity is the same as in the ordinary PSP.

Size Estimate involves carrying out a conceptual design (i.e., producing a module [class] structure). Each class is refined into a list of methods and the size of each method is estimated. For this we do as in ordinary PSP (i.e., we use proxies which utilize a categorization of the method according to its size and the functional kind of the corresponding class). Categories of size of methods are very small, small, medium, large, or very large; functional kinds of classes are Calc, Logic, IO, Set-Up, and Text. Thus, using the structure of classes, the number of methods in each class and the category of the class, we arrive at an estimation of the LOCs of the program.

Now, using (Verified) Design by Contract requires us to formally write down the pre- and post-conditions of each method and the invariant of each class. And we do not yet possess methods akin to the above-mentioned proxies for estimating the size of these formal specifications. Therefore, we are not in a position to produce such estimation. However, it can be argued that VDbC modifies the process by which we arrive at design and code, but not the size of the final program in LOCs, at least if we consider the latter to be the executable code that is necessary to achieve the required functionality. In this sense, formal specifications could be treated as formal comments to the code (i.e., not to be counted into the size of the program).

Resource Estimate estimates the amount of time needed to develop the program. For this, the method PROBE is used, which employs historical records and linear regression for producing the new estimation, as well as for measuring and improving the precision of the estimations. In our adaptation, the activity remains conceptually the same, but will employ records associated to the new phases incorporated into PSP_{DC}. Therefore, once sufficient time data has been gathered, we shall be able to estimate the time to be employed in formal specification, as well as in program proof.

Task and Schedule Planning is for long-term projects. These are subdivided into tasks and the time is estimated for each task. This is unchanged in PSP_{DC}.

Defect Estimate Base is for estimating the number of defects injected and removed at each phase. Historical records, as well as the estimated size of the program, are utilized for performing this estimation. In PSP_{DC} we must keep new records in order to estimate the defects removed and injected at each new phase.

4.5 Design

During Design, we define the data structures, classes and methods, interfaces, components, and the interactions among all of them. Formal specification of methods and of invariants of classes could be carried out within the Design phase. This, however, presents the disadvantage of not allowing us to keep records of the time employed specifically in Design, as well as in Formal Specification. Instead, we would just record a likely significant increase in Design time. Therefore we prefer to separate the phase of Formal Specification.

4.6 Design Review

This is the same as in ordinary PSP.

4.7 Formal Specification

This phase must be performed after Design Review. The reason for this is that reviews are very effective in detecting defects injected during design, and we want these to be discovered as early as possible.

In this phase we start to use the computerized environment supporting VDbC. We propose to carry out two activities within this phase, namely Construction and Specification. The activity of Construction consists of preparing the computerized environment and defining within it each class with its method headers. This could have been done during Design, in which case we would, of course, omit it here. The choice is a valid personal one. The second activity is Specification, in which we write down in the carrier language the pre- and post-conditions of each method as well as the class invariant. Notice that, within the present approach, the use of formal methods begins once the design has been completed. It consists of the formal specification of the produced design and the formal proof that the final code is correct with respect to this specification.

4.8 Formal Specification Review

Using a formal language for specifying conditions is not a trivial task and both syntactic and semantic defects can be injected. To avoid the propagation of these errors to further stages and thereby increasing the cost of correction, we propose to carry out a phase of Formal Specification Review.

The script that corresponds to this phase contains the activities of Review, Correction, and Checking. The Review activity consists in inspecting the sentences of the specification using a checklist. In the activity of Correction all defects detected during Review are removed. Finally, Checking consists of looking over the corrections to verify their adequacy.

4.9 Formal Specification Compile

Any computerized tool supporting VDbC will be able to compile the formal specification. Since this allows an early detection of errors, we consider it valuable to explicitly introduce this phase into PSP$_{DC}$. In particular, it is worthwhile to detect all possible errors in the formal specifications before any coding is carried out. A further reason to isolate the compilation of the formal specification is that it allows recording the time spent in this specific activity.

4.10 Code

Just as in ordinary PSP, this phase consists of translating the design into a specific programming language.

4.11 Code Review

This phase does not differ from the corresponding one in ordinary PSP.

4.12 Compile and Proof

The phase Code Compile of PSP is modified in PSP$_{DC}$ in order to provide, besides the compiled code, evidence of its correctness with respect to the formal specification (i.e., its formal proof). As already said, we here use the computerized tool (verifying compiler) which compiles the code, derives proof obligations, and helps to carry out the proofs themselves.

4.13 Unit Test

This phase is the same as in ordinary PSP. We consider it relevant for detecting mismatches with respect to the original, informal requirements to the program. These defects can arise in several points during the development, particularly as conceptual or semantic errors of the formal specifications. The test cases to be executed must therefore be designed right after the requirements are established, during the phase of Planning.

4.14 Post-Mortem

This is the same as in ordinary PSP. Several modifications have to be made to the infrastructure supporting the new process. For instance, all new phases must be included into the support tool to keep control of the time spent at each phase as well as to record defects injected, detected, and removed at each phase. Our intention in this paper is to present the changes in the process in order to incorporate VDbC. The adaptation of the supporting tools, scripts, and training courses is a matter for a separate work.

4.15 Conclusions and Future Work

We have presented PSP$_{DC}$, a combination of PSP with Verified Design by Contract (VDbC), with the aim of developing better quality products.

In summary we propose to supplement the design with formal specifications of the pre- and post-conditions of methods as well as class invariants. This gives rise to three new phases that come after the Design phase, namely Formal Specification, Formal Specification Review, and Formal Specification Compile. We also propose to verify the logical correctness of the code by using an appropriate tool, which we call a *verifying compiler*. This motivates the modification of the Compile phase, originating the new Compile and Proof phase, which provides evidence of the correctness of the code with respect to the formal specification.

The process can be carried out within any of several available environments for VDbC.

By definition, in Design by Contract (and thereby, also of VDbC) the specification language is seamlessly integrated with the programming language, either because they coincide or because the specification language is a smooth extension of the programming language. As a consequence, the conditions making up the various specifications are simple to learn and understand Boolean expressions. We believe that this makes the approach easier to learn and use than the ones in other proposals, like Babar and Suzumori [Babar 2005, Suzumori 2003]. Nonetheless, the main difficulty associated with the whole method resides in developing a competence in carrying out the formal proofs of the written code. This is, of course, common to any approach based on formal methods. Experience shows, however, that the available tools are generally of great help in this matter. There are reports of cases in which the tools have generated the proof obligations and discharged up to 90% of the proofs in an automatic manner [Abrial 2006].

We conclude that it is possible in principle to define a new process that integrates the advantages of both PSP and Formal Methods, particularly VDbC. Our future work consists of completing the adaptation of PSP by writing down in detail the scripts to be used in all phases, adapt the logs, specify and carry out the modifications to the support tool, and modify or define interesting metrics.

We must evaluate the PSP_{DC} in actual practice by carrying out measurements in case studies. The fundamental aspect to be measured in our evaluation is the quality of the product, expressed in the amount of defects injected and removed at the various stages of development. We are also interested in measures of the total cost of the development.

4.16 Author Biographies

Silvana Moreno
School of Engineering, Universidad de la República

Silvana Moreno is a teaching and research assistant at the Engineering School at the Universidad de la República (UdelaR). She is also a member of the Software Engineering Research Group (GrIS) at the Instituto de Computación (InCo.). Moreno holds an Engineer title in computer science from UdelaR and is currently enrolled in their Master of Science program in computer science.

Álvaro Tasistro
School of Engineering, Universidad ORT Uruguay

Álvaro Tasistro is a professor at the engineering school of the Universidad ORT Uruguay. He is a chair of Theoretical Computer Science and is also the director of studies for the Master of Engineering Sciences program at the university. He holds a PhD from Chalmers University in Gothenburg, Sweden. He has several articles published in international conferences, journals, and books. His main research topics are type theory and formal methods.

4.17 References/Bibliography

[Abrial 2006]
Abrial, Jean-Raymond. "Formal Methods in Industry: Achievements, Problems, Future," 761–768. *28th International Conference on Software Engineering (ICSE'06*). Shanghai, Republic of PRC, May 2006. ACM Press, 2006.

[Babar 2005]
Babar, Abdul & Potter, John. "Adapting the Personal Software Process (PSP) to Formal Methods," 192-201. *Australasian Software Engineering Conference (ASWEC'05*), Brisbane, Australia, March-April, 2005. IEEE, 2005.

[Barnett 2004]
Barnett, Mike; Rustan, K.; Leino, M.; & Schulte, Wolfram. "The Spec# Programming System: An Overview" *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), Lecture Notes in Computer Science 3362*, pages 49–69. Springer, 2004.

[Cok 2005]
Cok, David & Kiniry, Joseph. "ESC/Java2: Uniting ESC/Java and JML." *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004). Lecture Notes in Computer Science 3362*, pp. 108–228. Springer-Verlag, 2005.

[Crocker 2003]
Crocker David, *Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement*, 2003.

[Eiffel 2012]
Eiffel. Definition of “Modern Eiffel.” http://tecomp.sourceforge.net/index.php?file=doc/papers/lang/modern_eiffel.txt. Retrieved August 16, 2012.

[Frege 1967]
Frege, G. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle a. S.: Louis Nebert, 1879. Translated as *Concept Script, a formal language of pure thought modelled upon that of arithmetic*, by S. Bauer-Mengelberg. Edited by J. vanHeijenoort, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Cambridge, MA: Harvard University Press, 1967.

[Galeotti 2010]
Galeotti Juan; Rosner, Nicolás; Pombo, López; &. Frias, Marcelo F. “Analysis of invariants for efficient bounded verification,” 25-36. *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis* (ISSTA 2010), Trento, Italy, July 2010. Schloss-Dagstuhl, 2010.

[Hoare 1969]
Hoare, C. A. R. “An Axiomatic Basis for Computer Programming.” *Communications of ACM 12*, 10 (October 1969): 576-580.

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005.

[Humphrey 2006]
Humphrey, Watts S. *TSP: Coaching Development Teams.* Addison-Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/201731134.cfm

[Meyer 1992]
Meyer, Bertrand. “Applying Design by Contract.” *IEEE Computer* 25, 10 (October 1992): 40-51.

[Schwalbe 2007]
Schwalbe, Kathy. *Information Technology Project Management,* 5th ed. Course Technology, 2007.

[Suzumori 2003]
Suzumori, Hisayuki; Kaiya, Haruhiko; & Kaijiri, Kenji. “VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development,” *Proceedings of the 27th Annual International Computer Software and Applications Conference*. Hong Kong, PRC, September–October 2003. IEEE Computer Society, 2003.