Formal Extreme (and Extremely Formal) Programming

Ángel Herranz and Juan José Moreno-Navarro

Univ. Politécnica de Madrid
Campus de Montegancedo s/n, Boadilla del Monte 28660, Spain
+34 9133674{52,58}
{aherranz,jjmoreno}@fi.upm.es

Abstract. This paper is an exploratory work were the authors study how the technology of Formal Methods (FM) can interact with agile process in general and with Extreme Programming (XP) in particular. Our thesis is that most of XP practices (*pair programming*, *daily build*, *the simplest design* or *the metaphor*) are technology independent and therefore can be used in FM based developments. Additionally, other essential pieces like *test first*, *incremental development* and *refactoring* can be improved by using FM. In the paper we explore in a certain detail those pieces: when you write a formal specification you are saying *what* your code must do, when you write a test you are doing the same so the idea is to use formal specifications as tests. Incremental development is quite similar to the refinement process in FM: specifications evolve to code maintaining previous functionality. Finally FM can help to remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, and this is refactoring.

Keywords Extreme Programming, Formal Methods, Incremental Development, Formal Testing, Refactoring.

1 Motivation

At first sight, XP [1] and FM [11, 8] are water and oil: an *impossible* mixture. Maybe the most relevant discrepancy is that while one of the strategic motivation of XP is "spending later and earning sooner" FM require "spending sooner and earning later". However, a deeper analysis reveals that FM and XP can benefit their selves.

The use of formal specifications is perceived as improving reliability at the cost of lower productivity. XP and other agile processes focus on productivity so, in principle, using FM following XP practices could improve its efficiency. In particular, *pair programming*, *daily build*, *the simplest design* or *the metaphor* are XP practices that in our view are independent of the concrete development technology used to produce software and the declarative technology and FM is just a different development technology.

On the other hand, the main criticism to XP is that it has been called *systematic hacking* and, probably, the underlying problem is the lack of a formal or even semi-formal approach. But, what XP practices are liable to incorporate a formal approach? We think that *unit testing*, *incremental development* and *refactoring* are three main XP practices where FM can be successfully applied:

– When you write a formal specification you are saying *what* your code must do, when you write a test you are doing the same so one idea is to use formal specifications as tests.
– Incremental development is quite similar to the refinement process in FM: specifications evolve to code maintaining previous functionality.
– Finally FM can help to remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, and this is refactoring.

After all, it might be possible to dilute FM in XP. We would like to point out that we are not claiming to *formalise* XP (as could be understood from the joke in the title), but just to study how the declarative technology can be integrated in XP and how XP can take advantages of this technology.

Before exploring the above XP practices from a formal approach, SLAM (our formal tool) is presented in Section 2. In Sections 3.1 and 3.3 we briefly present how formal specifications can be used in the practices of testing and refactoring. Section 3.2 focuses in the formalisation of the incremental development under the prism of FM.

2 Formal Methods and SLAM

In spite of the great expectations generated around *declarative technologies* (formal methods, and functional and logic programming) years ago, these have not penetrated the mass market of software development. One of the main causes is a deficient software tool support for integrating formal methods in the development process. Since 2001 we are involved in the development of the SLAM [9] system, a *modern* comfortable tool for *specifying*, *refinement* and *programming*.

The formal notation SLAM-SL [7] embedded in the whole system is an object-oriented specification language valid for the *design and programming stages* in the software construction process. Although the main ideas in the paper could have been presented using any other FM and its associated notation, we think that the design of SLAM-SL gives to our notation important advantages.

For this paper, other of the most relevant features of SLAM-SL is that it has been designed as a trade-off between the expressiveness of its underlying logic and the possibility of code synthesis. From a SLAM-SL specification the user can obtain code in a high level programming language (let us say Java), a code that is *readable* and, of course, correct with respect to the specification. Because the code is readable, it can be modified and, we expect, improved by human programmers.

A complete SLAM-SL description is out of the scope of this work, but let us sketch some relevant elements for the goals of the paper.

2.1 Data Modelling

SLAM-SL is a *model based formal notation* [2] where *algebraic types* (*free types* under Z terminology) are used to specify a model for representing instances. From the point of view of an object-oriented programmer, data are modelled following the design pattern *State* ([4]):

```
class Order
state pending (customer : Customer,
               product : Product,
               quantity : Positive)
state delivered (customer : Customer,
                 product : Product,
                 quantity : Positive,
                 payment : Transfer)
```

Informally, an order instance can be in state *pending* so members *customer*, *product* and *quantity* are meaningful, or in state *delivered* and *customer*, *product*, *quantity* and *payment* are meaningful. Even more, *pending* and *delivered* are order constructors, and *customer*, *product*, *quantity* and *payment* are *getter* methods (the last one is partial). Automatically, the SLAM-SL compiler synthesised the following human understandable Java code:

```
class Order {
  private OrderState state;
  ...
}

class OrderState {
  private Customer customer;
  private Product product;
  private int Quantity;
  ...
}

class PendingOrderState extends OrderState {
}

class DeliveredOrderState extends OrderState {
  private Transfer payment;
}
```

Class invariants associated to every state are allowed, invariants that can be used to statically (through a theorem prover) or dynamically (through assertions) to check the specification and the implementation consistency.

2.2 Method Specification

The general scheme of a method specification is this one:

class A
...
method m (T_1, ..., T_n) : R
pre $P(self, x_1, \ldots, x_n)$
call $self$.m (x_1, ..., x_n)
post $Q(self, x_1, \ldots, x_n, result)$
chk $T_1(self, x_1, \ldots, x_n, result)$

...
chk $T_m(self, x_1, \ldots, x_n, result)$
sol $S(self, x_1, \ldots, x_n, result)$

As we can see, a method specification involves a *guard* or a *precondition* (the formula $P(self, x_1, \ldots, x_n)$) that indicates if the rule can be triggered, an *operation call scheme* (`self.m (`x_1, ..., x_n`)`); and a *postcondition* (given by the formula $Q(self, x_1, \ldots, x_n, result)$) that relates input state and output state. The formal meaning of this specification is given by the following property:

$$\forall s, x_1, \ldots, x_n.\, \texttt{pre}{-}m(s, x_1, \ldots, x_n) \;\Rightarrow\; \texttt{post}{-}m(s, x_1, \ldots, x_n, s.m(x_1, \ldots, x_n)$$

where pre and post predicates are defined in this way:

$$\begin{aligned} \texttt{pre}{-}m(s, x_1, \ldots, x_n) &\equiv P(self, x_1, \ldots, x_n) \\ \texttt{post}{-}m(s, x_1, \ldots, x_n, r) &\equiv Q(self, x_1, \ldots, x_n, r) \end{aligned}$$

The procedure to calculate the result of the method is called a *solution* in the SLAM-SL terminology and it has been indicated by the reserved word **sol** following by the formula $S(self, x_1, \ldots, x_n, result)$. Notice that the formula is written in the same SLAM-SL notation, but must be an *executable expression* (a condition that can be syntactically checked). The SLAM-SL compiler synthesised efficient and readable imperative code from solutions. The key concept is the operational use of *quantifiers* (extending usual logic quantifiers). Quantifiers allow the expressiveness of logic while the basis for their efficient implementation as traversal operations on data.

Once it is proved that the postcondition entails the solution it is ensured the correctness of the obtained code. However, the automatically generated code could not be enough efficient and, as we mentioned previously, the programmer can modify the generated code.

Formulas prefixed with the reserved word **chk** are extra properties that will hold in the program. Each T_i must be an *executable* formula and can be considered as tests (for instance that a prime number greater than 2 must be odd). Theoretically, they are not needed because they must be entailed by the postcondition, however, important errors in specifications can be caught. They can also be completed with some values (concrete values, intervals, etc.) what can provide automatic tests to be executed during the execution. Proof obligations are generated in order to prove that every T_i holds under the given postcondition and assertions can be generated in order to check that hand-coded modifications fulfil those properties.

2.3 Support for Testing and Debugging

Executable code is obtained from solutions and using similar techniques pre and postconditions are used to generate debugging annotations (assertions and exceptions) [6]. Notice that the postcondition can be complex enough to prevent code generation. However, test can always be checked. This feature can be used both to prevent errors in the case of programmer's modifications and to implements runtime tests. Furthermore, up to now the SLAM system is not automatically proving that the postcondition entails the

solution, so test can help to find wrong solutions. Nevertheless, as soon as this feature will be incorporated to the system the automatically generated code is always correct and no test checking is needed.

3 XP Practices

As mentioned in the motivation, most XP practices are technology independent. In our opinion, the XP process could be adopted by using SLAM (or any other FM tool) instead of an ordinary programming language and tool. In other words, we propose to write formal specifications instead of programs. A number of advantages appear:

- Rephrasing a XP rule, "The specification is the documentation" because we have a high level description with a formal specification of the intended semantics of the future code. One of the bigger efforts in the SLAM development has been to ensure that the generated code is readable enough. Therefore, the "answer is still in the code" (but also in the specification).
- FM tools (theorem provers, model checkers, etc.) help to maintain the consistency of the specification and the correctness of the implementation.
- Important misunderstandings and errors can be captured in the early stages of the development but close enough to code generation.

While in Agile Methods the emphasis is on staying light, quick, and low ceremony in the process, FM could make it sometimes heavier, sometimes not. Even in the first cases we have that: i) it is still can be considered a light method in the FM area, and ii) the benefits should compensate in many cases the increase of work.

Let us focus on in three XP pieces where we consider that FM can play an interesting role.

3.1 Unit Testing

In XP the role of writing the tests in advance is similar to the role of writing a precise requirement: it is used to indicate *what* the program is expected to do. Tests in XP solves two different problems:

- The detection of misunderstandings in the *intended specifications*.
- The detection of errors in the implementation.

The perspective under both problems is completely different when using FM. The detection of inconsistencies in formal specifications are supported by formal tools, mainly by *a generator of proof obligations* and by *a theorem prover assistant*. With both tools the user get information about possible inconsistencies.

The detection of errors in the implementation is absolutely unneeded thanks to the *verified design process*: a process that ensures that the code obtained from an original specification is correct with respect to it. Notice that the use of tests do not ensure that requirements are satisfied, just "convince" the programmer that it happens. The FM approach overcome this limitation.

So we propose to replace the tests by **chk** formulas expressed in SLAM-SL. There are several advantages of this approach:

1. tests can be complex enough but the SLAM system takes care of the code generation is feasible,
2. tests are executed automatically every time the program is run in debugging mode,
3. testing properties can be carried out in all the incremental versions of the code, i.e. they are automatically checked in all the iterations, and
4. automated formal tools can be used to improve the behaviour, for instance proving that some test are inconsistent with the specification by using a theorem proving.

3.2 Incremental Development

In this section we present the logical properties that the iterative development of software by the incremental addition of requirements must fulfil. We have called the set of those properties the *Combination Property* and it formally establishes that the combination of the code already obtained to solve the previous requirements and the code needed to solve the new one must fulfil all the requirements. The incremental development of XP needs to ensure that: i) at every step we develop the minimal code needed to solve the corresponding requirement, and ii) this code is combined with the previous code in such a way that the old requirements still hold. To solve this goal we establish the minimal properties that must be proved to ensure a correct behaviour.

We will call $story_i$ the formula expressing requirements at step i. At every step we want to develop a *function* f_i that covers all the requirements $story_1, \ldots, story_i$. To obtain f_i we depart from:

- the function $f_{i-1}(\overline{x}, \overline{y})$ with postcondition $post_{i-1}(self, \overline{x}, \overline{y}, result)$, and
- a function $g_i(\overline{x}, \overline{z})$ that solves requirement $story_i$.

Additionally, function f_i computes "more things" than f_{i-1}, i.e. the result of f_i includes the result of f_{i-1}, and maybe more data. Formally, there exists a projection π_i that relates both results.

Let us discuss some remarks with respect to these formulas before establishing the main properties. The fact that g_i is developed for requirements $story_i$ means that its postcondition entails $story_i(self, \overline{x}, \overline{z}, result)$. We assume that some of the arguments for g_i are still present in the previous code, i.e. arguments represented by variables $\overline{x}$ are still present in f_{i-1}, while some previous arguments $\overline{y}$ are not needed for $story_i$ and some new $\overline{z}$ are required.

Now, the main property to be proved can be formulated. Let us assume that the function $f_i(\overline{x}, \overline{y}, \overline{z})$ has been specified with postcondition $post_i(self, \overline{x}, \overline{y}, \overline{z}, result_i)$. To ensure that this function is correctly defined we must prove the *Combination Property*:

$$\begin{aligned} post_i(self, \overline{x}, \overline{y}, \overline{z}, result_i) \ \Rightarrow \\ \quad story_i(self, \overline{x}, \overline{z}, result_i) \wedge \\ \quad post_{i-1}(self, \overline{x}, \overline{y}, result_{i-1}) \wedge \\ \quad \pi_i(result_i) = result_{i-1} \end{aligned}$$

Now we can formally establish that this is the only property (at every step i) needed to ensure that the final code (i.e. f_n) entails all the requirements.

Theorem 1. *For every $i \in \{1, \ldots, n-1\}$ the following formulas hold:*

$$post_n(self, \overline{x}, \overline{y}, \overline{z}, result_n) \;\Rightarrow\; story_i(self, \overline{x}, \overline{z}, result_i)$$

$$post_{i+1}(self, \overline{x}, \overline{y}, \overline{z}, result_{i+1}) \;\wedge\; post_i(self, \overline{x}, \overline{y}, \overline{z}, result_i) \;\Rightarrow\; \pi_{i+1}(result_{i+1}) = result_i$$

The proof proceeds by induction on i.

A Simple Example

In the following example, we will show three customer stories for the development of a small telephone database ([10]). The customer wants a telephone database where information can be added and looked up maintaining two different tables: one with the persons and other one with the entries (pairs of person and phone). The specification written by development is the following one:

```
class Phone_DB

state (members : {Person},
       phones  : {(Person,Phone)})

constructor make_phone_DB
call make_phone_DB
post result.members = {}

modifier add_entry (Person, Phone)
pre  person in self.members and
     not (person, phone) in self.phones
call add_entry(person, phone)
post result.phones = self.phones + {(person,phone)}

modifier add_member (Person)
pre  not person in members
call add_member(person)
post members = self.members + {person}

observer find_phones (Person) : {Phone}
pre  person in dom(phones)
call find_phones(person) = self.phones(person)
```

In the second story, the customer asks for including a way to remove entries in the data base and this is the result of the development task:

```
modifier remove_entry (Person, Phone)
pre  (person,phone) in phones
call remove_entry(person, phone)
post phones = self.phones - {(person,phone)}
```

The combination property in this case is trivial to prove because we only have added a new operation. A consistency check is also trivial.

In the third customer story, she asks for removing the person from the database of members if its removed entry is the last one:

```
modifier remove_entry (Person, Phone)
pre  (person,phone) in phones
call remove_entry(person, phone)
post phones = self.phones - {(person,phone)} and
     if (exists phone : Phones with (person, phone) in phones)
     then members = self.members
     else members = self.members - {person}
     end
```

In this step, the postcondition of `remove_entry` must be proved to entail the previous postcondition. A theorem prover can automatically do the work: let A be the formula `phones = self.phones - {(person,phone)}` and B the right hand side of the conjunction, the proof obligation is

$$A \wedge B \Rightarrow A$$

what is directly the scheme of an inference rule in first order logic.

3.3 Refactoring

The declarative technology makes easier to find and remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, i.e to refactor code [3]. Thanks to the reflective properties of SLAM-SL, generic patterns can be specified and it can be proved that a specification is an instantiation of such a generic pattern. The idea it is having a relevant collection of generic patterns trust the prover technology of FM were able to *match* specifications with specifications in those patterns. Some works in formalising design patterns [4] have been done using SLAM-SL [5].

However, we need to be sure that the resulting code from refactoring is still readable enough. In any case, taking into account that it is for free, the programmer can spend some time in documenting it.

4 Conclusions

We have presented how some XP practices can admit the integration of Formal Methods and declarative technology. In particular, *unit testing*, *refactoring*, and, in a more detailed way, *incremental development* have been studied from the prism of FM.

Probably there is more room for FM ideas helping agile methodologies and XP, and we will study this as a future work.

One of the goals of the SLAM system is to make FM and their advantages closer to any kind of software development. Obviously FM are specially needed for critical applications but combining it with rapid prototyping and agile methodologies could make them affordable for any software construction. Up to know we have not equipped

SLAM with an automatic interface generator that precludes the use of our system for heavy graphical interface applications. The automatic generation of graphical interfaces is another matter of future work.

References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Pearson Education, 2000. ISBN 201-61641-6.
2. H. Ehrig, F. Orejas, and J. Padberg. Relevance, intergration and classification of specification formalism and formal specification techniques. In *Proc. FORMS, Formale Techniken für die Eisenbahnsicherung*, Fortschritt-Berichte VDI, Reihe 12, Nr. 436, VDI Verlag, 2000, pages 31 – 54, 1999.
3. M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
5. A. Herranz, J. Moreno, and N. Maya. Declarative reflection and its application as a pattern language. In M. Comini and M. Falaschi, editors, *11th. International Workshop on Functional and Logic Programming (WFLP'02)*, Grado, Italy, June 2002. University of Udine.
6. A. Herranz and J. J. Moreno. Generation of and debugging with logical pre and post conditions. In M. Ducasse, editor, *Workshop on Automated and Algorithmic Debugging 2000*. TU Munich, 2000.
7. A. Herranz and J. J. Moreno. On the design of an object-oriented formal notation. In *Fourth Workshop on Rigorous Object Oriented Methods, ROOM 4*. King's College, London, March 2002.
8. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.
9. The SLAM website. `http://lml.ls.fi.upm.es/slam`.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
11. J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.