Towards an Evolutionary Formal Software Development

Dieter Hutter and Axel Schairer
German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3, D 66213 Saarbrücken, Germany

Abstract

Although formal methods have been successfully applied in various industrial applications, their use in software development is still restricted to individual case studies. To overcome this situation we aim at a methodology for an evolutionary formal software development which allows for a stepwise and incremental development process along the line of rapid prototyping. The approach is based on work on a formal management of change for formal developments which is able to maintain proofs when changing specifications.

1. Introduction

Software development is usually organized by a life cycle model which structures and guides the activities between an initial idea of a product and its final implementation or performance testing. The most prominent model is the waterfall life cycle model in which the development process is organized as a sequence of steps from the initial software concept, requirement analysis, etc. through implementation and testing. Each phase is separated; reviews are hold at the end of each phase to determine whether the project is ready to advance to the next phase. However applying the waterfall life cycle model requires a correct and complete understanding of the project already from the beginning since backing up from mistakes, made in previous phases, is a difficult and expensive task. To overcome these restrictions and to cope with changing needs of the customers, life cycle models like evolutionary prototyping or evolutionary delivery have been developed which allow the development of the system concept as one moves through the project.

Life cycle models for formal methods are typically organized analogously to the waterfall model. Formal software development (as it is for instance incorporated in VSE [1], KIV [3] or SPECWARE [7]) is considered as a top-down approach, starting with a formal requirement specification and ending with an executable (with respect to the underlying abstract machine) specification. For each two successive specifications a formal refinement relation has to guarantee that the needs of the upper specification are taken up in the more detailed lower specification. More formally speaking, the axioms forming the upper specification – once they are mapped (via some morphism) to the language of the lower specification – have to be theorems of this more detailed specification. This is established by computing sufficient conditions for this to be the case, so called proof obligations, and constructing formal proofs for these obligations. Although the arising proof obligations are tackled with the help of automated reasoning techniques, they lengthen significantly the development process. Fully automated techniques like model checking are often used successfully in the context of hardware verification, however, for software development often (partially) interactive theorem proving has to be used because the non-finite domains can not be treated directly by fully automatic techniques.

Changing specifications inside a formal development will endanger the refinement relations and invalidate related, already existing proofs. As it takes months or even years to perform these proofs for a small or medium sized industrial project, an evolutionary formal software development is impracticable without appropriate techniques to (quickly) adjust the proof work in a changing environment. Various techniques have been developed to calculate and restrict the impacts of changes within formal software developments. The notion of development graphs [6, 2] allows for a logical encoding of structured specifications incorporating a management of change to minimize proof work in case of changing specifications. Recent extensions of development graphs incorporate the notion of hidings [10] and translations between different logic formalisms [9]. While development graphs cope with the validity of formal relations between theories and related proofs, other techniques have been developed to adapt existing proofs to changed specifications. Strategic proof information, encoded into tactics used for the proof search, as well as structural information of the given proof, are reused to tackle the proof of a changed problem (e.g. [11, 5]).

Although these techniques have been developed to cope with “small” changes, caused for instance by fixing errors

of the specifications, their success within this domain may also indicate that the dream of an evolutionary formal development will come true in the near future. In this paper, we outline a methodology to combine evolutionary prototyping and formal developments by stepwise refinements. Instead of considering a formal development as a sequence of specifications, we construe an evolutionary formal software development as a sequence of formal developments. Each development follows from the previous ones by applying some predefined transformation technique whereby transfering as much proof work as possible automatically. In section 2 we will investigate the nature of these transformations on formal developments and in section 3 we will present an example of such a transformation.

2. Evolutionary Formal Development

In practice, formal software development, as it is common today, allows only to increment the existing development by adding new specification parts or providing new proofs. In all existing systems the change or removal of specification parts will usually invalidate all related proofs and causes the developers to redo most of the work again. But adding new parts to the development is constrained by the notion of formal refinements. Since the executable specification has to satisfy the axioms of the requirement specification, adding new features to a program or introducing new exception handling is usually impossible without changing the formal requirement specification. Thus, already the top level formal specification must take into account design decisions usually made during implementation in a non-formal setting.

To overcome the limitations, mentioned above, we view an evolutionary formal software development process $\mathcal{D} = D_1, \ldots, D_n$ as a sequence of individual (self contained) formal developments $D_1, \ldots, D_n$. Thus, each of these developments D_i consists of layers of specifications and proofs of the refinement relations between successive layers. The soundness of the final program is guaranteed only by the soundness of the last formal development D_n but does not depend on previous versions D_i ($1 \leq i < n$). (Therefore, at each point in time in the development process we only have to store and work on the last formal development, unless an earlier one is of independent interest.) It may seem as if we have increased our work as we now have to provide n developments instead of a single one. However, the idea is to reuse most of a development D_i to construct the development D_{i+1}. More formally speaking, we are interested in transformations τ_i which map a development D_i (consisting of specifications and proofs) into the next development D_{i+1}. The more we can automate this mapping, i.e. the more we can map existing proofs, the less overhead we incur by evolutionary formal software development. Ideally, we can reuse all existing proof work such that for each new development D_{i+1} we only have to care about the *differences* of D_i and D_{i+1} rather the whole of D_{i+1}. Furthermore we would "only" have to adjust the proofs to the more complex environment rather than doing the proofs from scratch.

In the following we will discuss different types of such mappings between formal developments.

2.1 Changes of Logical Representations

The efficiency of using formal methods relies on the choice of an appropriate logical representation which eases the natural specification of the intended system and the guidance of the proofs to be done. The more expressive the logic is, however, the more difficult it is typically to guide the automatic proof search. This distinction is a motivation to switch between different logical representations.

An example is the use of abstractions for model checking, e.g. [4]. A specification in an expressive formalism is transformed into a finite state abstraction which can then be verified using model checking. In the reverse direction, i.e. if D_i is the abstraction of D_{i+1}, by construction of the abstraction certain properties proved in D_i carry over to D_{i+1}. In other words, τ_i is a transformation such that certain proven properties are invariant with respect to τ_i, and proofs for the properties in D_{i+1} can be computed from the proofs in D_i and knowledge about the abstraction.

2.2 Changes Inside Logical Representations

As part of the formal development process, specifications are changed, either because errors have to be corrected and changed requirements have to be taken into account, or because this is part of the specification methodology. An example for the latter are changes in specifications as required by fault transformations [8].

Changing specifications is usually done by free editing of the specifications. No logical relation between the specification before and after the editing can be guaranteed because there is no restriction on what the user can change in such an editing step. After the specification has been changed, proofs that were already present have to be patched and repaired by techniques of management of change. This works for some cases, but not for others. Many editing steps that occur regularly are very hard to handle this way. This is due to the fact that the tools to patch the proofs do not know about the intention of the user's changes. Also, typically it is very hard for the user to forsee what effects repairing the proofs will have.

Instead of editing specifications freely, we propose to view editing of specifications as a sequence of transformations on specifications and proofs. Each of a set of predefined transformations corresponds to a very simple editing

step. Now, users are restricted in the ways they can edit the specifications using these predefined transformations only. On the other hand, the tools can be built to know exactly what each transformation does and how the proofs can be repaired and patched accordingly. This has the advantage that as part of editing a specification the proofs are transformed automatically in a foreseeable manner. This is a generalization of what has been worked out for the special case of editing terminating SML programs in [12]. In the following section we will sketch some transformations that can be used to this effect in the setting of more general formal software development.

3. An Example

As mentioned in Sect. 2.2 editing specifications in a (partial) formal development can be seen as a series of transformations on the whole formal development. In the following we sketch two transformations that can be used to incrementally build specifications. Assume we have constructed a specification for a Unix-like file system. To keep the presentation manageable we will only look at a part of the very simplified version of the specification. A state of the file system, s, is represented by a mapping *s.files* from file names to files and a set of directories *s.dirs*. State transitions defined on the file system are operations to create and delete files and directories, and for reading and updating the contents of files. Let us look at *addfile* which adds an empty file with name n to a given directory d in the file store. Its definition as a predicate on predecessor and successor states s and s' is

$$
\begin{array}{l}
\forall d : Dir; n : Name; s, s' : State. \\
\quad addfile(d, n, s, s') \Leftrightarrow \\
\qquad \text{if } \neg direxists(d, s) \text{ then } s' = s \\
\qquad \text{elseif } \ldots \text{/* other error cases */} \\
\qquad \text{else } \; s'.dirs = s.dirs \wedge \\
\qquad\qquad s'.files = insert(file(d, n, Empty), s.files)
\end{array}
$$

One of many properties that we prove as an invariant of the system is that for each file in the store its parent directory exists: $\forall s : State.\ I(s) \Leftrightarrow \forall f : File.\ f \in s.files \Rightarrow f.parent \in s.dirs$. Amongst other things we have to prove that I is maintained by *addfile*: $\forall d : Dir; n : Name; s, s' : State.\ I(s) \wedge addfile(d, n, s, s') \Rightarrow I(s')$. Assume that we have completed some or even all of the proofs for this and many other invariants and operations. Then we decide to add access permissions to the model, i.e. add permissions to be stored with files and directories, add permissions to be associated with operations, and change the state transitions to check whether an operation is allowed or not. Note that these changes have effects on all operations, because all operations mention the state, which we change. We briefly sketch some of the transformations we use to accomplish this without losing the proofs that we have already done.

First, we use the transformation ADD-SLOT to add an additional slot to the entries stored in *s.files*. Where before entries had the slots *parent*, *name*, and *contents*, they will now have an additional slot *perm* for their access permissions, i.e. the constructor *file* expects an additional argument. ADD-SLOT transforms the specification and the proofs in such a way that they are again in a consistent state. For our example, throughout the specification wherever there is an occurrence of a term of the form $file(t_1, t_2, t_3)$, the term is transformed into $file(t_1, t_2, t_3, p)$. The dummy parameter $p : Perm$ is a new constant introduced by the transformation. The definition of *addfile* now reads $\forall d : Dir; n : Name; s, s' : State.\ addfile(d, n, s, s') \Leftrightarrow \ldots\ file(d, n, Empty, p)\ \ldots$ The same transformation is applied to all proofs in the formal development by adding the dummy parameter p to occurrences of terms of the form $file(\ldots)$. Since we only replace terms in a tightly controlled fashion, each inference step in the transformed proof is again a valid inference step. This is fairly straightforward, the only non-obvious point being proof obligations of the form $file(t_1, t_2, t_3) = file(t_1', t_2', t_3')$. However, proving the new obligation $file(t_1, t_2, t_3, p) = file(t_1', t_2', t_3', p)$ works in the same way as proving the original obligation except that $p = p$ occurs as an additional goal or unification constraint and can be handled uniformly because the same constant p has been used in all places. I.e. the proofs can be extended in a uniform way to close the additional (trivial) subgoal. This transformation has left the user with a state of the formal development that is well-formed again, and in this case, no new open subgoals have been introduced. However, the formulation of *addfile*, including the constant p, is not what is ultimately intended. Also, considering that p is inserted everywhere where there is a term involving the constructor *file*, e.g. in other operations, it is clear that we need to replace different occurrences of the newly inserted constant p by different terms, which we accomplish with the transformation CHANGE-TERM sketched below.

Before we apply CHANGE-TERM we add a new argument to the operation *addfile* to represent the user that executes the command. This is done by a transformation similar to ADD-SLOT: terms involving *addfile* are transformed throughout the specification and proofs by adding another constant u at the appropriate places. The definition of *addfile* then looks like $\forall d : Dir; n : Name; u : User; s, s' : State.\ addfile(d, n, u, s, s') \Leftrightarrow \ldots\ file(d, n, Empty, p)$. (For lack of space we have not shown the transformation to add u to the variables that are universally closed over.) We are now ready to replace the occurrence of p by $stdperm(u)$, which denotes the standard access permissions of subject u. The user applies the transformation CHANGE-TERM which transforms the specification and proofs by replacing

all occurrences of p that depend on the particular occurrence of p we have applied the transformation on. In the invariant proof for *addfile* the changes are again straightforward except for one place in the proof where we use a lemma about file mappings: $\forall x,y : File; r : List.\ x \neq y \Rightarrow x \in y :: r \Rightarrow x \in r$. In the original proof we used the lemma with y instantiated to $file(d,n,Empty,p)$ and later unified the term with the term from the definition of *addfile*. So, in this special case we know that we should change the instantiation of y in the new proof. However, in general it is not decidable locally what instantiation to use. CHANGE-TERM, therefore, does not attempt to guess. When, later on in the proof, the two terms $file(d,n,Empty,r)$ and $file(d,n,Empty,stdperm(u))$ do not match it introduces a subgoal that reads $file(d,n,Empty,p) \Leftrightarrow file(d,n,Empty,stdperm(u))$. With this subgoal, the main proof can be carried ahead and transformed, but there is an additional open subgoal. The user has to come back to this point and correct the instantiation of y in the lemma using an appropriate transformation.

All these example transformations just do one simple thing, and use special knowledge to transform the whole development in a sensible and expectable way. Although each transformation is simplistic, a combination of several transformations can be used to accomplish the more complicated changes the user wants to make in a step-by-step fashion. As is expected in general, transformations will introduce open subgoals or even invalidate subproofs, but the majority of the proof effort is preserved and the open subgoals can be closed using a combination of inference steps and other transformations. The examples we have worked out on paper so far indicate that this approach works quite well and intuitively, especially when compared to the alternative of doing the proofs from scratch.

4. Conclusion

In this paper we have presented first steps towards a genuinely evolutionary formal software development based on stepwise transformations on formal developments. The results we got are very promising and we are confident that the technique proves to be helpful once it is worked out in full detail. Since a formal development documents all information about how top-level requirements are related to the concrete implementation, formal development may, therefore, be an appropriate way to devise *adaptable* secure software libraries. In particular, the adaption of such a library using transformations may prove to be more time effective than the (non-formal) development from scratch.

In our future work we intend to develop sets of transformation rules for example domains. As noted in Sect. 2 there are many such domains which motivate transformation rules. However, the benefit of keeping proof effort over transformations can only be realized if the set of transformation rules is complete in the sense that application of transformation rules does not have to be interrupted by other editing steps which invalidates proofs.

References

[1] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special Issue on Mechanized Theorem Proving for Technology*, 3(1):66–77, 2000.

[2] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy and D. Bert, editors, *Recent Trends in Algebraic Development Techniques, (WADT-99)*, pages 73–88. Springer, LNCS 1827, 2000.

[3] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. Kiv 3.0 for provably correct systems. In *Current Trends in Applied Formal Methods*. Springer LNCS 1641, 1999.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[5] D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence (AMAI). Special Issue on Strategies in Automated Deduction*, 29:183–222, 2000.

[6] D. Hutter. Management of change in verification systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, pages 23–34. IEEE Computer Society, 2000.

[7] R. Juellig, Y. Srinivas, and J. Liu. SPECWARE: An advanced environment for the formal development of complex software systems. *Lecture Notes in Computer Science*, 1101, 1996.

[8] H. Mantel and F. Gärtner. A case study in the mechanical verification of fault tolerance. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 12(4), 2000.

[9] T. Mossakowski. Heterogeneous development graphs. Submitted for publication, 2001.

[10] T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In *Proceedings of Fundamental Approaches to Software Engineering (FASE2001)*. Springer, LNCS 2029, 2001.

[11] A. Schairer, S. Autexier, and D. Hutter. A pragmatic approach to reuse in tactical theorem proving. In *Proceedings Workshop on Strategies, 1st International Joint Conference on Automated Reasoning, IJCAR-2001*, Siena, Italy, 2001.

[12] J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML editor based on proofs-as-programs. In *Proc. 14th Int. Conf. Automated Software Engineering (ASE-1999)*, 1999.