On Models and Code
A Unified Approach to Support Large-Scale Deductive Program Verification

Marieke Huisman(✉)

University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl

Abstract. Despite the substantial progress in the area of deductive program verification over the last years, it still remains a challenge to use deductive verification on large-scale industrial applications. In this abstract, I analyse why this is case, and I argue that in order to solve this, we need to soften the border between models and code. This has two important advantages: (1) it would make it easier to reason about high-level behaviour of programs, using deductive verification, and (2) it would allow to reason about incomplete applications during the development process. I discuss how the first steps towards this goal are supported by verification techniques within the VerCors project, and I will sketch the future steps that are necessary to realise this goal.

1 The Problem: Scaling Deductive Program Verification

Deductive program verification is a technique to prove the correctness of a program w.r.t. its specification, which is given in terms of pre- and postconditions of the methods occurring in the program, following the Design-by-Contract principle [20]. Typically deductive program verification uses (an extension or variant of) Hoare logic [12] or dynamic logic [8] as its underlying verification technique.

Over the last years, enormous progress has been made on the use of such deductive program verification techniques for non-trivial examples, such as for example the discovery of a bug in Timsort [11], the verification of a Linux's USB keyboard driver [25], the verification of avionics software [7], and the various VerifyThis challenges (see *e.g.*, [14,17]). There are many different factors that have contributed to this progress, such as:

- the increase in power of automated provers,
- efficient use of multi-core hardware for formal verification tools,
- developments in specification languages, and
- the development of new verification theories, such as the use of concurrent separation logics to reason in a modular way about concurrent programs [3, 18,21].

Of course, there exist other formal analysis techniques that provide a much higher level of automation than deductive program verification, but the attractiveness of deductive program verification lies in that (1) it can be used to reason

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 111–118, 2018.
https://doi.org/10.1007/978-3-030-03418-4_7

about a very large and flexible class of program properties, and (2) it allows to reason about programs with an unbounded state space, and in particular about parametrised programs, i.e., it is possible to prove that a method `void m (int n)` respect its specification for any possible value of its parameter `n`.

Therefore, I believe it is important to investigate why the use of deductive program verification on large-scale industrial examples remains difficult, and what can be done to improve this situation. To understand why the use of deductive program verification remains difficult, many different reasons can be given, but I believe the most important ones are the following.

- Applications are often simply *too large* to handle, and the verifier lacks the overview of the complete application. Deductive program verification is traditionally quite closely connected to the concrete code, and as a result, it can be difficult to reason about the application at a suitable level of abstraction, because too many low-level details have to be dealt with.
- Deductive program verification typically requires a high number of auxiliary annotations (loop invariants, intermediate assertions), which require a detailed understanding of the code and of the verification process.
- For large applications, if we wish to use deductive program verification *during* the development process, when not all components are available yet, typically the deductive program verification tools require at least some stubs (for example, method contracts for the unimplemented methods) for the missing parts before the available components can be verified.
- To reason about *global* system properties (which is necessary if we wish to show that the program requirements are fulfilled) we need to have some way to reason about the missing components as well.
- As mentioned above, deductive program verification techniques are developed for pre-postcondition-style specifications, which usually do not match well with how high-level program requirements are expressed. We need formal techniques to connect these two levels of specifications.

In this position paper, I propose to work on the unification of models and code, to provide a solution to this problem. I will then sketch the first steps towards this solution, which we are currently developing within in the VerCors tool set. Finally, I will conclude by outlining further research challenges that need to be addressed to fully achieve my proposed solution.

2 The Solution: Unification of Models and Code

I believe that to make the use of deductive program verification on large-scale industrial examples possible, we need to soften the border between program and model (or global/high-level specification). Ideally, one can have different views on the components of an application, see Fig. 1 for a visualisation. First of all, there should of course be a code view, which is executable and deterministic, and which provides many low-level details. But one also needs to have a specification view on the same component, which is high-level, declarative and abstract, possibly

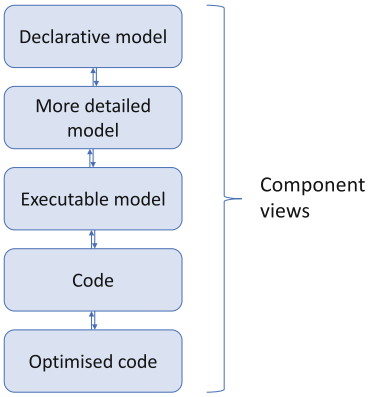

Fig. 1. Multiple views on a system component

non-deterministic, and leaves out many details. And many more different views for the same component should be possible: a high-level specification view can be refined into a more detailed, but still declarative model, which can further be refined into an executable model and finally into a code view. And this code view might be further refined, into a program that is further optimised, e.g. for performance or memory usage.

When we take this approach, there are two crucial requirements:

1. we need techniques to *connect* the views at all the different levels, and this connection needs to be *provably correct*, and
2. we need to be able to *compose* the components at all different view levels, i.e. it should be possible to "build" an application, where some of its components are only described by a high-level specification, and to combine these with code-level components, in order to reason about properties of the application as a whole, as visualised in Fig. 2.

There already exists some work on refinement between different views, such as done in VDM [5,10,15], Z [16], or EventB [1]. However, most of these approaches focus on refinement between different models, and if they go all the way to executable code, typically the code is extracted from a low-level model description, which is close to the code in spirit, but the connection between the code and the low-level model is not proven correct. Two exceptions that I am aware of are:

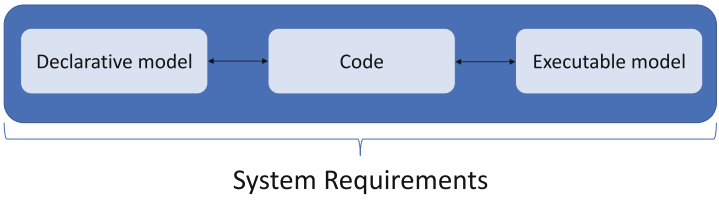

Fig. 2. Global verification with multiple component views

(1) the work by Dalvandi et al. [9], which aims at extracting code and annotations from an Event-B-model and then proving these correct using Dafny, and (2) the work of Tran-Jörgensen et al. [26], which generates JML annotations from VDM specifications.

Also, in the nineties early ideas on the transformation from models to programs, and from program to program have been developed, see e.g. [6,22,24]. These existing ideas need to be studied carefully, and it needs to be investigated if and how they can be incorporated in the current state-of-the-art deductive program verification tools.

Finally, also the work on the CompCert project, see e.g. [4,19] can be a source of inspiration. In this project, a verified tool chain for C is developed in Coq. However in the approach I advocate, also practical verification and applicability to different programming languages should be a major driving force.

3 First Steps: The VerCors Approach

Within the VerCors tool set, we have started to develop techniques to support this idea. In particular, we allow to specify an abstract model view of a component using process algebra, and then we use an extension of concurrent separation logic to prove that the concrete code behaves according to this model [23], while model checking technology can be used to derive global properties of the program from the process algebra models.

To illustrate this idea, let us consider the small code example in Fig. 3. Suppose we have a shared variable `x` protected by a lock `lck`, and two threads that manipulate `x`: one thread multiplies `x` by 4, the other thread adds 4 to `x`. The specifications of the two threads capture the thread's behaviours abstractly: assuming that the behaviour of the thread before this method call was equal to the process algebra term H (written *Hist(H)*), execution of the method adds the action *mult(4)* or *add(4)* to this behaviour (where $H.a$ denotes a process algebra term H, followed by action a, see the thread postconditions in lines 4 and 16).

```
1  class Mult extends Thread {
2
3  //@ requires Hist(H);
4  //@ ensures Hist(H.mult(4));
5  public void run() {
6       //@ action mult(4) {
7         lock(lck);
8         x = x * 4;
9         unlock(lck);
10      //@ }
11    }
12 }
```

```
13 class Add extends Thread {
14
15 //@ requires Hist(H);
16 //@ ensures Hist(H.add(4));
17 public void run() {
18      //@ action add(4) {
19        lock(lck);
20        x = x + 4;
21        unlock(lck);
22      //@ }
23   }
24 }
```

Fig. 3. Example: abstract behaviour specifications

```
1  //@ assume true;
2  //@ guarantee x == \old(x) * k;
3  action mult(k);
4
5  //@ assume true;
6  //@ guarantee x == \old(x) + k;
7  action add(k);
```

Fig. 4. Example action specifications

The *action* annotations (lines 6–10, and lines 18–22) inside the method body indicate the concrete code fragments that corresponds to the abstract actions. Given the action specifications that describe the effect of the actions *mult* and *add* in Fig. 4, we use our program logic to prove that the action implementations behave as specified.

Moreover, the program logic can also be used to verify that a process algebra term describes the global behaviour of the program. Suppose we have a `main` method, which starts the two threads and then waits for them to terminate. We can prove that the behaviour of this `main` method is to execute the *mult* and the *add* action in any order (see the postcondition in line 2 below, where $P + Q$ denotes a non-deterministic choice between P and Q and *empty* denotes an empty history). Finally, we can use existing model checking technology to reason about this abstract model, combined with the action specifications, to derive that the possible final values of variable x are 4 and 16.

```
1  //@ requires Hist(empty) & x == 0;
2  //@ ensures Hist(mult(4).add(4) + add(4).mult(4));
3  public void main(...) {
4    Thread t1 = new Mult(); Thread t2 = new Add();
5    t1.fork(); t2.fork();
6    t1.join(); t2.join();
7  }
```

This example is very simple, but we have used the same approach on larger and non-terminating programs [2,23,27].

4 Future Steps

The approach described above is still in its early stages. To fully realise the goal to have a seamless integration of code and models, more work is needed. I believe that the theory of how to make a connection between different levels of abstract models is reasonably well-understood [1,5,15,16], but to make a provably correct transformation from model (or high-level specification) to code is less clear. There are approaches to generate a model from code, but correctness of the extraction is then typically a meta-property, and cannot be proven for the model and code directly (and thus, in particular depends on whether the extraction is

correctly implemented[1]). There also exists work on (provably correct) model-to-code generation, see e.g. [28], but the generated code is still very close to the model, and needs to be improved to achieve a reasonable performance.

Therefore, I believe we need to address the following research challenges:

- we need to further develop refinement techniques that from an abstract model can generate annotated and verifiable code, where it is important that the generated code can be executed efficiently;
- we need techniques to prove that a program that is transformed to optimise it for performance remains correct after the transformation, see [13] for further ideas;
- we need to consider whether it is possible to automatically derive a model or abstract view from a concrete program; and
- we need to further develop the abstract model theory for concurrent software, in particular making the abstract models compositional, such that it is possible to reason about the global behaviour of a system that is composed of both abstract models and concrete code components.

Acknowledgements. The author is supported by NWO VICI 639.023.710 Mercedes project.

References

1. Abrial, J.-R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press (2010)
2. Amighi, A., Blom, S., Huisman, M.: VerCors: a layered approach to practical verification of concurrent software. In PDP, pp. 495–503 (2016)
3. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1) (2015)
4. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1
5. Bjørner, D.: The vienna development method (VDM). In: Blum, E.K., Paul, M., Takasu, S. (eds.) Mathematical Studies of Information Processing. LNCS, vol. 75, pp. 326–359. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09541-1_33
6. Bowen, J.P., Olderog, E.-R., Fränzle, M., Ravn, A.P.: Developing correct systems. In: Fifth Euromicro Workshop on Real-Time Systems, RTS 1993, Oulu, Finland, 22–24 June 1993, Proceedings, pp. 176–187. IEEE (1993)
7. Brahmi, A., Delmas, D., Essousi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: Embedded Real-Time Software and Systems (ERTS^2) (2018)
8. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing 1974, pp. 308–312. Elsevier, North-Holland (1974)

[1] Of course, a similar argument can be made here, but the advantage is that if the annotated code is available, correctness can be reverified by other tools.

9. Dalvandi, M., Butler, M.J., Rezazadeh, A.: Transforming Event-B models to Dafny contracts. In: Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015), Volume 72 of Electronic Communications of the EASST (2015)
10. Dawes, J.: The VDM-SL Reference Guide. Pitman (1991)
11. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
13. Huisman, M., Blom, S., Darabi, S., Safari, M.: Program correctness by transformation. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 365–380. Springer, Cham (2018)
14. Huisman, M., Klebanov, V., Monahan, R., Tautschnig, M.: VerifyThis 2015 a program verification competition. Int. J. Softw. Tools Technol. Transfer (2016)
15. International Organisation for Standardization. Information technology–Programming languages, their environments and system software interfaces–Vienna Development Method-Specification Language–Part 1: Base language, December 1996. ISO/IEC 13817–1
16. International Organisation for Standardization. Information technology–Z Formal Specification Notation-Syntax, Type System and Semantics (2000). ISO/IEC 13568:2002
17. Joosten, S.J.C., Oortwijn, W., Safari, M., Huisman, M.: An exercise in verifying sequential programs with VerCors. In: Summers, A.J. (ed.) 20th Workshop on Formal Techniques for Java-like Programs (FTfJP) (2018)
18. Jung, R., et al.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL, pp. 637–650. ACM (2015)
19. Kästner, D., et al.: CompCert: practical experience on integrating and qualifying a formally verified optimizing compiler. In: ERTS 2018: Embedded Real Time Software and Systems. SEE (2018)
20. Meyer, B.: Object-Oriented Software Construction, 2nd Edn. Prentice-Hall (1997)
21. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
22. Olderog, E.-R., Rössig, S.: A case study in transformational design of concurrent systems. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993. LNCS, vol. 668, pp. 90–104. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56610-4_58
23. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 191–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_12
24. Partsch, H.: Specification and Transformation of Programs - A Formal Approach to Software Development. Texts and Monographs in Computer Science. Springer, Heidelberg (1990). https://doi.org/10.1007/978-3-642-61512-2
25. Penninckx, W., Mühlberg, J.T., Smans, J., Jacobs, B., Piessens, F.: Sound formal verification of Linux's USB BP keyboard driver. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_21
26. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. STTT **20**(2), 211–235 (2018)

27. Zaharieva-Stojanovski, M.: Closer to reliable software: verifying functional behaviour of concurrent programs. Ph.D. thesis, University of Twente (2015)
28. Zhang, D.: From concurrent state machines to reliable multi-threaded Java code. Ph.D. thesis, Technische Universiteit Eindhoven (2018)