Correctness-by-Construction and Post-hoc Verification: Friends or Foes?

Maurice H. ter Beek[1(✉)], Reiner Hähnle[2], and Ina Schaefer[3]

[1] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] TU Darmstadt, Darmstadt, Germany
[3] TU Braunschweig, Braunschweig, Germany

Abstract. While correctness-by-construction and post-hoc verification are traditionally considered to provide two opposing views on proving software systems to be free from errors, nowadays numerous techniques and application fields witness initiatives that try to integrate elements of both ends of the spectrum. The ultimate aim is not merely to improve the correctness of software systems but also to improve their time-to-market, and to do so at a reasonable cost. This track brings together researchers and practitioners interested in the inherent 'tension' that is usually felt when trying to balance the pros and cons of correctness-by-construction versus post-hoc verification.

Motivation and Aim

Correctness-by-Construction (CbC) sees the development of software (systems) as a scientific discipline of engineering. Originally intended as a mere means of programming algorithms that are correct *by construction* [22,27], the approach found its way into commercial development processes of complex systems [24,25]. In this larger context, we can say that CbC advocates a step-wise refinement process from specification to code, ideally by CbC design tools that automatically generate error-free software implementations from rigorous and unambiguous specifications of requirements. Afterwards, testing only serves the purpose of validating the CbC process rather than to find bugs. (Of course, bugs might still be present outside the boundaries of the verified system: libraries, compilers, hardware, etc.).

In Post-hoc Verification (PhV), on the other hand, formal methods and tools are applied only after the (software) system has been constructed, not during the development process. Typically, a formal specification of (an abstraction of) the implemented system describes how it should behave, after which validation and verification techniques like testing [12,32,33], bug finding [4,26,29], model checking [3,15,17], and deductive verification [7,23,35] are used to check whether the implementation indeed satisfies the specifications and meets the user's needs.

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 723–729, 2016.
DOI: 10.1007/978-3-319-47166-2_51

While two independent system models that are verified against each other can provide additional assurance that the designers' intentions have been captured correctly, PhV is notoriously difficult to carry out.

Recently, numerous techniques and fields of application witness initiatives that attempt to integrate elements of both ends of the spectrum, ultimately aiming to satisfy the holy grail of improving the correctness of software systems as well as their time-to-market, and to do so at a reasonable cost. This track brings together researchers and practitioners interested in the inherent 'tension' that is usually felt when trying to balance the pros and cons of CbC vs. PhV. In particular, we invited researchers and practitioners working in the following communities to shed their light on CbC vs. PhV:

- People working in Software Product Line Engineering (SPLE), who try to lift successful formal methods and verification tools from single product (system) engineering to SPLE in order to say something about the correctness of all products (programs, variants) of an SPL (whose population is exponential in the number of features).
- People working in System-of-Systems Engineering (SoSE), who address the verification (correctness, but also issues like reliability, resilience, robustness, security, and sustainability) of networks of interacting legacy and new software (systems).
- People working on (system) synthesis, who aim for transforming a logical specification into a system that is guaranteed to satisfy the specification in all possible environments.
- People working on deductive verification, who typically require a detailed understanding of why the system works correctly before actual verification to be able to express the correctness of a program as a set of verification conditions to be discharged.
- People working on 'bug-finding' lightweight verification, who trade off full functional verification for being able to deal with real-world languages and large programs as well as to avoid having to write formal specifications.
- People working on Design for Verification (DfV), mostly in hardware design, who advocate the usage of design methodologies, languages, patterns, etc., that make PhV a realistic option.
- People working on Statistical Model checking (SMC), who trade off model checking's verification accuracy, which however requires the entire state space to be known upfront, for scalability by resorting to the computationally more efficient sampling of simulations of (dynamic, black-box, infinite-state) systems until sufficient statistical evidence has been found.

Contributions

Watson et al. [36] argue for the marriage of CbC with PhV in order to leverage the advantages and to mitigate the disadvantages of both approaches. CbC specifies a problem in terms of its pre- and postconditions and then develops a final algorithmic solution in small, tractable refinement steps. The paper advocates a lightweight approach to proving the correctness of each refinement step. The consequent risk of errors should then be minimised by relying on a PhV system that now obtains *for free* the pre- and postconditions, as well as loop variants and invariants that it requires for proving partial correctness as well as termination.

Beckert et al. [6] address the following important question in the specific setting of legacy code: why is it so hard to perform PhV (deductive verification in particular) and what can be done to make it any easier? They answer the first part of this question by presenting a collection of (known) insights in a systematic way, larded with examples. Subsequently, they contribute to answering the second part of the question by first discussing possible means to tackle the challenges offered by legacy code verification and then suggesting a strategy for deductive PhV, together with possible improvements to existing deductive verification methodologies and tools like KeY [2,7] and VCC [20].

Cleophas et al. [18] discuss how CbC-based development may lead to a deep comprehension of algorithm families, based on the fact that organising the refinements obtained during CbC-based design in a taxonomy leads to a classification of common and varying properties within a family of algorithms and thus to insight in the relations among its elements. They also argue that using taxonomies in the implementation of toolkits, i.e. a library of all variants, for TABASCO [19] has the additional benefit of providing a meaningful starting point for extractive and proactive SPLE. For both, a concrete methodology is presented.

Kleijn et al. [9] consider systems of systems, represented as team automata, whose components interact by the synchronised execution of common actions [8]. They study conditions for the compatibility of components, defined as being free from message loss and deadlocks, relative to notions of synchronisation other than mandatory synchronised execution. They focus on various kinds of master-slave synchronisations, which require input actions (for 'slaves') to be driven by output actions (from 'masters'). Team automata composed according to this notion of synchronisation are exemplified and studied in some detail, including an extensive discussion of (potential) applications.

Legay et al. [34] introduce DynBLTL, an extension of time-bounded LTL [16], as a new 3-valued logic specifically aimed to reason over dynamically evolving software architectures. Here dynamism is understood as allowing components to be removed, added or (re)connected differently. The third value (undefined) is used to deal with components that are absent in a system configuration being evaluated. The semantics is that of (un)timed traces of graphs seen as snapshots of the architecture at a specific moment of computation. One can quantify over

all connections and components of a specific type or a specific component or connection, and the modalities allow to reasoning over a bounded number of steps or a bounded amount of time. Since the number of components is unknown upfront, SMC by means of an integration into the PLASMA statistical model checker [30] is an obvious choice. An example illustrates the approach.

Méry et al. [14] aim to show how CbC and PhV can possibly be combined in a productive way by describing a general framework that integrates the following two different approaches to software verification: program refinement as supported by Event-B [1] and program verification as supported by the Spec# programming system [5]. In particular, they describe a plug-in for Event-B's RODIN toolset that is able to automatically transform a given abstract Event-B specification into a recursive algorithm that is correct-by-construction and which can be directly translated into executable code.

Schaefer et al. [28] investigate the feasibility of generalizing the concept of proof-carrying code to proof-carrying apps as a means to verify extensible software platforms at deployment time. Rather than global safety policies, contracts are used to specify functional properties of the API of the base software platform, leaving it to the provider of the extension to verify that all API calls adhere to the contract. The resulting proof artefacts are used at deployment time to allow proof checking. After discussing the criteria that enable a verification technique for contract-based deployment-time verification, the applicability of deductive verification with KeY [2,7] and data-flow analyses with Soot [31] to the proof-carrying apps scenario is examined for a simple Java implementation.

De Vink et al. [10] illustrate the idea of supervisory controller synthesis for SPLE by applying the CIF 3 toolset [11] to an example. They show how to automatically synthesise an SPL model (in the form of an automaton for each valid product of the SPL) starting from a so-called attributed feature model, component behaviour models associated with the features, and additional behavioural requirements (like state invariants, event orderings, and guards on events). The resulting CIF 3 model then satisfies all feature-related constraints as well as all behavioural requirements, by construction. Further behavioural properties can be verified by exporting such SPL models in the input format of the mCRL2 model checker [21].

Beyer [13] outlines a few existing verification approaches that, in an attempt to try to increase the impact of formal verification, combine the advantages of automatic verification techniques and interactive verification techniques. The former, which usually expect the user to set the parameters while the prover computes the necessary invariants and the proof, work well for large systems, whereas the latter, which usually expect the user to provide invariants while the prover establishes a formal correctness proof, work well for sophisticated specifications.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Ahrendt, W., et al.: The KeY platform for verification and analysis of java programs. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 55–71. Springer, Heidelberg (2014)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. ACM SIGPLAN Not. **37**(1), 1–3 (2002)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
6. Beckert, B., Bormer, T., Grahl, D.: Deductive verification of legacy code. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 749–765. Springer, Heidelberg (2016)
7. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach. Springer, Heidelberg (2007)
8. ter Beek, M.H., Kleijn, J.: Team automata satisfying compositionality. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 381–400. Springer, Heidelberg (2003)
9. ter Beek, M.H., Kleijn, J., Carmona, J.: Conditions for compatibility of components: the case of masters and slaves. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 784–805. Springer, Heidelberg (2016)
10. ter Beek, M.H., Reniers, M.A., de Vink, E.P.: Supervisory controller synthesis for product lines using CIF 3. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 856–873. Springer, Heidelberg (2016)
11. van Beek, D.A., Fokkink, W.J., Hendriks, D., Hofkamp, A., Markovski, J., van de Mortel-Fronczak, J.M., Reniers, M.A.: CIF 3: model-based engineering of supervisory controllers. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 575–580. Springer, Heidelberg (2014)
12. Bertolino, A., Inverardi, P., Muccini, H.: Software architecture-based analysis and testing: a look into achievements and future challenges. Computing **95**(8), 633–648 (2013)
13. Beyer, D.: Partial verification and intermediate results as a solution to combine automatic and interactive verification techniques. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 874–880. Springer, Heidelberg (2016)
14. Cheng, Z., Méry, D., Monahan, R.: On two friends for getting correct programs: automatically translating event-B specifications to recursive algorithms in Rodin. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 821–838. Springer, Heidelberg (2016)
15. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. Commun. ACM **52**(11), 74–84 (2009)
16. Clarke, E.M., Faeder, J.R., Langmead, C.J., Harris, L.A., Jha, S.K., Legay, A.: Statistical model checking in BioLab: applications to the automated analysis of T-Cell receptor signaling pathway. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 231–250. Springer, Heidelberg (2008)

17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
18. Cleophas, L., Kourie, D.G., Pieterse, V., Schaefer, I., Watson, B.W.: Correctness-by-construction ∧ taxonomies ⇒ deep comprehension of algorithm families. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 766–783. Springer, Heidelberg (2016)
19. Cleophas, L.G., Watson, B.W., Kourie, D.G., Boake, A., Obiedkov, S.A.: TABASCO: using concept-based taxonomies in domain engineering. S. Afr. Comput. J. **37**, 30–40 (2006)
20. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
21. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS). LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
22. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT Numer. Math. **8**(3), 174–186 (1968)
23. Filliâtre, J.-C.: Deductive software verification. Int. J. Softw. Tools Technol. Transfer **13**(5), 397–403 (2011)
24. Hall, A.: Correctness by construction: integrating formality into a commercial development process. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, p. 224. Springer, Heidelberg (2002)
25. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. IEEE Softw. **19**(1), 18–25 (2002)
26. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 291–301. ACM (2002)
27. Hoare, C.A.R.: Proof of a program: FIND. Commun. ACM **14**(1), 39–45 (1971)
28. Holthusen, S., Nieke, M., Thüm, T., Schaefer, I.: Proof-Carrying Apps: Contract-based deployment-time verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 839–855. Springer, Heidelberg (2016)
29. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Not. **39**(12), 92–106 (2004)
30. Jegourel, C., Legay, A., Sedwards, S.: A platform for high performance statistical model checking – PLASMA. In: Flanagan, C., König, B. (eds.) TACAS 2012 (ETAPS). LNCS, vol. 7214, pp. 498–503. Springer, Heidelberg (2012)
31. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (2011)
32. Mathur, A.P.: Foundations of Software Testing, 2nd edn. Addison-Wesley, Boston (2014)
33. Pezzè, M., Young, M.: Software Testing and Analysis: Process Principles and Techniques. Wiley, Hoboken (2007)
34. Quilbeuf, J., Cavalcante, E., Traonouez, L.-M., Oquendo, F., Batista, T., Legay, A.: A logic for the statistical model checking of dynamic software architectures. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 806–820. Springer, Heidelberg (2016)

35. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning. MIT Press, Cambridge (2001)
36. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.G.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 730–748. Springer, Heidelberg (2016)