X-by-Construction

Maurice H. ter Beek[1], Loek Cleophas[2,3],
Ina Schaefer[4], and Bruce W. Watson[3,5]

[1] ISTI–CNR, Pisa, Italy
m.terbeek@isti.cnr.it
[2] TU Eindhoven, Eindhoven, The Netherlands
l.g.w.a.cleophas@tue.nl
[3] Stellenbosch University, Stellenbosch, South Africa
loek@sun.ac.za, bwwatson@sun.ac.za
[4] TU Braunschweig, Braunschweig, Germany
i.schaefer@tu-bs.de
[5] CAIR, Stellenbosch, South Africa

Abstract. After decades of progress on Correctness-by-Construction (CbC) as a scientific discipline of engineering, it is time to look further than correctness and investigate a move from CbC to XbC, i.e., considering also *non-functional properties*. *X-by-Construction (XbC)* is concerned with a step-wise refinement process from specification to code that automatically generates software (system) implementations that *by construction* satisfy specific non-functional properties concerning security, dependability, reliability or resource/energy consumption, to name but a few. This track brings together researchers and practitioners that are interested in CbC and the promise of XbC.

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* [4, 8], the approach found its way into commercial development processes of complex software systems [6, 7]. In this larger context, we can say that CbC advocates a step-wise correctness-preserving refinement process from specification to code, ideally by CbC design tools that automatically generate error-free (system) implementations from rigorous and unambiguous specifications of requirements. Afterwards, testing only serves the purpose of validating the CbC process rather than finding bugs.

A lot of progress has been made in this domain during the last four decades or so, implying the time has come to look further than correctness and investigate a move from CbC to XbC, i.e., considering also *non-functional properties*. *X-by-Construction (XbC)* is concerned with a step-wise refinement process from specification to code that automatically generates software (system) implementations that *by construction* satisfy specific non-functional properties concerning security, dependability, reliability or resource/energy consumption, etc.

Building on the highly successful track "Correctness-by-Construction and Post-hoc Verification: Friends or Foes?" at ISoLA 2016 [1], which focussed on the combination of post-hoc verification with CbC, the aim of the current XbC track at ISoLA 2018 is to bring together researchers and practitioners that are interested in CbC and the promise of XbC. Therefore, we invited researchers and practitioners working in the following communities to discuss moving from CbC to XbC:

- ☑ People working on system-of-systems, who address modelling and verification (correctness, but also non-functional properties concerning security, reliability, resilience, energy consumption, performance and sustainability) of networks of interacting legacy and new software systems, and who are interested in applying XbC techniques in this domain in order to prove non-functional properties of systems-of-systems by construction (from their constituent systems satisfying these properties).
- ☑ People working on quantitative modelling and analysis (e.g. through probabilistic or real-time systems and probabilistic or statistical model checking) in particular in the specific setting of dynamic, adaptive or (runtime) reconfigurable systems with variability. These people work on lifting successful formal methods and verification tools from single systems to families of systems, i.e., modelling and analysis techniques that need to cope with the complexity of systems stemming from behaviour, variability, and randomness—and which focus not only on correctness but also on non-functional properties concerning safety, security, performance or dependability properties. As such, they are likely interested in applying XbC techniques in this domain to prove non-functional properties of families of systems by construction (from their individual family members satisfying these properties).
- ☑ People working on generative software development, who are concerned with the automatic generation of software from specifications given in either general formal languages or domain-specific languages, leading to families of related software (systems). Also in this setting, the emphasis so far has typically been on functional correctness, but the restricted scope of the specifications—especially for domain-specific languages—definitely offers a suitable ground for reasoning about non-functional properties, and for using XbC techniques to guarantee such properties.
- ☑ People working on systems security, privacy and algorithmic transparency and accountability, who care about more than correctness. The application of XbC techniques could provide security guarantees from the outset when designing critical systems. It could also enforce transparency when developing algorithms for automated decision-making, in particular those based on data analytics—thus reducing algorithmic bias by avoiding opaque algorithms. For instance, people working on *Privacy by Design* start from the presupposition that any personal data processing environment should be designed such that privacy is taken into account already in the requirement elicitation and earliest design phases [3].

Contributions

In his keynote, Poll [12] discusses some positive and negative experiences in applying formal methods to security. Based on insights from the language-based security paradigm LangSec, he provides directions for leveraging formal methods to improve security.

Johnsen et al. [2] plead for a more central, high-level role of deployment decisions as part of a by-construction development process and call it *Deployment-by-Construction (DbC)*. According to this paradigm, deployment decisions should be expressed as part of a program's high-level model and evaluated in terms of how they affect program performance. To illustrate their proposal, the authors apply DbC in the context of parallel architectures with shared memory and caches. More concretely, they present a toolbox for evaluating deployment decisions for specific high-level parallel programs when interacting with existing memory structures. Based on the input, the toolbox scores the program execution with a performance indicator to compare different implementations and environments. The authors also provide a proof-of-concept implementation of the toolbox in Maude. Given a set of deployment decisions, this paper thus provides a means to abstractly represent and analyse a program's memory accesses with respect to their impact on runtime behaviour.

Steffen et al. [10] propose an approach to obtain a property X by construction via model transformations. The main idea of the approach is that an initial model is transformed into another model that guarantees the property X. The model transformations are very much into the spirit of aspect-oriented programming where an aspect is weaved into a base program such that the aspect implements a specific behaviour at certain base program points. As an instantiation of this idea, the authors consider their graphical modelling language framework CINCO with which graphical modelling languages and transformations can easily be specified and implementations thereof can be generated and generate models that guarantee the properties of model checkability and learnability.

Méry [11] describes a procedure for formal system modelling and step-wise refinement of algorithms using Event-B. He distinguishes several best practices for modelling as *design patterns*. The idea is to provide an abstract specification of the problem in Event-B, which is then refined progressively and transformed into an algorithm which is correct-by-construction in the sense that it satisfies the properties defined in the abstract specification. The patterns are categorised in paradigms (Inductive, Call-as-Event, Call-as-Service), and for each category a pattern is described in an abstract way, together with an example in the form of a problem solution developed using the introduced pattern. The crucial refinement steps are shown on an abstract level during the description of the pattern, but the actual application of a pattern is given by showing the resulting algorithm, invariants and properties.

Steinhöfel and Hähnle [15] describe an approach for compilation that is verified to be correct, but done in a modular way. This is achieved by a rule-based approach, relating each source language construct to the corresponding target language construct. The translation between the two is done at the level of sym-

bolic execution trees, and proving the correctness of the translation can be done automatically provided both source and target language have been formalised in terms of symbolic execution (SE) rules. Proving the correctness is done via a program logic that is capable of SE of abstract programs. A new source or target language can easily be added by formalising such a language in terms of SE rules. The authors exemplify this by re-using a previously available SE formalisation of a subset of the Java language, and define one for the LLVM IR.

Huisman et al. [9] consider deductive program verification in contexts where high-level programs need to be refined to lower-level programs as a compilation step or for performance reasons such as introducing multi-core computing constructs, e.g., loop parallelizations. The authors argue for not just transforming program code in such cases, but to transform existing program annotations in concert. This is proposed to ensure that the result of the transformation is suitably annotated, i.e., to ensure the result can be proven correct provided the original program with its annotations could be. The authors sketch a diverse range of example settings for such annotation-aware transformations, and provide a research agenda of the open questions that need to be answered for a fully-fledged methodology around such annotation-aware transformations, including automation and genericity beyond the settings discussed in the paper.

Tribastone [16] provides an interesting perspective on software performance and ensuring it by construction. The main idea of this extended abstract is to extract a software performance model from the code, analyse it and propose fixes to the code if the performance properties are not met. Open research challenges for this approach are discussed which include which quantitative models to use and how to derive those performance models faithfully from the code base.

Schneider [14] discusses the idea of privacy-by-design (PbD) as one of the biggest challenges in finding suitable ways to handle personal data. Current legislation makes it mandatory to comply with the privacy requirements set in the regulation. PbD is one approach to build a system in a way such that it satisfies those privacy requirements from the inception of the system and also is a possibility to show to regulators that the privacy requirements are met. Schneider asks the question how much privacy can effectively be achieved by PbD, and if it cannot, if there are other ways to achieve privacy by construction.

Legay et al. [5] consider a move from correctness-by-construction to security-properties-by-construction via calculi for reasoning about such properties. The authors begin with straightforward password-entry code (in C), in which a Boolean access control variable is set depending on password correctness. The resulting calculus expresses the equivalence of that variable with the validity of the entered password. That example is neatly extended to capture timing properties which form some of the well-known attacks on such C code. The calculus is also seamlessly combined with the correctness argument. A considerably more complex example is also worked out, in which AES-style cryptography is considered. While the standard code in that example is remarkably compact, it hides a number of security vulnerabilities. Again, the calculus allows for reasoning about the known potential problems.

Schaefer et al. [13] propose an approach to confidentiality-by-construction (C14bC). The main idea is to recast information flow control properties in a CbC-style set of refinement rules. Hence, a program observing information flow control properties can be derived in a constructive fashion that preserves the desired security policy *ab initio* rather than rejecting a program as non-compliant *ex post facto*. The authors give a set of those refinement rules for a simple guarded command language and also consider the notion of declassification in order to build a more expressive constructive framework. This work paves the way for future extensions to more generic information flow policies also considering trust-by-construction.

Conclusion

The X-by-Construction track shows that a number of different, complementary approaches can be used to ensure that non-functional (X-)properties are guaranteed to hold in a system by construction. Formal methods for classical functional correctness can be extended to apply to X-properties as well [5, 12, 14]. Refinement-based approaches from classical functional CbC can be extended to X-properties [13], or model transformations can be used to introduce an X-property into the transformed model [10]. Refinement-based approaches can be generalised by using patterns capturing well-established design decisions to incorporate X-properties into a deployed system [2,11]. Correctness of those transformations can be ensured by correctly transforming annotations that are used for verification of the transformed program [9] or by proving the transformation rules correct themselves [15]. In the other direction, models extracted from implementation artifacts can be used for analysis of X-properties and results can be used to fix the implementation [16]. It remains a challenge for future research to see which of these approaches is applicable to what kind of X-property and which additional ideas are necessary to arrive at a holistic development approach for X-by-Construction.

References

1. M. H. ter Beek, R. Hähnle, and I. Schaefer. Correctness-by-Construction and Post-hoc Verification: Friends or Foes? In T. Margaria and B. Steffen, editors, *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16)*, volume 9952 of *LNCS*, pages 723–729. Springer, 2016.
2. S. Bijo, E. B. Johnsen, K. I. Pun, C. Seidl, and S. L. T. Tarifa. Deployment by Construction for Multicore Architectures. 2018. In this volume.
3. A. Cavoukian. Privacy by Design. *IEEE Technology and Society Magazine*, 31(4):18–19, 2012.
4. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.
5. T. Given-Wilson and A. Legay. X-by-C: Non-Functional Security Challenges. 2018. In this volume.

6. A. Hall. Correctness by Construction: Integrating Formality into a Commercial Development Process. In L. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume 2391 of *LNCS*, pages 224–233. Springer, 2002.
7. A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1):18–25, Jan/Feb 2002.
8. C. A. R. Hoare. Proof of a Program: FIND. *Communications of the ACM*, 14(1):39–45, 1971.
9. M. Huisman, S. Blom, S. Darabi, and M. Safari. Program Correctness by Transformation. 2018. In this volume.
10. M. Lybecait, D. Kopetzki, and B. Steffen. Design for 'X' through Model Transformation. 2018. In this volume.
11. D. Méry. Modelling by Patterns for Correct-by-Construction Process. 2018. In this volume.
12. E. Poll. (Some) Security by Construction through a LangSec Approach. 2018. In this volume.
13. I. Schaefer, T. Runge, A. Knüppel, L. Cleophas, D. Kourie, and B. W. Watson. Towards Confidentiality-by-Construction. 2018. In this volume.
14. G. Schneider. Is Privacy by Construction Possible? 2018. In this volume.
15. D. Steinhöfel and R. Hähnle. Modular, Correct Compilation with Automatic Soundness Proofs. 2018. In this volume.
16. M. Tribastone. Towards Software Performance by Construction. 2018. In this volume.