Guest Editorial: Special Issue on Formal Methods and Models for System Design

30

Organized to develop a solid conceptual foundation and an engineering discipline for applying formal methods in hardware/software co-design, the first MEMOCODE conference was held in Mont Saint-Michel, France, in 2003. A major objective of MEMOCODE 2003 and of the MEMOCODE conferences held annually since 2003 has been to bridge the gap between formal methods theory and development practice. While tremendous progress has been made in formal methods for constructing, modeling, and verifying systems, adoption of formal methods in practice remains relatively rare—practitioners are often unaware of relevant results published by formal methods researchers or have difficulty applying these results in an industrial setting. Likewise, researchers are often unaware of the real problems that practitioners face, and typically their results cannot be directly applied in industry. To address this problem, the MEMOCODE conferences through the years have had a basic premise—that formal methods researchers, hardware designers, and system and software developers can all benefit from an exchange of ideas and experiences. While the challenges of transferring the results of research into practice have remained, since 2003, the boundaries between computer system components, such as hardware, software, firmware, middleware, and applications, have blurred. This evolution in system design and development practices led in 2014 to a change in the title and scope of the MEMOCODE conference from its original focus on hardware/software co-design to its new focus on formal methods and models for developing computer systems and computer system components.

This special issue contains extended versions of four of the best papers presented at the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15). These four papers were chosen from 21 technical papers presented at the conference, held in Austin, Texas, on September 21–23, 2015. The first article of this special issue, “Formal Requirement Elicitation and Debugging for Testing and Verification of Cyber-Physical Systems” by Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos, describes a framework for eliciting and debugging specifications written in the Metric Interval Temporal Logic (MITL). To represent the required system behavior, the developer uses a graphical interface in the ViSpec tool, which then converts the system input into MITL. The article presents methods for performing several checks on MITL specifications, including validity checks that determine if a specification is satisfiable and not a tautology, redundancy checks to identify redundant conjuncts in a specification, and vacuity checks to identify subformulas of the specification irrelevant to its satisfaction. It also presents an extension of the vacuity checking results to specifications in Signal Temporal Logic and experimental results showing the utility of these checks on a suite of example specifications.

The second article, “Symbolic Multi-Level Loop Mapping of Loop Programs for Massively Parallel Processor Arrays” by Alexandru Tanase, Michael Witterauf, Jürgen Teich, and Frank Hannig, presents a multi-level symbolic parallelization technique for mapping loop programs to massively parallel processor arrays. The method is a hybrid compile time–runtime technique, which, unlike many existing techniques, is not I/O or memory bound and provides results even when the number of processing elements is unknown until runtime. At compile time, a symbolic multi-level tiling is performed to create different levels of hierarchical tiles, followed by scheduling to produce a

2018 Copyright is held by the owner/author(s).

1539-9087/2018/01-ART30

<https://doi.org/10.1145/3162079>

set of feasible, latency-optimal symbolic loop schedule candidates. At runtime, based on the number of processing elements as well I/O and memory constraints, one of the symbolic scheduling candidates is selected for instantiation. The article presents experimental results showing that the approach quickly produces a latency-optimal schedule for real examples.

The third article, “Model and Program Repair via SAT Solving” by Paul Attie, Kinan Dak Al Bab, and Mouhammad Sakr, describes how a model, specified as a finite Kripke structure, can be repaired to satisfy a given CTL formula using a SAT solver. The repair method is subtractive: The repaired model is constructed by deleting transitions and/or states from the original model. The results are extended to abstraction mappings in which a more abstract version of the Kripke structure is repaired. Then, the repair is concretized to produce a repair of the original Kripke structure. The article also presents additional extensions for repairing hierarchical Kripke structures in which the repair is done level by level, and repairing concurrent Kripke structures in which repair is done pairwise for interacting processes. The repair methods are implemented in a tool called Eshmun, and experimental results are presented showing that the methods avoid state explosion.

The final article, “SCEst: Sequentially Constructive Esterel,” is authored by Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. The synchronous language Esterel ensures determinacy by requiring adherence to the “signal coherence rule.” Adherence to this rule ensures that signals have stable values throughout each reaction cycle (or tick). A disadvantage of this approach is that such a requirement can be overly restrictive in software development. This article describes Sequentially Constructive Esterel (SCEst), in which variables are allowed to change value within a tick as long as determinacy is still guaranteed. This relaxation of the Esterel rules allows more programs to have valid semantics than in standard Esterel. To describe the Sequentially Constructive model of computation used in SCEst and to define the semantics of SCEst, the authors describe a set of transformation rules for translating SCEst specifications into the Sequentially Constructive Language (SCL). The article also describes tool support for translating programs in SCEst to C and early experimental results using this tool support.

Elizabeth Leonard
Constance Heitmeyer
Guest Editors