Oleg Sokolsky
Serdar Taşıran (Eds.)

LNCS 4839

Runtime Verification

7th International Workshop, RV 2007
Vancover, Canada, March 2007
Revised Selected Papers

Springer

Oleg Sokolsky Serdar Taşıran (Eds.)

Runtime Verification

7th International Workshop, RV 2007
Vancover, Canada, March 13, 2007
Revised Selected Papers

Volume Editors

Oleg Sokolsky
University of Pennsylvania
Department of Computer and Information Science
3330 Walnut Street, Philadelphia, PA, USA
E-mail: sokolsky@cis.upenn.edu

Serdar Taşıran
Koç University
College of Engineering
Rumeli Feneri Yolu, Sariyer, 34450, Istanbul, Turkey
E-mail: stasiran@ku.edu.tr

Library of Congress Control Number: 2007941510

CR Subject Classification (1998): D.2, D.3, F.3, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-77394-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-77394-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12208111 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4839

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Preface

Runtime verification is a recent direction in formal methods research, which is complementary to such well-established formal verification methods as model checking. Research in runtime verification deals with formal languages suitable for expressing system properties that are checkable at run time; algorithms for checking of formal properties over an execution trace; low-overhead means of extracting information from the running system that is sufficient for checking of the property. Applications of runtime verification technology include post-deployment monitoring of system correctness and performance; construction of formally specified test oracles; collection of statistics about system behavior, among others.

The Workshop on Runtime Verification was started in 2001 and has been held annually since then. The workshop was co-located with the Conference on Computer-Aided Verification (CAV) in 2001–2003 and 2005–2006; and with the European Joint Conferences on Theory and Practice of Software (ETAPS) in 2004. In 2007, the workshop was held on March 13, 2007 in Vancouver, British Columbia, Canada, co-located to the Conference on Aspect-Oriented Software Development (AOSD) in order to explore the emerging connections between the two communities.

RV 2007 attracted contributions from the core area of runtime verification, as well as related research areas such as testing, static and dynamic analysis of programs, and aspect-oriented programming. The Program Committee selected 16 out of 29 submissions. Each submitted paper was reviewed by at least three Program Committee members. Submitted papers were supplemented by an invited talk given by Cindy Eisner (IBM Research Haifa). This volume contains expanded versions of the presentations made at the workshop. The expanded versions were again reviewed by the Program Committee.

September 2007

Oleg Sokolsky
Serdar Tasiran

Conference Organization

Program Committee

Mehmet Aksit, University of Twente, The Netherlands
Howard Barringer, University of Manchester, UK
Saddek Bensalem, VERIMAG Laboratory, France
Eric Bodden, McGill Univeristy, Canada
Bernd Finkbeiner, Saarland University, Germany
Cormac Flanagan, University of California, Santa Cruz, USA
Vijay Garg, University of Texas, Austin, USA
Klaus Havelund, NASA Jet Propulsion Laboratory/Columbus Technologies, USA
Gerard Holzmann, NASA Jet Propulsion Laboratory, USA
Moonzoo Kim, KAIST, Korea
Martin Leucker, Technical University of Munich, Germany
Oege de Moor, Oxford University, UK
Klaus Ostermann, Darmstadt University of Technology, Germany
Shaz Qadeer, Microsoft Research
Grigore Rosu, University of Illinois, Urbana-Champaign, USA
Henny Sipma, Stanford University, USA
Oleg Sokolsky (Co-chair), University of Pennsylvania, USA
Scott Stoller, State University of New York, Stony Brook, USA
Mario Südholt, Ecole des Mines de Nantes-INRIA, France
Serdar Tasiran (Co-chair), Koc University, Turkey

Steering Committee

Klaus Havelund, NASA Jet Propulsion Laboratory, USA
Gerard Holzmann, NASA Jet Propulsion Laboratory, USA
Insup Lee, University of Pennsylvania, USA
Grigore Rosu, University of Illinois, Urbana-Champaign, USA

External Reviewers

Andreas Bauer
Selma Ikiz
David Rydeheard
Christian Schallhart

Lecture Notes in Computer Science

Sublibrary 2: Programming and Software Engineering

For information about Vols. 1– 4218
please contact your bookseller or Springer

Vol. 4849: M. Winckler, H. Johnson, P. Palanque (Eds.), Task Models and Diagrams for User Interface Design. XIII, 299 pages. 2007.

Vol. 4839: O. Sokolsky, S. Taşıran (Eds.), Runtime Verification. VIII, 215 pages. 2007.

Vol. 4834: R. Cerqueira, R.H. Campbell (Eds.), Middleware 2007. XIII, 451 pages. 2007.

Vol. 4829: M. Lumpe, W. Vanderperren (Eds.), Software Composition. VIII, 281 pages. 2007.

Vol. 4824: A. Paschke, Y. Biletskiy (Eds.), Advances in Rule Interchange and Applications. XIII, 243 pages. 2007.

Vol. 4807: Z. Shao (Ed.), Programming Languages and Systems. XI, 431 pages. 2007.

Vol. 4799: A. Holzinger (Ed.), HCI and Usability for Medicine and Health Care. XVI, 458 pages. 2007.

Vol. 4789: M. Butler, M.G. Hinchey, M.M. Larrondo-Petrie (Eds.), Formal Methods and Software Engineering. VIII, 387 pages. 2007.

Vol. 4767: F. Arbab, M. Sirjani (Eds.), International Symposium on Fundamentals of Software Engineering. XIII, 450 pages. 2007.

Vol. 4764: P. Abrahamsson, N. Baddoo, T. Margaria, R. Messnarz (Eds.), Software Process Improvement. XI, 225 pages. 2007.

Vol. 4762: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Automated Technology for Verification and Analysis. XIV, 566 pages. 2007.

Vol. 4758: F. Oquendo (Ed.), Software Architecture. XVI, 340 pages. 2007.

Vol. 4757: F. Cappello, T. Herault, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. XVI, 396 pages. 2007.

Vol. 4753: E. Duval, R. Klamma, M. Wolpers (Eds.), Creating New Learning Experiences on a Global Scale. XII, 518 pages. 2007.

Vol. 4749: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Service-Oriented Computing – ICSOC 2007. XIX, 629 pages. 2007.

Vol. 4748: K. Wolter (Ed.), Formal Methods and Stochastic Models for Performance Evaluation. X, 301 pages. 2007.

Vol. 4741: C. Bessière (Ed.), Principles and Practice of Constraint Programming – CP 2007. XV, 890 pages. 2007.

Vol. 4735: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems. XV, 698 pages. 2007.

Vol. 4716: B. Meyer, M. Joseph (Eds.), Software Engineering Approaches for Offshore and Outsourced Development. X, 201 pages. 2007.

Vol. 4680: F. Saglietti, N. Oster (Eds.), Computer Safety, Reliability, and Security. XV, 548 pages. 2007.

Vol. 4670: V. Dahl, I. Niemelä (Eds.), Logic Programming. XII, 470 pages. 2007.

Vol. 4652: D. Georgakopoulos, N. Ritter, B. Benatallah, C. Zirpins, G. Feuerlicht, M. Schoenherr, H.R. Motahari-Nezhad (Eds.), Service-Oriented Computing ICSOC 2006. XVI, 201 pages. 2007.

Vol. 4640: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development IV. IX, 191 pages. 2007.

Vol. 4634: H. Riis Nielson, G. Filé (Eds.), Static Analysis. XI, 469 pages. 2007.

Vol. 4620: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development III. IX, 201 pages. 2007.

Vol. 4615: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems IV. XIV, 435 pages. 2007.

Vol. 4610: B. Xiao, L.T. Yang, J. Ma, C. Muller-Schloer, Y. Hua (Eds.), Autonomic and Trusted Computing. XVIII, 571 pages. 2007.

Vol. 4609: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming. XIII, 625 pages. 2007.

Vol. 4608: H.W. Schmidt, I. Crnković, G.T. Heineman, J.A. Stafford (Eds.), Component-Based Software Engineering. XII, 283 pages. 2007.

Vol. 4591: J. Davies, J. Gibbons (Eds.), Integrated Formal Methods. IX, 660 pages. 2007.

Vol. 4589: J. Münch, P. Abrahamsson (Eds.), Product-Focused Software Process Improvement. XII, 414 pages. 2007.

Vol. 4574: J. Derrick, J. Vain (Eds.), Formal Techniques for Networked and Distributed Systems – FORTE 2007. XI, 375 pages. 2007.

Vol. 4556: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part III. XXII, 1020 pages. 2007.

Vol. 4555: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part II. XXII, 1066 pages. 2007.

Vol. 4554: C. Stephanidis (Ed.), Universal Acess in Human Computer Interaction, Part I. XXII, 1054 pages. 2007.

Vol. 4553: J.A. Jacko (Ed.), Human-Computer Interaction, Part IV. XXIV, 1225 pages. 2007.

Vol. 4552: J.A. Jacko (Ed.), Human-Computer Interaction, Part III. XXI, 1038 pages. 2007.

Vol. 4551: J.A. Jacko (Ed.), Human-Computer Interaction, Part II. XXIII, 1253 pages. 2007.

Vol. 4550: J.A. Jacko (Ed.), Human-Computer Interaction, Part I. XXIII, 1240 pages. 2007.

Vol. 4542: P. Sawyer, B. Paech, P. Heymans (Eds.), Requirements Engineering: Foundation for Software Quality. IX, 384 pages. 2007.

Vol. 4536: G. Concas, E. Damiani, M. Scotto, G. Succi (Eds.), Agile Processes in Software Engineering and Extreme Programming. XV, 276 pages. 2007.

Vol. 4530: D.H. Akehurst, R. Vogel, R.F. Paige (Eds.), Model Driven Architecture - Foundations and Applications. X, 219 pages. 2007.

Vol. 4523: Y.-H. Lee, H.-N. Kim, J. Kim, Y.W. Park, L.T. Yang, S.W. Kim (Eds.), Embedded Software and Systems. XIX, 829 pages. 2007.

Vol. 4498: N. Abdennahder, F. Kordon (Eds.), Reliable Software Technologies - Ada-Europe 2007. XII, 247 pages. 2007.

Vol. 4486: M. Bernardo, J. Hillston (Eds.), Formal Methods for Performance Evaluation. VII, 469 pages. 2007.

Vol. 4470: Q. Wang, D. Pfahl, D.M. Raffo (Eds.), Software Process Dynamics and Agility. XI, 346 pages. 2007.

Vol. 4468: M.M. Bonsangue, E.B. Johnsen (Eds.), Formal Methods for Open Object-Based Distributed Systems. X, 317 pages. 2007.

Vol. 4467: A.L. Murphy, J. Vitek (Eds.), Coordination Models and Languages. X, 325 pages. 2007.

Vol. 4454: Y. Gurevich, B. Meyer (Eds.), Tests and Proofs. IX, 217 pages. 2007.

Vol. 4444: T. Reps, M. Sagiv, J. Bauer (Eds.), Program Analysis and Compilation, Theory and Practice. X, 361 pages. 2007.

Vol. 4440: B. Liblit, Cooperative Bug Isolation. XV, 101 pages. 2007.

Vol. 4408: R. Choren, A. Garcia, H. Giese, H.-f. Leung, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems V. XII, 233 pages. 2007.

Vol. 4406: W. De Meuter (Ed.), Advances in Smalltalk. VII, 157 pages. 2007.

Vol. 4405: L. Padgham, F. Zambonelli (Eds.), Agent-Oriented Software Engineering VII. XII, 225 pages. 2007.

Vol. 4401: N. Guelfi, D. Buchs (Eds.), Rapid Integration of Software Engineering Techniques. IX, 177 pages. 2007.

Vol. 4385: K. Coninx, K. Luyten, K.A. Schneider (Eds.), Task Models and Diagrams for Users Interface Design. XI, 355 pages. 2007.

Vol. 4383: E. Bin, A. Ziv, S. Ur (Eds.), Hardware and Software, Verification and Testing. XII, 235 pages. 2007.

Vol. 4379: M. Südholt, C. Consel (Eds.), Object-Oriented Technology. VIII, 157 pages. 2007.

Vol. 4364: T. Kühne (Ed.), Models in Software Engineering. XI, 332 pages. 2007.

Vol. 4355: J. Julliand, O. Kouchnarenko (Eds.), B 2007: Formal Specification and Development in B. XIII, 293 pages. 2006.

Vol. 4354: M. Hanus (Ed.), Practical Aspects of Declarative Languages. X, 335 pages. 2006.

Vol. 4350: M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework. XXII, 797 pages. 2007.

Vol. 4348: S. Tucker Taft, R.A. Duff, R.L. Brukardt, E. Plödereder, P. Leroy, Ada 2005 Reference Manual. XXII, 765 pages. 2006.

Vol. 4346: L. Brim, B.R. Haverkort, M. Leucker, J. van de Pol (Eds.), Formal Methods: Applications and Technology. X, 363 pages. 2007.

Vol. 4344: V. Gruhn, F. Oquendo (Eds.), Software Architecture. X, 245 pages. 2006.

Vol. 4340: R. Prodan, T. Fahringer, Grid Computing. XXIII, 317 pages. 2007.

Vol. 4336: V.R. Basili, H.D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, R.W. Selby (Eds.), Empirical Software Engineering Issues. XVII, 193 pages. 2007.

Vol. 4326: S. Göbel, R. Malkewitz, I. Iurgel (Eds.), Technologies for Interactive Digital Storytelling and Entertainment. X, 384 pages. 2006.

Vol. 4323: G. Doherty, A. Blandford (Eds.), Interactive Systems. XI, 269 pages. 2007.

Vol. 4322: F. Kordon, J. Sztipanovits (Eds.), Reliable Systems on Unreliable Networked Platforms. XIV, 317 pages. 2007.

Vol. 4309: P. Inverardi, M. Jazayeri (Eds.), Software Engineering Education in the Modern Age. VIII, 207 pages. 2006.

Vol. 4294: A. Dan, W. Lamersdorf (Eds.), Service-Oriented Computing – ICSOC 2006. XIX, 653 pages. 2006.

Vol. 4290: M. van Steen, M. Henning (Eds.), Middleware 2006. XIII, 425 pages. 2006.

Vol. 4279: N. Kobayashi (Ed.), Programming Languages and Systems. XI, 423 pages. 2006.

Vol. 4262: K. Havelund, M. Núñez, G. Roşu, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification. VIII, 255 pages. 2006.

Vol. 4260: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering. XII, 778 pages. 2006.

Vol. 4257: I. Richardson, P. Runeson, R. Messnarz (Eds.), Software Process Improvement. XI, 219 pages. 2006.

Vol. 4242: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development II. IX, 289 pages. 2006.

Vol. 4229: E. Najm, J.-F. Pradat-Peyre, V.V. Donzeau-Gouge (Eds.), Formal Techniques for Networked and Distributed Systems - FORTE 2006. X, 486 pages. 2006.

Vol. 4227: W. Nejdl, K. Tochtermann (Eds.), Innovative Approaches for Learning and Knowledge Sharing. XVII, 721 pages. 2006.

Table of Contents

Invited Paper

AOP-Related Papers

Core Runtime Verification Papers

PSL for Runtime Verification: Theory and Practice

Cindy Eisner

IBM Haifa Research Laboratory
eisner@il.ibm.com

Abstract. PSL is a property specification language recently standardized as IEEE 1850™-2005 PSL. It includes as its *temporal layer* a linear temporal logic that enhances LTL with regular expressions and other useful features. PSL and its precursor, Sugar, have been used by the IBM Haifa Research Laboratory for formal verification of hardware since 1993, and for informal (dynamic, simulation runtime) verification of hardware since 1997. More recently both Sugar and PSL have been used for formal, dynamic, and runtime verification of software. In this paper I will introduce PSL and briefly touch on theoretical and practical issues in the use of PSL for dynamic and runtime verification.

1 Introduction

PSL stands for Property Specification Language. Its temporal layer is a linear temporal logic that enhances LTL [19] with regular expressions and other useful features. PSL originated as the branching temporal logic Sugar at the IBM Haifa Research Laboratory, and in October 2005 was standardized as IEEE 1850-2005 (PSL).

PSL has four layers: the Boolean, the modeling, the temporal, and the verification layers. The *Boolean layer* is used to define Boolean expressions. For instance, `a & b` is a Boolean expression (in the Verilog flavor) indicating the conjunction of `a` and `b`. The Boolean layer comes in five flavors, corresponding to the hardware description languages VHDL, Verilog, SystemVerilog and SystemC and to GDL, the language of IBM's RuleBase model checker. Although other flavors are not yet an official part of the language, it is very easy to define new ones. See, for instance, [7], which describes a C flavor of PSL.

The flavor affects the syntax of the *modeling layer* as well, which is used to describe the environment of the design under test. For instance, constraints on the inputs would be described in the modeling layer. The modeling layer can also be used to describe auxiliary signals (in software: variables) that are not part of the design, but are used as part of the verification. For example, the modeling layer statement `assign a = b & c;` lets the signal name `a` be used in place of the Boolean expression `b & c`.

The *temporal layer* is the heart of the language, and consists of an LTL-based temporal logic incorporating regular expressions. A formula over this temporal

O. Sokolsky and S. Tasiran (Eds.): RV 2007, LNCS 4839, pp. 1–8, 2007.

logic is called a PSL *property*. For example, `always(req -> eventually! ack)` is a PSL property saying that whenever `req` is asserted, `ack` should be asserted sometime in the future, and `always {req ; ack ; grant} |=> {busy[*] ; done}` is a PSL property that says that whenever `req` is asserted followed by `ack` and then by `grant`, `busy` should be asserted and stay so until `done` occurs. The temporal layer also allows an elementary form of quantification, so that the property `forall i in {0:7}: always ((req & tag==i) -> eventually! (ack & tag==i))` says that whenever `req` is asserted, eventually an associated `ack` will occur, where the association is indicated by a matching value of `tag`. Other features include a clock operator that can be used to change the default view of time, and the `abort` operator, described in Section 3 below.

The *verification layer* contains directives that tell the verification tool what to do with a PSL property: e.g., should it be asserted (checked), or should it be assumed, or perhaps used as the basis for coverage measurement? The verification layer also provides a way to group sets of directives into a `vunit`, or verification unit, which can be referred to by name in the verification tool.

PSL is good for hardware verification, and various tools for both formal and dynamic hardware verification using PSL are available from companies such as IBM, Cadence, Mentor graphics, etc. PSL is also good for software verification, and PSL or its precursor, Sugar, has been used internally at IBM for software model checking [4][10][11], as well as within a `C++` based simulation environment [9]. More recently, it has also been used externally for runtime verification of software [7].

Intuitively, dynamic and runtime verification have a linear view of time. In the remainder of this paper, I will explain why the move from branching time Sugar to linear time PSL, a big deal in theory, was not a problem in practice and required no modification to our runtime simulation checker generator FoCs (nor to our model checker RuleBase). I will present the truncated semantics that were developed to support non-maximal finite paths as seen in dynamic and runtime verification, and show how they are related to the support of resets in a reactive system, and finally I will discuss the FoCs approach to the issue of how time "ticks" in software.

2 Masking Branching vs. Linear Time

In branching time logics such as CTL [8] and PSL's precursor, Sugar, time is *branching*. That is, the semantics are given with respect to a state in the model, and every possible future of that state is considered. In linear time logics such as LTL [19] and PSL, time is *linear*. That is, the semantics are given with respect to set of ordered states (a path) in the model, and thus every state has a single successor. In theory, this is a very big deal. The complexity of branching time model checking is better than that of linear time model checking [21], the expressive power of the two is incomparable [17], and of course, only linear time makes sense for dynamic and runtime verification.

In practice, however, the issue is not such an important one. The overlap between linear and branching time is a large one, and the vast majority of properties used in practice belong to the overlap. Furthermore, there is a simple syntactic test that can be used to confirm that a syntactically similar CTL/LTL formula pair is equivalent [17]. As an example, the test confirms that the CTL formula $AG(p \rightarrow AXq)$ is equivalent to the LTL formula $G(p \rightarrow Xq)$. The test does not work for every equivalent pair; for example, it does not confirm that the CTL formula $AG(\neg p \rightarrow AX\neg q)$ is equivalent to the LTL formula $G((Xq) \rightarrow p)$, even though the pair are equivalent. However, it works in enough cases to make it practically useful: for instance, the *simple subset* of PSL [5][12] obeys the test.

For this reason, the move from the original CTL-based semantics of Sugar to the current, LTL-based semantics of PSL was not a major issue in practice, neither for IBM's model checker RuleBase [20] nor for its dynamic verification tool FoCs [1]. In both cases, the move is masked by the *Sugar compiler*. For RuleBase, it checks whether a (linear) PSL formula passes the syntactic test of [17] and if so, uses the established (branching) algorithms. For FoCs, the tool has always used a syntactic test similar to that of [17] to weed out branching formulas that cannot be checked dynamically, and the same test weeds out linear formulas for which the dynamic checking is not trivial.

3 Finite Paths and the Truncated Semantics

In the sequel, I will use PSL syntax corresponding to the basic LTL operators, as follows: `always` is equivalent to the LTL operator G, and the PSL operators `eventually!`, `until`, `until!`, `next` and `next!` correspond to the LTL operators F, W, U, X and $X!$, respectively.

Traditionally, LTL semantics over finite paths [18] are defined for maximal paths in the model. That is, if we evaluate a formula over a finite path under traditional LTL finite semantics, it is because the last state of the path has no successor in the model. For a long time, finite paths on non-maximal paths were treated in an ad-hoc manner – see [6], for instance. In [14], we considered in detail the problem of reasoning with temporal logic on non-maximal paths.

A *truncated path* is a finite, not necessarily maximal path. Truncated paths are seen by incomplete formal methods, such as bounded model checking, and also by dynamic and runtime verification (at any point before the program ends we have seen a partial, non-maximal path). In the *truncated semantics*, there are three *views* of a finite path. The *weak* view takes a lenient view of truncated paths – a property holds even if there is doubt about the status of the property on the full path. The *strong* view is a strict view of truncated paths – a property does not hold if there is doubt about the status of the property on the full path. The *neutral* view of a truncated path is simply the traditional semantics for a maximal path.

For example, on a finite path such that p holds at every state on the path, the property `always p` might or might not hold on the full path: if it turns out that the truncated path continues with a p at every state, our property will hold

on the full path, but if there is even one future state with no p, it will not hold. Thus, the property holds in the weak view, and does not hold in the strong view. It holds in the neutral view, because `always p` holds on our path if we consider it to be maximal.

As another example, consider a finite path such that q holds at no state on the path. The property `eventually! q` might or might not hold on the full path: if there is a future q, the property holds, otherwise it does not. Thus, `eventually! q` holds in the weak view, and does not hold in the strong view. It does not hold in the neutral view, because `eventually! q` does not hold on such a path if we consider it to be maximal. If q does hold for some state on the path, then the property holds in the neutral view, and there is no doubt that it will hold as well on any continuation of the path. Thus, on such a path the property holds in the weak, neutral and strong views.

Consider now a finite path on which p holds at states 2, 4 and 20, and q holds at state 15. As with our previous examples, the property `always (p -> eventually! q)` might or might not hold on the full path, depending on how the truncated path continues. Thus the property holds in the weak view and does not hold in the strong view. It does not hold in the neutral view, because `always (p -> eventually! q)` does not hold on our path if we consider it to be maximal – the p that holds at state 20 is missing a q. Even if there were such a future q, for instance if p held at states 2, 4 and 20 and q held at states 15 and 25 – then there still would be doubt about whether the property holds on the full path, because there might be a future p that does not see an appropriate q. Thus, our property would still hold in the weak view and not hold in the strong view. However, it does hold in the neutral view on our new path, because the neutral semantics do not worry about possible futures – they consider the path to be maximal.

The weak view can be understood as a weakening of all operators (assuming negation-normal form) [13], and the strong view can be understood as a strengthening of all operators (under the same assumption). Thus, it is easy to see that `eventually!` φ holds weakly on any path for any φ (including *false*): `eventually!` φ is equivalent to `true until!` φ. Weakening this gives `true until` φ, which holds on any path for any φ. Similarly, we can show that `always` φ does not hold strongly on any path for any φ (including *true*), because `always` φ is equivalent to φ `until false`. Strengthening that gives φ `until! false`, which holds on no path for no φ.

In practice, very few formulas hold strongly on any path, because most formulas begin with the `always` operator. Thus, the weak view of truncated paths is most useful in practice. However, the strong view is dual to the weak, and giving it up would result in a logic not closed under negation.

On an infinite path, the weak, neutral and strong views coincide [14]. Nevertheless, the truncated semantics can be useful in the context of infinite paths, because an infinite path may contain finite, non-maximal segments. They can be useful in the context of finite maximal paths for the same reason. For instance,

a hardware reset or a software event such as "clear form", "start over", or "new query" may partition a path into two parts: a finite, truncated part until the reset or software event, and a possibly infinite, possibly maximal part (depending on whether or not the original path was maximal) afterwards. The PSL `abort` operator truncates a path and moves to the weak view. Thus, the property `always (`φ `abort reset)` partitions the path into segments at every occurrence of `reset` (discarding the states on which `reset` occurs). The property φ must hold neutrally on the final segment, and weakly on the remaining segments (each of which was followed in the original path by a state on which `reset` held).

The point of the `abort` operator can be best appreciated by comparing it to the `until` operator. Both of the following properties:

$$(\texttt{always (p -> eventually! q))}\ \texttt{abort reset} \tag{1}$$

$$(\texttt{p -> eventually! q})\ \texttt{until reset} \tag{2}$$

need `(p -> eventually! q)` to hold up until `reset` holds. However, they differ with respect to what happens at that point. Consider a path π of length 20 such that `p` holds at states 2 and 10, `q` holds at state 4, and `reset` holds at state 15. Property 1 holds on such a path, because the `abort` operator truncates the path at the occurrence of `reset` and takes us to the weak view. Since the sub-property `(always (p -> eventually! q))` holds weakly on the truncated path, Property 1 holds on the original path. However, Property 2 does not hold on path π, because sub-property `(p -> eventually! q)` does not hold at state 10.

The behavior of the `abort` operator is very easy to describe in an informal manner, but very difficult to formalize. Our first try, the *abort semantics* [14], defined simply

$$\begin{aligned} w \models \varphi \text{ abort } b \iff & \\ & \text{either } w \models \varphi \text{ or} \\ & \text{there exist } j < |w| \text{ and word } w' \text{ such that } w^j \models b \text{ and } w^{0..j-1}w' \models \varphi \end{aligned}$$

This looks intuitive, but turns out not to be what we wanted. Consider the property `(eventually! false) abort b` on a path where `b` occurs at some point. We want the property to hold on such a path, because we want `eventually! false` to hold in the weak view on a finite path. To see this, recall that `eventually! false` is equivalent to `true until! false`. If we weaken the `until!` operator we get `true until false` which holds on any path. However, looking back to the proposed semantics, there is no w' we can choose that will give us what we want. Others [3] were more successful, but ended up with a semantics that required intricate manipulations of two contexts within the semantics of the existing LTL operators.

We have since presented two simple and elegant formulations. The original truncated semantics, presented in [14], directly defines semantics for each of the three views (weak, neutral and strong). The result is a semantics that is

equivalent to the *reset semantics* of [3], but whose presentation is much cleaner and easier to grasp. The $\top$, $\bot$ approach to the truncated semantics, presented in [15], takes another tack, and folds the three views into an equivalent but more compact representation. It does so by adding two new letters, $\top$ and $\bot$, to the alphabet, such that everything holds on $\top$, including `false`, and nothing holds on $\bot$, including `true`. With these two new letters, the original formulation of the semantics presented above works because we can choose a w' consisting entirely of the letter $\top$. While the $\top$, $\bot$ approach uses a slightly more cryptic formulation than the original truncated semantics, we have found it useful in characterizing the relation between the weak and strong views, as described in [13].

4 The FoCs Approach to the Ticking of Time

PSL does not dictate how time ticks. The formal semantics (see for instance Appendix B of [12]) is based on a sequence of states, but how those states are derived from the hardware or software under verification is not defined. This is good news for software, because it means that the formal semantics can be used as is. However, it does not provide any practical answers.

FoCs is a tool that takes PSL properties and translates them into monitors that allow the use of PSL in event-based software. Originally, FoCs was designed for hardware simulations [1], but it can work with other event-based software as well. In the FoCs approach, the responsibility for time belongs to the application. If the user has embedded a PSL property in C (or other) code, FoCs will translate the property into a state machine embedded in the code at the location where the property originally appeared. Then, time is considered to have ticked when the state machine is reached at runtime. The FoCs approach is a *generic* solution for any language, but of course it is not a *general* solution for any application, in the case that some other definition of the ticking of time is desired.

Acknowledgements

PSL was and continues to be the work of many people. I would particularly like to acknowledge my IBM colleagues Ilan Beer, Shoham Ben-David, Dana Fisman and Avner Landver for their early work on Sugar, and Dana Fisman, Avigail Orni, Dmitry Pidan and Sitvanit Ruah for more recent work on PSL.

The members of the Accellera FVTC (Formal Verification Technical Committee) and the IEEE P1850 PSL Working Group are too numerous to mention by name – a complete list can be found in the respective standards [2] [16] – but I would particularly like to thank Harry Foster and Erich Marschner, chairman and co-chairman of the FVTC and chairman and secretary of the IEEE P1850 Working Group, for leading the process that led to standardization.

The work described in Section 3 was joint work with Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, Johan Mårtensson and David Van Campenhout (in various combinations).

Thank you to Dmitry Pidan for his explanation of the FoCs approach to the ticking of time.

References

1. Abarbanel, Y., Beer, I., Gluhovsky, L., Keidar, S., Wolfsthal, Y.: FoCs - automatic generation of simulation checkers from formal specifications. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
2. Accellera property specification language reference manual, `http://www.eda.org/vfv/docs/psl_lrm-1.1.pdf`
3. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Aborts vs resets in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, Springer, Heidelberg (2003)
4. Barner, S., Glazberg, Z., Rabinovitz, I.: Wolf - bug hunter for concurrent software using formal methods. In: CAV, pp. 153–157 (2005)
5. Ben-David, S., Fisman, D., Ruah, S.: The safety simple subset. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) First International Haifa Verification Conference. LNCS, vol. 3875, pp. 14–29. Springer, Heidelberg (2005)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, Springer, Heidelberg (1999)
7. Cheung, P.H., Forin, A.: A C-language binding for PSL. In: Technical Report MSR-TR-2006-131, Microsoft Research (2006)
8. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
9. Dahan, A., Geist, D., Gluhovsky, L., Pidan, D., Shapir, G., Wolfsthal, Y., Benalycherif, L., Kamdem, R., Lahbib, Y.: Combining system level modeling with assertion based verification. In: ISQED, pp. 310–315 (2005)
10. Eisner, C.: Model checking the garbage collection mechanism of SMV. In: Stoller, S.D., Visser, W. (eds.) Electronic Notes in Theoretical Computer Science, vol. 55, Elsevier, Amsterdam (2001)
11. Eisner, C.: Formal verification of software source code through semi-automatic modeling. Software and Systems Modeling 4(1), 14–31 (2005)
12. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
13. Eisner, C., Fisman, D., Havlicek, J.: A topological characterization of weakness. In: Proc. 24th Annual ACM Symposium on Principles of Distributed Com puting (PODC), pp. 1–8 (2005)
14. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
15. Eisner, C., Fisman, D., Havlicek, J., Mårtensson, J.: The $\top$, $\bot$ approach for truncated semantics. Technical Report 2006.01, Accellera (January 2006)
16. IEEE standard for property specification language (PSL). IEEE Std 1850-2005