9th Workshop PSSV

Proceedings

Nikolay Shilov, Vladimir Zakharov (Eds.)

9th Workshop “Program Semantics,
Specification and Verification: Theory and Applications”
dedicated to the memory
of B. A. Trakhtenbrot, M. I. Dekhtyar, and M. K. Valiev

(Yaroslavl, Russia, June 21-22, 2018)

Yaroslavl
Yaroslavl State University
2018

UDC 519.68

9th Workshop PSSV: proceedings / Edited by Nikolay Shilov, Vladimir Zakharov. – Yaroslavl: Yaroslavl State University, 2018. – 98 pages. – (9th Workshop "Program Semantics, Specification and Verification: Theory and Applications" dedicated to the memory of B. A. Trakhtenbrot, M. I. Dekhtyar, and M. K. Valiev, Yaroslavl, Russia, June 21-22, 2018).

ISBN 978-5-8397-1149-5

Workshop on Program Semantics, Specification and Verification: Theory and Applications is the leading event in Russia in the field of applying of the formal methods to software analysis. Proceedings of the ninth workshop dedicated to formalisms for program semantics, formal models and verification, programming and specification languages, algebraic and logical aspects of programming.

ISBN 978-5-8397-1149-5

Preface

The volume contains the papers selected for presentation at the IX International Workshop on *Program Semantics, Specification and Verification: Theory and Applications* (PSSV-2018). The Workshop took place on June 21-22, 2018 in Yaroslavl, Russia. PSSV Workshops were successfully held in Kazan (2010), St. Petersburg (2011, 2016), Nizhni Novgorod (2012), Yekaterinburg (2013), Moscow(2014, 2017), Kazan (2015). In 2010–14 and 2016 PSSV Workshops were affiliated with the International Symposiums *Computer Science in Russia* (CSR); in 2015 and 2017 it was affiliated with the International Conference *Perspectives of System Informatics* (CSR).

The topics of the Workshop include formal models of programs and systems, methods of formal semantics of programming languages, formal specification languages, methods of deductive program verification, model checking method, static analysis of programs, formal approaches to testing and validation, program testing, analysis and verification tools.

In 2018 the Workshop is dedicated to the memory of B. A. Trakhtenbrot (1921–2016), M. I. Dekhtyar (1946–2018), and M. K. Valiev (1942–2018). Three memorial papers which survey the substantial contribution in computer science made by these remarkable mathematicians are included in these Proceedings. Thirteen research papers have been submitted to PSSV 2018. Program Committee accepted 5 papers as regular ones, 2 as short presentations, and 3 more papers — for the poster session. Abstracts of 3 invited talks are also included in these Proceedings.

Nikolay Shilov, Vladimir Zakharov,
June 2018

Organization

Program Chairs

Vladimir Zakharov	Lomonosov Moscow State University, Russia
Nikolay Shilov	Innopolis University, Innopolis, Russia

Steering Committee

Valery Nepomniaschy	Institute of Informatics Systems, Novosibirsk, Russia
Valery Sokolov	Demidov Yaroslavl State University, Russia

Program Committee

Natasha Alechina	University of Nottingham, UK
Alexander Bolotov	University of Westminster, UK
Igor Konnov	INRIA Nancy (LORIA), France
Victor Kuliamin	Institute for System Programming, Moscow, Russia
Egor Kuzmin	Demidov Yaroslavl State University, Russia
Alexei Lisitsa	University of Liverpool, UK
Irina Lomazova	Higher School of Economics, Moscow, Russia
Manuel Mazzara	Innopolis University, Innopolis, Russia
Alexander Okhotin	St. Petersburg State Universit, Russia
Nina Yevtushenko	Tomsk State University and Institute for System Programming, Moscow, Russia

Organizing Committee

Egor Kuzmin	Demidov Yaroslavl State University, Russia
Valery Sokolov	Demidov Yaroslavl State University, Russia

Invited Speakers

Daniel de Carvalho	Innopolis University, Russia
Dmitry Vlasov	Sobolev Institute of Mathematics, Novosibirsk, Russia
Nina Yevtushenko	Tomsk State University and Institute for System Programming, Moscow, Russia

Table of Contents

In memory of Boris Trakhtenbrot, Mars Valiev and Michael Dekhtyar

Vladimir Sazonov

The University of Liverpool

1 Boris (Boaz) Trakhtenbrot (20.02.1921–19.09.2016)

The scientific history of Boris Abramovich Trakhtenbrot has already been well described in [1,2,4]. So, I would like to focus on some aspects that seem important from the humanistic point of view, touching on other things only very slightly.

Moldova, WW2, Chernovtsy. Boris Abramovich Trakhtenbrot was born in Brichevo village in Northern Bessarabia (now Moldova). In 1940 he started studying mathematics in the Moldavian Pedagogical Institute in Kishinev, then evacuation. . . , returning back. . . , then enrolling in the University of Chernovtsy (Ukraine) in 1945 and getting a master's degree in 1947.

This was also a happy time because he married Berta Isaakovna Rabinovich (1921–2013). Later, many of us enjoyed the warm hospitality of this family, particularly of truly remarkable woman Berta Isaakovna, with two sons Mark and Josef and, now, many grandchildren.

Kiev-Moscow 1947–50, PhD degree under direction of prominent Soviet mathematician and logician P. S. Novikov from Moscow. This was a brilliant scientific start with the famous Trakhtenbrot's finite version of Church's Theorem about non-recursive enumerability of first order logic truths on finite models.

Meanwhile, this was also a difficult time for survival of mathematical logic in the USSR due to an absurd semi-official ideological accusations against such greatest logicians and philosophers of the world, actually founders of the subject, as B. Russel ("warmonger") and A. Tarski ("militant bourgeois"). Trakhtenbrot was only a passive witness of such attacks at that time, but soon. . .

Penza 1950–1960, the Pedagogical Institute. Once, after giving a seminar talk "The method of symbolic calculi in mathematics", Trakhtenbrot was blamed of being "an idealist of Carnap-species"—quite a dangerous accusation in 50-th. Fortunately, Moscow's colleagues P. S. Novikov, A. A. Lyapunov, A. N. Kolmogorov and others were able to defend him. By the way, Trakhtenbrot always priced highly and recalled warmly the deep relationship with Lyapunov, his influence on him and support, particularly in his earlier period of research. In the more quiet 1970s he told me this story even humorously. The very idea of algorithmically undecidable problems was an "ideological crime", because "there could not be anything insoluble for the great Soviet people!" Then, to protect himself in the future, he published an educational article "Algorithms and automatic problem solving" in a "Mathematics in School" journal in 1956. Later it was reworked (1957, 1960, 1974) as a widely popular introductory textbook in the USSR and even abroad "Algorithms and Computational Automata".

Siberian Period 1960–1980 became the last one for Trakhtenbrot in the USSR. He was engaged in research at the Institute of Mathematics of the USSR Academy of Sciences' Siberian Branch (IM SB AS USSR) in the *Department of Theoretical Cybernetics* (through the initiative and guidance of A. A. Lyapunov (1911–1973)) and also lecturing at the Novosibirsk State University in Akademgorodok. In 1967 he and A. Gladky jointly established the Department *Automata Theory and Mathematical Linguistics*.

Collectively, in Academgorodok, staff, undergraduate and PhD students related with the Department were in various periods: V. Agafonov, J. Barzdins, N. Belyakin, V. Boyarkin, M. Dekhtyar, A. Dikovsky R. Freivalds, A. Korshunov, J. Hodjaev, M. Kratko, Z. Litvintseva, I. Lomazova, Matveeva, L. Modina, V. Nepomniaschy, L. Orekhovskaya, V. Sazonov, M. Sokolovskiy, A. Vaiser, M. Valiev. The departmental seminar *"Algorithms and Automata"* was visited by many guests: S. Artemov, M. Kanovich, E. Kinber, V. Kotov, L. Levin, L. Lisovik, A. Nepomniaschy, G. Plesnevich, R. Pliuškevičius, A. Slisenko, M. Taitslin, M. Trakhtenbrot, G. Tseitlin—just to mention some of them.

At this highly fruitful period Trakhtenbrot was working on automata theory (publishing two books in co-authorship of one with N. Kobrinsky (1910–1985) and another with J. Barzdins), complexity theory, semantical and logical problems of high level programming languages, etc. See much more in [2,3]. As the result of his teaching and research on complexity theory, he published Lecture Notes "The Complexity of Algorithms and Computations" (1967).

Unfortunately, this period finished in 1980 as Trakhtenbrot emigrated to Israel. Actually, rather dramatical events preceded this decision. Although this story did not touch me personally, I felt myself this as a kind of highly unpleasant quasi-scientific politics from which Trakhtenbrot and some colleagues, e.g. Michael Dekhtyar, were suffering. Also, Trakhtenbrot was deprived of the management of his Department in 1977. In the new edition of the Soviet Mathematical Encyclopedia Trakhtenbrot's participation was stopped. It is awful when the situation, starting seemingly from quite harmless, even reasonable scientific discussions on formalizations of new intuitions (around the problem of 'perebor' —the problem of eliminating 'brute force search') developed to confrontation on various levels [6,7].

After Trakhtenbrot's emigration to Israel in 1980 some of our colleagues Dektyar, Valiev and, later, Agafonov, Lomazova and me eventually moved to the European part of USSR. Others, already working in different places, stayed in Novosibirsk. M. Trakhtenbrot emigrated to Israel at some later time than his father. The entire life for those who moved has changed. Dikovsky and Modina had already relocated to Kalinin (nowadays, Tver) in 1978. At the same time we were always connected to each other and some of us cooperated scientifically in various ways, e.g. participating in joint projects.

Israel, 1981, Professor of Computer Science at Tel Aviv University. Trakhtenbrot's emigration at that political time could really mean that we never meet again. This was at least a very strange and really painful game of fate. However, the history of the Soviet Union had been changing so dramatically and rapidly that Trakhtenbrot came back to the USSR for a conference in 1989. This was like a miracle! We could not even dream of it! Since then, he visited Russia again and invited many of his colleagues to attend conferences in Israel…

It is unrealistic to present all scientific achievements of Boris Trakhtenbrot here and his invaluable role in Computer Science both in the Soviet Union and in the World. For example, see the description of his world-wide role as a 'Pillar of Computer Science' e.g. in [1,4]. The Friedrich Schiller University in Jena bestowed a degree of doctor *honoris causa* on Trakhtenbrot in October 1997. He was also honoured with the prestigious *EATCS annual Distinguished Achievements Award of 2011* [5] "to acknowledge extensive and widely recognized contributions to theoretical computer science over a life long scientific career".

Boris Abramovich had a happy life in spite of all problems and created his own school—*the Trakhtenbrot's School* with a great scientific and moral atmosphere. He passed away at the age of 95 surrounded by his loving family.

2 Mars Kotdusovich Valiev (01.01.1942–31.01.2018)

Mars was born in Adaevo village, Tatarstan, Russia—the place where the family survived during the WW2 in hardship and poverty while his father was fighting in the frontline. He eagerly started study at school at 6 (very unusual in that time), and graduated in the village Poisevo with a silver medal. Quite young, at 16, he became a student of mathematics at Kazan State University.

On the last, fifth year of study, he (with a few best students) was seconded to practice at the Novosibirsk State University. Mars happily used this chance to become a PhD student under direction of Trakhtenbrot, and then he got a junior research position in the Institute of Mathematics SB AS USSR. His PhD Thesis was: "On Complexity of Word Problem for Finitely Presented Groups" (1969). This subject mostly prevailed in his works till 1978. Then his interests were extended widely and radically and can be described briefly as application of mathematical logic to programming and database theory, computational complexity, multi-agent systems and distributed computing. From 90-th he did his research in a close and fruitful cooperation with Dekhtyar and Dikovsky.

After emigration of Trakhtenbrot, Mars moved to Moscow where he worked in various places: 1982—Moscow Institute of Electronic Engineering, 1984—Institute of System Analysis of RAS (formerly ВНИИСИ). Scientifically most important were 1994—M.V. Keldysh's Institute of Applied Mathematics of RAS (Senior Researcher at Department of Information Modeling and Control Systems) and 2005—Russian State University for the Humanities (known as РГГУ or RGGU; Associate Professor, reading the courses of lectures on "Programming", "Mathematical Linguistics" and "Mathematical Logic").

Mars was highly talented and fruitful scientist. He is the author of at least 68 publications. See also his 'Research Gate' [10]. Very responsive and friendly he was always ready to help his young colleagues. How many times did Mars give me useful tips for improving the style of my first articles! He actually became a heart and soul of the team and the person who always remembered about everybody living here-and-there. Usually it was Mars who informed us about everything important what happened to anyone in the team. I remember he was worried not getting contact with Miroslav Kratko in the Summer of 2017. Mars died of a heart attack at the age of 76—so painful loss.

3 Michael Iosifovich Dekhtyar (18.11.1946–17.03.2018)

Michael was born in Zhitomir (Ukraine) were he finished high school with the golden medal. During his school years he was also the winner of the Ukrainian Republican Mathematical Olympiad. Quite independently, both he and his future wife Rika simultaneously enrolled to the Novosibirsk State University in 1964. Besides studying, they worked in the line of the Komsomol on organizing mathematical "circles" (math clubs) in schools of Academgorodok. This way… they fell in love and married in 1969 in the last year of their studies. They were a very happy family with son Alexander (and now with two grandsons).

As an undegraduate student, Michael quite early became a permanent and successful participant of the Trakhtenbrot's seminar. During this period he made his first research work on 'perebor'. That was the time when the very concept had only a very intuitive level, and required some first mathematical approaches. Trakhtenbrot wrote in [2], page 24: "…the inevitability of *perebor* could be explained in terms of computational complexity of the reduction process. The conjecture was proved by M. I. Dekhtyar in his Master's Thesis (1969)…one can say that his construction implicitly provided the proof of the relativised version of the P $\neq$ NP conjecture" thereby anticipating one of the results by Backer, Gill and Soloway (1975). This work eventually resulted in his PhD Thesis: “On the Complexity of Relativised Computations”, Moscow State University (1977).

However, before enrolling to postgraduate study, Michael had to serve two years 1969–1971 in the Soviet Army as a lieutenant in the city of Semipalatinsk.

After Trakhtenbrot’s emigration the Dekhtyars moved in Tver where Michael was eager to reunite and work with Dikovsky. But it turned out to be impossible right away, and Michael began to work on software projects (against his mathematical interests) in: (1982–1987)—Tver Special Design Office of Control Systems (Chief Designer), then (1987–1991) Tver NPO the Centre of Program Systems (Leading Researcher). At last, the official activity of Michael became coinciding with his research interests: (1991–2015) The Department of Informatics of Tver State University (Associate Professor and then Professor since 2010). Michael received Doctor of Science Degree in mathematics in 2009 with the dissertation “Semantics and the complexity analysis of algorithmic problems of dynamic systems and languages using logic programming”.

Overall, Michael was one of the outstanding computer scientists in Russia with wide range of research interests: Complexity of Computations and Algorithms, Kolmogorov’s Complexity, Data Bases (active, deductive, probabilistic and temporal), Artificial Intelligence, Logic Programming, Intellectual Program Agents and Multiagent Systems, Bioinformatics, Computational Linguistics. He has 109 publications including 7 tutorial books. His two successful PhD students are S. Dudakov (2000) and B. Karlov (2012). See more in [8,9]. Being an exceptional mathematician Michael could explain complicated concepts very clearly even to non-specialists. He was a charismatic person who radiated kindness, generosity and calm confidence.

In 2017 Michael and Rika moved to USA to join the family of their son Alexander Dekhtyar, Professor at the Dept. of Computer Science at California Polytechnic State University. So, he has followed his father’s scientific footsteps.

Michael died after a long illness at the age of 71.

Recalling meetings with Mars and Michael in Tver. Regular meetings of old friends and colleagues where usually held in the hospitable house of Michael and Rika Dekhtyar in Tver. These were long and interesting discussions on various topics connected not only to research news. Once in 2013 Mars, Michael and I were watching a dramatic TV discussion featuring a reform of Russian Academy of Sciences. An extreme anxiety and emotions overpowered us. Then Alexandr Dikovsky from Nantes in France (who was already critically ill) joined us by the Skype and our hot and intense conversation continued. . . The last meetings were especially touching and emotional.

Boris Trakhtenbrot, Mars Valiev and Michael Dekhtyar are greatly missed. No doubts, all of us who knew them, are devastated and heartbroken by the loss of our friends and colleagues. Now, we have no opportunity to find ourselves in their circle again, “to warm up by the fire of their hearts”, to restore those wonderful feelings when we began our scientific life and friendship together in that amazing atmosphere of our spiritual unity. They will remain in our hearts and memories forever, together with other colleagues and friends who also left us: V. A. Agafonov (1940–1997), B. I. Trakhtenbrot (1921–2013), M. A. Taitslin (1936–2013), A. Ja. Dikovsky (1945–2014), N. V. Belyakin (1936–2016), R. V. Freivalds, (1942–2016) L. S. Modina (1945–2017), and just in the most recent months—M. I. Kratko (1936–2018) and A. V. Gladky (1928–2018).

Acknowledgments. I am very grateful to Mark Trakhtenbrot, Venera Erikova and Rika Dekhtyar for providing their memories and commenting on the text.

References

1. Avron, A., Dershowitz, N., Rabinovich. A. (Eds.): Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, Lect. Notes in Comp. Sci. vol. 4800, Springer Berlin, Heidelberg (2008)
2. Trakhtenbrot, B. A.: From Logic to Theoretical Computer Science—An Update. In [1], pp. 1–38.
3. Avron, A., Dershowitz, N., Rabinovich. A. (Eds.): Boris A. Trakhtenbrot: Academic Genealogy and Publications. In [1], pp. 46–57.
4. ACM NEWS In Memoriam: Boris Trakhtenbrot, 1921-2016 https://cacm.acm.org/news/207650-in-memoriam-boris-trakhtenbrot-1921-2016/fulltext
5. EATCS annual Distinguished Achievements Award of 2011; http://eatcs.org/index.php/eatcs-award
6. Trakhtenbrot, B. A.: A Survey of Russian Approaches to Perebor. Annals of the History of Computing. **6**(4), 384–400 (1984)
7. Trakhtenbrot, B. A.: In memory of Andrei P. Ershov. In: Ershov, a Scientist and Human Being (In Russian), Novosibirsk, Nauka, (2006) pp. 343–352
8. Dekhtyar, M. I.: http://homepages.tversu.ru/~p000103/
9. Dekhtyar, M. I.: https://www.researchgate.net/profile/Michael_Dekhtyar
10. Valiev, M. K.: https://www.researchgate.net/profile/Mars_Valiev

Mikhail Iosifovich Dekhtyar (1946–2018)

Sergey M. Dudakov and Boris N. Karlov

CS Faculty, Tver State University, 33, Zhelyabova str., Tver, 170100, Russia
(sergeydudakov@yandex.ru, bnkarlov@gmail.com)

Abstract. The text is a tribute to Mikhail Iosifovich Dekhtyar.

On march 17, 2018, we have lost Mikhail Iosifovich Dekhtyar, one of the great computer scientists in Russia. We would like to remember his main achievements. Certainly our short paper can't describe the full contribution of Mikhail Iosifovich into Computer Sciences, but it gives an idea of his investigations and main results. The list of his publication is also non-complete and contains only few works.

Mikhail Iosifovich Dekhtyar was born in Zhitomir, Ukraine, in 1946. There he graduated from secondary school, then he entered the Department of Mathematics of Novosibirsk State University and finished it in 1969 with distinction. Later he entered postgraduate studies in the Institute of Mathematics, Russian (Soviet Union) Academy of Sciences (RAS), Siberian Branch. He was a student of B. A. Trakhtenbrot. In 1977 he got the PhD degree (the candidate of sciences in physics and mathematics) on the Department of Mechanics and Mathematics, Moscow State University. Since 1974 until 1981 he worked as junior researcher in the Institute of Mathematics. At the end of 1981 M. I. Dekhtyar moved to Tver (Kalinin) where he continued theoretical studies. Simultaneously he gained an experience of a real software development as a chief software designer in SPKB SU and a lead researcher in Centerprogramsystem (both are software development companies). Since 1987 he worked at the Computer Sciences (algorithmic languages and system programming) Faculty, Department of Applied Mathematics and Cybernetic, Tver State University. At first he was a senior lecturer, since 1992 this was his main job and he became an associate professor. In 2009 he got the Doctor of Science Degree in physics and mathematics and became a full professor. Thus he worked until 2015 when he ought to leave the job due to health issues.

M. I. Dekhtyar was a well-known specialist in computational and Kolmogorov complexity, modern databases theory, artificial intelligence, functional and logical programming, multiagent systems, computational linguistics and in other branches of Computer Sciences. He published more than 80 scientific works. M. I. Dekhtyar was a member of American Mathematical Society and International Association of Logical Programming, long-term referent of Abstract Journal (VINITI, RAS) and Mathematical Review. Many times he participated in Russian and international conferences and seminars. A number of times he was involved in international investigations in universities of France (Paris, Nantes) and USA (Maryland, Kentucky).

In the early works M. I. Dekhtyar investigated some problems of computational and Kolmogorov complexity. He studied some problems of relativized computations, the structure of degrees of bounded reductions, relationships between relativized versions of complexity classes P, NP, PSPACE, density of hard sets etc. One of these tasks is to calculate function with respect to its graph. M. I. Dekhtyar proved that solution of this problem necessarily requires some kind of "brute force". He defined the notion of hard sets approximation (see [1]). It means there are sets of "low" polynomial bounded Kolmogorov complexity and their initial segments are equal to the ones of the hard set. Exponential lower bounds of such approximation are obtained for hard sets in some complexity classes. As a corollary it was proved that formulas of some well-known theories can't be easily approximated in this manner.

A group of linear logic problems was investigated by M. I. Dekhtyar with M. A. Taitslin, D. A. Archangelsky and some other researchers (see [3]). One of these tasks is to find an inference of a sequent in the Horn fragment of Girard's linear logic. NP-completeness was proved for this task. To simplify the problem they proposed several different concepts of sequent concurrency. This idea connects sequents of linear logic and nets with bounded resources. It was proved that different concurrency properties lead to different complexities of inference problem. But the other obtained result is a high complexity of concurrency recognition in any case.

Many works of M. I. Dekhtyar are devoted to functional and logical programming. For example, he investigated a semantic of text processing functional language REFAL. He proposed a correct denotational semantic for this language and established NP-completeness of its interpretation (see [2]).

Another practically important problem is intelligent updates of dynamic deductive databases (DDDB). Such database must automatically restore its state to satisfy an integrity constraint (given by a logical program) when this constraint is broken by user actions. Together with A. Ya. Dikovsky and N. Spiratos an axiomatic definition of DDDB updates semantics was developed. For these semantics operators of forced updates were introduced and algorithms were developed for computing such operators (see [5]). The analysis of decidability and computational complexity for these operators was made together with S. M. Dudakov.

Also the behavior of DDDB was analyzed in the case of interacting with an unpredictable external environment. Two following notions were proposed and investigated (see [4]). One is a stability, when this environment tries to restore a correct database state after user ruinous actions. The other is homeostaticity, when the environment behavior is destructive and a correctness is restored by user. The complexity of both problems is established for different kinds of integrity constraints.

M. I. Dekhtyar studied a problem of probabilistic satisfiability for logical programs. For example, databases with interval probabilities are investigated with A. M. Dekhtyar. Non-soundness was established for the fixed-point semantic proposed by V. S. Subrahmanian and R. Ng. The correct semantic of possible worlds was introduced instead (see [6]).

The group of M. K. Valiev, A. Ya. Dikovsky and M. I. Dekhtyar made series of researches of intelligent program multiagent systems (MAS). The problem is to verify a behavior of MAS (see [7]). First-order temporal logic and temporal calculus were used to describe the MAS behavior. Verification complexity was established for different classes of MAS. For probabilistic and fuzzy MAS obtained results allow to use known methods of finite Markov chains verification.

One of the last series of researches was performed with A. Ya. Dikovsky and B. N. Karlov (see [10]). An object of these investigations is categorial dependency grammars (CDG). This concept is a generalization of classical formal grammars, but it permits to parse more complex constructions of natural languages. For example, CDG can describe non-projective dependencies when connected items are separated. Parsing complexity of CDG was established, in particular it was proved that the membership problem for CDG is NP-complete in general case, but CDG can be parsed in polynomial time under some natural restrictions. In subsequent studies further generalization of CDG was found, it is multimodal categorial dependency grammars (mmCDG). Some results on closure properties were found. It was proved that the class of mmCDG-languages is an abstract family of languages (AFL), and the class of CDG-languages is closed under all AFL-operations except maybe the iteration.

For CDG and mmCDG corresponding equivalent classes of formal automata were proposed. These are the push-down automata with independent counters or with stacks of independent counters.

M. I. Dekhtyar had some singular works. One of them is dedicated to bioinformatics. He designed, developed and tested an algorithm for searching of strong promoters in bacterial genomes (see [8]). The other is devoted to markup languages (see [9]) and was written with M. K. Valiev.

M. I. Dekhtyar was very talented lecturer. He designed and taught a lot of courses: discrete mathematics, development of efficient algorithms, mathematical foundations of computer sciences, Kolmogorov complexity and randomness, methods of artificial intelligence, markup languages and many other. A number of workbooks were written. For example, the book "VBA and Office 97. Office programming" was written together with V. A. Billig. In 1998 this book won a diploma in the contest "Business book of Russia" in nomination "Computer Sciences". In 2007 he made a workbook "Lectures on discrete mathematics" published by "BINOM", and in 2011 he made a workbook "Graph algorithmic tasks".

M. I. Dekhtyar was science supervisor for S. M. Dudakov and B. N. Karlov when they were PhD studying, and for many graduate and undergraduate students.

We condole with Rosalia Vitalievna and Alexander Mikhailovich Dekhtyar over the loss of Mikhail Iosifovich.

References

1. Dekhtjar M. I. Bounds on computational complexity and approximability of initial segments of recursive sets. In: Proc. 8-th Symp. Mathematical Foundations of Computer Science, Olomouc (Czech.), 1979, LNCS 74, 277–283.
2. Dekhtjar M. I. On Semantic and Properties Proof for REFAL-programs. In: Proc. All-Soviet Conf. on Program Synthesis, Testing, Verification and Debugging, Riga (Latvia), 1986, 102–103.
3. Archangelsky D. A., Dekhtyar M. I., Taitslin M. A. Linear logic for nets with bounded resources. Annals of Pure and Applied Logic, 1996, No. 78 (1–3), 3–28.
4. Dekhtyar M. I., Dikovsky A. Ja. On Homeostatic Behavior of Dynamic Deductive Data Bases. In: D. Bjomer, M. Broy, I. Pottosin (eds.) Proc. 2-nd Int. A. P. Ershov Memorial Conference "Perspectives of System Informatics", 1996, LNCS 1181, 420–432.
5. Dekhtyar M., Dikovsky A., Dudakov S., Spyratos N. Maximal State Independent Approximations to Minimal Real Change. Annals of Mathematics and Artificial Intelligence, 2001, No. 33 (2–4), 157–204.
6. Dekhtyar A., Dekhtyar M. I. Possible Worlds Semantics for Probabilistic Logic Programs, in Proc., International Conference on Logic Programming (ICLP)' 2004, LNCS 3132, 137–148.
7. Dekhtyar M. I., Dikovsky A. J., Valiev M. K. Temporal verification of probabilistic multi-agent systems. In: Avron A., Dershowitz N., Rabinovich A. (eds.) Pillars of Computer Science, 2008, LNCS 4800, 256–265.
8. Koloyan A. O., Hambardzumyan A. A., Hovsepyan A. S., Dekhtyar M. I., Saghiyan A. S., Sakanyan V. A. Virtual screening of coryneform bacteria genome for the presence of strong promoters. Biotechnologiya, 2010, No. 1, 41–50.
9. Dekhtyar M. I., Valiev M. K. Concurrent Use of OWL-onthologies and RIF-rules. In: Proc. of 2-nd Symp. "Onthological Modeling", Moscow, 2011, 151–173.
10. Dekhtyar M., Dikovsky A., Karlov B. Categorial dependency grammars. Theor. Comput. Sci., 2015, No. 579, 33–63.

Michael I. Dekhtyar's Contributions to the Theory of Interval Probabilistic Programs

Alex Dekhtyar

Department of Computer Sicence and Software Engineering
California Polytechnic State Universy, San Luis Obispo,CA, USA
(`dekhtyar@calpoly.edu`)

Abstract. This paper presents a brief overview of my joint work with Michael I. Dekhtyar on the topic of the semantics of Interval Probabilistic Programs.

1 Introduction

Interval probabilistic programs are logic programs in which clauses have a form

$$H_1 : [a_1, b_1] \vee \ldots H_m : [a_m, b_m] \longleftarrow F_1 : [l_1, u_1] \wedge \ldots \wedge F_n : [l_n, u_n],$$

where H_i and F_j are either atomic formulas or simple conjunctions, and $[a_i, b_i]$ and $[l_j, u_j]$, called *annotations*, are subintervals of $[0, 1]$. Intuitively, the clause above is understood as a statement that *if the probabilities of events described by formulas $F_1, \ldots, F_n$ fall, respectively within the ranges $[l_1, u_1], \ldots, [l_n, u_n]$, then the probability of at least one of events described by formulas $H_1, \ldots, H_m$ falls within the respective range $[a_1, b_1]$, or $[a_2, b_2$, or $\ldots$, or $[a_m, b_m]$*. Various subclasses of interval probabilistic programs were proposed by Ng and Subrahmanian in [10,11] and Dekhtyar and Subrahmanian in [4,5].

The Generalized Annotated Programs (GAPs) of Subrahmanian and Kifer have pioneered the use of annotations to represent truth values in logical programs [9]. The GAP framework uses annotations from a lattice of truth values and shows how to construct both model-theoretic and matching fixpoint semantics for annotated logic programs. Early interval probabilistic program frameworks [10,5] represented the extensions of the GAP framework, with probability interval annotations forming a lattice over the subset inclusion operation.

Years later, Dekhtyar and Dekhtyar noticed that there is a more precise way to interpret interval probabilistic programs. Starting with a simplified case [3] and eventually expanding their work to encompass more expressive frameworks [2,1] they described the precise semantics of interval probabilistic programs in new terms, and developed efficient procedures for computing the new semantics.

In this paper, we provide a brief description of Michael Dekhtyar's contributions to the theory of interval probabilistic logic programs, concentrating on his work on defining their precise semantics and developing efficient algorithms for constructing it. In Section 2 I provide a brief overview of the history of my collaboration with Michael Dekhtyar in the area of probabilistic logic programming. Section 3 contains a brief outline of the technical contributions.

2 Historic Notes

In October 1998, at the invitation of my Ph.D. advisor V.S. Subrahmanian, Michael I. Dekhtyar visited University of Maryland, where I was a Ph.D. student at the time[1]. During

[1] The immediate reason for this visit was the birth of Michael's first grandson, Victor.

this visit, we introduced him to Hybrid Probabilistic Logic Programs (hp-programs) [4], an emerging topic of my Ph.D. dissertation. Over the course of Michael Dekhtyar's one-month stay in Maryland, we jointly extended the semantics of hp-programs to the case where the uncertainty was temporal [6], and have answered a number of important complexity questions related to hp-programs [7].

This would have been the extent of our joint work if it wasn't for a series of conversations Michael Dekhtyar and I had during his visit to the US in 2003, when it occurred to us that the semantics of hp-programs, and in general, of any probabilistic interval programs as described in our prior work [4,7], as well as in the work of Ng and Subrahmanian [10,11] was not precise. We developed the precise semantics for a simple fragment of interval probabilistic programs first [3], and during Michael's subsequent visit to the US in 2004-2005[2] we extended the semantics to a significantly broader class of programs[2]. In 2007, we worked remotely on summarizing our results on this topic, broadening them, and publishing them as a single extended paper [1].

3 Semantics of Interval Probabilistic Programs

The GAP-style semantics of interval probabilistic programs [10,4,5] is briefly described as follows. Each atom a from the Herbarnd base of a program P is associated with a probability interval $h(a) = [a, b] \subseteq [0, 1]$. We say that $h \models a : [l, u]$ iff $h(a) \subseteq [l, u]$. A conjunction $a \wedge b : [l, u]$ and a disjunction $a \vee b : [l', u']$ is satisfied by h if $h(a)$ and $h(b)$ interpreted as interval probabilities of a and b make it possible for the probability of $a \wedge b$ and $a \vee b$ to be respectively in the intervals $[l, u]$ and $[l', u']$. $\longleftarrow$ is interpreted as a strict implication. The fixpoint semantics starts by assigning each atom in a program P the probability value $[0, 1]$ ($\perp$ in the lattice of subintervals of [0,1] based on set inclusion), and narrows down the interval by "firing" the facts $a : [l, u]$ and clauses $a : [l, u] \longleftarrow Body$ when possible. The key mechanism for narrowing the probability range of a formula is this: if clauses with the heads $H : [l, u]$ and $H : [l', u']$ are "fired", then we infer $H : [\max(l, l'), \min(u, u)] = [l, u] \cap [l', u']$. For the frameworks of [10,5] it is shown that given a probabilistic program P, the fixpoint procedure converges to its minimal interval interpretation.

It turns out, however, that this is largely due to the weakness of both the definition of a minimal interval interpretation, and the fixpoint procedure itself. A well-known approach to reasoning with probabilities is the possible world semantics, in which a probability distribution over a set of possible worlds (sets of atoms that are true) is given, and a probability of a specific atom is computed as the sum of probabilities of possible worlds the atom is true in. Dekhtyar and Dekhtyar noted in [3] that if point probability models associating a single probability value with each atom/formula in an interval probabilistic program are used as interpretations of even the simplest programs, the set of valid models can no longer be faithfully represented by assigning each atom a single probability interval. A simple program illustrating this is shown below:

$a : [0.2, 0.4] \longleftarrow .$ $\qquad$ $b : [0.3, 0.5] \longleftarrow .$

$b : [0.6, 0.7] \longleftarrow a : [0.2, 0.3]$ $\quad$ $b : [0.6, 0.7] \longleftarrow a : [0.3, 0.4]$

The fixpoint procedure of [10] fires both facts $a : [0.2, 0.4]$ and $b : [0.3, 0.5]$ but is unable to fire either of the rules, because $[0.2, 0.4]$ is not a subset of either $[0.2, 0.3]$ or $[0.3, 0.4]$, thus leaving the probability interval for b as $[0.3, 0.5]$. Yet, it is pretty clear that if probability of a is in the interval $[0.2, 0.4]$, it must be either in the interval $[0.2, 0.3]$, in which case the first rule must fire, or in the interval $[0.3, 0.4]$, in which case the second rule must fire. In either

[2] On the occasion of the birth of his second grandson, Gregory.

case, $b : [0.6, 0.7]$ becomes confirmed, contradicting the already established $b : [0.3, 0.5]$, i.e., the program above cannot have a satisfying possible world model.

In [3], we provide a precise description of the set $Mod(P)$ of the possible worlds models of an interval probabilistic program P. We describe each program in terms of a collection of sets of inequalities – each set corresponding to one possible template of assignment of point probabilities to the atoms in P that satisfies all rules (all atoms are satisfied, and for each rule, either both the head and the tail are satisfied, or the tail is not satisfied). We show that the set of all possible solutions of such a system of inequalities ($INEQ(P)$), associates with each atom a *a union* of open and closed subintervals of $[0, 1]$.

In [1] we show the following results related to the computation of the set $Mod(P)$ of possible worlds models of interval probabilistic programs:

– Given the Herbrand base $L = \{a_1, \ldots, a_n\}$ of a program P, the number of disjoint subsets of $[0, 1]^n$ in $Mod(P)$ has an upper bound exponential in n.
– For some families of interval probabilistic programs, this exponential bound is reached.
– A Gelfond-Lifschitz-style [8] procedure can be used to construct $Mod(P)$ in a way that enumerates all disjoint subsets of $Mod(P)$ and guarantees that each of them is considered *exactly once*.
– For a non-trivial class of simple interval probabilistic programs there exists a straightforward necessary and sufficient condition for $Mod(P)$ to coincide with the result of the fixpoint procedure.

The results of [3,2,1] are somewhat pessimistic: they suggest that a relatively simple and intuitive framework for reasoning with interval probabilities has very complex, both from the descriptive, and computational points of view semantics. At the same time, this work crosses the "t"s and dots the "i"s in close to 20 years of active research in the area of interval probabilistic logic programs.

References

1. A. Dekhtyar and M. I. Dekhtyar, (2009), The Theory of Interval Probabilistic Logic Programs, *Annals of Mathematics and Artificial Intelligence*, Vol. 55 (3-4), pp. 355-388.
2. A. Dekhtyar and M. I. Dekhtyar, (2005), Revisiting the Semantics of Interval Probabilistic Logic Programs, in *Proceedings, Eighth International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'05*, LNAI, Vol. 3662, pp. 330-342, September 2005, Diamante, Italy.
3. A. Dekhtyar and Michael I. Dekhtyar, (2004), Possible Worlds Semantics for Probabilistic Logic Programs, in *Proceedings, International Conference on Logic Programming (ICLP'2004)*, pp. 137–148, St. Malo, France, September 2004.
4. A. Dekhtyar and V.S. Subrahmanian,(1997), Hybrid Probabilistic Programs, in *Proceedings, International Conference on Logic Programming (ICLP'97) (ed. L. Naish)*, MIT Press, pp. 391–405, July 1997, Leuven, Belgium.
5. A Dekhtyar and V.S. Subrahmanian, (2000), Hybrid Probabilistic Programs, *Jorunal of Logic Programming*, Vol. 43, No. 3, pp. 187-250.
6. A. Dekhtyar, M.I. Dekhtyar and V.S. Subrahmanian, (1999), Temporal Probabilistic Logic Programs in *Proceedings, International Conference on Logic Programming (ICLP'99)*, pp. 109–123, December 1999, Las Cruces, NM.
7. M.I. Dekhtyar, A. Dekhtyar and V.S. Subrahmanian, (1999), Hybrid Probabilistic Programs: Algorithms and Complexity, in *Proceedings of Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pp. 160–169, July 1999, Stockholm, Sweden.
8. M. Gelfond and V. Lifschitz (1988) The stable model semantics for logic programming. in *Proceedings Fifth International Conference and Symposium on Logic Programming (ICLP/SLP'1988)*, pp. 1070-1080.

9. M. Kifer, V.S. Subrahmanian (1992) Theory of Generalized Annotated Logic Programming and its Applications, *Journal of Logic Programming*, Vol. 12, No. 4, pp. 335–368.
10. R. Ng and V.S. Subrahmanian. (1993) Probabilistic Logic Programming, *Information and Computation*, 101, 2, pps 150–201, 1993.
11. R. Ng and V.S. Subrahmanian. A Semantical Framework for Supporting Subjective and Conditional Probabilities in Deductive Databases, JOURNAL OF AUTOMATED REASONING, 10, 2, pps 191–235, 1993.

Taylor expansion of proofs and static analysis of time complexity
(abstract of invited talk)

Daniel de Carvalho

Innopolis University, Russia
d.carvalho@innopolis.ru

Gentzen introduced (1934) two different formalisms for proofs in intuitionistic logic: natural deduction (NJ) and sequent calculus (LJ). The Curry-Howard-de Bruijn isomorphism between NJ and simply-typed lambda-terms (1940) arose in the 60's, while there was no such an isomorphism for LJ, essentially because equality between LJ proofs is "evil". In 1971, Girard extended this isomorphism to the second-order considering an expressive polymorphic programming language (System F). Fifteen years later, inspired by Berry's notion of stability (1978), which refines Scott's notion of continuity (1972), he introduced coherence spaces as a model of System F; that is how linear logic was born. Linear logic is a refinement of intuitionistic and classical logic that distinguishes among all the proofs those using exactly once their assumptions. Taylor expansion of proofs, which expresses proofs as series of their linear approximants, was considered by Girard from a semantic viewpoint. Since the 2000s Ehrhard has been investigating topological vectorial spaces as models of linear logic. This investigation led him with Regnier to introduce differential nets (2006), which allow to express syntactically Taylor expansion.

I will address the two following related problems: 1) Friedman proved (1975) that the standard model for the lambda-calculus with sets and functions is complete for beta-reduction; can we obtain a similar result for cut-elimination in linear logic? It was conjectured twenty years ago that it is the case with the model of sets and relations. This conjecture has finally been proved. This result shows in particular that we have a good notion of equality of proofs in linear logic. 2) If two proofs have the same Taylor expansion, are they equal? Furthermore, inspired by the closed relation between Taylor expansion and Krivine's machine, I will introduce a typing system allowing a static analysis of time complexity.

Russell logical framework: proof language, usability and tools
(abstract of invited talk)

Dmitry Vlasov

Sobolev Institute of Mathematics, Novosibirsk, Russia
vlasov@academ.org

When one develops a proof language, the following features are of the highest degree of importance: usability, reliability and flexibility. Language Russell was designed with these features put in the forefront. Different aspects of these features are discussed, and Russell system is compared with other popular formal math systems like Isabelle, HOL, Mizar, Metamath etc.

Software testing:
Finite State Machine based test derivation strategies
(abstract of invited talk)

Nina Yevtushenko

Ivannikov Institute for System Programming, RAS, Moscow Russia
nyevtush@gmail.com

There is a big body of works in software testing (see, for example, Springer publications of the IFIP International Conference on Testing Software and Systems, ICTSS, 1991 – 2017) devoted to test derivation based on Finite State Machines (FSMs). The reason is that FSMs on one hand, include a "natural reactivity" and thus, can be eventually used for testing systems which work in the "request-response" mode. On the other hand, for FSMs, the test derivation is a long standing problem and a number of methods have been developed for deriving test suites with guaranteed fault coverage, i.e., test suites which guarantee to detect at least critical faults. The main deterministic FSM based methods deal with the case when the state number of an implementation FSM is limited and somehow most of these methods are based on the so-called W-method or the Henni method when deriving a checking sequence. However, nowadays FSM based test derivation is extended with novel powerful methods for different FSM types and in fact, an H-method or a SPY-method can be hardly seen as modifications of W-method; moreover, another interesting body of works appeared for testing nondeterministic FSMs. For nondeterministic, possibly partial FSMs, novel interesting techniques for deriving adaptive tests with guaranteed fault coverage have been elaborated. FSM based test suites were effectively used for testing protocol software implementations and a number of inconsistencies were found in those implementations. Extensions to the W-based methods are also considered in the context of hybrid systems including systems with timed constraints. In order to have a test suite with guaranteed fault coverage usually the behavior of such a system is described by an appropriate FSM. We also note that lately FSM extensions are used for checking not only functional but also non-functional requirements for a system under test.

Making Verification in KeYmeara Easier – A Graphical Approach for Better Usability

Thomas Baar[1] and Sergey Staroletov[2]

[1] Hochschule für Technik und Wirtschaft (HTW) Berlin, Germany thomas.baar@htw-berlin.de
[2] Polzunov Altai State Technical University, Barnaul, Russia
serg_soft@mail.ru

Abstract. KeYmeara is an interactive theorem prover for verifying safety properties of cyber-physical systems (CPSs). It implements a Dynamic Logic for Hybrid Programs (HPs), while a HP models a CPS very precisely. Verifying properties of a given system in KeYmeara can become a challenge for the user since the proof is authored in a classical sequent calculus framework and a successful proof requires from the user intimate knowledge on the available calculus rules. Another barrier for widespread application of KeYmeara is the purely textual representation of current proof goals, what requires from the user very good training, experience, and patience. In this paper, we present an alternative verification approach based on KeYmeara, which drastically improves usability and minimizes user interaction. The main idea is to let the user annotate contracts to states of the hybrid automaton. Thus, the user can employ the graphical representation of the modelled system and is not bound to the purely textual form of hybrid programs as in KeYmeara. Based on the user-provided contracts, one can generate proof obligations, which are much simpler than the original proof goal in KeYmeara.

1 Motivation

A cyber-physical system (CPS) is a system that tightly combines software with physical components. The state of a CPS consists of the discrete state of its software and the analogous state of its physical parts. Safety analysis of CPSs must take into account physical laws, which apply to physical parts as well as the code structure of the software part [6].

The notion of hybrid automaton (HA) [3,7] has proven to be useful for the precise description of the behaviour of CPSs. Logic-based analysis of a given hybrid automaton has been thoroughly investigated by Platzer in [10] and became practically feasible by the tool KeYmeara [12]. This tool is an interactive prover and allows the user to formally prove safety properties taken both discrete and continuous state variables into account. KeYmeara implements a Dynamic Logic for hybrid programs (HPs), which can be seen as a textual representation of hybrid automata. While the notation of HP is compact and concise, it can become painful to be bound on the purely textual notation when proving even rather obvious properties of a CPS.

In this paper, we propose an approach to overcome some of the obstacles the user faces when authoring a proof using KeYmeara. Based on a very simple example, we show how a graphical representation of a HP can be obtained. Our graphical representation is inspired by classical hybrid automata and shows the states of the system together with applied physical laws. Based on our graphical representation, the user can rather easily provide additional facts, which are necessary to formally verify the whole system. These additional facts are annotated in our approach as contracts to the system states. Based on the provided contracts, one can generate proof obligations, which are much simpler than the original proof goal in KeYmeara.

2 Verification of Cyber-Physical Systems using KeYmeara

In KeYmeara, a CPS is modelled in form of a Hybrid Program (HP), for which properties expressed in Dynamic Logic can be proven. A HP is built on variables (always of type float), derivations of (continuous) variables, arithmetic expressions, first-order formulas for conditions on the current state, and a simple execution language with operators for assignment (:=), sequential execution (;), non-deterministic choice ($\cup$), and non-deterministic repetition ($*$). For a detailed introduction to HP and the logic of KeYmeara, the reader is referred to [11].

2.1 Running Example: Simple Velocity Controller

As an illustrative example, we introduce a simple velocity controller. The velocity v of the controlled system (e.g. a car or a train) is set by the controller either to a fixed velocity v_0 or to 0 (zero). The controller readjusts the choice periodically based on the current position z. If z is far enough from an obstacle m, the system will keep velocity $v = v_0$, otherwise the system is stopped. The safety property we want to prove is, that the controller never stops the system too late, i.e. under all circumstances we will have $z < m$.

Our example is actually a simplified version of the tutorial example given in [11] and formulated as a Hybrid Program as follows

$$\alpha = \begin{array}{ll} q := \mathit{start}; & \\ (& (?q = \mathit{start}; SB := m - \epsilon v_0; t := 0; q := \mathit{diamond}) \\ & \cup\, (?q = \mathit{diamond} \wedge z < SB; v := v_0; q := \mathit{driving}) \\ & \cup\, (?q = \mathit{diamond} \wedge z \geq SB; v := 0; q := \mathit{stopped}) \\ & \cup\, (?q = \mathit{driving}; z' = v, t' = 1 \& t \leq \epsilon) \\ & \cup\, (?q = \mathit{driving}; q := \mathit{start}) \\ & \cup\, (?q = \mathit{stopped};) \\ )* & \end{array}$$

The Hybrid Program starts in a state *start* ($q := \mathit{start}$) and then chooses non-deterministically often from the following program branches: if the current state is *start*, then SB is assigned to $m - \epsilon v_0$, t is assigned to 0 and the current state is switched to *diamond*; if the current state is *diamond*, the current state switches to *driving* or *stopped* depending on the value of z; if the current state is *driving*, the continuous variables z and t changes their values according to $z' = v$ and $t' = 1$ but only as long as $t \leq \epsilon$ holds; if the current state is *driving*, the state can also switch at any time to *start*; if the current state is *stopped*, the system will remain in this state. The safety property one would like to prove is $z < m \wedge \epsilon > 0 \wedge v_0 > 0 \rightarrow [\alpha] z < m$

3 Our Approach: Graphical Representation and Contracts

The simple velocity controller specified by HP α above can be equivalently specified as a Hybrid Automaton (HA) as shown in Fig.1. In addition to traditional HAs ([7]), our diagram formulates not only the behaviour of the system, but also the safety property to be proven in form of a pre-/post-condition pair. Furthermore, we use not only continuous states (*driving*, *stopped*), but also a discrete state (labelled with $SB := \ldots$) in order to execute assignments. The dashed arrows from *driving* and *stopped* to the exit state should express, that the system can abort at any time; thus the post-condition we formulate can rather be read as an invariant of the system.

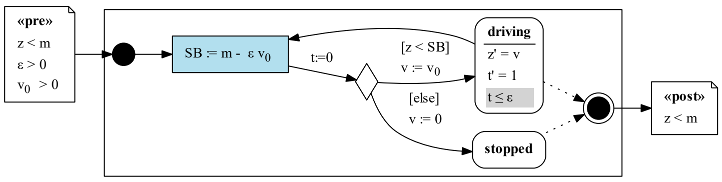


Fig. 1. System specification in Hybrid Automaton - like notation

When trying to verify the system specified in Fig.1 with KeYmeara we would run in the same problems as in the original case: the user has to provide the proof arguments to KeYmeara in form of idiosyncratic proof rules.

Thus, in our approach, we will provide to KeYmeara a system specification that already contains key facts for proving the correctness as shown in Fig.2.

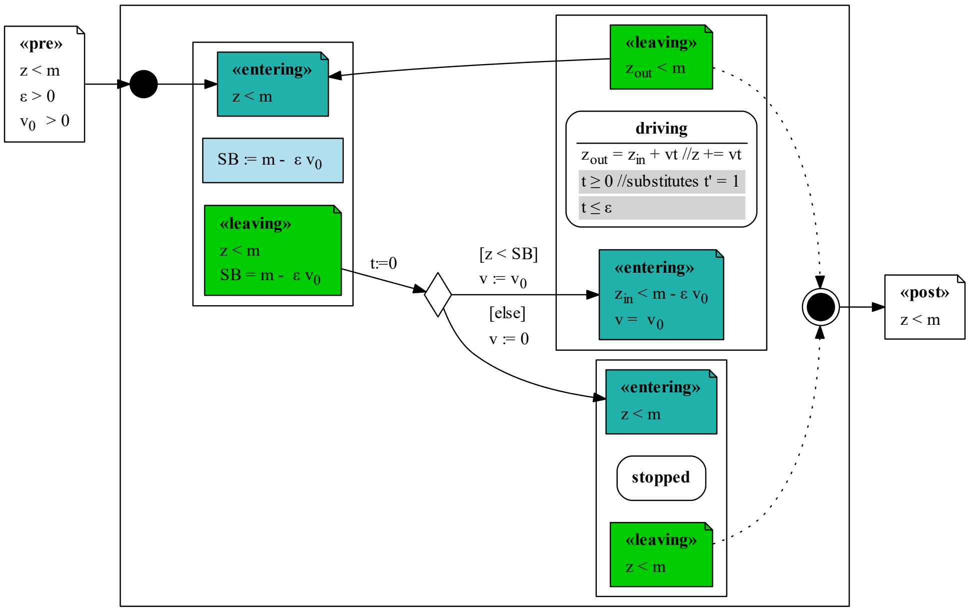


Fig. 2. Proof contracts have been added

We allow the user to add pre-/post-conditions to every state in the Hybrid Automaton. For example, we added $z_in < m - \epsilon v_0 \wedge v = v_0$ as pre-condition and $z_out < m$ as post-condition to state *driving*. Instead of proving $pre \rightarrow [\alpha]post$ for the whole system α, we have now to prove for each state S the proof obligation $pre_S \rightarrow [\alpha_S]post_S$ and for all transitions

T going from state $S1$ to state $S2$ the proof obligation $post_{S1} \wedge cond_T \rightarrow [act_T]pre_{S2}$ where $cond_T$, act_T denote the condition and actions annotated with transition T.

To summarize, instead of one rather complex proof obligation $(pre \rightarrow [\alpha]post)$ we use KeYmeara now to prove many much smaller (and often even trivial) proof obligations, mostly of them can be proven without any further user interaction.

4 Related Work

According to the nature of CPSs, we can identify three main techniques for their specification: (1) transition automata with discrete jumps, (2) continuous dynamic calculations in each state (what makes the system to be a hybrid system) and (3) requirements or goals which are interesting for a user to check.

In this section we browse among some known techniques to verify these parts which issues we can face. We will use a simple demo system with the velocity controller and a simple goal such as "z<m".

Firstly, the user's goals about constant or unexpected behaviour can be easily transformed to the LTL formulas with temporal operators "always" and "eventually". For example, a goal (z<m) during execution a CPS system from the example can be expressed as a globally [](z<m) LTL predicate. If we need a goal to be always right in the special states, we can add the state variables and construct an LTL formula with logical operators like "and" and "implication". This is a good and standard way to express the goals [13].

But after expressing the goals we will get a major issue when we will try to describe the behaviour model of a system since it is hard to impossible to implement the continuous (or at least close to continuous) dynamic behaviour in each state, even if we hard code the mathematical expressions and solve it without loose of accuracy. The main problem here is an explosion of the number of internal states and memory being used in a verifier to express such a system.

According to the design of a well-known tool in Model Based Checking world, Spin's Promela language doesn't include floating point arithmetic to the models, because the purpose of the language is to encourage abstraction from the computational aspects and focusing on the verification of process interaction [2]. So we need a more than integer-based arithmetic and it can't be done in the most of the cases.

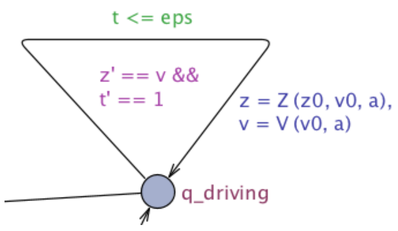
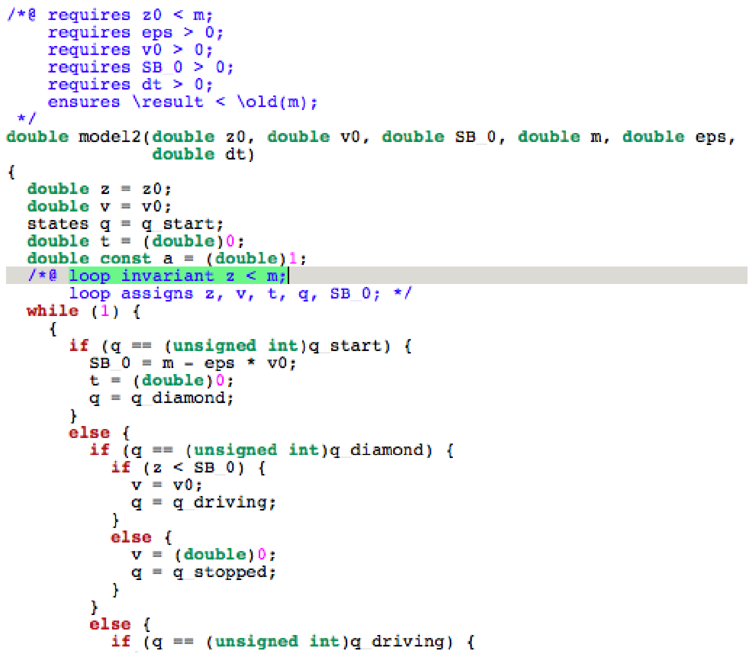


Fig. 3. Derivatives in an invariant on a Uppaal model

Next, we can move to tools that use complicated automaton models, especially timed automata. One great representative is Uppaal [4]. It offers construction of such extended automata, check invariants and can verify properties expressed with modalities (i.e. always predicate). For example, if we would like to test the (z<m) property during particular system run, we can check it dynamically by putting (z<m) as an invariant in desired states or statically verify that goal by using a query with E[] (z<m) property. To describe the system in the Uppaal, we should implicitly create the behaviour automaton. We can introduce control variables and during transitions we can update them by calling our functions that are being written in a code which almost looks like C. The creators of Uppaal made a big step from ordinary discrete automaton - they introduce a SMC extension [5] that offers making controlled non-determined transitions, adds double datatype to user's code, adds floating-point clocks type (user can specify the delta step for it in the settings) and they even targeted to modelling and verifying hybrid systems by introducing time based derivatives in the invariants.

The main disadvantage of writing code for hybrid systems in Uppaal (as in some other tools) is that we should program it almost implicitly using the offered language and it is hard to write complicated ODEs or other types of mathematical models. Uppaal supports time-based derivatives, we can use for checking invariants when staying in a state (as an additional way to check the correctness of a mathematical model implementation, see Figure 3).



```
/*@ requires z0 < m;
    requires eps > 0;
    requires v0 > 0;
    requires SB_0 > 0;
    requires dt > 0;
    ensures \result < \old(m);
 */
double model2(double z0, double v0, double SB_0, double m, double eps,
              double dt)
{
  double z = z0;
  double v = v0;
  states q = q_start;
  double t = (double)0;
  double const a = (double)1;
  /*@ loop invariant z < m;
      loop assigns z, v, t, q, SB_0; */
  while (1) {
    {
      if (q == (unsigned int)q_start) {
        SB_0 = m - eps * v0;
        t = (double)0;
        q = q_diamond;
      }
      else {
        if (q == (unsigned int)q_diamond) {
          if (z < SB_0) {
            v = v0;
            q = q_driving;
          }
          else {
            v = (double)0;
            q = q_stopped;
          }
        }
        else {
          if (q == (unsigned int)q_driving) {
```

Fig. 4. Annotations in the simple hybrid model in C

Lastly, we refer to the rich world of verification tools for C. Note, that a huge amount of mathematical libraries has been written in C. The modular platform for static analysis Frama-C [8] can prove a lot of types of C programs, it uses the deductive approach and extends the Hoare logic to work with pointers, memory and various type conversions. The floating point arithmetic is supported. They use a Weakest Precondition (WP) method and the verification of the program in this case will consist of calculating the weakest precondition from the end to the beginning of the function code and setting up the problem of proving the reverse derivation to the theorem prover (an internal and some externals interactive provers can be used). So, it is a very strict method and to prove the function, all the precondition, post-conditions, changes the variables, loops, internal function calls must be annotated in a special form (see an invariant with (z<m) on Figure 4). The ISO-standardized language ACSL [1] is used.

To verify the hybrid system with Frama-C, the mathematical model for it should be solved (by direct or numerical methods) and annotated in C (and annotations can occupy huge places in a code). There is no explicit way to write it in the terms of mathematics.

A similiar approach is pursuited by Ariadne [9], a framework implemented in C++. The user can encode a CPS in form of a hybrid automaton including its requirements as instances of C++ classes. More precisely, there are special classes for declaring states, transitions and invariants of the modeled system. After defining the system, the user can execute code to do reachability analysis and prove properties of the described system, especially for safety verification. Ariadne allows parametric verification, which exhaustively checks all possible values of the parameters and determines for which values the component obeys the guarantees. For setting up physical formulas in a model, the user has to use overloaded operations, which are not fully supported by the framework yet. Also, a graphical system representation is not supported yet and enforces the user to work with plain C++-code all time.

5 Conclusion and Future Work

In this paper, we discussed one of the biggest barrier of verification tools such as KeYmeara to get widely acceptance in industry: They assume the user to be highly trained in mathematical logic and to know in detail the system's proof rules. In addition, a particular problem of KeYmeara is the representation of a proof. The actual proof of a system property can be saved by KeYmeara, but inspection of it by the user is very hard, since the key ideas of a proof are cluttered by many other proof rule applications, which are necessary to get a formal proof.

Based on a simple but typical example, we illustrated our new approach to let the user annotate key proof facts in the system description itself. As a result, there are much more proof obligations to be proven by KeYmeara, but they are much simpler now and require much less user interaction while the formality of the proof is preserved.

So far, we treated the illustrated example as a pen-and-pencil case study. The next step will be to build a prototypical front-end tool, that allows the user to specify the system graphical as shown in Fig.2 and which will generate the proof obligations for KeYmeara automatically.

References

1. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliatre, Claude Marche, Benjamin Monate, Yannick Moy, Virgile Prevosto. ACSL: ANSI/ISO C Specification Language Version 1.12. https://frama-c.com/download/acsl_1.12.pdf
2. Spin: Promela reference. Float - floating point numbers. http://spinroot.com/spin/Man/float.html
3. R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In Grossman et al. [6], pages 209-229.
4. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In Formal methods for the design of real-time systems, pages 200-236. Springer, 2004.

5. A. David, K. G. Larsen, A. Legay, M. MikuËĞcionis, and D. B. Poulsen. Uppaal SMC tutorial. International Journal on Software Tools for Technology Transfer, 17(4):397-415, 2015.
6. R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. Hybrid Systems, volume 736 of Lecture Notes in Computer Science. Springer, 1993.
7. T. A. Henzinger. The Theory of Hybrid Automata. In Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996, pages 278-292. IEEE Computer Society, 1996.
8. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. Formal Aspects of Computing, 27(3):573-609, 2015.
9. P. Nuzzo, A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. Proceedings of the IEEE, 103(11):2104-2132, 2015.
10. A. Platzer. Logical Analysis of Hybrid Systems: Proving Theorems for ComplexDynamics. Springer, Heidelberg, 2010.
11. A. Platzer. Logic and Compositional Verification of Hybrid Systems (Invited Tutorial). In G. Gopalakrishnan and S. Qadeer, editors, CAV, volume 6806 of LNCS, pages 28-43. Springer, 2011.
12. J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer. How to Model and Prove Hybrid Systems with KeYmaera: A tutorial on safety. STTT, 18(1):67-91, 2016.
13. S. Salamah, A. Gates, and V. Kreinovich. Validated templates for specification of complex LTL formulas. Journal of Systems and Software, 85(8):1915-1929, 2012.

On the expressive power of some extensions of Linear Temporal Logic

Anton Gnatenko[1], Vladimir Zakharov[2]

[1] Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, RU-119899, Russia
[2] National Research University Higher School of Economics (HSE), Moscow, 101000, Russia
(Corresponding author: gnatenko.cmc@gmail.com)

Abstract. One of the most simple models of computation which is suitable for representation of reactive systems behaviour is a finite state transducer which operates over an input alphabet of control signals and an output alphabet of basic actions. A behaviour of such a reactive system displays itself in the correspondence between flows of control signals and compositions of basic actions performed by the system. We believe that behaviour of this kind requires more suitable and expressive means for formal specifications than conventional LTL. In this paper we define some new (as far as we know) extension $\mathcal{LP}$-LTL of Linear Temporal Logic specifically intended for describing the properties of transducers computations. In this extension the temporal operators are parameterized by sets of words (languages) which âĂŃâĂŃrepresent distinguished flows of control signals that impact on a reactive system. Basic predicates in our variant of temporal logic are also languages in the alphabet of basic actions of a transducer; they represent the expected response of a transducer to the specified environmental influences. In our earlier papers we considered model checking problem for $\mathcal{LP}$-LTL and $\mathcal{LP}$-CTL and showed that this problem has effective solutions. The aim of this paper is to estimate the expressive power of $\mathcal{LP}$-LTL by comparing it with some well known logics widely used in computer science for specification of reactive systems behaviour. We discovered that a restricted variant $\mathcal{LP}$-1-LTL of our logic is more expressive than LTL and another restricted variant $\mathcal{LP}$-n-LTL has the same expressive power as monadic second order logic S1S.

1 Introduction

Finite state transducers find applications in many branches of computer science, software engineering, computational lingustics. They also provide the most simple model of computation which is suitable for representation of reactive systems behaviour. At every step of computation a transducer takes at its input a letter from an input alphabet and produces a sequence of letters (a word) from another output alphabet. The input letters may be regarded as control signals received from the environment and the output letters may be viewed as basic actions performed by a reactive system. A behaviour of such a reactive system is characterized by a transformation of flows of control signals into compositions of basic actions carried out by the system.

To construct a reliable information processing system it is crucial to be sure at the very early stages of its designing that it will have a correct behaviour. In the case of a sequential reactive system we assume that it behaves correctly when it gives adequate responses to certain flows of control signals. Finite state transducers operating over semigroups of basic actions, introduced and studied in [18], proved to be a suitable formal model for representing such computations of sequential reactive system. But we need also some expressive and convenient for use specification language which is adequate to this formal model. Then one could develop verification algorithms for these models to solve such problems as equivalence checking, deductive verification, or model checking.

When verification of transducers is concerned, to the extent of our knowledge, no special purpose study of specification languages and model checking problem for this formal model of computing system has been conducted so far. We think that this is due to the following reason. The letters of input and output alphabets of a transducer can be regarded as valuations of some set of basic predicates. Therefore, a transducer can be viewed as but a some special representation of a labeled transition system (Kripke structure), and it is not worthy of any particular treatment.

But our viewpoint is quite different. A behaviour of a transducer displays itself in the correspondence between input and output words. A typical property of such behaviour to be checked is whether for every (some) input word from a given pattern a transducer outputs a word from another given pattern. When formally expressing the requirements of this kind one needs not only temporal operators to specify an order in which events occur but also some means to refer to such patterns. To this end Temporal Logics like LTL or CTL should be modified in such a way as to acquire an ability to express such correspondences between the sets of input words and the sets of output words. This could be achieved by supplying temporal operators with patterns as parameters. Every such pattern is a formal description (by means of automata, formal grammars, regular expressions, language equations, etc.) of a language L over an input alphabet $\mathcal{C}$. A basic property of output words can be also represented by a language P over an output alphabet $\mathcal{A}$. Then, for instance, an expression $\mathbf{G}_L P$ can be understood as the requirement that for every input word w from the language L the output word h of a transducer belongs to the language P.

The advantages of this approach are twofold. Such extensions of Temporal Logics makes it possible to express explicitly relationships between input and output words and thus specify desirable behaviours of transducers. Moreover, it also assimilates some well-known model checking techniques (see [1]) developed for conventional temporal logics. This idea was first implemented in [8] where an $\mathcal{LP}$-LTL specification language based on LTL temporal logic was introduced. Next in [5] this approach was extended to branching time logics and a variant of $\mathcal{LP}$-CTL was introduced and studied.

The aim of this paper is to estimate the expressive power $\mathcal{LP}$-LTL by comparing it with some well known logics widely used in computer science for specification of reactive systems behaviour. It is not possible to make such a comparative analysis in a straightforward way, since the semantics of conventional temporal logics is defined on the structures different from those used for $\mathcal{LP}$-LTL. To overcome this difficulty we introduce two fragments of $\mathcal{LP}$-LTL (namely, $\mathcal{LP}$-1-LTL and $\mathcal{LP}$-n-LTL). The semantics of both fragments are adapted to ω-words and they become comparable with Linear Temporal Logic (LTL) and the Second Order Monadic Logic of One Successor (S1S). We discovered that $\mathcal{LP}$-1-LTL is more expressive than LTL and that $\mathcal{LP}$-n-LTL has the same expressive power as S1S.

The paper is organized as follows. In Section 2 we define the concept of finite state transducer as a formal model of sequential reactive systems. Next we describe the syntax and the semantics of $\mathcal{LP}$-LTL as a formal language for specifying behaviour of sequential reactive systems. In Section 4 we introduce the fragments $\mathcal{LP}$-1-LTL and $\mathcal{LP}$-n-LTL of $\mathcal{LP}$-LTL and relate their semantics with ω-words. In this section we also establish the main results of the paper and sketch their proofs. In Section 5 we make a brief comparison of $\mathcal{LP}$-LTL with other extensions of LTL and outline some tasks for our further research.

2 Finite state transducers as models of reactive systems

Let $\mathcal{C}$ and $\mathcal{A}$ be finite sets of *signals* and *basic actions* respectively. A reactive computing system receives control signals from the environment and reacts to these signals by performing basic actions. Finite words over $\mathcal{C}$ are called *signal flows*; the set of all signal flows is denoted by $\mathcal{C}^*$. Finite words over $\mathcal{A}$ are called *compound actions*; they denote sequential compositions of basic actions. The set of all compound actions is denoted by $\mathcal{A}^*$. Given a pair of words u and v we write uv for their concatenation, and denote by ε the empty word.

Definition 1. *A* ***Finite State Transducer*** *(FST) over the alphabets $\mathcal{C}$ and $\mathcal{A}$ is a quintuple $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$, where 1) Q is a finite set of control states, 2) $q_{init} \in Q$ is an initial control state, and 3) $T \subseteq Q \times \mathcal{C} \times Q \times \mathcal{A}^*$ is a finite transition relation.*

Each tuple (q', c, q'', h) in T is called a *transition*: when a transducer is in a control state q' and receives a signal c, it changes its state to q'' and performs a compound action h. We denote such transition by $q' \xrightarrow{c,\,h} q''$. A behaviour of a FST Π can be defined in terms of runs and trajectories. A *run* of a FST Π is any infinite sequence of transitions: $run = q_1 \xrightarrow{c_1,\,h_1} q_2 \xrightarrow{c_2,\,h_2} \cdots \xrightarrow{c_n,\,h_n} q_{n+1}, \ldots$; if $q_1 = q_{init}$ then the run is called *initial run.*

Definition 2. *Given a compound action s_0 and a run of a FST Π we define a* ***trajectory*** *of FST Π as the pair $tr = (s_0, \alpha)$, where the sequence $\alpha = (c_1, s_1), (c_2, s_2), \ldots, (c_i, s_i), \ldots$, is such that $s_i = s_{i-1}h_i$ holds for every i, $i \geqslant 1$.*

A trajectory represents a possible scenario of a behaviour of a sequential reactive system in response to the signal flow $w = c_1c_2 \ldots c_i \ldots$ in the event that it has previously performed the compound action s_0. When *run* is an initial run of Π and $s_0 = \varepsilon$ then the corresponding trajectory is called *initial.* The set of all initial trajectories of a FST Π is denoted by $Tr(\Pi)$. By $tr|^i$ we mean a trajectory $(s_i, \alpha|^i)$, where $\alpha|^i = (c_{i+1}, s_{i+1}), (c_{i+2}, s_{i+2}), \ldots$ is a suffix of α.

3 $\mathcal{LP}$-LTL specification language

When designing sequential reactive systems one should be provided with a suitable formalism to specify the requirements for their desirable behaviour. Many requirements which refer to the correspondences between control flows and compound actions in the course of FST runs can be specified by means of Temporal Logics. When choosing a suitable temporal logic as a formal specification language of FST behaviours one should take into account two principal features of our model of sequential reactive systems:

1. since a FST operates over compound actions, the basic predicates must be interpreted over the set $\mathcal{A}^*$, and
2. since a behaviour of a FST depends not on the time flow itself but on a signal flow, temporal operators must be parameterized by descriptions of admissible signal flows.

To adapt traditional temporal logic formalism to these specific features of FST behaviours the authors of [8] introduced a new variant of Linear Temporal Logic (LTL). In general case one may be interested in checking the correctness of FST's responses to an arbitrary set of signal flows. Every set of control flows may be regarded as a language over the alphabet $\mathcal{C}$ of signals. Therefore, it is reasonable to supply temporal operators ("globally" **G**, "eventually" **F**, etc.) with certain descriptions of such languages as parameters. These languages will be called *environment behaviour patterns.*

A reactive system performs compound actions in response to control signals from the environment. Therefore, *basic predicates* used in LTL formulae may be viewed as some sets of such compound actions P, $P \subseteq \mathcal{A}^*$, i.e. languages over the alphabet $\mathcal{A}$. As in the case of environment behaviour patterns we may distinguish a certain class $\mathcal{P}$ of languages and use them as specifications of basic predicates. When these languages are used in temporal formulae then it will be assumed that they are defined constructively by means of automata, grammars, Turing machines, etc.

Thus, we arrive at the concept of $\mathcal{LP}$-variants of Temporal Logics.

Definition 3. *Select an arbitrary family of environment behaviour patterns $\mathcal{L}$ and a family of basic predicates $\mathcal{P}$. The set of $\mathcal{LP}$-LTL formulae are defined as follows:*

1. *each basic predicate P from $\mathcal{P}$ is a formula;*
2. *if φ_1, φ_2 are formulae then $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$ are formulae;*
3. *if φ_1 and φ_2 are formulae, $c \in \mathcal{C}$, and $L \in \mathcal{L}$ then $\mathbf{X}_c\varphi_1$, $\mathbf{Y}_c\varphi_1$, $\mathbf{F}_L\varphi_1$, $\mathbf{G}_L\varphi_1$, and $\varphi_1\,\mathbf{U}_L\varphi_2$ are formulae.*

*The **specification language** $\mathcal{LP}$-LTL is the set of all formulae as defined above.*

Now we introduce the semantics of $\mathcal{LP}$-LTL. The formulae are interpreted over the trajectories of FSTs. Let Π be a FST, and $tr = (s_0, \alpha)$ be a trajectory of Π such that $\alpha = (c_1, s_1), (c_2, s_2), \ldots, (c_i, s_i), \ldots$. Then for every formula ψ we write $tr \models \psi$ to denote the fact that the assertion ψ holds for the trajectory tr of Π.

Definition 4. *The **satisfiability relation** $\models$ is defined by induction on the height of formulae:*

1. $tr \models P \iff s_0 \in P$;
2. $tr \models \neg\varphi \iff$ *it is not true that* $tr \models \varphi$;
3. $tr \models \varphi_1 \wedge \varphi_2 \iff tr \models \varphi_1$ *and* $tr \models \varphi_2$;
4. $tr \models \mathbf{X}_c\varphi \iff c = c_1$ *and* $tr|^1 \models \varphi$;
5. $tr \models \mathbf{Y}_c\varphi \iff$ *either* $c \neq c_1$, *or* $tr|^1 \models \varphi$;
6. $tr \models \mathbf{F}_L\varphi \iff \exists i \geqslant 0\colon c_1c_2\ldots c_i \in L$ *and* $tr|^i \models \varphi$;
7. $tr \models \mathbf{G}_L\varphi \iff \forall i \geqslant 0\colon$ *if* $c_1c_2\ldots c_i \in L$ *then* $tr|^i \models \varphi$;
8. $tr \models \varphi\,\mathbf{U}_L\psi \iff \exists i \geqslant 0\colon c_1c_2\ldots c_i \in L$ *such that* $tr|^i \models \psi$ *and* $\forall j,\ 0 \leqslant j < i$, *if* $c_1c_2\ldots c_i \in L$ *then* $tr|^j \models \varphi$.

Observe, that operators $\mathbf{X}_c$ and $\mathbf{Y}_c$, as well as $\mathbf{F}_L$ and $\mathbf{G}_L$, are dual to each other. As usual, other Boolean connectives like $\vee, \rightarrow, \equiv$ may be defined by means of $\neg$ and $\wedge$. Some other LTL operators like, for example, $\mathbf{R}$ (release) or $\mathbf{W}$ (weak until) may be parametrized by environmental behaviour patterns in the same way.

Definition 5. *The **model checking problem for FSTs against $\mathcal{LP}$-LTL specifications** is as follows: given a FST Π and an $\mathcal{LP}$-LTL formula φ, check whether $tr \models \varphi$ holds for every initial trace tr from $Tr(\Pi)$.*

In [8] it was shown that model checking problem for $\mathcal{LP}$-LTL is decidable in double exponential time when environment behaviour patterns and basic predicates are specified by deterministic finite state automata. In the same way as it was made above the syntax and the semantics of $\mathcal{LP}$-CTL was introduced in [5]. In this paper it was proved that model checking problem for $\mathcal{LP}$-CTL can be solved in exponential time. Basic actions of FSTs may have a more sophisticated interpretation: they may be considered as generating elements of some semigroup. In [4] it was proved that model checking problem for FSTs operating over free commutative semigroups of actions is undecidable.

4 On the expressive power of some fragments of $\mathcal{LP}$-LTL

In the previous sections, we refer to FST, first of all, to emphasize the origin and purpose of $\mathcal{LP}$-LTL. By choosing different classes of languages âĂŞâĂŞfor defining environment behaviour patterns and basic predicates, we can vary the expressive possibilities of $\mathcal{LP}$-LTL. We would like to find out how widely the expressiveness of this logic can vary. The aim of this paper is to compare the expressive power of three logics LTL, S1S, and $\mathcal{LP}$-LTL used as specification languages of reactive systems.

Definition 6. *When the semantics of two logics $\mathscr{L}_1$ and $\mathscr{L}_2$ are defined on the same class of interpretations then we say that a formula ψ_1 from $\mathscr{L}_1$ is* ***equivalent*** *to a formula ψ_2 from $\mathscr{L}_2$ if $I \models \psi_1 \Leftrightarrow I \models \psi_2$ holds for every interpretation I. A logic $\mathscr{L}_1$ is* ***at least as expressive*** *as $\mathscr{L}_2$ ($\mathscr{L}_2 \preccurlyeq \mathscr{L}_1$ in symbols) if for any formula in $\mathscr{L}_2$ there exists an equivalent formula in $\mathscr{L}_2$. Two logics $\mathscr{L}_1$ and $\mathscr{L}_2$ are* ***equally expressive*** *($\mathscr{L}_2 \equiv \mathscr{L}_1$ in symbols) if $\mathscr{L}_2 \preccurlyeq \mathscr{L}_1$ and $\mathscr{L}_1 \preccurlyeq \mathscr{L}_2$. We say that $\mathscr{L}_1$ is* ***more expressive*** *than $\mathscr{L}_2$ ($\mathscr{L}_2 \prec \mathscr{L}_1$ in symbols) if $\mathscr{L}_2 \preccurlyeq \mathscr{L}_1$ and $\mathscr{L}_2 \not\equiv \mathscr{L}_1$.*

As it can be seen from these definitions, only those pairs of logics are comparable, the semantics of which are defined over the same class of interpretation. The semantics of LTL and S1S can be defined over ω-words. This makes LTL and S1S comparable, and, as it was shown in [14,16], $LTL \prec$ S1S. But the semantics of $\mathcal{LP}$-LTL is defined over another class of interpretations — trajectories, — that, in fact, may be viewed as pairs of ω-words. Therefore, to compare the expressive power of $\mathcal{LP}$-LTL, LTL and S1S we single out two fragments of $\mathcal{LP}$-LTL, the semantics of which can be defined over ω-words.

The formulae of the first fragment $\mathcal{LP}$-1-LTL are defined over 1-letter alphabet $\mathcal{C} = \{c\}$ of signals and an arbitrary finite alphabet of basic actions $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$. A class $\mathcal{L}$ of environment behaviour patterns used in the formulae of this fragment is the family of all regular languages over $\mathcal{C}$. A class $\mathcal{P}$ of basic predicates includes only n such predicates which are regular languages of the form $P_i = \{ha_i : h \in \mathcal{A}^*\}, 1 \leqslant i \leqslant n$. Then we say that ω-word $w = a_{i_1}a_{i_2}\dots a_{i_m}\dots$ satisfies a formula ψ' from $\mathcal{LP}$-1-LTL iff $tr' \models \psi'$, where $tr' = (\varepsilon, \alpha'_w)$, and $\alpha'_w = (c, a_{i_1}), (c, a_{i_1}a_{i_2}), \dots, (c, a_{i_1}a_{i_2}\dots a_{i_m}), \dots$.

The second fragment $\mathcal{LP}$-n-LTL differs from the first one only in that the alphabets $\mathcal{A}$ and $\mathcal{C}$ coincide, i.e. $\mathcal{C} = \mathcal{A} = \{a_1, a_2, \dots, a_n\}$. Then we assume that ω-word $w = a_{i_1}a_{i_2}\dots a_{i_m}\dots$ satisfies a formula ψ'' from $\mathcal{LP}$-n-LTL iff $tr'' \models \psi''$, where $tr'' = (\varepsilon, \alpha''_w)$, and $\alpha''_w = (a_{i_1}, a_{i_1}), (a_{i_2}, a_{i_1}a_{i_2}), \dots, (a_{i_m}, a_{i_1}a_{i_2}\dots a_{i_m}), \dots$.

To carry out a comparative analysis of the above-mentioned logics within the framework of common concepts and for the sake of clarity we give below an alternative definition of their semantics solely on the basis of ω-words. Let $\Sigma = \{a_1, a_2, \dots, a_n\}$ be a finite alphabet.

Definition 7. *An ω-**word** w is any mapping $w \colon \mathbb{N}_0 \to \Sigma$, where $\mathbb{N}_0$ is the set of non-negative integers.*

The set of all ω-words is denoted by Σ^ω. We use the notation $w|^k$ for the k-th suffix of w which is an ω-word v such that $v(i) = w(i+k)$ for every i, $i \geqslant 0$. We also write $w[0\dots k]$ to denote the k-th prefix of w which is a finite word $w(0)w(1)\dots w(k)$. By Reg_1 we mean the family of all regular languages over a 1-letter alphabet $\mathcal{C} = \{c\}$. Since we are only concerned with the expressive power of logics, the ways of specifying these languages âĂŞâĂŞare indifferent. When L is a language from Reg_1 we will write $i \in L$ instead of $c^i \in L$. By Reg_n we denote the family of all regular languages over the alphabet Σ.

The formulae of $\mathcal{LP}$-1-LTL are built of letters from Σ (they are regarded as atomic propositions) by means of Boolean connectives $\neg, \wedge$ and temporal operators $\mathbf{X}$, $\mathbf{F}_L$, $\mathbf{G}_L$, and $\mathbf{U}_L$ parameterized by languages L from Reg_1.

Definition 8. *The formal semantics of $\mathcal{LP}$-1-LTL formulae is defined through the* ***satisfiability relation*** $w \models \psi$ *on the set of ω-words:*

1. *for an atomic formula $a \in \Sigma$: $w \models a \iff w(0) = a$;*
2. $w \models \neg\varphi \iff$ *it is not true that* $w \models \varphi$*;*
3. $w \models \varphi \wedge \psi \iff w \models \varphi$ *and* $w \models \psi$*;*
4. $w \models \mathbf{X}\varphi \iff w|^1 \models \varphi$*;*
5. $w \models \mathbf{F}_L\varphi \iff \exists\, i \geqslant 0$ *such that* $i \in L$ *and* $w|^i \models \varphi$*;*
6. $w \models \mathbf{G}_L\varphi \iff \forall\, i \geqslant 0$ *if* $i \in L$ *then* $w|^i \models \varphi$*;*
7. $w \models \varphi\,\mathbf{U}_L\psi \iff \exists\, i \geqslant 0$ *such that* $i \in L$ *and* $w|^i \models \psi$*, and* $\forall\, j,\ 0 \leqslant j < i$*, if* $j \in L$ *then* $w|^j \models \varphi$*.*

An example of a formula in $\mathcal{LP}$-1-LTL is an expression $\mathbf{G}_{(cc)^*}a$. As it may be seen from the definition above, an ω-word w satisfies this formula iff $w(2i) = a$ for every $i,\ i \geqslant 0$.

It is easy to see that temporal logic LTL is a subset of $\mathcal{LP}$-1-LTL which allows the only regular language $\mathcal{C}^*$ for parameter of temporal operators. Thus, $LTL \preccurlyeq \mathcal{LP}$-1-$LTL$.

Theorem 1. $LTL \prec \mathcal{LP}$-1-LTL

Proof. (*Sketch*). In [16] it was shown that no LTL formula can express the property of an ω-word to have a given letter in every k-th position for any $k, k \geq 2$. As a minor innovation we give an alternative proof of this fact by means of Ehrenfeucht-Fraïssé games (see [2]). $\square$

The syntax of $\mathcal{LP}$-n-LTL formulae differs from that of $\mathcal{LP}$-1-LTL in three aspects:

1. the neXttime operator $\mathbf{X}$ has the dual counterpart $\mathbf{Y}$;
2. both operators $\mathbf{X}$ and $\mathbf{Y}$ are parametrized by letters from Σ: $\mathbf{X}_a,\ \mathbf{Y}_b$;
3. temporal operators $\mathbf{F}_L$, $\mathbf{G}_L$ and $\mathbf{U}_L$ are parametrized by regular languages from Reg_n.

Definition 9. *The* ***satisfiability*** *of $\mathcal{LP}$-n-LTL formulae on ω-words is defined as follows:*

1. *the satisfiability of atomic formulae, negation $\neg$ and conjunction $\wedge$ is defined exactly as in the case of $\mathcal{LP}$-1-LTL;*
2. $w \models \mathbf{X}_c\varphi \iff w(0) = c$ *and* $w|^1 \models \varphi$*;*
3. $w \models \mathbf{Y}_c\varphi \iff w(0) \neq c$ *or* $w|^1 \models \varphi$*;*
4. $w \models \mathbf{F}_L\varphi \iff \exists\, i \geqslant 0$ *such that* $w[0 \dots i] \in L$ *and* $w|^i \models \varphi$*;*
5. $w \models \mathbf{G}_L\varphi \iff \forall\, i \geqslant 0$ *if* $w[0 \dots i] \in L$ *then* $w|^i \models \varphi$*;*
6. $w \models \varphi\,\mathbf{U}_L\psi \iff \exists\, i \geqslant 0$ *such that* $w[0 \dots i] \in L$ *and* $w|^i \models \psi$*, and* $\forall\, j,\ 0 \leqslant j < i$*, if* $w[0 \dots j] \in L$ *then* $w|^j \models \varphi$*.*

It is clear that $\mathcal{LP}$-n-LTL is at least as expressive as $\mathcal{LP}$-1-LTL. We will show that it has the same expressive power as $S1S$. It is well-known (see a survey in [12]) that three following assertions concerning an ω-language L are equivalent:

– there exists a S1S formula Φ such that $L = \{w : w \models \Phi\}$,
– L is an ω-regular language,
– L is recognizable by a Buchi (Rabin, Muller, Street) automaton.

Therefore, instead of dealing with S1S we compare the expressive power of $\mathcal{LP}$-n-LTL with that of finite state ω-automata.

Definition 10. *An* ***ω-automaton*** *is a quintuple $A = (\Sigma, Q, Q_0, \Delta, \mathcal{F})$, where Σ is a finite alphabet, Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $Q_0 \in Q$ is a set of initial states, $\mathcal{F}$ is an* acceptance rule.

An ω-automaton is called *deterministic* if $Q_0 = \{q_0\}$, and for each pair of a state q and a letter a there exists only one state $q' \in Q$, such that (q, a, q') is a transition of the automaton. A *run* of an automaton A on an infinite word w is a map $\rho\colon \mathbb{N}_0 \to Q$ such that

1. the first state is initial, i. e. $\rho(0) \in Q_0$, and
2. for each i the transition from $\rho(i)$ to $\rho(i+1)$ by a letter $w(i)$ is done with respect to Δ, i. e. $(\rho(i), w(i), \rho(i+1)) \in \Delta$.

By $\inf(\rho)$ we denote a set of all those states that occur infinitely often in ρ.

In this paper we consider Büchi automata and Muller automata which differ in their acceptance rules. An acceptance rule of a (generalized) Büchi automaton B is a subset a family $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$, of subsets $F_i,\ F_i \subseteq Q,\ 1 \leqslant i \leqslant m$, of states of B.

Definition 11. *A Büchi automaton B accepts an ω-word w iff there exists a run ρ of B on w, such that $\inf(\rho) \cap F_i \neq \varnothing$ for each i, $1 \leqslant i \leqslant m$.*

An acceptance rule of a Muller automaton M is a family $\mathcal{F} = \{E_1, E_2, \ldots, E_m\}$ of subsets $E_i,\ E_i \subseteq Q,\ 1 \leqslant i \leqslant m$, of states of M.

Definition 12. *A Muller automaton M accepts ω-word w iff there exists a run ρ of M on w, such that $\inf(\rho) \in \mathcal{F}$. A set of ω-words accepted by an automaton A is denoted by $L(A)$.*

Theorem 2. $\mathcal{LP}$-*n*-$LTL \equiv$ S1S.

Proof. (*Sketch*) It is sufficient to show that the properties of ω-words expressed by $\mathcal{LP}$-*n*-LTL formulae are exactly all ω-regular languages. It can be proved by constructing, for every $\mathcal{LP}$-*n*-LTL formula ψ, a nondeterministic Büchi automaton B_ψ such that $L(B_\psi) = \{w \mid w \models \psi\}$. On the other hand, for every deterministic Muller automaton M it there can be constructed such an $\mathcal{LP}$-*n*-LTL formula ψ_M that $L(M) = \{w \mid w \models \psi_M\}$. □

5 Related papers and conclusion

Since the publication of the paper [3] when the expressive capabilities of LTL logic were fully realized, several attempts have been made to increase its expressiveness while preserving the complexity of decision procedures. The authors of [6] defined a Process Logic which subsumes a number of other logics (Propositional Dynamic Logic (PDL), ParikhâĂŹs $SOAPL$, NishimuraâĂŹs process logic, and LTL) but but it turned out that the complexity of decision problems for this logic is very high. In [11] quantifiers on basic propositions were added to the syntax of LTL. In [16] right-linear grammar patterns were offered to define new temporal operators. The same kind of temporal patterns but specified by means of finite state automata were introduced in [9,13]. Of course, fixed-point extension μ-LPL has not been ignored, it is the topic of research in [15]. The authors of [10] presented a Regular LTL which generalizes LTL with the ability to use regular expressions in the context of Until temporal operator. For all these extensions (but μ-LTL) it was proved that they have the same expressiveness as S1S and retain PSPACE-complexity of satisfiability checking problem. In our case we did not set ourselves the goal of merely expanding the expressive possibilities of LTL; we just tried to make LTL more adequate for describing the behaviour of reactive systems. Nevertheless, as it follows from Theorem 2 we achieve almost the same effect as the other extensions of LTL in [9,11,13,15,16].

The idea of providing parametrization of temporal operators is also not new: almost the same kind of parametrization is used in Dynamic LTL [7]. In fact, the fragment $\mathcal{LP}$-*n*-LTL is almost the same as $DLTL$ defined in [7]. But our extension of LTL differs from that which was developed in [7] in that basic predicates are also parameterized. Such a parametrization gives us a possibility to introduce the fragment $\mathcal{LP}$-1-LTL which could display some interesting features. Every extension of LTL introduced in [7,11,13,15,16] is as expressive as LTL or S1S. But $\mathcal{LP}$-1-LTL is more expressive than pure LTL and it is doubtful that it has the same expressive power as $\mathcal{LP}$-n-LTL. If our hypothesis turned out to be correct, then we could for the first time discover such an extension of LTL, which occupies an intermediate position between $FO(<)$ and S1S. Another line of further research is to estimate how much (or less) succinct are these fragments of $\mathcal{LP}$-1-LTL in comparison with other extensions of LTL.

The authors of the article thank the anonymous reviewers for their valuable comments and advice on improving the article. This work was supported by the Russian Foundation for Basic Research, Grant N 18-01-00854.

References

1. E. M. Clarke, Jr., O. Gramberg, D. A. Peled. *Model Checking.* MIT Press, 1999.
2. K. Etessami, T. Wilke. *An Until Hierarchy and Other Applications of an Ehrenfeucht-Fraisse Game for Temporal Logic.* Information and Computation, v. 160, p. 88-108. Elsevier, 2000.
3. D. Gabbay, A. Pnueli, S. Shelach, J. Stavi. *The temporal analysis of fairness.* Proceedings of 7-th ACM Symposium on Principles of Programming Languages, 1980, p. 163-173.
4. A. Gnatenko, V.A. Zakharov. *On the complexity of verification of finite state machines over commutative semigroups.* Proceedings of the 18-th International Conference "Problems of Theoretical Cybernetics" (Penza, June 20-24, 2017), p. 68-70.
5. A. Gnatenko, V.A. Zakharov. *On the model checking of finite state transducers over semigroups.* To be published in Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering, 2018.
6. D. Harel, D. Kozen, R. Parikh. *Process logic: Expressiveness, decidability, completeness.* Journal of Computer and System Science. v.25(2), 1982, p. 144-170.
7. J.G. Henriksen, P.S. Thiagarajan. *Dynamic linear time temporal logic.* Annals of Pure and Applied Logic, 1999, v. 96, p. 187-207
8. D.G. Kozlova, V.A. Zakharov. *On the model checking of sequential reactive systems.* Proceedings of the 25th International Workshop on Concurrency, Specification and Programming (CS&P 2016), CEUR Workshop Proceedings, 2016, vol. 1698, p. 233-244.
9. O. Kupferman, N. Piterman, M.Y. Vardi. *Extended Temporal Logic Revisited.* Proceedings of 12-th International Conference on Concurrency Theory, 2001, p. 519-535.
10. M. Leucker, C Sanchez. *Regular Linear Temporal Logic.* Proceedings of the 4-th International Colloquium on Theoretical Aspects of Computing, 2007, p. 291-305.
11. Z. Manna, P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* Logic of Programs. 1981, p. 253-281.
12. W. Thomas. *Automata on infinite objects.* In Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics, 1990, p. 133-192.
13. M.Y. Vardi, P. Wolper. *Yet Another Process Logic.* Logic of Programs, 1983, p. 501-512.
14. M.Y. Vardi, P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* Proceedings of the First Symposium on Logic in Computer Science, 1986, p. 322-331.
15. M.Y. Vardi. *A Temporal Fixpoint Calculus.* Proceedings of the 15-th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1988, p. 250-259.
16. P. Wolper. *Temporal Logic Can Be More Expressive.* Information and Control, 1983, v. 56, N 1/2, p. 72-99.
17. P. Wolper, B. Boigelot. *Verifying systems with infinite but regular state spaces.* Proceedings of the 10-th Int. Conf. on Computer Aided Verification (CAV-1998). LNCS. **1427** (1998), p. 88-97.
18. V. A. Zakharov. *Equivalence checking problem for finite state transducers over semigroups.* Proceedings of the 6-th International Conference on Algebraic Informatics (CAI-2015), p. 208-221, LNCS 9270, 2015.

Polyprograms and polyprogram bisimulation

Sergei Grechanik

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
`sergei.grechanik@gmail.com`

Abstract. A polyprogram is a generalization of a program which admits multiple definitions of a single function. Such objects arise in different transformation systems, like the Burstall-Darlington framework or equality saturation. In this paper we introduce the notion of a polyprogram in a non-strict first-order functional language. We define denotational semantics for polyprograms and describe some possible transformations of polyprograms. We also introduce the notion of polyprogram bisimulation which enables a powerful transformation called merging by bisimulation, corresponding to proving equivalence of functions by induction or coinduction.

1 Introduction

Many program transformation methods can be seen as special cases of the Burstall-Darlington framework [2]. The idea behind this framework consists in viewing a program as a set of equations and then transforming this set by inferring new equations. Such a set of equations is essentially a program without the uniqueness constraint on the definitions of its functions, thus we propose to call it a *polyprogram* (short for "polyvariant program", a term coined by Bulyonkov).

Equality saturation [7] may also be considered an instance of the Burstall-Darlington framework if we restrict ourselves to so-called *decomposed polyprograms*. In decomposed polyprograms every definition has a very simple form containing only one nontrivial language construct, thus decomposed polyprograms are essentially closer to ASTs and E-PEGs (E-PEG is a Program Expression Graph with an Equivalence relation on nodes [7]), definitions of polyprograms corresponding to nodes and outgoing edges of E-PEGs, and functions corresponding to classes of node equivalence. DecomposedD polyprograms can be represented as graphs, so they are better for implementation and formulation of transformations. Every complex definition can be split into several simple definitions by introducing intermediate functions, so every polyprogram can be transformed into a decomposed one.

One of the transformation rules of the Burstall-Darlington framework is called redefinition. It allows replacing one function for another if they have isomorphic recursive definitions. In this paper we show how this rule can be formulated using the notion of polyprogram bisimulation, which has the benefit of dealing with situations when the definitions are not exactly isomorphic (e.g. the functions are equal only up to argument permutation). Thus, we prefer to call this transformation rule *merging by bisimulation*.

This paper is a continuation of work on equality saturation for functional languages [4]. The main contributions of this paper are:

1. articulation of the notion of a polyprogram;
2. a polyprogram-based formulation of equality saturation which shows the connection between equality saturation and the Burstall-Darlington framework;

3. the notion of polyprogram bisimulation and an algorithm for enumerating polyprogram bisimulations.

The paper is structured as follows: first of all, we describe the language we will use throughout the paper and give the definitions of a polyprogram and a decomposed polyprogram in this language (Section 2), then we show some basic transformation rules (Section 3), and after that we introduce the notion of polyprogram bisimulation and present an algorithm enumerating bisimulations (Section 4).

2 Polyprograms

In this paper we will use a simple first-order language. We will denote variables with letters x, x_i (from $\mathcal{X}$), functions names with f, f_i (from a set of functional symbols $\mathcal{F}$), and constructors with C, C_i. A set of functional symbols is just a set F equipped with an arity function $arity : F \to \mathbb{N}$.

Definition 1. A *polyprogram* (in our language) is a set of definitions of the form $f(x_1..x_n) \equiv e$ where e has the following form[1]:

$$e ::= x \mid f(e_1..e_m) \mid C(e_1..e_m) \mid \textbf{case } e_0 \textbf{ of } \{ \overline{C_i(\overline{x_{ij}}) \ \to \ e_i;} \}$$

where there is no variable duplication in case patterns and in left hand sides of definitions.

In a polyprogram each function is allowed to have any number of definitions. The intention is that such definitions should be semantically equal, but may be different performance-wise. Here is an example of a polyprogram:

$$\begin{aligned}
&\text{not}(t) &&\equiv \textbf{case } t \textbf{ of } \{F \to T; T \to F\} \\
&\text{even}(x) &&\equiv \textbf{case } x \textbf{ of } \{Z \to T; S(y) \to \text{odd}(y)\} \\
&\text{even}(x) &&\equiv \text{not}(\text{odd}(x)) \\
&\text{odd}(x) &&\equiv \textbf{case } x \textbf{ of } \{Z \to F; S(y) \to \text{even}(y)\} \\
&\text{odd}(x) &&\equiv \text{not}(\text{even}(x))
\end{aligned}$$

Definition 2. A polyprogram in decomposed form, or just *decomposed polyprogram*, is a polyprogram such that all right hand sides of its definitions have the following form:

$$\begin{aligned}
&e ::= r \mid x \mid f(r_1..r_m) \mid C(r_1..r_m) \mid \textbf{case } r_0 \textbf{ of } \{ \overline{C_i(\overline{x_{ij}}) \ \to \ r_i;} \} \\
&r ::= f(x_1..x_l), \text{ where all } x_j \text{ differ from each other}
\end{aligned}$$

This form is not very human-readable, but it is better suited for reasoning and implementation. Consider the following polyprogram consisting of one definition:

$$f(x, z) \equiv \textbf{case } x \textbf{ of } \{Z \to Z; S(y) \to f(y, y)\}$$

To transform it into a decomposed polyprogram, we have to factor out subexpressions x, Z, y, and $f(y, y)$ (the last one because it has variable duplication). This gives us the following decomposed polyprogram:

$$\begin{aligned}
&f(x, z) \equiv \textbf{case } id(x) \textbf{ of } \{Z \to g(); S(y) \to h(y)\} \\
&id(x) \equiv x \\
&g() \equiv Z \\
&h(y) \equiv f(id(y), id(y))
\end{aligned}$$

[1] $\overline{e\langle i\rangle}$ expands to $e\langle 1\rangle..e\langle n\rangle$ for some $n = \max i$

2.1 Polyprogram semantics

It is straightforward to define denotational semantics for polyprograms. However, instead of considering only the least fixed point, we will consider all fixed points, or models, because this makes semantics compositional, i.e. we can replace polyprogram fragments with semantically equivalent fragments without changing the meaning of the whole polyprogram.

Let A be the greatest fixed point of the following equation:

$$A = \{C(a_1..a_n) \mid a_i \in A,\ C \text{ is a constructor}\} \cup \{\bot\}.$$

Let D be the set of continuous functions over A of arbitrary arity (i.e. $D = \bigcup_n [A^n \to A]$). Now let's define the notion of an interpretation.

Definition 3. Let P be a polyprogram with the set of function names F. Then an *interpretation* of P is a function $\eta : F \to D$ such that $arity(\eta(f)) = arity(f)$.

Now let's define the valuation of a term t given an interpretation η and a valuation of variables $\nu : \mathcal{X} \to A$, written $[\![t]\!]_{\eta,\nu}$:

$$[\![x]\!]_{\eta,\nu} = \nu(x)$$
$$[\![f(e_1, \ldots, e_n)|]\!]_{\eta,\nu} = \eta(f)([\![e_1]\!]_{\eta,\nu} \ldots [\![e_n]\!]_{\eta,\nu})$$
$$[\![C(e_1..e_n)]\!]_{\eta,\nu} = C([\![e_1]\!]_{\eta,\nu} \ldots [\![e_n]\!]_{\eta,\nu})$$
$$[\![\textbf{case } e_0 \textbf{ of } \{\overline{C_i(y_1..y_m) \to e_i}\}]\!]_{\eta,\nu} = [\![e_k]\!]_{\eta,\nu\{y_i \to a_i\}}$$
$$\text{where } [\![e_0]\!]_{\eta,\nu} = C_k(a_1..a_m) \text{ for some } k \in \{1..\max\ i\}$$

Given a definition d, its valuation $[\![d]\!]_\eta$ is a boolean defined as follows:

$$[\![e_1 \equiv e_2]\!]_\eta = (\forall \nu. [\![e_1]\!]_{\eta,\nu} = [\![e_2]\!]_{\eta,\nu})$$

Definition 4. An interpretation μ is called a *model* of a polyprogram P if for every definition $d \in P$, $[\![d]\!]_\mu$ is true.

3 Polyprogram transformation rules

According to both the Burstall-Darlington framework and equality saturation, polyprograms should be transformed with some rules which add new function definitions to a polyprogram. After that a new program may be extracted from the polyprogram by choosing a single definition for each function.

We will write rules as $P_1 \mapsto P_2$. Application of such a rule to a polyprogram consists in replacing the subpolyprogram corresponding to the left hand side up to function and variable renaming with the right hand side. We will assume that rules may add new definitions and functions, and also remove some definitions

We will consider only a couple of basic rules in two different styles: in the style of the Burstall-Darlington framework and in the style of equality saturation. The latter one assumes that polyprograms are in decomposed form.

3.1 Rules in the style of the Burstall-Darlington framework

Unfolding

$$\left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \end{array} \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \\ f(\overline{x_i}) \equiv E\langle H\{\overline{y_j \mapsto e_j}\}\rangle \end{array} \right\}$$

This rule substitutes a function call with the function's body.

Folding

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv H\langle g(\overline{z_j})\rangle \end{array}\right\}$$

This rule does the inverse: it replaces an instance of some function's body with a call of this function. Note that it is less powerful than the corresponding rule from the paper by Burstall and Darlington.

3.2 Rules in the style of equality saturation

Decomposed polyprograms are very similar to E-PEGs from the work of Tate et al. on equality saturation. They enable more effective sharing of subexpressions, which is crucial for big polyprograms, especially when a simple heuristic-free rule application strategy is used, as in equality saturation.

However, rules in the form from the previous subsection cannot be applied to decomposed polyprograms simply because their right hand sides are not in decomposed form. The easiest solution is to rewrite rules in such a way that both their sides are in decomposed form. In this case rules will not only be applicable to decomposed polyprograms but also will preserve them in decomposed form.

Transitivity with symmetry

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv g(\overline{y_j}) \end{array}\right\}$$

This rule is analogous to folding. It infers function equivalence from their having coinciding definitions.

Congruence

$$\left\{\begin{array}{l} g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \\ D\langle g(\overline{e_j})\rangle \end{array}\right\} \mapsto \left\{\begin{array}{l} g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \\ D\langle h(\overline{e_{\theta(j)}})\rangle \end{array}\right\}$$

This rule allows propagating information about function equivalence by replacing one function with another. This rule is best applied to every call site of the function being replaced at once: in this case we can simply remove the old function from the polyprogram. We will call such a procedure *merging by congruence* since the functions are effectively merged.

Deduplication

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ f(\overline{x_i}) \equiv E \end{array}\right\} \mapsto \left\{ f(\overline{x_i}) \equiv E\right\}$$

This rule is used after merging by congruence to remove coinciding definition of a function. Its only purpose is to reduce memory consumption. The three aforementioned rules together implement congruence closure [5] which lies at the heart of equality saturation.

The unfolding rule breaks apart into several simpler rules. For brevity we show only one of these rules. The most interesting thing is that we don't need a separate operation for substitution into an expression ($E\{x \mapsto e\}$) since non-elementary function calls play the role of explicit substitutions.

Unfolding of function consisting of a function call

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \\ f(\overline{x_i}) \equiv g(\overline{q_k(\overline{x_i})}) \\ \hline q_k(\overline{x_i}) \equiv d_k(\overline{e_j(\overline{x_i})}) \end{array}\right\}$$

4 Polyprogram bisimulation

Merging by bisimulation is a generalization of the redefinition rule from the Burstall-Darlington framework. Consider the following polyprogram:

$$\begin{array}{l} f() \equiv S(f()) \\ g() \equiv S(h()) \\ h() \equiv S(g()) \end{array}$$

The goal is to infer $f() \equiv g()$. Turns out, this cannot be done directly with the simple rules mentioned above, so we need something more powerful. In this case the definitions of these functions are not even isomorphic, so we need a more general relation than isomorphism, which we will call a bisimulation because it remotely resembles the notion of bisimulation for labeled transition systems.

4.1 The notion of polyprogram bisimulation

First of all, let's give some auxiliary definitions. If θ is a function from $\{1..m\}$ to $\{1,\ldots,n\}$ (which we will write just as $\theta : m \to n$) then it can be applied to any functional symbol to permute, omit and duplicate its parameters. We will write this application simply as θf and will use the following reduction rule:

$$(\theta e_0)(e_1..e_n) \rightsquigarrow e_0(e_{\theta(1)}..e_{\theta(m)})$$

Definition 5. A *morphism of functional symbols* from F_1 to F_2 is a function ϕ that maps each functional symbol $f \in F_1$ to a pair $\phi(f) = (\theta, h)$ where $h \in F_2$ and $\theta : \mathit{arity}(h) \to \mathit{arity}(f)$.

Morphisms of functional symbols can be applied to definitions and polyprograms. If d is a definition then $\phi(d)$ is obtained by replacing all function names f in d with θh, where $(\theta, h) = \phi(f)$, with subsequent normalization with respect to $\rightsquigarrow$. If P is a polyprogram then $\phi(P) = \{\phi(d) \mid d \in P\}$.

Note that even if P is a polyprogram, $\phi(P)$ is not necessarily a polyprogram, because ϕ may introduce variable duplication in left hand sides of definitions. We will call such objects *quasipolyprograms.*

We will call two definitions *α-equivalent*, written $d_1 \approx d_2$, if they are equal up to variable renaming. We will use the notation $P_1 \sqsubseteq P_2$ if P_1 is a subpolyprogram of P_2 up to α-equivalence of its definitions.

Definition 6. A *polyprogram bisimulation* over a polyprogram P is a polyprogram B with two morphisms of functional symbols ϕ and ψ such that $\phi(B) \sqsubseteq P$, $\psi(B) \sqsubseteq P$, and if a function $f \in B$ has no definitions then $\phi(f) = \psi(f)$.

Polyprogram bisimulations are useful for proving equivalence of functions (and subsequently merging them, the transformation we call *merging by bisimulation*) using the following theorem.

Theorem 1. Let B with ϕ and ψ be a polyprogram bisimulation over P. If for every interpretation ν of B's functions without definitions there is only one model μ that coincides with ν on B's functions without definitions then it is possible to add into P definitions of the following form for each function $f \in B$ without changing the set of models of P:

$$g(x_{\theta(1)}..x_{\theta(m)}) \equiv h(x_{\xi(1)}..x_{\xi(n)})$$

where $(\theta, g) = \phi(f)$, $(\xi, h) = \psi(f)$, $m = arity(g)$, $n = arity(h)$.

Not every bisimulation is good enough for merging by bisimulation, because it must also satisfy the model uniqueness property. To test this property some decidable sufficient conditions may be used, like structural and guarded recursion [1], or the presence of ticks [6]. This topic is out of scope of this paper.

4.2 Enumerating bisimulations

Here we will assume that polyprograms are in decomposed form. There is an infinite number of polyprogram bisimulations, so we are going to present an algorithm that produces an infinite stream polyprogram bisimulations. It can be informally outlined in the following way: first enumerate *prebisimulations*, quasipolyprograms that are precursors of bisimulations, and then transform each prebisimulation into a polyprogram bisimulation if possible.

Prebisimulations will be quasipolyprograms consisting of products of definitions of the original polyprogram. The notion of a product of two definitions *def-product*(d_1, d_2) is formally specified as follows.

Definition 7. Let d and d' be two decomposed definitions with the same language construct, the same number of function calls and the same number of variables bound by corresponding patterns. Assume also that corresponding variables in corresponding patterns of the two definitions have coinciding names, and also if the right hand sides have the form x then x is the same for both definitions, but there are no more variable collisions between the definitions (these requirements may be satisfied by applying α-conversion). Then the *product of these definitions* is a definition which is presented as follows:

$$\begin{aligned}&\textit{def-product}(f(x_1, \ldots, x_n) \equiv x_i, f'(y_1, \ldots, y_m) \equiv y_j) = \\ &\quad = \langle f, f' \rangle (x_1 \ldots x_n, y_1 \ldots y_{j-1}, x_i, y_{j+1} y_m) \equiv x_i\end{aligned}$$

$$\begin{aligned}\textit{def-product}(f(\overline{x}) &\equiv h(g_1(\overline{x}_1), \ldots, g_n(\overline{x}_n)), \\ f'(\overline{y}) &\equiv h'(g'_1(\overline{y}_1), \ldots, g'_n(\overline{y}_1))) = \\ &= \langle f, f' \rangle (\overline{x}, \overline{y}) \equiv \langle h, h' \rangle (\langle g_1, g'_1 \rangle (\overline{x}_1, \overline{y}_1), \ldots)\end{aligned}$$

$$\begin{aligned}\textit{def-product}(f(\overline{x}) &\equiv C(g_1(\overline{x}_1), \ldots, g_n(\overline{x}_n)), \\ f'(\overline{y}) &\equiv C(g'_1(\overline{y}_1), \ldots, g'_n(\overline{y}_1))) = \\ &= \langle f, f' \rangle (\overline{x}, \overline{y}) \equiv C(\langle g_1, g'_1 \rangle (\overline{x}_1, \overline{y}_1), \ldots)\end{aligned}$$

$$\begin{aligned}\textit{def-product}(f(\overline{x}) &\equiv \mathbf{case}\ g_0(\overline{x}_0)\ \mathbf{of}\ \{C_1(\overline{z}_1) \to g_1(\overline{x}_1); \ldots\}, \\ f'(\overline{y}) &\equiv \mathbf{case}\ g'_0(\overline{y}_0)\ \mathbf{of}\ \{C_1(\overline{z'}_1) \to g'_1(\overline{x'}_1); \ldots\}) = \\ &= \langle f, f' \rangle (\overline{x}, \overline{y}) \equiv \mathbf{case}\ g_0(\overline{x}_0, \overline{x'}_0)\ \mathbf{of} \\ &\quad \{C_1(\overline{z}_1) \to \langle g_1, g'_1 \rangle (\overline{w}_1, \overline{w'}_1\{\overline{z'}_1 \mapsto \overline{z}_1\}); \ldots\}\end{aligned}$$

Here $\langle f, g\rangle$ denotes a unique functional symbol corresponding to f and g with arity $arity(f)+arity(g)$.

Its idea is to combine every pair of corresponding functions into a single one with their parameter lists concatenated, e.g.:

$$\begin{aligned}
&\textit{def-product}((f(x) \equiv x), (g(y) \equiv y)) = (\langle f, g\rangle(x, x) \equiv x)\\
&\textit{def-product}((f(x, z) \equiv \textbf{case } h() \textbf{ of } \{S(y) \to g(x, y)\}),\\
&(f'(x) \equiv \textbf{case } h'(x) \textbf{ of } \{S(y) \to g'(y, x)\})) =\\
&= (\langle f, f'\rangle(x, z, x') \equiv \textbf{case } \langle h, h'\rangle(x') \textbf{ of } \{S(y) \to \langle g, g'\rangle(x, y, y, x')\})
\end{aligned}$$

A function enumerating prebisimulations in the form of trees with back edges growing from a pair of functions is formally defined as follows.

$$\begin{aligned}
&prebisimulations(P, f, f') = B\\
&\qquad \text{where } (\_, B) = prebisimulations'(P, f, f', \{\})
\end{aligned}$$

$$\begin{aligned}
&prebisimulations'(P, f, f', history)\\
&\quad = \{(f_{new}, \{\}) \mid f = f'\}\\
&\quad \cup \{(f_{old}, \{\}) \mid (f, f', f_{old}) \in history\}\\
&\quad \cup \{(f_{new}, \{q'\} \cup B_1 \cup \ldots \cup B_n) \mid\\
&\qquad\qquad d = (f(\ldots) \equiv L(\ldots)) \in P,\\
&\qquad\qquad d' = (f'(\ldots) \equiv L(\ldots)) \in P,\\
&\qquad\qquad \text{such that } def-product(d, d') \text{ is defined},\\
&\qquad\qquad q = def-product(d, d'),\\
&\qquad\qquad (\langle f, f'\rangle(\ldots) \equiv L((\lambda\overline{y} \to \langle g_1, g_1'\rangle(\ldots)), \ldots)) = q,\\
&\qquad\qquad history' = history \cup \{(f, f', f_{new})\},\\
&\qquad\qquad (h_i, B_i) \in prebisimulations'(P, g_i, g_i', history'),\\
&\qquad\qquad q' \text{ is } q \text{ with } \langle f, f'\rangle \text{ replaced with } f_{new}\\
&\qquad\qquad\qquad \text{and } \langle g_i, g_i'\rangle \text{ replaced with } h_i\}\\
&\quad \text{where } f_{new} = \langle f, f', l\rangle \text{ where } l \text{ is a fresh unique label}\\
&\quad f_{new}\ textrmhasarity\ arity(f) + arity(f')
\end{aligned}$$

Its idea is to traverse the polyprogram P in depth-first order simultaneously from the functions f and f'. A branch may be finished if we encounter a pair of coinciding functions (reflexivity) or a pair of functions which we have already visited (folding). If we do not finish the branch then we choose a pair of definitions of these functions such that the product of these definitions is defined, and descend to pairs of corresponding functions in their right hand sides.

Now let's define two morphisms, $\pi_1(\langle f_1, f_2, l\rangle) = (\gamma_1, f_1)$ where $\gamma_1(i) = i$, and $\pi_2(\langle f_1, f_2, l\rangle) = (\gamma_2, f_2)$ where $\gamma_2(i) = i+arity(f_1)$. A prebisimulation with π_1 and π_2 is not a bisimulation mainly because it may contain duplicate variables introduced by $def-product$. To transform it to a polyprogram bisimulation these variables should be merged. To do so, we will propagate variable equivalence with the following transformation:

Definition 8. Let Q be a quasipolyprogram and $f(x_1..x_n)$ be a term from some of Q's definitions such that x_i and x_j coincide. The following transformation is called *variable equivalence propagation step*: for every definition containing a term $f(y_1..y_n)$ replace y_j with y_i in the whole definition.

If variable equivalence is fully propagated then parameter positions of every function may be partitioned into equivalence classes such that for every term $f(x_1..x_n)$, x_i and x_j coincide iff i and j are from the same class. Then the arity of the function may be reduced by collapsing parameters from the same class into one. The resulting quasipolyprogram will be a polyprogram. Note that we still need to check if the resulting polyprogram B is a polyprogram bisimulation, because variable equivalence propagation may equate too much, resulting in $\pi'_i(B)$ not being subpolyprograms of P.

5 Conclusion

In this paper we have introduced the notions of a polyprogram, a decomposed polyprogram and polyprogram bisimulation. Polyprograms are essentially systems of equations from the Burstall-Darlington framework [2]. Decomposed polyprograms are closer to AST and need transformation rules formulated in a different style. Decomposed polyprograms can be used to implement equality saturation for functional languages [7,4], which indicates that equality saturation may be seen as another instance of the Burstall-Darlington framework.

Our definition of polyprogram bisimulation is not relational and instead it is based on the notion of a span, although it can be reformulated in relational form. It should be noted that polyprogram bisimulation and LTS bisimulation are quite different since nondeterminism in polyprograms is not related to nondeterminism in LTS. Polyprogram bisimulation is used to implement merging by bisimulation, a generalization of the redefinition rule.

We have also presented a bisimulation enumeration algorithm which enumerates polyprogram bisimulations of a specific form. This algorithm is related to finding intersection of two languages of term equalities [3]. It is also structurally very similar to supercompilation [8].

This work was supported by the Russian Foundation for Basic Research, Grant N 18-31-00412.

References

1. Abel, A., Altenkrich, T.: A predicative analysis of structural recursion. Journal of Functional Programming 12(1), 1–41 (2002).
2. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24, 44–67 (1977)
3. Emelyanov, P.: Analysis of equality relationships for imperative programs. CoRR abs/cs/0609092 (2006), http://arxiv.org/abs/cs/0609092
4. Grechanik, S.: Inductive prover based on equality saturation (extended version). In: Klimov, A., Romanenko, S. (eds.) Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation. pp. 26–53. Pereslavl Zalessky: Publishing House "University of Pereslavl", Pereslavl-Zalessky, Russia (July 2014)
5. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. JACM: Journal of the ACM 27(2), 356–364 (1980)
6. Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Trans. Program. Lang. Syst. 18(2), 175–234 (1996)
7. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. SIGPLAN Not. 44, 264–276 (January 2009), http://doi.acm.org/10.1145/1594834.1480915
8. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(3), 292–325 (1986)

On Safety of Unary and Non-Unary Inflationary Fixed Point Operators

Sergey M. Dudakov

CS Faculty, Tver State University, 33, Zhelyabova str., Tver, 170100, Russia
sergeydudakov@yandex.ru

Abstract. We investigate unary inflationary fixed point (IFP) operators and establish some differences between the unary IFP-operators and general ones.

1 Introduction

The modern SQL language (see [8]) supports first order features (see [1]), and some extensions. For example, recursive queries correspond exactly to inflationary fixed point (IFP) operators (see [5]). Also SQL allows to use universe functions and relations so we have a finite structure (database tables) embedded into an infinite universe (see [6]).

But if recursive queries contain universe functions and relations then its execution can fall into an infinite loop. This distinguishes recursive queries from traditional ones when any query can be calculated after finitely many steps. Such queries are called *safe*, and the property is called *safety*. Really the infinitely growing successor function allows to model any algorithm. Hence unsafe queries exist for very simple universes.

In [2] we introduced a class of universes where unsafe queries are impossible and any IFP-operator can be calculated in finitely many steps. Such universes are called *safe*.

In previous investigations we have established some properties of the safety. In [4] we propose a necessary and sufficient condition for the universe safety. Since this condition contains only first order properties, we obtain that the safety is also a property of complete theories. In [3] we demonstrate that the safety of all unnested IFP-queries implies the safety of the universe.

The proofs of these results include IFP-operators of arbitrary arities. But a well-known feature of first order logic is that a unary variant of something may differ from a general one. Here we establish some properties of unary IFP-operators and show that both previous results don't hold for them: safety of the same unary IFP-operator maybe different in elementary equivalent structures, and nested unary IFP-operators can be unsafe unless all unnested ones are safe. Our example also demonstrates that binary unnested IFP-operators can be unsafe even when all unary unnested IFP-operators are safe. Therefore the arity of IFP-operators is essential for the safety.

2 Definitions

We consider the expansion of first order (FO) logic by inflationary fixed point (IFP) operators.

Definition 1 (see [5]). *Formulas of IFP-logic are constructed in the same way as FO-formulas (see [7]) and by* IFP-operators*: if $\phi(\bar{x}, \bar{y})$ is a formula with free variables $\bar{x}$, $\bar{y}$ and ϕ contains a new relation symbol Q then* $\mathrm{IFP}_{Q(\bar{x})}(\phi)$ *is a formula of old language with free variables $\bar{x}$, $\bar{y}$. Here the arity of Q must be equal to the length of $\bar{x}$, it is an* arity *of the IFP-operator.*

Evaluations of terms, atomic formulas, boolean operations and quantifiers are defined similar to FO-logic (see [7]).

Definition 2 (see [5]). *Let $\mathfrak{A}$ be a structure, $\phi(\bar{x}, \bar{y})$ be a formula, semantic of ϕ be already defined, and ϕ contain a new relation symbol Q. Let $\bar{a} \in \mathfrak{A}$ be values for $\bar{y}$. Then the* value *of the formula* $\mathrm{IFP}_{Q(\bar{x})}(\phi)$ *is defined the next way. Let*

$$Q_0^{\bar{a}} = \varnothing; \quad Q_{i+1}^{\bar{a}} = Q_i^{\bar{a}} \cup \{\bar{b} \in |\mathfrak{A}| : (\mathfrak{A}, Q_i^{\bar{a}}) \models \phi(\bar{b}, \bar{a})\},$$

for all $i \in \omega$.

Let $Q_n^{\bar{a}} = Q_{n+1}^{\bar{a}}$ for some natural n and $\bar{b}$ be values for $\bar{x}$. In this case the formula $\mathrm{IFP}_{Q(\bar{x})}(\phi)(\bar{x}, \bar{a})$ *is* true *if $\bar{b} \in Q_n^{\bar{a}}$, and it is* false *if $\bar{b} \notin Q_n^{\bar{a}}$. If there isn't such natural n then the value of* $\mathrm{IFP}_{Q(\bar{x})}(\phi)$ *is* undefined.

If the value of $\mathrm{IFP}_{Q(\bar{x})}(\phi)(\bar{x}, \bar{a})$ *is defined for all $\bar{a} \in \mathfrak{A}$ then the operator* $\mathrm{IFP}_{Q(\bar{x})}(\phi)$ *is called* safe *in $\mathfrak{A}$. The structure $\mathfrak{A}$ is called* safe *if all IFP-operators are safe in $\mathfrak{A}$.*

Easy to see that for each natural i the formula $Q_i^{\bar{a}}(\bar{x})$ is equivalent to the next formula $\psi_i(\bar{x}, \bar{a})$ correspondingly:

$$\psi_0 \equiv [\text{false}]; \quad \psi_{i+1} \equiv \left[\psi_i \vee (\phi)^{Q(\bar{t})}_{\psi_i(\bar{t}, \bar{a})}\right]. \tag{1}$$

Here the formula $(\phi)^{Q(\bar{t})}_{\psi_i(\bar{t}, \bar{a})}$ is obtained from ϕ by replacing every subformula of kind $Q(\bar{t})$ with $\psi_i(\bar{t}, \bar{a})$.

3 Main result

We consider a logic language containing two unary functional symbols s and d which mean the next and the previous item correspondingly. Our theory T of this language has an axiom

$$(\forall x)(s(d(x)) = x \wedge d(s(x)) = x)$$

and for every natural $n > 0$ two axioms of the form

$$(\exists x) \underbrace{\left(s^n(x) = x \wedge \bigwedge_{0<i<n} s^i(x) \neq x\right)}_{\phi_n(x)};$$

$$(\forall x)(\forall y)\left(\phi_n(x) \wedge \phi_n(y) \to \bigvee_{0 \le i < n} y = s^i(x)\right).$$

Here $s^i(x)$ means

$$\underbrace{s(s(\cdots s(}_{i \text{ times}} x) \cdots)),$$

in particular $s^0(x) = x$. Hence any model of T contains a unique "cycle" of length n for each natural $n > 0$. Let us call this cycle "n-cycle". The formula $\phi_n(x)$ says that x belongs to the n-cycle. Hence we can expand T by definable unary relations R_n: $R_n(x) \equiv \phi_n(x)$.

Theorem 1. *The theory T admits quantifier elimination.*

Proof. As usual (see [7]) it is enough to eliminate an existential quantifier from any formula of the form $(\exists u)\theta$ where θ is a quantifier-free elementary conjunction.

In this proof we simplify formulas writing $u+n$ or $u-n$ instead of $s^n(u)$ or $d^n(u)$ correspondingly. Then we have the equality $(u+n)+m = u+(n+m)$ for any integers n and m. Also easy to see that the equality $u+n=t$ is equivalent to $u=t-n$ and $R_m(u+n)$ is equivalent to $R_m(u)$. Then we can suppose that all atomic formulas in θ are of types $u=u+n$, $u=t$, or $R_n(u)$ where t doesn't contain u.

Let us note that $u=u+n$ is equivalent to the disjunction

$$\bigvee_{d|n} R_d(u),$$

where $d|n$ means divisibility. Hence the equalities $u=u+n$ can be replaced with the corresponding disjunction of $R_d(u)$.

Now we can suppose that the formula $(\exists u)\theta$ has the following form:

$$(\exists u)\theta \equiv (\exists u)\Bigg(\bigwedge_j R_{n_j}(u) \wedge \bigwedge_j \neg R_{m_j}(u) \wedge \bigwedge_k u=t_k \wedge \bigwedge_\ell u \neq r_\ell\Bigg), \tag{2}$$

where all t_k and r_ℓ don't contain u.

If at least one equality $u=t_k$ presents in (2) then the quantifier can be eliminated by replacing u with t_k: $(\exists u)\theta \equiv (\theta)^u_{t_k}$. In the following we suppose that there is no equality $u=t_k$ in (2).

All $R_{n_j}(u)$ in (2) for different n_j are not pairwise compatible hence $R_n(u)$ must be unique in (2) or absent at all. For the same reasons if $R_n(u)$ presents in (2) then all $\neg R_{m_j}(u)$ are implied for $m_j \neq n$. Thus we have three cases for the formula (2): it (a) contains a unique $R_n(u)$ and doesn't contain any $\neg R_{m_j}(u)$, or (b) doesn't contain $R_n(u)$ and contains some $\neg R_{m_j}(u)$, or (c) doesn't contain both.

In the case (b) we have the formula

$$(\exists u)\theta \equiv (\exists u)\Bigg(\bigwedge_j \neg R_{m_j}(u) \wedge \bigwedge_\ell u \neq r_\ell\Bigg). \tag{3}$$

Let us suppose that every r_ℓ belongs to the corresponding q_ℓ-cycle. Because there are n-cycles for all positive n then we can select a natural n such that $n \neq m_j$ and $n \neq q_\ell$ for all j, ℓ. Since each item u of this n-cycle satisfies $\neg R_{m_j}(u)$ and $u \neq r_\ell$ thus the formula (3) is true.

The case (c) is a partial case of (b) and we can use the same reasoning.

In the case (a) we have the formula of the form

$$(\exists u)\theta \equiv (\exists u)\Bigg(R_n(u) \wedge \bigwedge_\ell u \neq r_\ell\Bigg). \tag{4}$$

Let us add to (4) the true disjunctions $(R_n(r_\ell) \vee \neg R_n(r_\ell))$ for all ℓ, and construct the disjunctive normal form. Now we have a disjunction of formulas

$$(\exists u)\Bigg(R_n(u) \wedge \bigwedge_\ell u \neq r_\ell\Bigg) \wedge \bigwedge_\ell R^*_n(r_\ell),$$

where $R^*_n(r_\ell)$ is $R_n(r_\ell)$ or $\neg R_n(r_\ell)$. From $R_n(u)$ and $\neg R_n(r_\ell)$ it follows that $u \neq r_\ell$. Hence we can exclude the inequality $u \neq r_\ell$ when $\neg R_n(r_\ell)$ presents. So we have

$$(\exists u)\Bigg(R_n(u) \wedge \bigwedge_{\ell'} u \neq r_{\ell'}\Bigg) \wedge \bigwedge_{\ell'} R_n(r_{\ell'}) \wedge \bigwedge_{\ell''} \neg R_n(r_{\ell''}) \tag{5}$$

where $\ell' \neq \ell''$ for all ℓ' and ℓ''.

If there is no inequality in (5) then (5) is obviously equivalent the following quantifier-free formula

$$\bigwedge_{\ell'} R_n(r_{\ell'}) \wedge \bigwedge_{\ell''} \neg R_n(r_{\ell''})$$

because $(\exists u) R_n(u)$ is true.

In the other case there is at least $u \neq r_1$ in (5). Then the formula (5) is true if and only if the n-cycle contains items except $r_{\ell'}$. Hence (5) is equivalent to

$$\Big(\bigvee_{i=1}^{L} \bigwedge_{\ell'} \Big(R_n(r_{\ell'}) \wedge r_1 + i \neq r_{\ell'}\Big)\Big) \wedge \bigwedge_{\ell''} \neg R_n(r_{\ell''}) \tag{6}$$

where L is a number of ℓ' if its amount is less than n or $L = n - 1$ otherwise. If $n = 1$ then the empty disjunction is false.

Now we have considered all possible cases for the formula (2) and have eliminated the quantifiers in all cases. Therefore the theory T admits quantifier elimination.

Because the language of theory T has no constant so

Corollary 1. *The theory T is complete.*

Let a formula ϕ contain atomic formulas of the form $R_{n_u}(u)$ and $v = u + k_{u,v}$ for integers n_u and $k_{u,v}$, $\bar{y}$ be some fixed variables which are free in ϕ. Then *$\bar{y}$-weight of ϕ* (denote it as $w_{\bar{y}}(\phi)$) is the maximal of all such n_u and $k_{u,v}$ when u and v are not in $\bar{y}$.

Corollary 2. *In the proof of theorem 1 we have $w_{\bar{y}}(\theta') \leq 2 w_{\bar{y}}(\theta)$ where θ' is a result of quantifier elimination in $(\exists x)\theta$.*

Proof. Weight grows in substitution $(\theta)^u_{t_k}$ and in (6) but no more than twice.

An atomic model $\mathfrak{A}_0$ of T contains the unique n-cycle for every $n > 0$ and nothing more.

Theorem 2. *For any FO-formula $\phi(x, \bar{y})$ (with a new unary predicate Q) the unary operator* $\mathrm{IFP}_{Q(x)}(\phi)$ *is safe in $\mathfrak{A}_0$.*

Proof. Let the formula ϕ contain q quantifiers, and in the tuple $\bar{y}$ every y_j belong to the corresponding m_j-cycle. Let us select such natural w that $m_j \leq w$ for all j and $w_{\bar{y}}(\phi) \leq w$.

Consider a sequence of formulas $\psi_i(x, \bar{y})$ defined as (1). Let us prove that all formulas $\psi_i(x, \bar{y})$ are equivalent quantifier-free ones of $\bar{y}$-weight $2^q w$ or less. For $i = 1$ it follows immediately from theorem 1 and corollary 2. Let the formula $\psi_i(x, \bar{y})$ be a boolean combination of $R_m(x)$ and $x = y_j + k_j$ where $n \leq 2^q w$. When we substitute $\psi_i(t, \bar{y})$ instead of $Q(t)$ in ϕ we obtain new subformulas of the form $R_m(u)$ and $u = y_j + k'_j$. Hence we can't obtain new formulas $u = u + n$ and new $R_d(u)$ when u is not in $\bar{y}$. We can't increase weight in the substitution $(\theta)^u_{y+k'_j}$. And we can't increase L (amount of ℓ') in (6) unless $n = m_j$ for some m_j. But in the last case $L < m_j \leq w$. Therefore after quantifier elimination ψ_{i+1} is of weight no more than $2^q w$.

Thus every $\psi_i(x, \bar{y})$ is a boolean combination of $R_n(x)$ and $x = y_j + k_j$ where $n \leq 2^q w$. Since $y + m_j = y$ so we can suppose that $k_j < m_j$ for all j. But there are finitely many pairwise non-equivalent formulas of such form. Hence $\psi_i \equiv \psi_{i+1}$ for some i and the operator $\mathrm{IFP}_{Q(x)}(\phi)$ is safe.

The previous claim doesn't hold for binary IFP-operators.

Theorem 3. *The binary operator*

$$\mathrm{IFP}_{Q(x,y)}(x = y \vee Q(x, d(y)))$$

is not safe in $\mathfrak{A}_0$.

Proof. The set Q_i contains pairs $(a, a+i)$, $i = 0, \dots, i-1$. If a belongs to the $n+1$-cycle then $(a, a+i) \in Q_{i+1} \setminus Q_i$. Hence Q_i grows infinitely and the IFP-operator is undefined.

Also the safety doesn't hold for nested unary IFP-operators.

Theorem 4. *There is a nested unary IFP-operator that is not safe in* $\mathfrak{A}_0$.

Proof. Let us consider the following formula $\phi(y, z, x)$:

$$\mathrm{IFP}_{Q(x)}(x = y \vee x = z \vee Q(d(x)) \wedge \\ \wedge (\exists x_1, x_2)(x_1 \neq x_2 \wedge \neg Q(x_1) \wedge Q(d(x_1)) \wedge \neg Q(x_2) \wedge Q(d(x_2)))).$$

Let a and b be values for y and z correspondingly. Here we add a and b to $Q_1^{a,b}$ and then add at least two items to $Q_{i+1}^{a,b}$ for each i. This process stops when one of cycles $(a, s(a), s^2(a), \dots)$ or $(b, s(b), s^2(b), \dots)$ ends. Hence $\phi(y, z, x)$ means that for some m all $s^i(y)$ and $s^i(z)$ are pairwise different, $i = 0, \dots, m$, and $x = s^m(y)$ or $x = s^m(z)$. If these cycles are different then the predecessor of a or of b is not in $Q^{a,b}$ and the corresponding cycle is longer than other.

Hence the formula

$$\theta(y, z) \equiv [y \neq z \wedge \phi(y, z, d(y))]$$

means that the cycle of y is shorter than one of z. So we have a pre-order and the factor-order is isomorphic to the set of naturals. Then we can define the predecessor function D for cycles:

$$[D(z) = y] \equiv [\theta(y, z) \wedge \neg(\exists u)(\theta(y, u) \wedge \theta(u, z))].$$

Therefore the operator $\mathrm{IFP}_{P(x)}(x = s(x) \vee P(D(x)))$ is unsafe.

The theory T has non-atomic models $\mathfrak{B}$, which must contain both side infinite "lines". Unary IFP-operators can be unsafe in $\mathfrak{B}$.

Theorem 5. *The unary operator* $\mathrm{IFP}_{Q(x)}(x = y \vee Q(d(x)))$ *is unsafe in* $\mathfrak{B}$.

Proof. If a value a of y belongs to an infinite "line" then Q_i^a contains $a+j$ for $j = 0, \dots, i-1$. Hence there is no natural i such that $Q_i^a = Q_{i+1}^a$.

4 Conclusion

We have proved that properties of unary IFP-operators distinguish from properties of arbitrary IFP-operators. All unary IFP-operators can be safe unless binary ones are not, all unnested unary IFP-operators can be safe unless nested ones are not, and unary IFP-operators can be safe in some structure but unsafe in another elementary equivalent one.

We are interested in the following questions:

– is there a structure where all binary IFP-operators are safe but ternary ones are not;
– is there a complete theory T such that all unary IFP-operators are safe in all T models, but binary ones are not.

Answers help construct more reliable database queries.

References

1. Codd E.F. *Relational completeness of data base sublanguages.* Database Systems (ed. Rustin R.), Prentice-Hall, (1972), 33–64.
2. Dudakov S.M. *On safety of recursive queries.* Vestnik TvGU. Seriya: Prikladnaya matematika, **4**, (2012), 71–80, [Russian].
3. Dudakov S.M. *On safety of IFP-operators and recursive queries.* Vestnik TvGU. Seriya: Prikladnaya matematika, **2**, (2013), 5–13, [Russian].
4. Dudakov S.M. *On inflationary fix-point operators safety.* Lobachevskii J. Math., **36(4)**, (2015), 328–331.
5. Gurevich Y., Shelah S. *Fixed-point extensions of first-order logic.* Annals of Pure and Applied Logic, **32**, (1986), 265–280.
6. Kanellakis P., Kuper G., Revesz P. *Constraint query languages.* J. of computer and system sciences, **51**, (1995), 26–52.
7. Marker D. *Model theory: an introduction.* New York: Springer-Verlag, 2002.
8. ISO/IEC 9075-2:2016: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation) International Organization for Standardization, Geneva, Switzerland. `https://webstore.iec.ch/preview/info_isoiec9075-2{ed5.0}en.pdf`

Towards Loop Invariant Elimination for Definite Iterations over Changeable Data Structures in C Programs Verification

D. A. Kondratyev*, I. V. Maryasov, and V. A. Nepomniaschy

A. P. Ershov Institute of Informatics Systems, Novosibirsk, Russia
`{ivm, vnep}@iis.nsk.su, apple-66@mail.ru`

Abstract. This work represents the further development of definite iteration verification [9,10]. It extends the mixed axiomatic semantics method [1] suggested for C-light program verification. This extension includes a verification method for definite iteration over changeable arrays possibly with loop exit in C-light programs. The method includes an inference rule for the iteration without invariants. This rule was implemented in verification conditions generator. An example which illustrates the application of this methods is considered.

1 Introduction

C program verification is an urgent problem at the present time. Some projects (e. g. [2,4]) suggests different solutions. But none of them contains any methods for automatic verification of loop-containing programs without invariants. As it is known, in order to verify loops the user has to provide invariants whose construction is a challenge. Tuerk [13] suggested to use pre- and post-conditions for while-loops but the user still has to construct them himself.

We consider loops with certain restrictions [12]. We extend our mixed axiomatic semantics of the C-light language[1] with a rule for verification of such loops based on replacement operation [12]. The special verification conditions are generated with the help of this rule.

In our paper [10] we considered:

1. Program verification over unchangeable arrays with a loop exit.
2. The recursive algorithm for replacement operation construction.
3. The strategy of interactive proving of verification conditions.

During our research two goals appeared: to consider changeable data structures and to develop automated proof methods. This paper represents our results in this area:

1. Program verification over changeable arrays with a loop exit.
2. The recursive algorithm for replacement operation construction for such programs.
3. The strategy of automation of verification conditions proving.

Note that at proof stage we use the proof assistant ACL2[6] instead of SMT-solver as it was previously. This led to new algorithm for replacement operation construction.

* This author is partially supported by RFBR grant 17-01-00789.

2 Definite Iteration over Changeable Data Structures and Replacement Operation

The method of loop invariants elimination for definite iteration was suggested in [12]. It includes four cases:

1. Definite iteration over unchangeable data structures without loop exits.
2. Definite iteration over unchangeable data structures with a loop exit.
3. Definite iteration over changeable data structures with / without a loop exit.
4. Definite iteration over hierarchical data structures with a loop exit.

The first case was considered in [9], the second case could be found in [10]. This paper deals with the third case.

Let us remind the notion of data structures which contain a finite number of elements. Let $memb(S)$ be the multiset of elements of the structure S and $|memb(S)|$ be the power of the multiset $memb(S)$. For the structure S the following operations are defined:

1. $empty(S) = true$ iff $|memb(S)| = 0$.
2. $choo(S)$ returns an element of $memb(S)$ if $\neg empty(S)$.
3. $rest(S) = S'$, where S' is a structure of the type of S and $memb(S') = memb(S) \setminus \{choo(S)\}$ if $\neg empty(S)$.

Sets, sequences, lists, strings, arrays, files and trees are typical examples of the data structures.

Let $\neg empty(S)$, then $vec(S) = [s_1, s_2, \ldots, s_n]$ where $memb(S) = \{s_1, s_2, \ldots, s_n\}$ and $s_i = choo(rest^{i-1}(S))$ for $i = 1, 2, \ldots, n$.

Consider the statement

$$\textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x})\ \textbf{end}$$

where S is a structure, x is the variable of the type "an element S", v is a vector of loop variables which does not contain x and $body$ represents the loop body computation, which does not modify x and which terminates for each $x \in memb(S)$. The structure S can be modified as described below. The loop body can contain only the assignment statements, the **if** statements, possibly nested, and the **break** statements. Such **for** statement is named a definite iteration.

The operational semantics of such statement is defined as follows. Let v_0 be the vector of initial values of variables from v. If $empty(S)$ then the result of the iteration $v = v_0$. Otherwise, if $vec(S) = [s_1, s_2, \ldots, s_n]$, then the loop body iterates sequentially for x taking the values $s_1, s_2, \ldots, s_n$, and $body(v, s_j)$ can modify $s_1, s_2, \ldots, s_{j-1}$.

To express the effect of the iteration let us define a replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v$, $rep(v, S, body, i) = body(rep(v, S, body, i-1), s_i)$ for all $i = 1, 2, \ldots, n$ if $\neg empty(S)$.

A number of theorems, which express important properties of the replacement operation, were proved in [12].

The inference rule for definite iterations has the form:

$$\frac{E, SP \vdash \{P\}\, \mathbf{A};\{Q(v \leftarrow rep(v, S, body))\}}{E, SP \vdash \{P\}\, \mathbf{A};\ \textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x})\,\textbf{end}\{Q\}}$$

Here A are program statements before the loop. We use backward tracing: we move from the program end to its beginning and eliminate the rightmost operator (at the top level) applying the corresponding rule of the mixed axiomatic semantics [1] of C-light. E is the environment which contains an information about current function (its identifier, type and body) which is verified, an information about current block and label identifier if **goto** statement occurred earlier. SP is program specification which includes all pre-conditions, post-conditions, and invariants of loops and labeled statements.

3 The Algorithm of Generation of *rep* Function

Let S be a one-dimensional array of n elements. Consider the special case of definite iteration

$$\mathbf{for}\ (\mathbf{i = 0; i < n; i++})\ \mathbf{v := body(v, i)}\ \mathbf{end}$$

where $\mathbf{v := body(v, i)}$ consists of assignment statements, **if** statements (possibly nested) and **break** statements.

In order to generate verification conditions we have to determine v, $body(v, i)$, and the function *rep*.

Let the loop body has the form

$$\begin{gathered}\{\mathbf{x_1 = expr_1(x_1, x_2, \ldots, x_k)};\\ \mathbf{x_2 = expr_2(x_1, x_2, \ldots, x_k)};\\ \cdots\\ \mathbf{x_k = expr_k(x_1, x_2, \ldots, x_k)};\}\end{gathered}$$

where $expr_j (j = 1, 2, \ldots k)$ are some C-light expressions.

The vector v of loop variables consists of all variables from left-hand sides of assignment statements: $v = (x_1, x_2, \ldots, x_k)$ including $S[i], i = 1, 2, \ldots, n$. From the statements before the loop, we can get the initial value of v.

Let `generate_rep` be the function which generates *rep*, including all auxiliary functions and data structures. The variables from v are simulated by the data structure of the type $frame$ (see app. A[8]) whose definition is generated by the function `generate_frame`. The execution of `generate_rep` starts with `generate_frame` call.

The function $frame\_init$ receives the values of v and returns the structure of the type $frame$ whose fields coincide with the arguments of $frame\_init$. Therefore, $frame\_init$ could be considered as constructor in terms of C++ language. The macro $make - frame$ allows to specify the function $frame\_init$.

The definition of the function $frame\_init$ is generated by the function `generate_frame_init`. After `generate_frame` call the execution of the function `generate_rep` continues with `generate_frame_init` call. An example of work of these functions can be found in app. A[8].

The first argument of *rep* is the iteration number. In verification condition it is always equal to n. The second argument is data structure fr of the type $frame$. In verification condition it is always equal to $frame\_init$, which creates the structure of the type $frame$, whose fields coincide with the initial values of v and of the other unchangeable variables and data structures. Such *rep* call returns the structure of the type $frame$, whose fields (except for *break* statement) coincide with the v after a number of iterations given by the first argument. Note that *rep* is defined recursively. If its first argument is equal to 0 then *rep* returns the structure of the type $frame$, whose fields coincide with the initial values of v.

Let a loop exit occurs for some i such that $0 < i \leq n$, then we define *rep* that for all j such that $i \leq j \leq n$:

$$rep(i, frame\_init(x_{10}, x_{20}, \ldots, x_{k0})) = rep(j, frame\_init(x_{10}, x_{20}, \ldots, x_{k0}))$$

It means that iterations is continued but v is not changed for $i > j$. In such case the field $loop - break$ is introduced into the data structure of the type $frame$. This field is true if and only if a loop exit occurred in previous iterations.

The body of rep can be defined in ACL2 by the construct $b*$ (see app. B[8]) when $i > 0$. Consider $rep(i-1, fr)$: a new variable fr of the type $frame$ is created, whose value is a result of such recursive call. Using the construct *when* we check the validity of $fr.loop - break$ because if a loop exit occurred in previous iterations then body statements are not executed on this iteration and returning value is the structure fr. Then the constructs corresponding to loop body statements follow. Finally, the value of $b*$ (so the returning value of rep) is equal to fr.

The function `generate_rep_one` generates the definition of rep as described above. In order to define the constructs corresponding to loop body the function `generate_rep_one` calls the function `generate_rep_body`. The function `generate_rep_one` is called inside `generate_rep` after `generate_frame` and `generate_frame_init` calls.

The definition of `generate_rep_one` in pseudocode similar to C++ language see in app. C[8]. The function `generate_rep_body` has the following form:

```
string generate_rep_body(list<instruction> instructions) {
  string result = "(b*\n(";
  for (instruction i = instructions.begin();
    i! = instructions.end(); i++) {
    if (is_if_expression(i)) {
      result += "(fr\n(if\n";
      result += generate_expression(get_condition(i));
      result += "\n";
      result += generate_rep_body(get_true_branch(i));
      result += generate_rep_body(get_false_branch(i));
      result += "))\n((when (frame->loop-break fr)) fr)\n";}
    else if (is_assignment_expression(i)) {
      result += "(fr (change-frame fr :" + get_lvalue(i) +
        " " + generate_expression(get_rvalue(i)) + "))\n";}
    else if (is_break_instruction(i)) {
    result += "(fr (change-frame fr :loop-break t))\n"; break;}}
  return result + ")\nfr)\n";}
```

The description of used constructs see in app. D[8].

This pseudocode was simplified in order to clarify our algorithm. In real implementation there are some additional error checks. Our pseudocode recursively translates C-light statements to ACL2 language (i. e. a dialect of LISP).

We apply such translation to list of C-light statements. Therefore the body of a loop under consideration is the argument of the function `generate_rep_body`. Note that it is necessary to define the translation for all statements of definite iteration of a special form. For example, to translate **if** statement the function `generate_rep_body` is called recursively for then-branch and for else-branch. The execution of any branch can lead to **break** statement. Therefore, in the body of the function `generate_rep_body` the construction *when* is generated, which checks whether the execution of **if** statement led to loop exit. Thus, the using of special ACL2 constructions (such as $b*$ and *when*) allowed to simplify the translation of C-light statements to ACL2 language.

4 Strategy of Automatic Verification Conditions Proving in ACL2

Inductive proofs appear in our verification method. Although ACL2 supports induction, we are confronted with difficulties during our experiments. We suggest heuristic method which allows us to prove successfully the partial correctness of a number of examples containing definite iteration over changeable arrays with / without loop exit.

The idea is to check the assumption that program post-condition describes the cases whether a loop exit occurred or not in a form of implications conjunction. ACL2 is able to check the validity of such assumption. If it is true it can help to define more precisely the cases in post-condition. Proving all detailed cases can help to prove the verification condition.

The input of the algorithm is the verification condition ϕ, the variable n, the definition of *rep*, underlying theory and post-condition.

The output of the algorithm is «ϕ is true» or «unknown».

The method has the form:

1. Let M be a tuple of implications from post-condition. Consider the tuple N such that the i-th element of N is the premise of the i-th element of M. Go to the step 2.
2. For each element of N execute step 3. If result is true, add to the theory a lemma which is a conjunction of ϕ and an equality of the i-th element of N and $rep(\ldots).loop-break$. Otherwise go to the step 4. If result is true, add to the theory a lemma which is a conjunction of ϕ and an equality of the i-th element of N and $\neg rep(\ldots).loop-break$. Go to the step 5.
3. Let θ be the i-th element of N. Let ω be the conjunction of ϕ and an equality of θ and $rep(\ldots).loop-break$. Check the validity of ω in ACL2. If ω was proved then return «true» otherwise return «false».
4. Let θ be the i-th element of N. Let ω be the conjunction of ϕ and an equality of θ and $\neg rep(\ldots).loop-break$. Check the validity of ω in ACL2. If ω was proved then return «true» otherwise return «false».
5. Check the validity of ϕ in ACL2 using obtained lemmas. If ϕ was proved then return «ϕ is true», otherwise return «unknown».

This method can be generalized for using with another interactive provers and SMT-solvers (for example CVC4 and Z3).

5 Example

Let us demonstrate the application of our methods. Consider the following function **negate_first**[5]. For given array of integers a it changes the sign of its first negative element.

```
void negate_first(int n, int* a) {
     int i;
     for (i = 0; i < n; i++) {
         if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

In ACL2 we represent arrays as lists. The pre-condition has the form:

```
(and (integer-listp a) (integer-listp a_0) (equal a a_0)
     (integerp n) (< 0 n) (<= n (length a_0)))
```

The post-condition has the form:

```
(let (((mv found-spec index-spec) (count_index n a_0)))
     (implies (not found-spec) (equal a a_0))
     (implies found-spec (equal a
               (update-nth index-spec (- (nth index-spec a_0)) a_0))))))
```

The definition of `count_index` see in app. E[8]. It returns a pair: the first component is true if and only if a has negative elements. In such case the second component is the index of negative element.

The data structure of the type `frame`, recursive definition of the function `rep` and the verification condition, based on replacement operation, was generated using the method described in 3. The verification condition `my-theorem1` (see in app. F [8]) is a conjunction of two cases: whether the array has no negative elements or it has a negative element. Note that the function post-condition is also based on these cases. Thus, the heuristic method from 4 was applied. Finally, the lemma about the equivalence of the statement about loop exit and the statement about existence of negative element in array was generated. The corresponding extensions of the underlying theory see in app. F [8]. ACL2 allowed to prove successfully the verification condition in obtained theory.

6 Conclusion

This paper represents an extension of the system for C-light program verification [11]. In the case of definite iteration over changeable arrays with loop exit this extension allows to generate verification conditions without loop invariants.

This generation is based on the inference rule for the C-light **for** statement which uses replacement operation. It expresses definite iteration in special form. The replacement operation is generated automatically by a special algorithm.

Obtained verification conditions are automatically proved in ACL2 with the help of suggested heuristic method.

Note that the verification of the functions implementing BLAS interface [3] is an important problem. Earlier we performed such experiments successfully [7]. Our methods allowed us to verify the function *asum* which implements the corresponding function from BLAS interface: it calculates the sum of absolute values of a vector.

The next step in our work will be the case of more complicated data structures and the verification of another functions of BLAS.

References

1. Anureev I. S., Maryasov I. V., Nepomniaschy V. A. C-programs Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2011. — Vol. 45 — Issue 7. — P. 485–500.
2. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. of 22nd Int. Conf. TPHOLs. — LNCS. — 2009. — Vol. 5674. — P. 23–42.
3. Dongarra J. J., van der Steen A. J. High-performance computing systems: Status and outlook // Acta Numerica. — 2012. — Vol. 21. — P. 379–474.
4. Filliâtre J.-C., Marché C. Multi-prover Verification of C Programs // Proc. of 6th ICFEM. — LNCS. — 2004. — Vol. 3308. — P. 15–29.
5. Jacobs B., Kiniry J. L., Warnier M. Java Program Verification Challenges // Proc. FMCO 2002. — LNCS. — 2003. — Vol. 2852. — P. 202–219.

6. Kaufmann M., Moore J. S. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp // IEEE Transactions on Software Engineering. — 1997. — Vol. 23. — Issue 4. — P. 203–213.
7. Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project // Proc. of PSI 2017. — LNCS. — 2018. — Vol. 10742. — P. 227–240.
8. Kondratyev D. A. Towards Loop Invariant Elimination for Definite Iterations over Changeable Data Structures in C Programs Verification. Appendices. — https://bitbucket.org/c-light/loop-invariant-elimination
9. Maryasov I. V., Nepomniaschy V. A. Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs // Modeling and Analysis of Information Systems. — 2015. Vol. 22 — Issue 6. — P. 773–782.
10. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Invariant Elimination of Definite Iterations over Arrays in C Programs Verification // Modeling and Analysis of Information Systems. — 2017. Vol. 24 — Issue 6. — P. 743–754.
11. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2014. — Vol. 48 — Issue 7. — P. 407–414.
12. Nepomniaschy V. A. Symbolic Method of Verification of Definite Iterations over Altered Data Structures // Programming and Computer Software. — 2005. — Vol. 31. — Iss. 1. — P. 1–9.
13. Tuerk T. Local Reasoning about While-Loops // VSTTE 2010. Workshop Proceedings. — 2010. — P. 29–39.

Verification Oriented Process Ontology [⋆]

Natalia Garanina and Igor Anureev

A.P. Ershov Institute of Informatics Systems, Lavrent'ev av., 6, Novosibirsk 630090, Russia
(`{garanina,anureev}@iis.nsk.su`)

Abstract. This paper presents the ontology of concurrent systems. This ontology is the part of the intellectual system for supporting verification of concurrent systems. The ontological representation of these systems is oriented both to applying formal verification methods and to extracting information from technical documentation. We describe the ontology classes and domains that define concurrent processes. Their formal semantics is given as a labelled transition system. We specialize several types of processes from the subject domain of automatic control systems. The concepts of ontology are illustrated by the example of a control part for a bottle-filling system.

1 Introduction

Our long-term goal is a comprehensive approach to extracting formal models and properties of concurrent systems from the texts of technical documentation, with their verification, i.e. ensuring the quality of a system using formal methods. For formal verification, we rely on the well-proven model checking and deductive methods. The drawback of the current systems for supporting the development and verification of requirements is that they offer only manual formulation of requirements and systems [2,11,14,15].

Our intellectual system being developed for supporting formal verification of concurrent system will automatically extract and generate both concurrent system and requirements for it. In this paper, we suggest an ontology of concurrent system. Another key component of the system is the ontology of specification patterns proposed in [6]. The content of these ontologies, i.e. the sets of their class instances, are ontological descriptions of some concurrent system and requirements for it. These descriptions are extracted from corpus of technical documentation by our system of information extraction from natural language text [3,4,5]. The description set of requirements can also be extended with descriptions of typical correctness requirements, automatically generated from the ontological description of the system. These descriptions of the system and requirements are the basis for formal verification of the concurrent system. To verify a system, it is necessary first to choose a suitable verifier (model checker) taking into account the formal semantics of the ontology-based requirement presentation. If it exists, we translate the ontological description of the system into the model specification input language of the verifier, and the requirements' description is translated into the property specification input language of the verifier (usually, this language is some temporal logic). If suitable verifier does not exist it is reasonable to develop a special specification patterns verifier based on the patterns' formal semantics which uses the simplest verification algorithms, if possible. If semantics of ontological representations of the system model and requirements, as well as the input language of a verification tool are strictly defined, and translating these representations into the input data of the verifier is correct, then the correctness of checking the requirements is guaranteed. Dealing with requirements involves not only the formal semantics of specification patterns, but also the presentation

⋆ The research has been supported by Russian Foundation for Basic Research (grant 17-07-01600).

of these patterns both in natural language and in graphical form. These representations are especially important, because due to the ambiguity of the natural language, it is possible that the extracted and generated requirements may not meet the engineer' expectations, and manual corrections may be required. The ontologies of concurrent systems and specification patterns with their formal semantics are the basis for a long-term development of a system for supporting formal verification. Our approach allow to model and verify a wide range of systems: from software distributed systems to business processes. The main limitation of the approach is the complexity, and often the impossibility to exactly extract information from technical documentation, and the extracted models of systems and requirements may require manual correction.

Our ontology of concurrent systems specifies a set of processes. Known approaches to ontological process description usually solve modeling problems. The ontologies PSL [13], Cyc [1], SBPM [7,8], and GFO [9] focus on tasks in the fields of production scheduling, process planning, workflow management, business process reengineering, simulation, process realization, process modelling, and project management. Taking into account the concepts of domains in describing the manufacturing, engineering and business processes is an important advantage of these approaches. Indicating the specific time and place of events and actions associated with these processes in some of these approaches allows one to describe the planning ontologies representing abstract and executable plans of organization of processes, which is another advantage. However, these advantages make verification of the logic of the processes underlying these ontologies very difficult. The intertwining of domain concepts with the logic of processes results in complication of extracting the latter for use in model-checking based verification systems. Moreover, the extracted logic still needs to be translated into the language of the verification system, based, as a rule, on transition systems that operate with the concepts of state and transition. In particular, for planning ontologies, the specific time and place of events and actions must be expressed in terms of states. In addition, these ontologies do not have a formal semantics for the logic of processes, which further complicates the task.

In this paper, we define the ontology classes and domains that characterize concurrent systems in general. We give the formal semantics of these classes including the operational semantics of system process actions. A syntax of ontology representation languages, in particular language OWL, is declarative, but the strictly defined operational semantics of our ontology allows to easily translate the representation of the process model into the input languages of verification tools, in particular SPIN [10]. On the way to deal with practical concurrent systems, we describe the particular method of using our general concurrent system ontology for specifying components of automatic control systems.

2 The Ontology

We consider an *ontology* as a structure, which includes the following elements: (1) a finite non-empty set of *classes*, (2) a finite non-empty set of *data attributes and relation attributes*, and (3) a finite non-empty set of *domains of data attributes*. Each class is defined by a set of attributes. Data attributes take values from domains, and relation attributes' values are instances of classes. *An instance of a class* is defined by a set of attribute values for this class. *Information content* of an ontology is a set of instances of its classes. The ontology population problem is to extract the information content of this ontology from input data. In our case, the input data for populating the ontology of concurrent systems is technical documentation. To populate this ontology, we use our system of semantic information extraction from natural language texts [3]. Note that some attribute values of extracted ontology instances may be not determined.

Our ontology is intended for an ontological description of a concurrent system using a set of instances. We consider a concurrent system as a set of sequential communicating processes. Processes (described by the class Process) are characterized by sets of local and shared variables; a list of actions on these variables which change their values; a list of channels for the process communication; and a list of communication actions for sending messages. The process variables (the class Variable) take values of the basic types (Booleans, finite subsets of integers or strings for enumeration types). Initial conditions of the variable values can be defined by comparison with constants. The actions of the processes (the class Action) include the base operations over the basic types. The enable of each action depends on the guard conditions (the class Condition) for the variable values and the content of the sent messages. The processes can send messages through channels (the class Channel) under the guard conditions (the class Condition). The communication channels are characterized by the type of reading messages, capacity, ways of writing and reading, and reliability. At Figure 1, classes are presented by white ovals. Relations between classes are shown as dashed arrows with names in grey ovals. These arrows are solid if the relation is one-to-many, and dotted, if the relation is one-to-one. Class data attributes placed in dash-dot rectangles are connected with their classes by dash-dot arrows. In the following sections, we give a formal semantics of the classes and domains of our ontology.

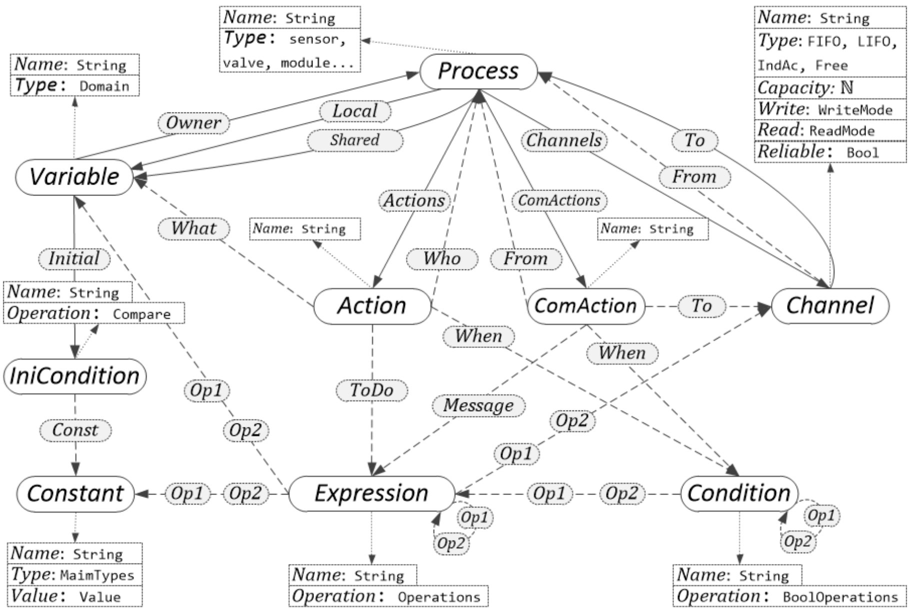


Fig. 1. The ontology of concurrent systems.

3 The Semantics

We consider a concurrent system which is a set of sequential processes communicating via shared variables and bounded channels. The values of the shared and local process variables determine a local state of the process. Depending on the subject area, we detect the process communication features (synchronous, asynchronous) and channel characteristics (FIFO, LIFO, reliable, bounded, etc.). The channel states are specified by the set of messages stored in it.

The ontological description O_S of the concurrent system S is the set of ontology instances which represent the process, channels and other elements of the system S. We define the formal semantics for O_S using the standard model of labeled transition systems $M_S = (D_S, D^0_S, T_S, Act_S)$, where D_S is a finite nonempty set of states, D^0_S is a nonempty set of initial states, T_S âĂŞ a transition relation between states, Act_S is the finite set of transition labels that describe the actions of the system. The model semantics of the ontological description O_S is a function that maps the values of the instance attributes to the elements of the labelled transition system $Sem : O_S \longrightarrow El(M_S)$.

Let us describe the set $El(M_S)$. Let the concurrent system S contain a finite set P_S of the processes and a finite set C_S of bounded channels. Let $\texttt{Domains} = \{\texttt{Integer}, \texttt{Bool}, \texttt{String}\}$ be finite version of the standard basic types for variable values with the standard type cast. For the process $P \in P_S$, we denote the following sets: the local variables L_P, the shared variables U_P, all variables $V_P = L_P \cup U_P$, the actions A_P, the channels C_P, communication actions (com-actions) S_P. The domain of the variable $v \in V_P$ is $Val(T_P(v)) \subseteq \texttt{Values}$ for some basic type $T_P(v) \in \texttt{Domains}$. Let L_S be the set of all local variables of processes from P_S, U_S be the set of all shared variables of processes from P_S, and the valuation function $V : L_S \cup U_S \longrightarrow 2^{\texttt{Values}}$ represent the value of the variable. The local state of the process P is a tuple of values of its variables: $st_P = (V(v_1), ...V(v_{n_V}))$. Let $st_P(v) = V(v)$ for $v \in V_P$ be the value of the variable v in the state st_P. The initial states of the process $st^0_P = (V^0(v_1), ...V^0(v_{n_V}))$ are defined by the initial values of its variables $V^0 : L_S \cup U_S \longrightarrow 2^{\texttt{Values}}$. The local state of the channel c is the ordered tuple of messages stored in it: $st_c = (m_1, ..., m_{n_c})$. Let $st_c(k) = m_k$ for $k \in [1..n_c]$ be the content of the k-th message of the channel c in the state st_c. The initial state of all channels is an empty set. The set of states D_S of the LTS-system M_S is the Cartesian product of the values of local and shared process variables and channel states. The set of initial states D^0_S is the Cartesian product of the initial values of process variables and channel states. We state the operational semantics of the system is the interleaving semantics, i.e. its state and the state of its channels can be changed only by a single system process. Hence, the set of system actions is $Act_S = \cup_{P \in S} A_P \cup S_P$. Due to interleaving, the transition relation T_S is defined by the local operational semantics of the particular process instances $P \in S_P$ described in the following sections.

Further, for every class, the attribute `Name` $\in$ `String` specifies the name of the class instance.

The class *Process* describes the system processes with the following attributes.
- `ProcType`: `{sensor, valve ...}`$\in$`String` contains the process specifier.
- *Local*$*$: *Variable* lists the variables visible only within the process actions.
- *Shared*$*$: *Variable* lists the variables visible only for process that share them.
- *Actions*$*$: *Action* lists the actions performed by the process on its variables.
- *Channels*$*$: *Channel* lists the channels connecting the processes.
- *ComActions*$*$: *ComAction* lists the sending message actions.

The value of any attributes of this class may not be specified. However, if *Local* and *Shared* are not specified, then the *Actions* value is not specified. If the value of *Channels* is not specified, the value of *ComActions* is not specified. Each instance of this class must have *Actions* and/or *ComActions* to be specified.

For the process P, the formal semantics of its ontological description $Sem(P)$ consists of the formal semantics of its attributes: local variables $Sem(P.Local)$, shared variables $Sem(P.Shared)$, actions $Sem(P.Actions)$, communication actions $Sem(P.ComActions)$, and channels $Sem(P.Channels)$. The process P can be unambiguously presented in M_S by these semantics. First, we define the following sets: $L_P = \{l\}_{l \in Local}$, $U_P = \{u\}_{u \in Shared}$, $A_P = \{a\}_{a \in Actions}$, $C_P = \{c\}_{c \in Channels}$, $S_P = \{s\}_{s \in ComActions}$. In the next sections, we give semantics of these attributes step by step.

The class *Variable* describes process variables by the following attributes.
- `Type`: `Domains` describes the type of a variable.
- *Initial*∗: *IniCondition* describes constraints on the initial value of the variable.
- *Owner*∗: *Process* refers to the processes having access to the variable.
The values of all attributes of this class, except *Initial*, must be specified.

The semantics of the variable v of the process P is defined as follows:
$Sem(v.Name) \in$ `String`, $Sem(v.Type) \in$ `Domains`, $Sem(v.Owner) \subseteq S_P$. If $Sem(v.Owner) = \{P\}$, then $v \in L_P$. The semantics of the initial condition specifies the set of initial values of the variable $V^0(v)$:

– If $v.Initial = \emptyset$, then $V^0(v) = \{false\}$ with $v.Type =$ `Bool`, $V^0(v) = \{0\}$ with $v.Type \in$`Integer`, $V^0(v) = \{'' \; ''\}$ with $v.Type \in$ `String`;
– if $v.Initial \neq \emptyset$, then $V^0(v) = \cap_{ini \in v.Initial} Sem(ini)$.

The class *IniCondition* defines the initial values of variables by the attributes:
- *IniConstant*: $\{Constant$, `Values`$\}$ defines a constant for the comparison.
- `Operation`: `Compare`$= \{<, >, =, \leq, \geq, \neq\}$ defines the comparison actions.
The values of *Const* and `Operation` attributes must be specified.

The semantics of the initial condition *ini* of the variable v is a subset of the set of its values: $Sem(ini.Name) \in$ `String` and $Sem(ini) = \{val \in Val(v.Type) \mid val \circ Sem(Const)$ for $\circ \in \{<, >, =, \leq, \geq, \neq\}\}$, where $Sem(Const) = Sem(cs)$ with $cs \in Constant$ or $Sem(Const) = s$ with $cs \in$ `Values`.

The class *Constant* describes constants, and contains the following attributes.
- `Domain`: `Domains` describes the type of a constant.
- `Value`: `Values` describes the value of a constant.
The values of all attributes must be defined.

The semantics of the constant cs is its value from the set of system possible values: $Sem(cs.Name) \in$ `String`, $Sem(cs.Domain) \in$ `Domains`, $Sem(cs.Value) \in$`Values`, and $Sem(cs) = cs.Value$.

The class *Channel* describes the communication channels by the attributes:
- *From*: *Process* specifies the sender.
- *To*∗: *Process* specifies the recievers of messages from the channel. If there are several recievers then the channel is broadcasting.
- `Type`: `Order = {FIFO, LIFO, IndAc, Free}` defines the order of reading from the channel, where `IndAc` means access by index and `Free` means random access.
- `Capacity`: `Integer` specifies the capacity of the channel.
- `Reliable`: `Bool` characterizes the external reliability of the channel in the sense that messages are not lost by external causes.
- `Write`∈`WriteMode = {NSent, Old, New, Some}` describes the writing mode when the channel is full.

– `NSent`: the message is not written to the channel, i.e. is lost;

- `New`: the message is written to the channel at the location of the message with the largest index, which is deleted;
- `Old`: the message is written to the channel at the message location with index 1, which is deleted;
- `Some`: the message is written to the channel in the place of some message that is deleted.

- `Read∈ReadMode = {Del, Keep, Tail, Head, Some}` specifies how to read from the channel.

- `Del`: the message is deleted from the channel after reading;
- `Keep`: the message is not deleted from the channel after reading;
- `Tail`: the message index after reading becomes the largest in the channel;
- `Head`: the index of the message after reading becomes 1;
- `Some`: after reading the message is moved to an arbitrary place in the channel.

The values of the $From$ and To attributes must be defined. By default, i.e. if the corresponding attribute is not defined then `Type` = `FIFO`, `Capacity` = 1, `Write` = `NSent`, `Read` = `Del`, `Reliable` = *true*.

The semantics of the channel c of the process P is defined as follows: $Sem(c.Name) \in$ `String`, $Sem(c.From) = P$, and $Sem(s.To) \subseteq S_P$. The semantics of the local channel states and their changes are defined below in the descriptions of actions and com-actions.

The class *Action* describes the actions performed by the processes.
- Who: *Process* refers to the process that performs this action.
- $What$: $Variable$ refers to a variable which may change as the action result.
- $ToDo$: *Expression* describes an expression whose value is assigned to the variable in $What$.
- $When$: *Condition* describes a Boolean formula that specifies the condition for performing an action in the local state.
Attribute values $ToDo$ and $When$ can use only process variables and messages from its channels. The values of the Who, $What$ and $ToDo$ attributes must be specified. If $When$ is not specified, this means that the action can be performed in any local state of the process.

The semantics of the action a of the process P is defined as follows. $Sem(a.Name) \in$ `String`, $Sem(a.Who) = P$, and $Sem(a.What) \in V_P$. Let $a.What = v$ be the variable that have to be changed by the action a, and $C_a \subseteq C_P$ be the process channels whose messages are read for the action. The transition relation between the local states both of the process P and its channels is defined as follows. If the local states of the process st_P and st_c of every channel $c \in C_a$ are such that the condition $a.When$ is satisfied ($Sem(a.When) = true$) and the action expression $a.ToDo$ has a meaning ($Sem(a.ToDo) \neq null$), then after the action a, the new local states st'_P of the process and its channels are such that $st'_P(v) = Sem(a.ToDo) \wedge \forall x \in V_P \setminus \{v\} : st'_P(x) = st_P(x) \wedge \forall y \in C_P \setminus C_a : st'_y = st_y$, and the new state st'_c of every channel c from C_a corresponds to the semantics of the read operation for channels (see the next section). Note, that in the process of calculating $Sem(a.When)$ and $Sem(a.ToDo)$, channel states do not change. If $Sem(a.ToDo) = null$ or $Sem(a.When) \neq true$, all states remains the same: $st'_P = st_P$ and $\forall c \in C_a : st'_c = st_c$.

The class *Expression* describes expressions whose value is assigned to process variables under actions, and contains the following attributes.
- The attributes $Op1, Op2$: $\{Expression, Variable, Constant, Channel,$ `Values`$\}$ specify the left and right operands of the expression.
- `Operation`: `Operations`$= \{+, -, *, \%, >, <, \leq, \geq, \neq, \neg, \wedge, \vee, \rightarrow\}$ makes standard integer or Boolean manipulations with the operands.

The values of $Op1$ and $Op2$ must operate on variables and channel data available to the process. The values of the $Op1$ and `Operation` attributes must be defined. If $Op2$ is not defined, then the `Operation` value must be a unary operation.

Let for the action a of the process P in the state st_P the action expression be $a.ToDo = e$. The semantics of the expression is the value of a base type or $null$: $Sem(e) \in \texttt{Values} \cup null$.

- For binary operations $\circ \in \{+, -, *, \%, >, <, \leq, \geq, \neq, \wedge, \vee, \rightarrow\}$: $Sem(e) = Sem(e.Op1) \circ Sem(e.Op2)$;
- For unary operations $\circ \in \{-, \neg\}$: $Sem(e) = \circ Sem(e.Op1)$;
- $Sem(e) = null$ iff $Sem(e.Op1) = null$ or $Sem(e.Op2) = null$.

Let $Op \in Op1, Op2$. $Sem(e.Op)$ is defined as follows:

- $e.Op = e' \in \{Expression, Constant, \texttt{Values}\}$: $Sem(e.Op) = Sem(e')$;
- $e.Op = w \in Variable$, $w \in V_P$: $Sem(e.Op) = st_P(w)$;
- $e.Op = c \in Channel$, $c \in C_a$: $Sem(e.Op) = Val$, where
  - If $st_c = \emptyset$ then $Val = null$ and $st'_c = st_c$;
  - If c.Type = `FIFO` then $Val = st_c(1)$ and if c.`Read` is equal to
    - `Del` then $st'_c = (m_2, ..., m_{n_c})$;
    - `Keep` then $st'_c = st_c$;
    - `Tail` then $st'_c = (m_2, ..., m_{n_c}, m_1)$;
    - `Head` then $st'_c = st_c$;
    - `Free` then $st'_c = (m_2, ..., m_1, ..., m_{n_c})$.

The semantics for other types of reading are defined in the similar way.

The class *Condition* describes the guard condition for a process actions and com-actions. Its description and semantics are almost the same as for the class *Expression* except that the *Condition* uses just boolean operation and values.

The class *ComAction* describes the com-actions for sending messages, and contains the following attributes.
- $From$: *Process* specifies the process that sends the message.
- To: *Channel* specifies a channel that delivers messages to a receiver.
- $Message$: *Expression* describes the message being sent.
- $When$: *Conditions* describes the guard condition for sending the message.

Note that our model of processes communication implements the receiving of messages when necessary, i.e. reading from the channel is performed only when a process calculates the guard conditions or the action expressions. Attribute values $Message$ and $When$ can use only the process variables and messages from its channels. The values of the $Message$, $From$, and To attributes must be defined. If $When$ is not specified, this means that the com-action can be performed in any local state of the process.

The semantics of sending action s of the process P in the state st_P is defined as follows. $Sem(s.Name) \in \texttt{String}$, $Sem(s.From) = P$, and $Sem(s.To) \in C_P$. Let $s.To = c$ be the channel of the process for sending the message and its local state be $st_c = (m_1, ..., m_{n_c})$, and $C_s \subseteq C_P$ are the process channels from which messages are read for the message. Let $Sem(s.Message) = Mes$ and $Sem(s.When) = Cnd$. The local operational semantics of sending a message is defined as follows. The local states of the process P and the unused channels do not change: $st'_P = st_P$ and $\forall cl \in C_P \setminus (C_s \cup \{c\}) : st'_{cl} = st_{cl}$. If the local states of P and the channels $cs \in C_s$ are such that $Mes = null$ or $Cnd \neq true$, then $\forall c \in C_P : st'_c = st_c$. If the condition $s.When$ is satisfied ($Cnd = true$) and the value of the message is calculated ($Mes \neq null$), then after com-action s, the new state of the channels st'_{cs} for $cs \in C_s$ corresponds to the semantics of the read operation defined above. The new local state st'_c of the channel c depends on the capacity of the channel $c.\texttt{Capacity}$ and the method of writing to the channel $c.\texttt{Write}$ as follows:

- If $n_c < c.\texttt{Capacity}$, then $st' = (m_1, ..., m_{n_c}, Mes)$.
- When the channel is full, if $c.\texttt{Write}$ is equal to
  - `NSent` then $\forall c \in C_P : st'_c = st_c$;
  - `Old` then $st'_c = (Mes, m_2, ..., m_{n_c})$;
  - `New` then $st'_c = (m_1, ..., m_{n_c-1}, Mes)$;
  - `Some` then $st'_c = (m_1, ..., m_{i-1}, Mes, m_{i+1}, ..., m_{n_c})$, $i \in [1..n_c]$.

Knowing the local semantics of the actions and com-actions of every process, the transition relation T_S can be specified.

4 Ontological Modeling Functional Elements of the Automatic Control System (ACS)

In this section, we represent the methodology of description for several basic functional elements of the automatic control system using our ontology: sensors, comparators and regulators, and represent an example of such a system.

A sensor detects events or changes in its environment (that is a part of the control object) observing some environment parameter and sends the information to other elements of ACS. The ontology process for a sensor has the following obligatory attributes. The attribute *Shared* contains the variable *ev* that specifies the value of the observing environment parameter. The attribute *Channels* describes the channels for sending information about the parameter. The attribute *ComActions* contains the actions of sending these messages.

A comparator compares the value of its input parameter with some reference value (setpoint). The ontology process for a comparator has the following obligatory attributes. The attribute *ComActions* contains the actions of sending messages with the result of comparing the value of the input parameter with the setpoint. The *Channels* describes the channels for sending this information.

A sensor with the comparison function combining the functions of a sensor and a comparator is modelled by the composite process.

A regulator maintains a designated characteristic of the control object. It directly influences the state of the control object by changing the parameters of the control object in accordance with the values of the input parameters of the regulator. Regulators are divided into ontological classes, depending on the number of modes of their operation. The ontology process for a regulator has the following obligatory attributes. The attribute *Local* contains the variable *mode* that specifies the current mode of the regulator and takes the values in an enumeration type consisting of n-th elements. The attribute *Shared* contains the variables that describe the parameters of the control object of the regulator. The attribute *Actions* contains the actions that change these parameters. The attribute *ToDo* of these actions contains the expression that computes the new values of these parameters. This expression depends on the value of the local variable *mode* and the messages with the values of the input parameters.

To illustrate ACS modelling by this methodology, we present a subsystem of bottle-filling system from [12] described in a natural language. We consider the following simplified subsystem: the sensors "lower-level sensor" and "upper-level sensor" with the comparison function are defined by the processes P_l and P_u, and the regulators "bottom valve" and "inlet valve" are defined by the processes P_b and P_i. We suggest that all channels are FIFO-type with capacity 1 and messages are removed after being read.

The process for the lower-level sensor is P_l=(`Type`: `sensor-comparator`, *Shared**: $\{ev\}$, *Channels**: $\{li, lb\}$, *ComActions**: $\{lo, lc\}$). The variable ev specifies the current fluid level information. It takes the `Integer` values from $min.Value$ to $max.Value$ where the constants min and max defines the almost empty and overflow condition. The channels li and lb connect the process-sensor P_l with the regulator processes P_i and P_b, respectively. The com-action lo sends the message "toOpen" to the inlet regulator P_i when the almost empty condition holds: lo = ($From$: P_l, To: li, $Message$: "toOpen", $When$: $ev \leq min$). The com-action lc is symmetric.

The process for the inlet valve is P_i=(`Type`: `regulator`, *Local**: $\{mode\}$, *Shared**: $\{ev\}$, *Actions**: $\{inc, io, ic\}$, *Channels**: $\{li, ui\}$). The variable $mode$ has the enumeration type {"open", "closed"}. The action inc increments the fluid level: inc = (Who: P_i, $What$: ev, $ToDo$: $ev + 1$, $When$: $mode$ = "open"). The action io opens the inlet valve: io = (Who: P_i, $What$: $mode$, $ToDo$: "open", $When$: li = "toOpen"). The closing action ic is symmetric. The processes P_u and P_b are defined in the similar way.

5 Conclusion

This paper represents the ontology of concurrent systems. This ontology is the part of the intellectual system for verification support. The system of information extraction from technical documentation texts populates this ontology with instances of processes found in the documents. Then, the translation module generates a representation of the system in the input language of some verifier. To ensure the correctness of this generation, in this paper, we define the formal semantics of our ontology by associating the labelled transition system to a set of ontological process instances by formal rules. The formal semantics of the ontological representation of processes, oriented to formal verification methods, distinguishes our approach from the state-of-art approaches to definitions of process ontologies.

Further, we plan to develop a system of object-oriented ontological process patterns for elements from various subject areas: automatic control systems, business processes, etc. As part of constructing our intellectual system for verification support, we will also develop a method of extracting typical system requirements from the ontological representation of processes to populate the associated ontology requirements. In addition, it is reasonable to produce a special verification tools using this ontology based on model checking and deduction verification methods, and their combination.

References

1. **Aitken, Stuart, Curtis J.** A Process Ontology. // Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, 2002. Vol. 2473. P. 263âĂŞ270. doi:10.1007/3-540-45810-7_13. ISBN 978-3-540-44268-4.
2. M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, A. Tang. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. In *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.
3. **N. Garanina, E. Sidorova, E. Bodin.** A Multi-agent Text Analysis Based on Ontology of Subject Domain. // In *Perspectives of System Informatics*, volume 8974 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2015.
4. **N. Garanina, E. Sidorova.** Context-dependent Lexical and Syntactic Disambiguation in Ontology Population. // In *Proceedings of the 25th International Workshop on Concurrency, Specification and Programming (CS&P)* pages 101–112. Humboldt-Universitat zu Berlin, 2016.
5. **N. Garanina, E. Sidorova, I. Kononenko, S. Gorlatch.** Using Multiple Semantic Measures For Coreference Resolution In Ontology Population. // In *International Journal of Computing*, 16(3):166–176, 2017.

6. **Garanina N., Zyubin V., Liakh T.** Ontological Approach to Organizing Specification Patterns in the Framework of Support System for Formal Verification of Distributed Program Systems // System Informatics, No. 9, 2017 (p. 111-132)
7. **Hepp M., Leymann F., Domingue J., Wahler A., Fensel D.** Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. // IEEE International Conference on e-Business Engineering(ICEBE 2005). Beijing, China, October 18-20, 2005. P. 535-540.
8. **Hepp M., Dumitru R.** An Ontology Framework for Semantic Business Process Management. // Proceedings of Wirtschaftsinformatik 2007, February 28 - March 2, 2007, Karlsruhe.
9. **Herre H.** General Formal Ontology(GFO): A Foundational Ontology for Conceptual Modelling. // Theory and Applications of Ontology: Computer Applications. P. 297-345.
10. **Holzmann G. J.** The Spin Model Checker: Primer and Reference Manual.// Addison Wesley Pub, 2003. P. 608
11. S. Salamah, A. Q. Gates, V. Kreinovich. Validated patterns for specification of complex LTL formulas. In *Journal of Systems and Software*, 85(8):1915–1929, 2012.
12. **Shanmugham S.G., Roberts C.A.** Application of graphical specification methodologies to manufacturing control logic development: A classification and comparison // International Journal of Computer Integrated Manufacturing, Vol. 11, Iss. 2. 1998. P. 142-152. https://doi.org/10.1080/095119298130886
13. **Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J., and Lee, J.** The Process Specification Language(PSL) Overview and Version 1.0 Specification. // NIST Report(NISTIR) 6459, Jan. 1999
14. P.Y.H. Wong, J. Gibbons. Property Specifications for Workflow Modelling. In *Proceedings of Integrated Formal Methods (IFM 2009)*, volume 5423 of *Lecture Notes in Computer Science*, pages 166–180. Springer, Berlin, Heidelberg, 2009.
15. J. Yu, T. P. Manh, J. Han et al. Pattern based property specification and verification for service composition. In *Proceedings of 7th International Conference on Web Information Systems Engineering (WISE)*, volume 4255 of *Lecture Notes in Computer Science*, pages 156–168. Springer-Verlag, 2006.

A new method of verification of security protocols

Andrew M. Mironov

Moscow State University
Faculty of Mechanics and Mathematics

amironov66@gmail.com

Abstract. In the paper we introduce a process model of security protocols, where processes are graphs with edges labelled by actions, and present a new method of specification and verification of security protocols based on this model.

1 Introduction

1.1 Security protocols and their properties

A **security protocol** (**SP**) is a distributed algorithm that determines an order of message passing between several agents. Examples of such agents are computer systems, bank cards, people, etc. Messages transmitted by SPs can be encrypted. We assume that encryption transformations used in SPs are perfect, i.e. satisfy some axioms expressing, for example, an impossibility of extraction of open texts from ciphertexts without knowing the corresponding cryptographic keys.

In this paper we present a new model of SPs based on Milner's Calculus of Communicating Systems [1] and theory of processes with message passing [2]. This model is a graph analog of a Calculus of Cryptographic Protocols (**spi-calculus**, [3]). It can serve as a theoretical foundation for a new method (presented in the paper) of **verification** of SPs, where verification means a constructing of mathematical proofs that SPs meet the desired properties. Examples of such properties are **integrity** and **secrecy**. These properties are defined formally, as some conditions expressed in terms of an observational equivalence.

1.2 Verification of security protocols

There are examples of SPs ([4]–[8]) which were used in safety-critical systems, however it turned out that the SPs contain vulnerabilities of the following forms:

– agents involved in these SPs can receive distorted messages (or lose them) as a result of interception, deletion or distortion of transmitted messages by an adversary, that violates the integrity property,
– an adversary can find out a confidential information contained in intercepted messages as a result of erroneous or fraudulent actions of SP agents.

These examples justify that for SPs used in safety-critical systems it is not enough informal analysis of required properties, it is necessary

– to build a **mathematical model** of an analyzed SP,
– to describe security properties of the analyzed SP as mathematical objects (e.g. graphs, or logical formulas), called a **formal specification**, and

– to construct a mathematical proof that the analyzed SP meets (or does not meet) the formal specification, this proof is called a **formal verification**.

In the process model described in the paper SPs and their formal specifications are represented by processes with message passing. Many important properties of SPs (in particular, integrity and secrecy) can be expressed as observational equivalence of such processes.

One of the most significant advantages of the proposed process model of SPs is a low complexity of proofs of correctness of SPs. In particular, there is no need to build a set of all reachable states of analyzed SPs, if the set of all these states and transmitted messages is unbounded.

Among other models of SPs most popular are logical models ([9]–[13]). These models provide possibility to reduce the problem of verification of SPs to the problem of proofs of theorems that analyzed SPs meet their specifications. Algebraic and logical approaches to verification are considered also in [14]–[16].

2 Description of a process model of security protocols

In the process model described below SPs and formal specifications of their properties are represented by graphs, whose edges are labeled by **actions**. Actions are expressions consisting of terms and formulas.

2.1 Variables, constants, terms

We assume that there are given a set $\mathcal{X}$ of **variables**, a subset $\mathcal{K} \subseteq \mathcal{X}$ of **keys**, and a set $\mathcal{C}$ of **constants**. A set $\mathcal{E}$ of **terms** is defined inductively:

– $\forall\, x \in \mathcal{X},\ \forall\, c \in \mathcal{C}\quad x$ and c are terms,
– for each list $e_1, \ldots, e_n$ of terms the record $e_1 \ldots e_n$ is a term,
  (if the above list is empty, then the corresponding term is denoted by ε),
– $\forall\, k \in \mathcal{K},\ \forall\, e \in \mathcal{E}$ the record $k(e)$ is a term (called an **encrypted message (EM)**, this term represents a result of an encryption of e on the key k).

Terms are designed for a representation of messages transmitted between participants of communications, a term of the form $e_1 \ldots e_n$ represents a composite message consisting of messages corresponding to the components $e_1, \ldots, e_n$. $\forall\, e \in \mathcal{E}$ the set of variables occurred in e is denoted by X_e. If terms e, e' have the form $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_{n'}$, respectively, then the record ee' denotes the term $e_1, \ldots, e_n e'_1, \ldots, e'_{n'}$, and $\forall\, e \in \mathcal{E}\ \ \varepsilon e = e\varepsilon = e$.

2.2 Formulas

Elementary formulas (EFs) are records of the form $e = e'$ and $e \in E$ (where $e, e' \in \mathcal{E}$, and E is a subset of $\mathcal{E}$). A **formula** is a conjunction of EFs. The symbols $\top$ and $\bot$ denote true and false formulas respectively (for example, $\top = (c_1 = c_1)$, $\bot = (c_1 = c_2)$, where c_1 and c_2 are different constants). A set of formulas is denoted by $\mathcal{B}$. $\forall\, b \in \mathcal{B}\ \ X_b$ is a set of all variables occurring in b.

$\forall\, b_1, b_2 \in \mathcal{B}\quad b_1 \leq b_2$ means that b_2 is a logical consequence of b_1 (where the concept of a logical consequence is defined by a standard way).

If $b_1 \leq b_2$ and $b_2 \leq b_1$, then b_1 and b_2 are assumed to be equal.

$\forall\, k, k' \in \mathcal{K},\ \forall\, e, e' \in \mathcal{E}$ the formulas $k(e) = k'(e')$ and $(k = k') \wedge (e = e')$ are assumed to be equal. The records $e_1 =_b e_2$ and $e \in_b E$ means that $b \leq (e_1 = e_2)$ and $b \leq (e \in E)$ respectively.

2.3 Closed sets of terms

Let $E \subseteq \mathcal{E}$ and $b \in \mathcal{B}$. The set E is said to be b–**closed** if

- $\big(\forall\, i = 1, \ldots, n \;\; e_i \in E\big) \Leftrightarrow e_1 \ldots e_n \in E$,
- $\forall\, k \in E \cap \mathcal{K} \;\; \big(e \in E \Leftrightarrow k(e) \in E\big)$,
- $\forall\, e, e' \in \mathcal{E} \quad (e =_b e') \;\Rightarrow\; \big(e \in E \;\Leftrightarrow\; e' \in E\big)$.

Closed sets of terms are used for representation of sets of messages which can be known to an adversary. The above conditions correspond to operations which an adversary A can perform with his available messages:

- if A has $e_1, \ldots, e_n$, then it can compose the message $e_1 \ldots e_n$,
- if A has $e_1 \ldots e_n$, then it may get its components e_1, …, e_n,
- if A has k and e, where k is a key, then it can create a EM $k(e)$,
- if A has an EM $k(e)$ and a key k, then it can decrypt $k(e)$, i.e. get e.

Theorem 1. $\forall\, E \subseteq \mathcal{E}$, $\forall\, b \in \mathcal{B}$ there is a least (w.r.t. an inclusion of sets) b–closed set $E^b \subseteq \mathcal{E}$, such that $E \subseteq E^b$. ■

Let $D_1, D_2 \subseteq \mathcal{E}$, and $b_1, b_2 \in \mathcal{B}$. A binary relation $\mu \subseteq D_1^{b_1} \times D_2^{b_2}$ is said to be a **similarity** between (D_1, b_1) and (D_2, b_2), if $\forall\,(e_1, e_2) \in \mu$

- $\forall\, e'_1, e'_2 \in \mathcal{E} \quad (e'_1, e_2) \in \mu \Leftrightarrow (e_1 =_{b_1} e'_1), \;\; (e_1, e'_2) \in \mu \Leftrightarrow (e_2 =_{b_2} e'_2)$,
- the conditions $\exists\; e_i^1, \ldots, e_i^n \in D_i^{b_i} : (e_i =_{b_i} e_i^1 \ldots e_i^n) \quad (i = 1, 2)$ are equivalent, and if these conditions hold, then $\forall\, i = 1, \ldots, n \;\; (e_1^i, e_2^i) \in \mu$,
- the conditions $\exists\, k_i, e'_i \in D_i^{b_i} : (e_i =_{b_i} k_i(e'_i)) \quad (i = 1, 2)$ are equivalent, and if these conditions hold, then $(k_1, k_2) \in \mu$ and $(e'_1, e'_2) \in \mu$.

A set of all similarities between (D_1, b_1) and (D_2, b_2) is denoted by the record $Sim\big((D_1, b_1), (D_2, b_2)\big)$.

2.4 Actions

An **action** is a record of one of the three kinds: an input, an output, an internal action. Inputs and outputs are associated with an **execution**, defined below.

- An **input** is an action of the form $e?e'$, where $e, e' \in \mathcal{E}$. An execution of this action consists of a receiving a message through a channel named e, and writing components of this message to variables occurring in e'.
- An **output** is an action of the form $e!e'$, where $e, e' \in \mathcal{E}$. An execution of this action consists of a sending a message e' through a channel named e.
- An **internal action** is an action of the form b, where $b \in \mathcal{B}$.

The set of all actions is denoted by $\mathcal{A}$, $\forall\, a \in \mathcal{A}$ a set of variables occurred in a is denoted by X_a.

2.5 Processes with a message passing

Processes with a message passing are intended for description of SPs and formal specifications of their properties.

A **process with a message passing** (called below briefly as a **process**) is a tuple $P = (S, s^0, R, b^0, D^0, H^0)$, where

- S is a set of **states**, $s^0 \in S$ is an **initial state**,
- $R \subseteq S \times \mathcal{A} \times S$ is a set of **transitions**, each transition $(s, a, s') \in R$ is denoted by the record $s \xrightarrow{a} s'$,
- $b^0 \in \mathcal{B}$ is an **initial condition**,
- $D^0 \subseteq \mathcal{E}$ is a set of **disclosed terms**, values of these terms are known to both the process P and the environment at the initial moment, and
- $H^0 \subseteq \mathcal{X}$ is a set of **hidden variables**.

A set of all processes is denoted by $\mathcal{P}$, $\forall\, P \in \mathcal{P}$ the records S_P, s_P^0, R_P, b_P^0, D_P^0, H_P^0 denote the corresponding components of P. A set of variables occurring in P is denoted by X_P. A process P such that $R_P = \emptyset$ is denoted by $\mathbf{0}$.

A transition $s \xrightarrow{a} s'$ is said to be an **input**, an **output**, or an **internal transition**, if a is an input, an output, or an internal action, respectively.

A process P can be represented as a graph (denoted by the same symbol P): its nodes are states from S_P, and edges are corresponded to transitions from R_P: each transition $s \xrightarrow{a} s'$ corresponds to an edge from s_1 to s_2 labelled by a. We assume that for each process P under consideration the graph P is acyclic.

2.6 An execution of a process

An execution of a process $P \in \mathcal{P}$ can be informally understood as a walk on the graph P starting from s_P^0, with an execution of actions that are labels of traversed edges. At each step $i \geq 0$ of this walk there are defined

- a state $s_i \in S_P$ of the process P at the moment i,
- a condition $b_i \in \mathcal{B}$ on variables of P at the moment i, and
- a set $D_i \subseteq \mathcal{E}$ of **disclosed messages** at the moment i, i.e. messages known to both the process P and the environment at the moment i.

An **execution** of a process $P \in \mathcal{P}$ is a sequence of the form

$$(s_P^0, b_P^0, D_P^0) = (s_0, b_0, D_0) \xrightarrow{a_1} (s_1, b_1, D_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (s_n, b_n, D_n)$$

where $\forall\, i = 1, \ldots, n \;\; (s_{i-1} \xrightarrow{a_i} s_i) \in R_P$, $(b_i, D_i) = (b_{i-1}, D_{i-1})a_i$, and

$$\forall\, b \in \mathcal{B}, D \subseteq \mathcal{E}, a \in \mathcal{A} \quad (b, D)a = \begin{cases} (b, D \cup \{e\}), \text{ if } a = d?e \text{ or } d!e, \text{ and } d \in D^b, \\ (b \wedge a, D), \text{ if } a \in \mathcal{B}, \\ \text{undefined, otherwise.} \end{cases}$$

We assume that a value of each variable $x \in H_P^0$ is unique and unknown to an environment of P at the initial moment of any execution of P.

A set of all executions of P can be represented by a labelled tree T_P, where

- a root t_P^0 of the tree T_P is labelled by the triple (s_P^0, b_P^0, D_P^0), and
- if the set of edges of P outgoing from s_P^0 is $\{s_P^0 \xrightarrow{a_i} s_i \mid i = 1, \ldots, m\}$, then for each $i \in \{1, \ldots, m\}$, such that $\exists\, (b_i, D_i) = (b_P^0, D_P^0)a_i$,
  - T_P has an edge of the form $t_P^0 \xrightarrow{a_i} t_i$, and
  - a subtree growing from t_i is T_{P_i}, where $P_i = (S_P, s_i, R_P, b_i, D_i, H_P^0 \setminus D_i^{b_i})$.

The set of nodes of T_P is denoted by the same record T_P. For each node $t \in T_P$ the records s_t, b_t, D_t denote corresponding components of a label of t.

$\forall\, t, t' \in T_P$ the record $t \to t'$ means that either $t = t'$, or there is a path in T_P of the form $t = t_0 \xrightarrow{a_1} t_1 \xrightarrow{a_2} \ldots \xrightarrow{a_m} t_m = t'$, where $a_1, \ldots, a_m \in \mathcal{B}$.

2.7 Observational equivalence of processes

In this section we introduce a concept of observational equivalence of processes. This concept has the following sense: processes P_1 and P_2 are observationally equivalent iff for any external observer (which can observe a behavior of P_1 and P_2 by sending and receiving messages) these processes are indistinguishable.

An example of a pair of observationally equivalent processes is the pair

$$P_i = (\{s_i^0, s_i\}, s_i^0, \{s_i^0 \overset{c\,!\,k_i(e_i)}{\to} s_i\}, \top, \{c\}, \{k_i\}) \quad (i = 1, 2). \tag{1}$$

P_1 and P_2 send a unique message via channel c and then terminate. Any process observing an execution of P_1 and P_2 is unable to distinguish them.

Processes $P_1, P_2 \in \mathcal{P}$, are said to be **observationally equivalent** iff there is a binary relation $\mu \subseteq T_{P_1} \times T_{P_2}$ satisfying the following conditions:

1. $\forall\, (t_1, t_2) \in \mu \;\; \exists\, \mu_{t_1,t_2} \in Sim\big((D_{t_1}, b_{t_1}), (D_{t_2}, b_{t_2})\big)$,
2. $(t^0_{P_1}, t^0_{P_2}) \in \mu$, $\forall\, (d_1, d_2) \in \mu_{t^0_{P_1}, t^0_{P_2}} \;\exists\, d \in \mathcal{E} : d_i =_{b^0_{P_i}} d \;\; (i = 1, 2)$,
3. $\forall\, (t_1, t_2) \in \mu$, for each edge $t_1 \overset{a_1}{\to} t_1'$, $\exists\, t_2' \in T_{P_2} : (t_1', t_2') \in \mu$, $\mu_{t_1,t_2} \subseteq \mu_{t_1',t_2'}$,
   - if $a_1 = d_1 \triangleright e_1$ ($\triangleright \in \{?, !\}$), then $\exists\, t, t' \in T_{P_2}$: $t_2 \to t$, $t' \to t_2'$, and $\exists\, d_2, e_2$: $t \overset{d_2 \triangleright e_2}{\to} t'$, $(d_1, d_2) \in \mu_{t_1',t_2'}$, $(e_1, e_2) \in \mu_{t_1',t_2'}$,
   - if $a_1 \in \mathcal{B}$, then $t_2 \to t_2'$,
4. a condition which is symmetric to condition 3: for each pair $(t_1, t_2) \in \mu$, and each edge $t_2 \overset{a_2}{\to} t_2'$ there is a node $t_1' \in T_{P_1}$, such that $(t_1', t_2') \in \mu$, etc.

For example, processes P_i $(i = 1, 2)$ from (1) are observationally equivalent, because in this case T_{P_i} has the form $(s_0^i, \top, \{c\}) \overset{c\,!\,k_i(e_i)}{\to} (s^i, \top, \{c, k_i(e_i)\})$, and the required μ is $\{(s_1^0, s_1), (s_2^0, s_2)\}$.

2.8 Operations on processes

In this section we define operations on processes which can be used for a construction of complex processes from simpler ones.

Prefix action $\forall\, a \in \mathcal{A}$, $\forall\, P \in \mathcal{P}$ $\;\; [a]P$ is a process defined as follows:

$$S_{[a]P} \overset{\text{def}}{=} \{s\} \sqcup S_P, \quad s^0_{[a]P} \overset{\text{def}}{=} s, \quad R_{[a]P} \overset{\text{def}}{=} \{s \overset{a}{\to} s^0_P\} \sqcup R_P,$$
$$b^0_{[a]P} \overset{\text{def}}{=} b^0_P,\; D^0_{[a]P} \overset{\text{def}}{=} X_a \cup D^0_P,\; H^0_{[a]P} \overset{\text{def}}{=} H^0_P.$$

An execution of $[a]P$ can be informally understood as follows: at first the action a is executed, then $[a]P$ is executed just like P.

Choice $\forall\, P_1, P_2 \in \mathcal{P}$ $\;\; P_1 + P_2$ is a process defined as follows: all states of P_1, that also belong to S_{P_2}, are replaced by fresh states, and

$$S_{P_1+P_2} \overset{\text{def}}{=} \{s\} \sqcup S_{P_1} \sqcup S_{P_2}, \quad s^0_{P_1+P_2} \overset{\text{def}}{=} s,$$
$$R_{P_1+P_2} \overset{\text{def}}{=} R_{P_1} \sqcup R_{P_2} \sqcup \{s \overset{a}{\to} s' \mid (s^0_{P_i} \overset{a}{\to} s') \in R_{P_i},\; i \in \{1, 2\}\},$$
$$b^0_{P_1+P_2} \overset{\text{def}}{=} b^0_{P_1} \wedge b^0_{P_2}, \quad D^0_{P_1+P_2} \overset{\text{def}}{=} D^0_{P_1} \cup D^0_{P_2}, \quad H^0_{P_1+P_2} \overset{\text{def}}{=} H^0_{P_1} \cup H^0_{P_2}.$$

An execution of $P_1 + P_2$ can be understood as follows: at first it is selected (non-deterministically) a process $P_i \in \{P_1, P_2\}$ which can execute its first action, and then $P_1 + P_2$ is executed as the selected process.

Parallel composition $\forall\, P_1, P_2 \in \mathcal{P}\ \ (P_1, P_2)$ is a process defined as follows: all variables in $X_{P_1} \setminus D^0_{P_1}$, that also belong to $X_{P_2} \setminus D^0_{P_2}$, are replaced by fresh variables, and

- $S_{(P_1,P_2)} \stackrel{\text{def}}{=} S_{P_1} \times S_{P_2}$, $s^0_{(P_1,P_2)} \stackrel{\text{def}}{=} (s^0_{P_1}, s^0_{P_2})$,
- $R_{(P_1,P_2)}$ consists of the following transitions:
  - $(s_1, s_2) \xrightarrow{a} (s'_1, s_2)$, where $(s_1 \xrightarrow{a} s'_1) \in R_{P_1}$, $s_2 \in S_{P_2}$,
  - $(s_1, s_2) \xrightarrow{a} (s_1, s'_2)$, where $s_1 \in S_{P_1}$, $(s_2 \xrightarrow{a} s'_2) \in R_{P_2}$,
  - $(s_1, s_2) \xrightarrow{\beta} (s'_1, s'_2)$, where $\beta = (d_1 = d_2) \wedge (e_1 = e_2)$ $(s_i \xrightarrow{a_i} s'_i) \in R_{P_i}$ $(i = 1, 2)$, $\{a_1, a_2\} = \{d_1!e_1, d_2?e_2\}$ (such transition is said to be **diagonal**),
- $b^0_{(P_1,P_2)} \stackrel{\text{def}}{=} b^0_{P_1} \wedge b^0_{P_2}$, $D^0_{(P_1,P_2)} \stackrel{\text{def}}{=} D^0_{P_1} \cup D^0_{P_2}$, $H^0_{(P_1,P_2)} \stackrel{\text{def}}{=} H^0_{P_1} \sqcup H^0_{P_2}$.

An execution of (P_1, P_2) can be understood as undeterministic interleaving of executions of P_1 and P_2: at each moment of an execution of (P_1, P_2)

- either one of P_1, P_2 executes an action, and another is in waiting,
- or one of P_1, P_2 sends a message, and another receives this message.

A process $(\ldots(P_1, P_2), \ldots, P_n)$ is denoted by $(P_1, \ldots, P_n)$.

Replication $\forall\, P \in \mathcal{P}$ a **replication** of P is a process $P^\wedge$ that can be understood as infinite parallel composition $(P, P, \ldots)$, and is defined as follows.

$\forall\, i \geq 1$ let P_i be a process which is obtained from P by renaming of variables: $\forall\, x \in X_P \setminus D^0_P$ each occurrence of x in P is replaced by the variable x_i, such that all the variables x_i are fresh. Components of $P^\wedge$ have the following form:

- $S_{P^\wedge} \stackrel{\text{def}}{=} \{(s_1, s_2, \ldots) \mid \forall\, i \geq 1\ \ s_i \in S_P\}$, $s^0_{P^\wedge} \stackrel{\text{def}}{=} (s^0_P, s^0_P, \ldots)$,
- $\forall\, (s_1, \ldots) \in S_{P^\wedge}$, $\forall\, i \geq 1$, $\forall\, (s_i \xrightarrow{a} s) \in R_{P_i}$ $\quad R_{P^\wedge}$ contains the transitions
  - $(s_1, \ldots) \xrightarrow{a} (s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots)$, and
  - $(s_1, \ldots) \xrightarrow{\beta} (s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_{j-1}, s', s_{j+1}, \ldots)$, where $\beta = (d_1 = d_2) \wedge (e_1 = e_2)$, $(s_j \xrightarrow{a'} s') \in R_{P_j}$ for some $j \neq i$, and $\{a, a'\} = \{d_1!e_1, d_2?e_2\}$,
- $b^0_{P^\wedge} \stackrel{\text{def}}{=} b^0_P$, $D^0_{P^\wedge} \stackrel{\text{def}}{=} D^0_P$, $H^0_{P^\wedge} \stackrel{\text{def}}{=} \bigsqcup_{i\geq 1} H^0_{P_i}$.

Hiding $\forall\, P \in \mathcal{P}$, $\forall\, X \subseteq \mathcal{X}$ $\quad P_X \stackrel{\text{def}}{=} (S_P, s^0_P, R_P, b^0_P, D^0_P \setminus X, H^0_P \cup X)$.

If $X = \{x_1, \ldots, x_n\}$, then P_X is denoted by $P_{x_1,\ldots,x_n}$.

Theorem 2. Observational congruence preserves operations of prefix action, parallel composition, replication and hiding. ■

2.9 A sufficient condition of an observational equivalence

Let $P \in \mathcal{P}$. A **labeling of states** of P is a set $\{(b_s, D_s) \mid s \in S\}$, such that

- $S \subseteq S_P$, $\forall\, s \in S\ \ b_s \in \mathcal{B}$ and $D_s \subseteq \mathcal{E}$, $s^0_P \in S$, $b_{s^0_P} = b^0_P$, $D_{s^0_P} = D^0_P$,
- for each transition $(s \xrightarrow{a} s') \in R_P$, if $s' \in S$ then $s \in S$, and in this case
  - if $a = d \triangleright e$, where $\triangleright \in \{?, !\}$, then $d \in D^{b_s}_s$, $b_s \leq b_{s'}$, $D_s \cup \{e\} \subseteq D^{b_{s'}}_{s'}$,
  - if $a \in \mathcal{B}$, then $b_s \wedge a \leq b_{s'}$ and $D_s \subseteq D_{s'}$.

$\forall\, s, s' \in S_P$ the record $s \to s'$ means that either $s = s'$, or there is a set of thansitions of the form $s = s_0 \overset{a_1}{\to} s_1 \overset{a_2}{\to} \ldots \overset{a_m}{\to} s_m = s'$, where $a_1, \ldots, a_m \in \mathcal{B}$.

Theorem 3 (a sufficient condition of an observational equivalence).
Let $P_1, P_2 \in \mathcal{P}$, where $S_{P_1} \cap S_{P_2} = \emptyset$. Then $P_1 \approx P_2$, if there are a binary relation $\mu \subseteq S_{P_1} \times S_{P_2}$ and labelings $\{(b_s, D_s) \mid s \in S_{P_1}\}$, $\{(b_s, D_s) \mid s \in S_{P_2}\}$ of states of P_1 and P_2 respectively, such that

1. each pair $(s_1, s_2) \in \mu$ is associated with $\mu_{s_1 s_2} \in Sim\big((D_{s_1}, b_{s_1}), (D_{s_2}, b_{s_2})\big)$,
2. $(s^0_{P_1}, s^0_{P_2}) \in \mu$, and each element of the set $\mu^0 \overset{\text{def}}{=} \mu_{s^0_{P_1} s^0_{P_2}}$ has the form (x, x), where $x \in D^0_{P_1} \cap D^0_{P_2}$,
3. for each pair $(s_1, s_2) \in \mu$, and each transition $(s_1 \overset{a_1}{\to} s'_1) \in R_{P_1}$ there is a state $s'_2 \in S_{P_2}$, such that $(s'_1, s'_2) \in \mu$, $\mu_{s_1 s_2} \subseteq \mu_{s'_1 s'_2}$, and
   - if a_1 is input or output, then $a_1 = x \rhd e_1$, where $\rhd \in \{?, !\}$, $(x, x) \in \mu^0$, $\exists\, s, s' \in S_{P_2}$: $s_2 \to s$, $s' \to s'_2$, $\exists\, e_2$: $s \overset{x \rhd e_2}{\to} s'$, $(e_1, e_2) \in \mu_{s'_1 s'_2}$,
   - if $a_1 \in \mathcal{B}$, then $s_2 \to s'_2$,
4. a condition which is symmetric to condition 3: for each pair $(s_1, s_2) \in \mu$ and each transition $s_2 \overset{a_2}{\to} s'_2$, $\exists\, s'_1 \in S_{P_1} : (s'_1, s'_2) \in \mu$, etc. ■

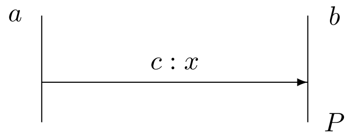
Theorem 4. Let P be a process, $\{(D_s, b_s) \mid s \in S\}$ be a labelling of P, and R_P has an edge $s \overset{a}{\to} s'$ such that $s, s' \in S$, and a has the form $d?k(e)$, where $D^{b_s}_s$ does not contain k and any term of the form $k(e')$. Then $P \approx P'$, where P' is obtained from P by removing the above edge and all unreachable (from s^0_P) states which appear after removing the edge. ■

3 Security protocols

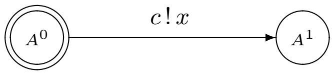
A **security protocol (SP)** is a process $P \in \mathcal{P}$ of the form $(P_1, \ldots, P_n)_X$, where $P_1, \ldots, P_n$ are processes corresponding to **agents** involved in the SP, and $X \subseteq \mathcal{X}$ is a **shared secret** of the agents. In this section we present an application of the proposed approach to description, specification of properties and verification of several examples of SPs, all of them are analogs of examples from [3].

3.1 A message passing through a hidden channel

First example is a simplest SP for a message passing through a hidden channel. This SP consists of a sending of a message x from an agent a to an agent b through a channel named c (where only a and b know the name c of this channel), b receives the message and stores it in variable y, then b behaves like a process P. This SP is represented by the diagram



A behavior of a and b is represented by processes A and B respectively, $A \overset{\text{def}}{=} [c\,!\,x]\,\mathbf{0}$, $B \overset{\text{def}}{=} [c\,?y]\,P$ (where $c \notin P$). The SP is represented by the process $Sys \overset{\text{def}}{=} (A, B)_c$. Graph representations of processes in Sys is the following:

– process A: 

– process B: $$\begin{tikzcd} B^0 \arrow[r, "c\,?\,y"] & P \end{tikzcd}$$

(where $\textcircled{P}$ denotes a subgraph corresponded to the process P)

– process (A, B): $$\begin{tikzcd} A^0B^0 \arrow[r, "c\,!\,x"] \arrow[d, "c\,?\,y"'] \arrow[dr, "y = x"] & A^1B^0 \arrow[d, "c\,?\,y"] \\ A^0P \arrow[r, "c\,!\,x"] & A^1P \end{tikzcd}$$

(where $\textcircled{A^0P}$ and $\textcircled{A^1P}$ denote subgraphs corresponded to copies of P (nodes of these graphs are denoted by A_is, where $i = 0, 1$, and $s \in S_P$), and the arrow from $\textcircled{A^0P}$ to $\textcircled{A^1P}$ denotes a set of corresponding transitions from A^0s to A^1s, where $s \in S_P$).

For the reason of theorem 4, the process $(A, B)_c$ is observationally equivalent to the process $$\begin{tikzcd} A^0B^0 \arrow[r, "y = x"] & A^1P \end{tikzcd}$$.

The process model allows us to formally describe and verify properties of integrity and secrecy of the above SP. These properties are as follows.

– **Integrity** of the SP is the following property: after a completion of the SP agent b receives the same message that has been sent by agent a.
– **Secrecy** of the SP is the following property:
  - for each pair x_1, x_2 of messages, which a can send b by this SP, and
  - for each two sessions of this SP, where the first session is a passing of x_1, and the second one is a passing of x_2,

  any external (i.e. different from a and b) agent, observing an execution of these sessions, is unable to extract from the observed information any knowledge about the messages x_1 and x_2: whether the messages are the same or different (unless these knowledges are not disclosed by participants a, b).
  More accurately, the secrecy property can be described as follows: for any pair x_1, x_2 of messages, which a can send b by an execution of this SP
  - if for any external observer the process $[y = x_1]\,P$ (which describes a behavior of the agent b after receiving x_1) is indistinguishable from the process $[y = x_2]\,P$ (which describes a behavior of b after receiving x_2),
  - then for any sessions of an execution of this SP, where the first one is a passing of x_1, and the second one is a passing of x_2, any external agent, observing the execution of these sessions, can not determine, are identical or different messages transmitted in those sessions.

A formal description and verification of the properties of integrity and secrecy of this SP is as follows.

1. A property of **integrity** is described by the proposition

$$Sys \approx \tilde{Sys} \tag{2}$$

where $\tilde{Sys}$ describes a SP which is defined like the original SP, but with the following modification of b: after receiving a message and storing it in a fresh variable y', the value of y is changed to the value that a really sent.

A behavior of modified b is described by the process $\tilde{B} \overset{\mathrm{def}}{=} [c\,?y']\,[y = x]\,P$, and the process $\tilde{Sys}$ has the form $(A, \tilde{B})_c$.

Now we prove (2). The definition of operations on processes implies that

$$Sys \approx [y = x]\,P, \quad \tilde{Sys} \approx [y' = x]\,[y = x]\,P, \tag{3}$$

that implies (2), because $y' \notin [y = x]\,P$. ■

2. A property of **secrecy** of this SP is described by the implication

$$\begin{array}{l}[y = x_1]\,P \approx [y = x_2]\,P \quad \Rightarrow \quad [x = x_1]\,Sys \approx [x = x_2]Sys \\ \text{(where } x_1, x_2 \text{ are fresh variables).}\end{array} \tag{4}$$

Now we prove (4). The the premise of implication (4) implies the statement

$$[y = x]\,[y = x_1]\,P \approx [y = x]\,[y = x_2]\,P,$$

which is equivalent to the statement

$$[x = x_1]\,[y = x]\,P \approx [x = x_2]\,[y = x]\,P. \tag{5}$$

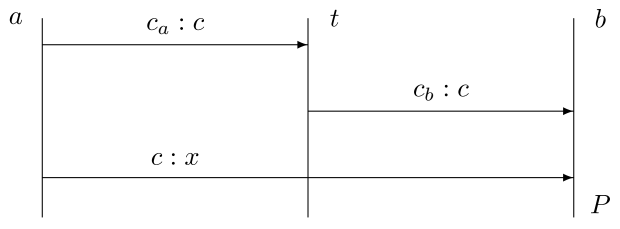
(5) and first proposition in (3) imply

$$[x = x_1]\,Sys \approx [x = x_1]\,[y = x]\,P \approx [x = x_2]\,[y = x]\,P \approx [x = x_2]\,Sys. ■$$

3.2 A SP with a creation of a new channel

Second SP consists of a message passing from a to b, with an assumption that a channel for this passing should be created during the execution of the SP. An auxiliary agent t is used in the SP (t is a trusted intermediary), and it is assumed that a name of a created channel must be known only to a, b, and t.

This SP is represented by the diagram



A behavior of agents a, t, b is represented by the processes A, T, B, where

$$A \overset{\mathrm{def}}{=} [c_a\,!\,c]\,[c\,!\,x]\,\mathbf{0}, \quad T \overset{\mathrm{def}}{=} [c_a\,?\,c]\,[c_b\,!\,c]\,\mathbf{0}, \quad B \overset{\mathrm{def}}{=} [c_b\,?\,c]\,[c\,?\,y]\,P.$$

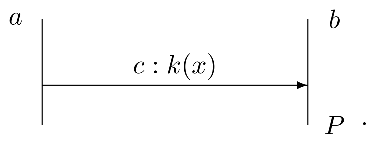
The SP is represented by the process $Sys \overset{\mathrm{def}}{=} (A_c, T, B)_{c_a, c_b}$.

A formal description of integrity and secrecy of the SP is represented by propositions (2) and (4), where $\tilde{Sys} \overset{\mathrm{def}}{=} (A_c, T, \tilde{B})_{c_a, c_b}$, $\tilde{B} \overset{\mathrm{def}}{=} [c_b\,?\,c]\,[c\,?\,y']\,[y = x]\,P$.

3.3 A passing of an encrypted message

Third example is a SP, which involves agents a and b having a common key k (only a and b know k), a and b can encrypt and decrypt messages by this key using a symmetric encryption system. The SP is as follows:

- a sends b a ciphertext $k(x)$ through an open channel c,
- b receives the ciphertext, decrypts it, stores the extracted message x in the variable y, then behaves as a process P.

This SP is represented by the diagram 

.

A behavior of agents a and b is represented by the processes A and B, where $A \overset{\text{def}}{=} [c\,!\,k(x)]\,\mathbf{0}$, $B \overset{\text{def}}{=} [c\,?k(y)]\,P$, and the SP is represented by $Sys \overset{\text{def}}{=} (A, B)_k$.

A formal description of the properties of integrity and secrecy of the SP is represented by (2) and (4), where $\tilde{Sys} \overset{\text{def}}{=} (A, \tilde{B})_k$, $\tilde{B} \overset{\text{def}}{=} [c\,?\,k(y')]\,[y = x]\,P$.

An integrity property of the SP is proposition (2), which in this case has the form $([c\,!\,k(x)]\,\mathbf{0},\ [c\,?k(y)]\,P)_k \approx ([c\,!\,k(x)]\,\mathbf{0},\ [c\,?k(y')]\,[y = x]\,P)_k$, and can be proven with use of theorem 3. To prove the secrecy property we prove implication (4). With use of theorem 3 it is not so difficult to prove that (3) and the premise of implication (4) imply $Sys \approx [y = x]\,P \approx [y = x']\,P$, that proves (4).

References

1. Milner R. A Calculus of Communicating Systems // Lecture Notes in Computer Science, 1980. Vol. 92. 172 p.
2. Mironov A., A Method of a Proof of Observational Equivalence of Processes, Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation, Pereslavl-Zalessky, 2014, p. 194-222. See also `http://meta2014.pereslavl.ru/papers`, `https://arxiv.org/abs/1009.2259`
3. Abadi M., Gordon A., A Calculus for Cryptographic Protocols: The Spi Calculus, Proceedings of the Fourth ACM Conference on Computers and Communications Security, (1997) 36-47, ACM Press.
4. Denning D., Sacco G., Timestamps in Key Distribution Protocols, Communications of the ACM, Vol. 24, No. 8, (1981) 533-536.
5. Needham R., Schroeder M., Using Encryption for Authentication in large networks of computers, Communications of the ACM, 21(12), (1978) 993-999.
6. Needham R., Schroeder M., Authentication revisited, Operating Systems Review, Vol. 21, No. 1, (1987).
7. Cervesato I., Jaggard A.D., Scedrov A., Tsay J.-K., Walstad C., Breaking and fixing public-key Kerberos, Information and Computation Volume 206, Issues 2-4, (2008), Pages 402-424.
8. Lowe G., Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR, In Proceedings of TACAS, (1996) 147-166, Springer Verlag.
9. Burrows M., Abadi M., Needham R., A Logic of Authentication, ACM Transactions on Computer Systems, 8(1), (1990) 18-36.
10. Syverson P., van Oorschot P.C., On Unifying some Cryptographic Protocol Logics, Proceedings of the 1994 IEEE Computer Security Foundations Workshop VII, (1994) 14-29, IEEE Computer Society Press.
11. Syverson P., Meadows C., A Logical Language for Specifying Cryptographic Protocol Requirements, Proceedings of the 1993 IEEE Computer Security Symposium on Security and Privacy, (1993) 165-177, IEEE Computer Society Press.

12. Paulson L., Proving Properties of Security Protocols by Induction, Proceedings of the IEEE Computer Security Foundations Workshop X, (1997) 70-83, IEEE Computer Society Press.
13. Brackin S., A State-Based HOL Theory of Protocol Failure, (1997), ATR 98007, Arca Systems, Inc., http://www.arca.com/paper.htm.
14. Mark D. Ryan and Ben Smyth, Applied pi calculus, in: Formal Models and Techniques for Analyzing Security Protocols, Edited by Veronique Cortier, 2011 IOS Press, p. 112-142.
15. M. Abadi, B. Blanchet, C. Fournet. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. [Research Report] ArXiv. 2016, pp.110. `hal-01423924`, `https://arxiv.org/abs/1609.03003`
16. Ricardo Corin, Analysis Models for Security Protocols, Enschede, The Netherlands, 2006.

Translation from Event-B into Eiffel

Sofia Reznikova and Victor Rivera

Innopolis University, Russian Federation
({s.reznikova,v.rivera}@innopolis.ru)

Abstract. Formal modeling languages play a key role in the development of software since they enable users to prove correctness of system properties. However, it is not always clear how to map a formal model to a specific programming language. This paper presents a translator tool from Event-B models into Eiffel programs, making use of Event-B's mathematical proofs and Eiffel's Design-by-Contract methodologies. The tool has been tested on several Event-B models.

1 Introduction

Modeling methodologies are used in software development to foresee how the resulting system will behave prior to building it. This stage might reveal hidden errors that can be handled at a smaller cost. Event-B is an example of such a methodology. Event-B is a formal method for system-level modeling and analysis [1]. It uses set-theory notation, refinement techniques to model different abstraction levels and mathematical proofs to ensure traceability from the requirements to the model. Event-B comes with an IDE, Rodin, which is based on Eclipse.

The main idea of the language, according to [1], is to structure the work on a system in a way that is similar to how physicists and architects do their work – creating initial representations that first appear in the mind and only after that constructing actual physical artifacts. The main components of any system are states and transitions that are represented as Contexts and Machines in Event-B. Contexts consist of sets, constants and axioms. Machines contain variables, invariants, and events including *guards* (a condition that must hold for an event to be executed), and *actions* (the actual semantics of an event). The IDE can be extended with various plug-ins, among them several translators from Event-B into programming languages have been developed: EventB2Java [10,9], EB2ALL [5], EventB2Dafny [2] and others.

The current paper presents the implementation of a Rodin plug-in that translates Event-B models into Eiffel [6], an object oriented programming language. Eiffel's main aim is to increase reliability of the software. One of its main principles is Design by Contract, a way of designing software with the main idea to define a contract between a client and a supplier. This contract ensures that if there is an error in the program, it is clear on whose part it originated and where [6]. This is made possible by class invariants, pre- and post-conditions in methods.

The paper is structured as follows. Section 2 describes the implemented components of the plug-in. Section 3 shoes the Event-B models used for testing and evaluation of the plug-in. Finally, Section 4 is devoted for conclusions, outlining potential improvements and directions for future work.

2 Implementation

The Rodin plug-in is an implementation of the rules described in [12]. Machines are translated to Eiffel classes. Event-B Events and variables are features and variables of the translated class, respectively. Guards and actions are translated as preconditions and body of the features. Contexts are being translated as class constants. The translation takes advantage of the embedded Design-by-Contract mechanism defined in Eiffel, for instance, Machine invariants are naturally translated to class invariants.

Another part of the implementation required translation of the mathematical language of Event-B. There are in total 90 symbols denoting different mathematical formulas. Some of them exist natively in Eiffel while others had to be implemented.

The implementation was done in several steps: first the structure for the Rodin IDE plug-in was set-up, then the information about the model was retrieved from the database and, finally, the resulting elements were translated into Eiffel.

2.1 Event-B and Rodin Structure

The plug-in's functionality is realized based on extensions and extension points that define the point of contact between different programs. The package responsible for Rodin extensions is `org.rodinp.core`. It provides an interface via which different extensions can communicate and provide extension points.

Event-B package – `org.eventb.core.ast` – provides an Abstract Syntax Tree (AST) of the system modeled in Rodin. This is a tree representation of the syntactic structure of an Event-B model. A visitor has been implemented to traverse it. It translates the mathematical notation into Eiffel code step by step.

2.2 The General Structure of the Tool

The packages included in the tool are `plug-in` and `rodinDB` (implementation of the tool can be found in [8]). The `plug-in` package contains `GenCodeEiffel.java` that is the entry point (defines the order of the translation), and `Translator.java` that implements a Visitor (`ISimpleVisitor2`) that parses the formulas and traverses the AST. The `rodinDB` package contains `RodinDBElements.java` that deals with retrieving information from the Rodin Database.

Each Rodin project has a set of children of class `IRodinElement` which also have `IInternalElements` in them. Depending on what type the internal elements is, it is possible to retrieve the information regarding machines and contexts from Rodin database. The package includes 14 methods that handle this task.

2.3 Traversing the Abstract Syntax Tree

Rodin provides the Abstract Syntax Tree (AST) of an Event-B model. It is then necessary to implement a Visitor to traverse the AST to translate the model parts into Eiffel.

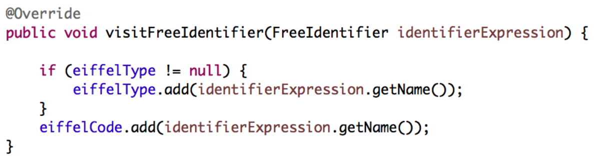
The next step after retrieving information from the Rodin database is to parse the received formulas and to organize them in a way suitable for Eiffel translation. The wrapper methods dealing with parsing are created for almost each method from the `rodinDB` package. The visitor implements *parsePredicate()*, that parses an Event-B predicate, *parseExpression()* that parses an Event-B expression and *parseAssignment()* that parses assignments.

As parameters they take a String-representation of a formula and launch the pass through the AST. The methods in `ISimpleVisitor2` are overridden to handle visits of different branches of the tree such as Atomic Expressions (covers standalone integers, natural numbers, empty sets, booleans and others), Binary Predicates (implication and equality), Becomes Equal To (assignment to a variable or a parameter), Associative Expressions (unions, intersections, backward and forward compositions, addition, multiplication) and many others. The full set of types is described in [1].

2.4 Translating into Eiffel

As each AST branch for a specific formula is visited, Eiffel code is generated and added to the `eiffelCode` Array List that aggregates the translation and then returns it to the code generator (`GenCodeEiffel.java`).

An excerpt of the visitor is shown in Figure 1. A free identifier is any variable from a machine or an event. First the method checks whether the translation is done to retrieve types or to translate event or machine parameters. Depending on the result, code is added to different places.



```java
@Override
public void visitFreeIdentifier(FreeIdentifier identifierExpression) {

    if (eiffelType != null) {
        eiffelType.add(identifierExpression.getName());
    }
    eiffelCode.add(identifierExpression.getName());
}
```

Fig. 1. Free Identifier Translation

3 Evaluation

Several Event-B models were used in the testing phase of the plug-in. This phase also captures how much and accurate (as without compilation errors in Eiffel) of the Event-B mathematical language gets translated. Table 1 shows the results for all the tested models: first column is the Event-B model taken from the literature; the second column indicates how much code is correctly translated. For instance, 94% of the `MIO` model is being translated, the remaining 6% belong to the limitations of the tool (discussed later); and the third column is how much code results in compilation errors in Eiffel.

Social Event Planner: This model was used for testing during the implementation stage. This model is described in more detail in [11]. It is a model for planing social events. The functionality includes creating events, inviting people to them and setting up permissions for inviting other people. This model defines its own sets (PERSON and CONTENTS). They are translated as Eiffel user-defined classes.

An example Event-B event `create_account` is shown on Figure 2. ANY declares parameters of the event, WHERE denotes guards (necessaries conditions to hold for the event to be triggered) of the event, THEN are the actions of the event. Figure 3 depicts the output of the plug-in. The output is a translation of Event-B event (in Figure 2) into Eiffel. There are two **require** statements corresponding to two guards (from Event-B) that ensure that the variables belong to the sets they are supposed to. The **do** statements assign translated expressions to the variables `contents, persons, owner and pages`, initializing them with an appropriate type. In Eiffel, **create** s a keyword used to create instances a classes, similar to **new** in Java or C++.

Class EBSET is a class created specifically for the translation of sets from Event-B. It inherits most of Eiffel set's functionality but allows more flexibility. It is part of `eb_math_lang` package that also includes natural numbers (EBNAT), integers, ranges and relations.

```
ANY
    c1
    p1
WHERE
    grd1 : $p1 \in PERSON \setminus persons$
    grd2 : $c1 \in CONTENTS \setminus contents$
THEN
    act1 : $contents := contents \cup \{c1\}$
    act2 : $persons := persons \cup \{p1\}$
    act3 : $owner := owner \cup \{c1 \mapsto p1\}$
    act4 : $pages := pages \cup \{c1 \mapsto p1\}$
END
```

Fig. 2. *create_ account* event

```
create_account (p1: PERSON; c1: CONTENTS)
    require
        grd1: PERSON.difference (persons).has (p1)
        grd2: CONTENTS.difference (contents).has (c1)
    do
        contents.assigns ((contents).union (create { EBSET[CONTENTS]}.singleton (c1)))
        persons.assigns ((persons).union (create { EBSET[PERSON]}.singleton (p1)))
        owner.assigns ((owner).union (create { EBREL[CONTENTS,PERSON]}.vals (<<(create
{ EBPAIR[CONTENTS,PERSON]}. make (c1, p1))>>))
        pages.assigns ((pages).union (create { EBREL[CONTENTS,PERSON]}.vals (<<(create
{ EBPAIR[CONTENTS,PERSON]}. make(c1, p1))>>))
    end
```

Fig. 3. Eiffel Code for *create_ account* event

MIO – Bus Transportation System: This model includes several entities: buses, bus stations, people, doors and sensors. It regulates bus transportation and is described in [3]. There are six refinements with each adding new entities and concepts. It is an extensive model with 21 actions in the initialisation event and 15 invariants for the last refinement.

Binary and Linear Search: These two models represented *binary* and *linear* search algorithms. The former one looks for a number, dividing each subsection into two, and the latter goes through a set of numbers in a linear way.

The binary search included three events apart from the initialisation (`inc`[rement], `dec`[rement] and `found`). The model for linear search included a similar `found` event as well as `progress` event that continued to search for the number if the correct one is not found.

In general, the tool is able to translate 86% of the 90 symbols that can be used with Event-B. Although the remaining 14% is not implemented yet, it is noted that they occur rarely in the models.

A common problem for most models (e.g. Social Event Planner, MIO, Binary Search and Sorting the Array) that results in compilation errors is due to the type translation of variables. For instance, the Event-B assignment $owner := owner \cup \{cmt \mapsto owner(rc)\}$ with a set extension ($\{\}$) including more than two expressions poses a problem as the types of the sets should be included into the Eiffel code before visiting the variable in the tree. Therefore, instead of a type, Java's `null` keyword is returned (as the tool is written in Java and for Eiffel this is an undefined keyword). This problem occurs for set extensions (declaring a set between two curly brackets), backward and forward composition of functions.

If the correct types are included into the Eiffel code manually, the compilation returns no errors. As Table 1 shows, these types of errors are not very common, e.g. for Social Event Planner there are only three cases of such statements, for MIO, Binary Search and Sorting the Array there is only one.

Model	**Translated**	**Compiled**
Social Event Planner	100%	98%
MIO	94%	88%
Binary Search	100%	92%
Linear Search	100%	100%
Reversing the Array	78%	78%
Sorting the Array	100%	96%
Finding a minimum	94%	94%
Square root	100%	100%

Table 1. Summary of the results.

Other models (namely, reversing an array and finding a minimum) require those parts of the mathematical language that have not been implemented (Bound Declarations and Bound Identifiers that are used as variables in Quantified Predicates – $\forall$ and $\exists$). This is where most of their errors occur.

The list of Event-B models used in this phase and their translation to Eiffel using the plug-in can be found in [8].

4 Conclusion

This paper presents a Rodin plug-in that implements a translation from Event-B models to the Eiffel programming language. The plug-in enables users to take advantages of Event-B (e.g. refinement) to then translation to Eiffel, which provides an actual implementation of the model, to take advantages of the language (e.g. Design by Contract). The main limitation of the plug-in is that no proof of soundness has been carried out. This paper shows a proof-of-concept and opens up a direction to carry out with the proof.

In order to be able to fully automate the translation, type retrieval for variables that are further down the AST need to be implemented. This corresponds to 14% of the Event-B mathematical language. One of the challenges is to translate choice from set ($x :\in S$) that arbitrarily chooses a value from the set S and choice by predicate ($z :\mid P$) that arbitrarily chooses values for the variable in z that satisfy the predicate P. For this, we plan to make use of ProB or Constraint Programming to assign values that satisfy a predicate. Another direction is related to the translation of Proof Obligations (POs), mathematical formulas, automatically generated by Rodin, to Eiffel. POs need to be proven in order to ensure that a machine is correct [4]. They can be proved either automatically or interactively. By translating Proof Obligations into Specification Drivers [7] it will be possible to formally verify the translated Eiffel code against its contracts.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Cataño, N., Leino, K.R.M., Rivera, V.: The EventB2Dafny Rodin Plug-in. In: Proceedings of the Second International Workshop on Developing Tools As Plug-Ins. pp. 49–54. TOPI '12, IEEE Press, Piscataway, NJ, USA (2012)
3. Catano, N., Rueda, C.: Teaching Formal Methods for the Unconquered Territory. In: Proceedings of the 2Nd International Conference on Teaching Formal Methods. pp. 2–19. TFM '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04912-5_2
4. Hallerstede, S.: On the purpose of event-b proof obligations. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) Abstract State Machines, B and Z. pp. 125–138. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
5. Méry, D., Singh, N.K.: Automatic code generation from event-b models. In: Proceedings of the Second Symposium on Information and Communication Technology. pp. 179–188. SoICT '11, ACM, New York, NY, USA (2011)
6. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (Oct 1992), http://dx.doi.org/10.1109/2.161279
7. Naumchev, A., Meyer, B.: Complete contracts through specification drivers. CoRR abs/1602.04007 (2016), http://arxiv.org/abs/1602.04007
8. Reznikova, S.: Innopolis thesis. https://github.com/sonyareznikova/InnopolisThesis (2018)
9. Rivera, V., Cataño, N.: Translating event-b to jml-specified java programs. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1264–1271. SAC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2554850.2554897
10. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for event-b. Int. J. Softw. Tools Technol. Transf. 19(1), 31–52 (Feb 2017)
11. Rivera, V., Catano, N., Wahls, T., Rueda, C.: Code generation for event-b. International Journal on Software Tools for Technology Transfer (January 2015), https://www.researchgate.net/publication/274036119_Code_Generation_for_Event-B
12. Rivera, V., Lee, J., Mazzara, M.: Mapping event-b machines into eiffel programming language. In: To appear in Proceedings of 6th International Conference in Software Engineering for Defence Applications - SEDA 2018, Rome, Italy, June 7th, 2018 (2018)

A summary of a case-study on platform-independent verification of the square root function in fix-point machine arithmetic*

Nikolay V. Shilov[1], Igor S. Anureev[2], Dmitry Kondratyev[2], Aleksey V. Promsky[2]

[1] Innopolis University, Innopolis, Russia
shiloviis@mail.ru
[2] A.P. Ershov Institute of Informatics Systems RAS, Novosibirsk, Russia
anureev@iis.nsk.su, apple-66@mail.ru, promsky@iis.nsk.su

Abstract. The paper is a progress report and outlines (human-oriented) specification and (pen-and-paper) verification of one particular algorithm to compute the square root function in machine fix-point arithmetics. The function implements Newton method and uses a look-up table for initial approximations. Specification is presented in terms of total correctness assertions with use of precise arithmetic (i.e. with unbounded precision) and the mathematical square root $\sqrt{\ldots}$, the algorithm is presented in pseudo-code with explicit distinction between precise and machine arithmetic, verification is done in Floyd-Hoare style. The primary purpose of the research is to make explicit (axiomatise) properties of the machine arithmetic (in terms of precise arithmetics) that are sufficient to verify one particular algorithm for the square root function. Please refer preprint [3] motivation of this study, literature survey, and for complete proofs. Computer-aided validation of the proofs (using some proof-assistant) is the topic for further studies.

Keywords: *machine arithmetic, exact functions, formal verification, total and partial correctness, Floyd-Hoare method, square root, Newton method, look-up table, fix-point representation*

1 Introduction

Let us specify a *generic* function (say $SQR(\ ,\ )$) for *generic* numeric data types with two parameters: the first parameter is for passing the argument value $Y \geq 0$ and the second — for passing the accuracy value $Eps > 0$; the function is for computing $\sqrt{Y}$ with the accuracy Eps.

The accuracy of this function SQR (i.e. the most wanted property and the only property specified in the standard) can be formally specified by any (or both) of the following two assertions:

- for all type-legal values $y \geq 0$ and $\varepsilon > 0$, $SQR(y, \varepsilon)$ differs from $\sqrt{y}$ by no more than ε, i.e. $|\sqrt{y} - SQR(y, \varepsilon)| \leq \varepsilon$;
- for all type-legal values $y \geq 0$ and $\varepsilon > 0$, $\big(SQR(y, \varepsilon)\big)^2$ differs from y by no more than ε, i.e. $\big|y - \big(SQR(y, \varepsilon)\big)^2\big| \leq \varepsilon$.

Let us fix the first specification since it compares numeric approximation against the precise function.

* This research is supported by Russian Basic Research Foundation grant no. 17-01-00789 *Platform-independent approach to formal specification and verification of standard mathematical functions.*

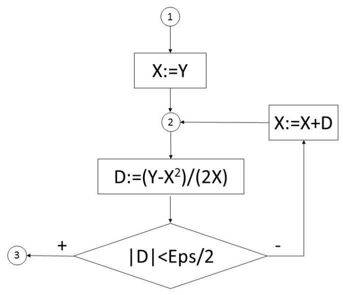


Fig. 1. A flowchart of the algorithm SQR

One may select any reasonable and feasible computation method to approximate $\sqrt{\ }$. (For example, [1] discusses 14 different algorithms and implementations.) For our case-study we select a very intuitive, easy-to-implement and popular in education Newton Method:

1. input the number (to compute the square root) and guess an initial approximation for the root;
2. compute the arithmetic mean between the guess and the number divided by the guess; let this mean be a new guess;
3. repeat step 2 while the difference between the new and the previous guesses isn't small enough (i.e. doesn't feet the use-defined accuracy).

Specification or a generic square root function with a generic numeric data type for input and output values follows below:

$$\begin{aligned}[TYPY\ is\ a\ numeric\ type,\ Y \geq 0:\ TYPE,\ and\ Eps > 0:\ TYPE]\\ SQR(Y,\ Eps)\ [|returned\ value\ - \sqrt{Y}| \leq Eps].\end{aligned} \tag{1}$$

Unfortunately, it is not easy to prove these specifications automatically and formally because of several reasons. The major one is an axiomatization of the computer-dependent floating-point arithmetic. Even a manual pen-and-paper verification of this specification in the precise arithmetic for real numbers (i.e. assuming $TYPY$ to be $\mathbb{R}$) is not a trivial exercise: please refer [3] for verification of the algorithm SQR depicted on Fig. 1 (that implements Newton Method) in the case when $1 < Y$.

2 Towards machine-oriented square root algorithm

The algorithm from Fig 1 may be improved (optimized). Firstly, since we study the case $1 < Y$, then $\sqrt{Y} \leq X \leq Y$ is part of the loop invariant, and hence it makes sense to compute directly the absolute value $AD := \frac{X^2-Y}{2X}$ of D instead of computing $D := \frac{Y-X^2}{2X}$ and then $|D|$ in the loop condition. Next, we may use a *fast* hash function $SUP : (1, \infty) \to (1, \infty)$ to compute *good* initial upper approximations instead of a very rough initial upper approximation used in the algorithm SQR. (For example, it may be rounded-up square roots.) While the first optimization just saves on each loop iteration, the second one reduces the number of loop iterations. Fig. 2 shows a flowchart of the improved algorithm that we refer as the algorithm $ISQR$ in the sequel. In [3] we prove that for every initial value $y > 1$ of the variable Y and every initial value $\varepsilon > 0$ of the variable Eps termination of the improved

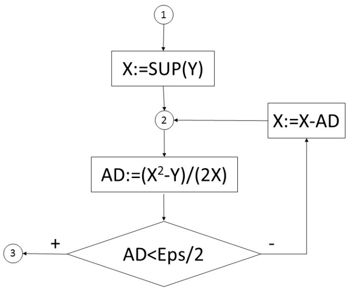


Fig. 2. A flowchart of the improved (optimized) algorithm $ISQR$

square root algorithm is guaranteed after (at most) $1 + \lceil \log_2 \frac{SUP(y) - \sqrt{y}}{\varepsilon} \rceil$ loop iterations, where $\lceil \dots \rceil$ is integer round-up function.

For example if the function SUP returns the rounded-up square roots, $y > 1$ is the initial (input) value of the variable Y, and $\varepsilon > 0$ is the initial (input) value of variable Eps (accuracy) then $0 \leq SUP(y) - \sqrt{y} < 1$ and, hence, an upper bound for the number of the loop iterations in the algorithm $ISQR$ is $1 - \log_2 \varepsilon$ instead of an upper bound $1 + \log_2 \frac{y - \sqrt{y}}{\varepsilon}$ for the number of the loop iterations in the non-optimized algorithm SQR.

Since termination of the improved square root algorithm is guaranteed after (at most) $1 + \lceil \log_2 \frac{SUP(y) - \sqrt{y}}{\varepsilon} \rceil$ loop iterations, then is possible to compute approximations for the square root by a non-adaptive for-loop-based algorithm $FSQR$ which flowchart depicted in Fig. 3. The corresponding correctness assertion is

$$\begin{aligned}&[Y > 1 \ \& \ Eps > 0 \ \& \ \forall y \in (1, +\infty) : \sqrt{y} \leq SUP(y) \leq y)] \\ &\qquad\qquad\qquad\qquad FSQR \ [|X - \sqrt{Y}| < \tfrac{Eps}{2}].\end{aligned} \tag{2}$$

Termination of the algorithm $FSQR$ is guaranteed by design since it is for-loop-based. Informally speaking the partial correctness of the algorithm follows from the partial correctness of the algorithm $ISQR$: while $\frac{X^2 - Y}{2X} \geq Eps$ values of X in both algorithms are equal in each iteration, and then $FSQR$ exercises several more iterations that move value of X closer to $\sqrt{Y}$. Please refer [3] for a formal proof of the partial correctness.

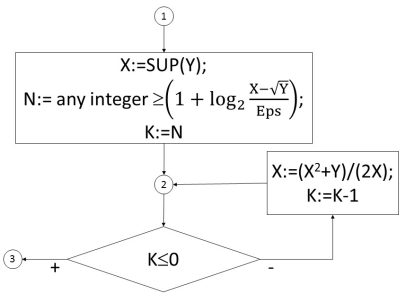


Fig. 3. A flowchart of the non-adaptive for-loop-based algorithm $FSQR$

3 Square root algorithm for fix-point arithmetics

3.1 Axiomatizing fix-point machine arithmetics

One of the problems with the improved and for-loop-based algorithms is how to implement an efficient function SUP. A hint is use of a numeric data type T with a (huge maybe) finite set of values $Val_T \subset \mathbb{R}$ instead of an infinite set $\mathbb{R}$. Then the function SUP may be implemented in two steps:

- define an efficient rounding up function $round : Val_T \to Val_T$,
- pre-compute and memorize a look-up table *root* with good upper approximations for the roots for each of the rounded values.

Further details and steps depend on selected numeric data type.

In our case-study we assume the following properties of the data type T:

- the set of values Val_T is a finite subset of mathematical reals $\mathbb{R}$ such that
  - it comprises all reals in some finite range $[-\inf_T, \sup_T]$, where $\inf_T > 2$, $\sup_T > 2$, with some fixed step $\frac{1}{2} > \delta_T > 0$,
  - and includes all integer numbers Int_T in this range $[-\inf_T, \sup_T]$;
- legal binary arithmetic operations are
  - addition and subtraction; if not the range overflow exception then these operations are precise: they equal to the standard mathematical operations assuming their mathematical results fall in the range $[-\inf_T, \sup_T]$ (and due to this reason are denoted as $+$ and $-$);
  - multiplication $\otimes$ and division $\oslash$; these operations are approximate but correctly rounded in the following sense: for all $x, y \in Val_T$
    - if $x \times y \in Val_T$ then $x \otimes y = x \times y$;
    - if $x/y \in Val_T$ then $x \oslash y = x/y$;
    - if $x \times y \in [-\inf_T, \sup_T]$ then $|x \otimes y - x \times y| < \delta_T/2$;
    - if $x/y \in [-\inf_T, \sup_T]$ then $|x \oslash y - x/y| < \delta_T/2$.
- legal binary relations are equality and all standard inequalities; these relations are precise, i.e. they equal to the standard mathematical relations (and due to this reason are denoted as $=$, $\neq$, $\leq$, $\geq$, $<$, $>$).

Due to the assumptions about the set of values

$$Val_T = \{n \times \delta_T \ : \ n \in \mathbb{Z} \text{ and } -\inf_T \leq n \times \delta_T \leq \sup_T\};$$

according the assumptions about integer values Int_T within the range of Val_T

$$[-2..2] \subseteq Val_T \text{ and } \frac{1}{\delta_T} \in \mathbb{N}.$$

In case when multiplication is guaranteed to be precise (the mathematical product is in Val_T) then let us use the standard notation $\times$ instead of $\otimes$; similarly in case when division is guaranteed to be precise (the mathematical dividend is in Val_T) then let us use the standard notation $/$ instead of $\oslash$.

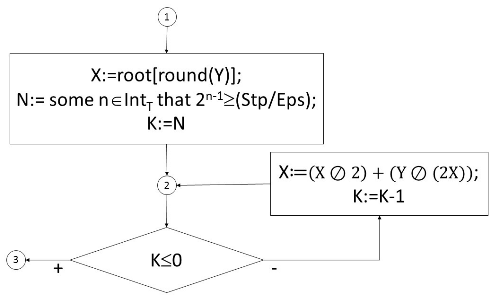


Fig. 4. A flowchart of the square root algorithm $fixSQR$ for fix-point arithmetic

3.2 Fix-point variant of the square root: algorithm, specification, and correctness

A non-adaptive algorithm $FSQR$ (Fig. 3) that uses mathematical operations transforms into algorithm $fixSQR$ (Fig. 4) that uses machine fix-point operations. This algorithm also (as $FSQR$) uses a non-deterministic assignment operator

$$N := \text{ some } n \in Int_T \text{ that } 2^{n-1} \geq \frac{Stp}{Eps}$$

that differs from the assignment by use of *some* instead of *any*: this difference means that later we *select* the value instead of use an *arbitrary* one.

In the new algorithm we use an additional variable Stp for a positive value in Val_T, an array *root*, and a function *round* that have the following properties:

STEP: value of Stp is a multiple of the accuracy Eps, divides $\sup_T$ and is used to define the set $Arg_{Stp} = \{n \times Stp \ : \ n \in \mathbb{N}, \text{ and } 1 < n \times stp \leq \sup_T\}$;

ROOT: *root* is a pre-computed look-up table indexed by Arg_{Stp} such that $root[v] - \delta_T < \sqrt{v} \leq root[v]$ for each index $v \in Arg_{Stp}$;

ROUND: the function $round : Val_T \to Arg_{Stp}$ is a rounding-up such that $round(u) - step < u \leq round(u)$ for each $u \in Val_T$, $u > 1$.

Comment on the STEP property: we consider as a very natural the assumption that

- Stp is a multiple of the accuracy Eps since in the "limit" case $Eps = \delta_T$ and this Eps divides any $Stp \in Val_T$;
- Stp divides the "extreme" value $\sup_T$ because this value should be provided with a pre-computed square root upper approximation.

Specification of the square root algorithm $fixSQR$ follows:

$$\begin{array}{r} [Y \in Val_T \ \& \ Y > 1 \ \& \ Eps \in Val_T \ \& \ Eps > 0 \ \& \\ \& \ \mathbf{STEP} \ \& \ \mathbf{ROOT} \ \& \ \mathbf{ROUND}] \\ fixSQR \ [|X - \sqrt{Y}| < \left(\frac{Eps}{2} \ + \ N \times \delta_T\right)]. \end{array} \quad (3)$$

Termination of the algorithm $fixSQR$ is straightforward since it is a for-loop-based algorithm. Please refer [3] for the pen-and-paper proof of the partial correctness.

4 Summary and conclusion

Firstly we take a very standard Newton method to compute square root, present is as an iterative algorithm SQR, specify it by Hoare total correctness assertion, and prove its validity in the case when input argument is greater than 1, accuracy is positive, and "computer" is precise (i.e. all computations are done in mathematical real numbers); the upper bound of loop iterations of the algorithm SQR is logarithmic.

Next we improve the algorithm SQR by using an auxiliary function to compute better initial approximations for square roots (it results in the algorithm $ISQR$) and then suggest a for-loop-based algorithm $FSQR$ that uses the same auxiliary function, computes a lower bound for the number of iterations that is sufficient to achieve the specified accuracy; both algorithms $ISQR$ and $FSQR$ work with precise arithmetic, but we prove that $FSQR$ achieves better accuracy than $ISQR$, and can achieves better accuracy if to increase the number of the loop iterations.

Then we convert for-loop-based algorithm with precise arithmetic $FSQR$ into algorithm $fixSQR$ with fix-point arithmetic, specify it by total correctness assertion and prove its validity by adjustment of its runs with runs of $FSQR$ with the same input data. Another specifics of the algorithm $fixSQR$ is use a look-up table (arrange as an array) for upper approximations of square roots and rounding-up function.

Use of a machine fix-point arithmetic instead of the precise arithmetic results in situation that more iterations of the loop doesn't always improve accuracy in contrast to $FSQR$. Due to this reason we suggest an other algorithm $mixSQR$ that is a specialised version of the algorithm $fixSQR$.

All proofs in our research were pen-and-paper proofs. So the first of the next topics for further research is to validate all these proofs with aid of some automated proof-assistant. We are going to use ACL2 due to industrial strength of this proof-assistant [2] for platform-specific verification of the standard mathematical functions (but don't rule out alternatives to this assistant).

Nevertheless remark that we attempt and present in this paper an approach that we call platform-independent. Also remark that we don't attempt to build an axiomatization of an "abstract" machine fix-point arithmetics. Instead we just make several explicit assumptions about machine arithmetic (and how it relates to the precise arithmetic) that are sufficient to validate specifications and algorithms with machine arithmetic by using its relations with specifications and algorithms with precise arithmetics. We believe that our assumptions about machine arithmetic are valid for many platforms and they are easy to check. So another topic for further research is to prove an "existence theorem", i.e. to give examples of platforms that use fix-point machine arithmetics that satisfies our axiomatization.

Finally let us mention one more research topic — to find an "optimal balance" between size of the array $root$ with initial upper approximations for square roots for selected arguments, number of iterations of the loop in the algorithm $fixSQR$, and accuracy of the square root approximation: if ε and s are values of the variables Eps and Stp then the array size is $\frac{\sup_T}{s}$, number of iterations may be any $n \geq \left(1 + \lceil \log_2 \frac{s}{\varepsilon} \rceil\right)$, and accuracy $|X - \sqrt{Y}|$ is less than $\left(\frac{s}{2^n} \ + \ n \times \delta_T\right)$.

Acknowledgement: Authors would like to thanks Dmitry Nadezhin for discussions, comments, suggestions, and consultations on the case study presented in this paper and in the preprint [3].

References

1. El-Magdoub M.H. Best Square Root Method – Algorithm – Function (Precision VS Speed). 2010. Available at https://www.codeproject.com/Articles/69941/Best-Square-Root-Method-Algorithm-Function-Precisi.
2. Grohoski G. Verifying Oracle's SPARC Processors with ACL2. Slides of the Invited talk for 14th International Workshop on the ACL2 Theorem Prover and Its Applications. Available at http://www.cs.utexas.edu/users/moore/acl2/workshop-2017/slides-accepted/grohoski-ACL2_talk.pdf.
3. Shilov N.V., Anureev I.S., Berdyshev M., Kondratev D.A., Promsky A.V. Towards platform-independent verification of the standard mathematical functions: the square root function. 2018. Available at https://arxiv.org/abs/1801.00969.

Building a process of trustworthy software developing based on BDD and ontology approaches with further formal verification

Sergey Staroletov

Polzunov Altai State Technical University, Barnaul, Russia
serg_soft@mail.ru

Abstract. This paper is devoted to propose an approach to building a reliable software with using specification based analysis, based on the well-known approach from industrial software developing, BDD (Behaviour-Driven Development), and ontology approach to try modeling requirements for developing software system with further verification tasks.

1 Motivation

Nowadays software production process is moving from an art which was firstly available among a small number of well-qualified software engineers to a normal production process in the teams of software developers with involved members of different skill and abilities, and there the problem of trustworthy code is becoming the major.

It is known that proof of program correctness - is an undecidable problem, so we constantly need methods to decrease the count of errors in production. And to improve the quality of software production, ordinary developers should follow an approved process of program creation, which should take in account possible methods of software creation, types of possible errors, typical requirements, methods of testing and verification of a complete code, etc. This process could be created and approved by the software community (from whom it is possible to get actual software production techniques) in cooperation with some academic institutions (from whom it is possible to get methods to model and prove properties of a program).

To study methods of avoiding the errors we definitely should know what is the error? In this paper, the error will be treated as an inconvenience to a given specification (it could come from a customer for a given software or from community experience in the field related to a produced software).

Furthermore here it exists one big issue in a modern software world - where to get a specification for software developing? Modern software creation processes (for example, the Scrum process) based on tasks which should be done in short iterations to produce the code which needs to be in production as soon as possible. That tasks could be decomposed by the business-analyst or project manager, and, in theory, they should be coming from the specification (from the document like the scope of work) but in the real life that documents are designed in the simplified form and don't cover the actual system is being developed.

The idea in current research in progress here is to develop a software based on specification (possible at the same time), then extract requirements from this specification and check the correctness of software with respect to the specification. To solve the issue with specification absence it is possible to use latest software engineering achievements. The BDD (Behaviour-Driven Developing) is an approach to make a software from scratch together with the specification writing and based on it (specified behaviour is driving us to write a code). It will be discussed further and it should be stressed here that the special language for behaviour specifications writing has already been developed and it is successfully being used in a large number of software companies. And the main advantage here that is the customer could be involved in the process and he can write those specifications by himself. So, we could use it and integrate into the process of robust software creation, because now the customer can see the possible errors as problems in specified behaviour.

With using it we could abandon the unnecessary and superfluous trying to create yet another language and tool-set which will never be used in the software companies and by customers who are trying to make a scope of work.

The results were obtained within the RFBR grant (project No. 17-07-01600).

2 Related Work

This work is based on Gherkin [1] - a BDD specification language. The language describes software system as a set of features and a set of behaviour scenarios inside each feature. The following example specifies a test for simple division method:

```
Feature: Software Calculator
  I want to create my software calculator
  Scenario:
  Given I have my software calculator
    When I have entered 10 as first operand
    And I have entered 2 as second operand
    And I press 'Div'
  Then The result should be 5
```

Here Feature, Scenario, Given, When, And, Then - keywords; 10, 2 - parameters, 'Div' - the name of a logical function in the future code, and the rest - just ordinary English words. Later, based on this specification with the existing tools the test code is being generated by the following rules: a code with 'Given' annotation should instantiate the object devoted to this feature, a code with 'And' (or possible, 'But') should pass the values from specification to the object, execute actions and a code with 'Then' should check to match the expected value to the result

```
public class MyStepdefs {
    @Given("^I have my software calculator\$")
    public void iHaveMySoftwareCalculator()
    {
    //code to create an instance of class Calculator
    }
    @When("^I have entered (\\d+) as first operand\$")
    public void iHaveEnteredAsFirstOperand(int arg)  {
    //code to pass the arg to a Calculator object
    }
    @And("^I have entered (\\d+) as second operand\$")
    public void iHaveEnteredAsSecondOperand(int arg)
    {
    //code to pass the arg to a Calculator object
    }
    @And("^I press 'Add'\$")
    public void iPressAdd()
    {
    //code to execute Add method
    }
    @Then("^The result should be (\\d+)\$")
    public void theResultShouldBe(int arg)
    {
    //code to check return value with the assert statement
    }
}
```

So the process starts from the specification, gets the urging 'step definition' code, and then the user is going to implement the actual system code and checking for correct assertions. The 'step definition code' is becoming a unit test for a given feature and started to execute in the order from top to bottom. Hence, here we got a 'runnable' specification, clear code, connection between the code and the specification, unit-testing with assertions. And of course, it improves the quality of developed software system.

To make correct assertions, it could be faced to a problem of correct requirements. Where to get them? In what form? In the work [2] we proposed a web-portal to software modeling with using requirements engineering approach. The portal could be a place to meet a customer, a business analyst and an engineer (Fig. 1). So, such portal may help to

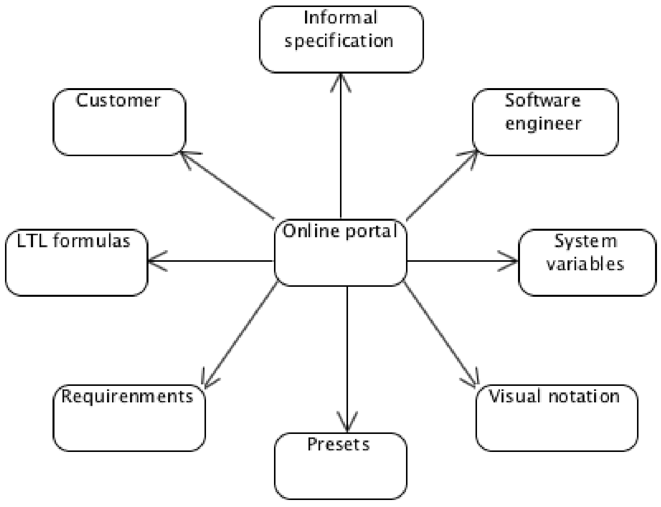


Fig. 1. Structure of the MDD portal for requirements engineering [2]

elaborate the requirements and think about program logic behaviour that could possible implemented as a runnable specification.

If we need to extract the system requirements from the existing specifications we could follow a lexical parsing approach. But because of lack of accepted format of specification/s-cope of work in the software engineering community, we could face semantic issues while resolving ambiguities. One possible way is trying to extract it by creating an ontology of possible classes of requirements parts and their relationships and then create algorithms to populate this ontology with classified data [3]. In the work [4] an ontology for various types of requirements based on templates on LTL and CTL propositional logics has been proposed. This ontology is shown in Fig. 2.

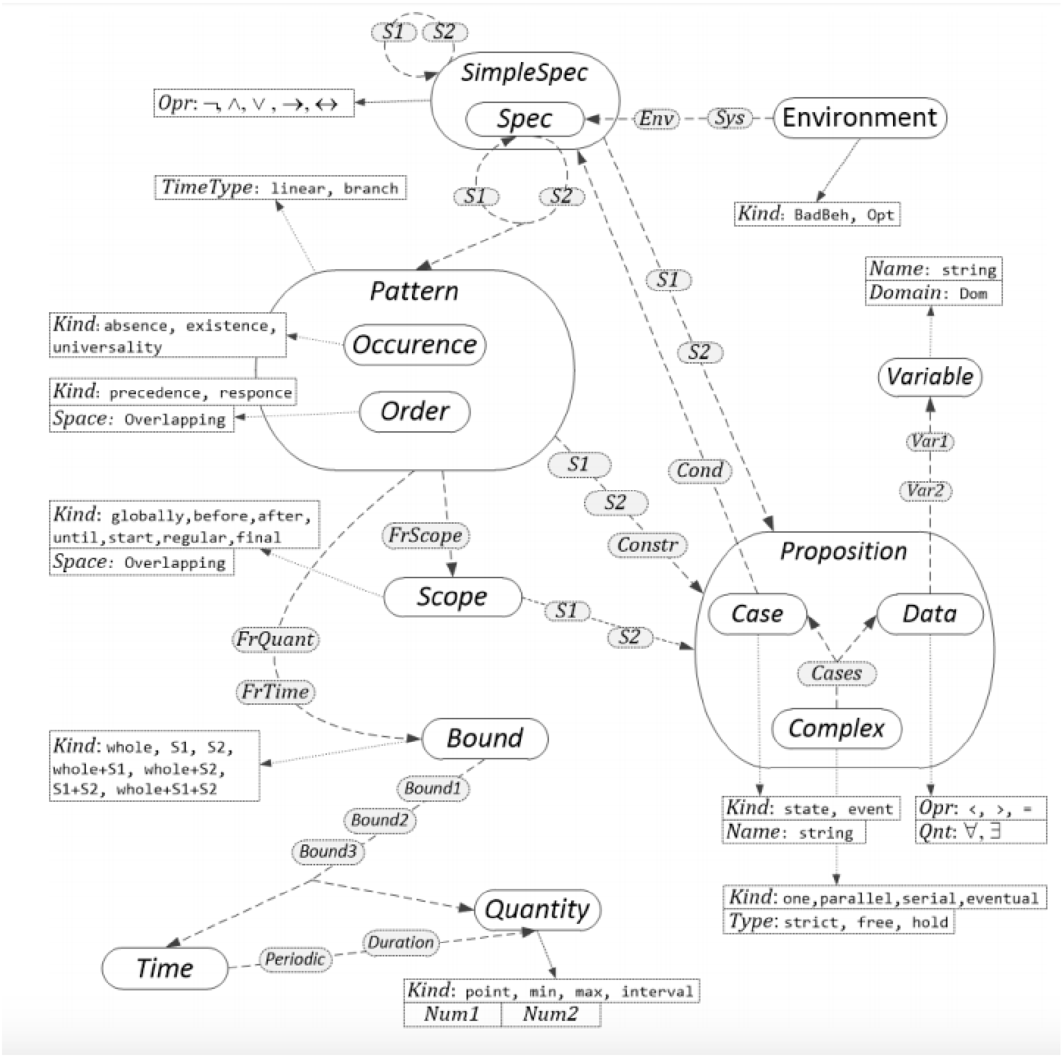


Fig. 2. The ontology for system requirements [4]

In the paper [5], a SCADA system for water treatment has been described. The developing process of the system was based on a real scope of work given by a customer. The fragment of it is as follows:

```
Step 2: Filling the distiller
The level sensor in the distiller is monitored for
(time of opening the valve) after opening valve N...
```

```
Yes / parameters OK:
1) Valve ... is closed;
2) Transition to step 3.
No / the parameters are not normal:
1) Message (Filling of filling of the distiller);
2) The automatic mode is switched off.
```

So we see here a specification in the form of transitions of a timed automaton based on control variables.

The goal of current research is to combine the last discussing approaches and propose a strong process to build a reliable software with it.

3 Proposed Approach: a Process and a BDD Extension

The proposed process is shown in Fig. 3.

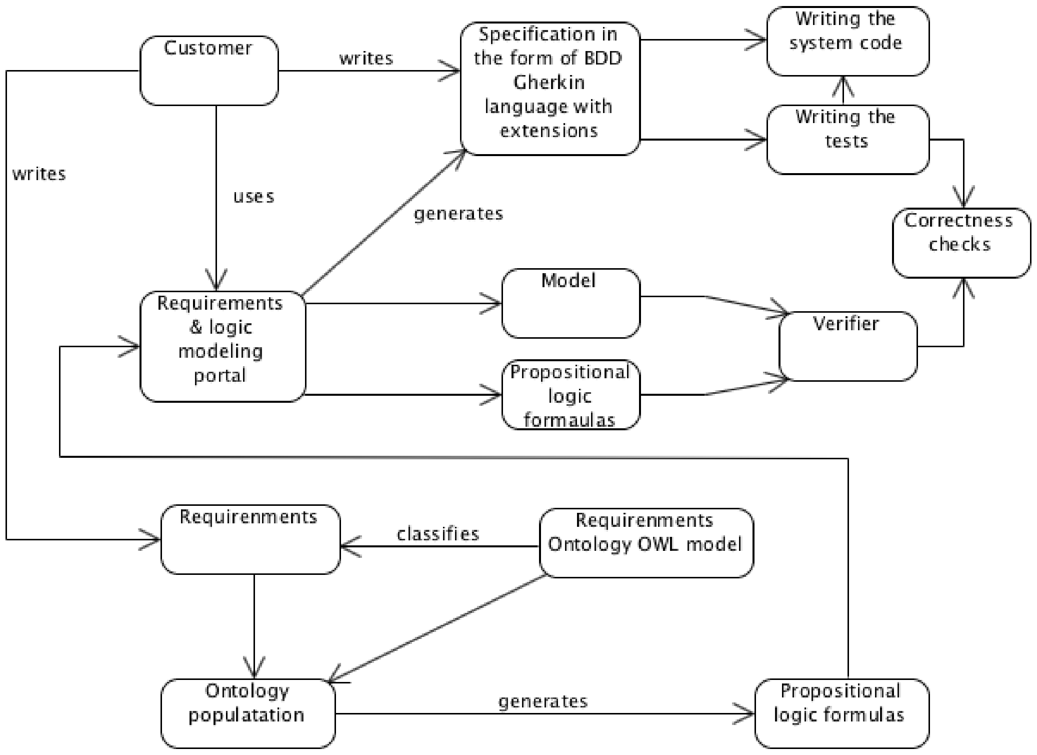

Customer
writes
Specification in the form of BDD Gherkin language with extensions
Writing the system code
Writing the tests
writes
uses
generates
Correctness checks
Model
Verifier
Requirements & logic modeling portal
Propositional logic formaulas
Requirenments
classifies
Requirenments Ontology OWL model
Ontology populatation
generates
Propositional logic formulas

Fig. 3. Possible use of the online portal for modeling, ontology for the requirements and a BDD process together

Thus, the customer can write a specification of the system in the text form of BDD language (with extensions), or he can do it implicitly by modeling the system behaviour in the diagram form with the specialized modeling portal. As a result, we will get a BDD specification and with using plugins in the IDEs we can start to write code of the system and the test code simultaneously. The customer (with help of requirements engineers) can specify the requirements for the system explicitly in or in the text form. The text form could be processed according to the developed ontology by classified the parts and make relations with them, so as a result, propositional formulas will be built. And if the system is modeled (by generating the model code from the diagrams) and the requirements are clear in the form of propositional logic (LTL/CTL), they can be verified with existing verifier (i.e. Spin).

What about the BDD specification, is the Gherkin language enough to specify controlling systems, like in the example? No. And we should make an extension to the BDD language and tools (based on an IDE plugin) to allow specify the behaviour in the form of an extended automaton:

```
Feature: <name of the feature, i.e. name of the mode>
Feature invariant: The ... should be ...
Step: <name of the state>
  Given ...
    When ...
    And ...
    And ...
  Then the system make transition to Step X.
```

4 Conclusion and Future Work

In this parer, the ideas of using the BDD process which is well known in software world, to build the trustworthy software systems are given. It combines the specification writing process, code writing, testing and verification together. Also, it is given the place of ontology for requirements in the process. Currently, the demo plugin for BDD language is being implemented on the bases of Intellij IDEA BDD plugin sources.

References

1. I. Dees, M. Wynne, and A. Hellesoy. Cucumber Recipes: Automate Anything with BDD Tools and Techniques. Pragmatic Bookshelf, 2013. – ISBN 978-1-93778-501-7
2. D. Lozhkina and S. Staroletov. An Online Tool for Requirements Engineering, Modeling and Verification of Distributed Software Based on MDD Approach. Preliminary Proceedings of the 11th Spring/Summer Young Researchersâ Colloquium on Software Engineering (SYR-CoSE 2017), June 5-7, 2017 â Innopolis, Republic of Tatarstan, Russian Federation. pp. 23-29. http://syrcose.ispras.ru/2017/SYRCoSE2017_Proceedings.pdf
3. N. O. Garanina and E. A. Sidorova. Ontology population as algebraic information system processing based on multi-agent natural language text analysis algorithms. Programming and Computer Software, 41(3):140â148, 2015.
4. N. Garanina, V Zyubin, T. Liakh. Ontological approach to organizing spec- ification patterns in the framework of support system for formal verification of distributed program systems. System Informatics, 9:111-132, 2017. In Russian. http://www.system-informatics.ru/files/article/garaninazubinliach_0.pdf
5. S. Staroletov. Design and implementation a software for water purification with using automata approach and specification based analysis. System Informatics, 10:33-34, 2017. http://www.system-informatics.ru/files/article/staroletov.pdf

Научное издание

9th Workshop PSSV

Материалы конференции

9-ый семинар «Семантика, спецификация и верификация программ: теория и приложения», посвященный памяти Б. А. Трахтенброта, М. И. Дехтяря и М. К. Валиева (Ярославль, Россия, 21-22 июня 2018)

Редакторы:
Шилов Николай Вячеславович, Захаров Владимир Анатольевич

Ярославль, ЯрГУ, 2018

На английском языке

Компьютерная верстка Н. В. Шилов, В. А. Захаров, Е. В. Кузьмин

Scientific edition

9th Workshop PSSV

Proceedings

9th Workshop "Program Semantics, Specification and Verification: Theory and Applications" dedicated to the memory of B. A. Trakhtenbrot, M. I. Dekhtyar, and M. K. Valiev (Yaroslavl, Russia, June 21-22, 2018)

Editors:
Nikolay Shilov, Vladimir Zakharov

Signed to print 14.06.2018. Format 70x100/16.
Print run 30 copies. Order 045/018.

Printed by Yaroslavl State University,
14, Sovetskaya str., 150003, Yaroslavl