LOGIC IN KNOWLEDGE REPRESENTATION AND EXPLORATION

Series: STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17(30)

Under the Auspices of the Polish Association
for Logic and Philosophy of Science

LOGIC IN KNOWLEDGE REPRESENTATION AND EXPLORATION

edited by
Dariusz Surowik

University of Białystok
Białystok 2009

Refereed by Marcin Koszowy, Kazimierz Trzęsicki and Slavian Radev

Series: STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17(30)
http://logika.uwb.edu.pl/studies/

Subseries issued by the Chair of Logic, Informatics and Philosophy of Science
Faculty of Mathematics and Informatics
University of Białystok

Cover design: Krzysztof Tur
Type-setting: Stanisław Żukowski

ISBN 978–83–7431–231–8
ISSN 0860–150X

WYDAWNICTWO UNIWERSYTETU W BIAŁYMSTOKU
15-097 Białystok, ul. Marii Skłodowskiej-Curie 14, tel. 0857457059
http://wydawnictwo.uwb.edu.pl, e-mail: ac-dw@uwb.edu.pl
Nakład 150 egz.

Druk i oprawa: PPHU TOTEM s.c., Inowrocław

CONTENTS

Dariusz Surowik
University of Białystok
College of Computer Science and Business Administration in Łomża

PREFACE: TEMPORAL ASPECT IN INFORMATION SYSTEMS

In the information society, there exists an irresistible need, one may even say a necessity, to acquire and comprehend knowledge. Equipped with a sufficient level of knowledge and defined rules of inference, we are able to undertake specific decisions. In order to attain better representation and exploration of the ever expanding knowledge resources and to undertake appropriate decisions, we create constantly improved means of representing knowledge and enhanced methods of modelling situations for the real world. The language of classical logic is an appropriate tool for the formal description of statistical incidents. Incidents taking place in time require a novel approach. The power of expression, enabling a formal description of the time component, is made possible by the language of temporal logic. When we ask: *What order should the complete information contained in a sentence have?* the answer is simple. The information should have such form as to obtain answers to the questions: *When? Why? How?* and *Where?* in the same order as the questions were asked. The ancient Romans already valued knowledge associated with the time of an event and put the question of the time of the event first. The temporal aspect was therefore not only important (among other aspects) but due to the priority of the question of time, we may say that beginning with ancient times, it was the most significant aspect.

The need for the analysis and investigation of temporal reasoning is noticeable in numerous scientific fields, among others, in computer sciences, geographical information systems, philosophy, psychology, and linguistics. In computer sciences, temporal reasoning is the key object of research in information systems, verification of programs, and artificial intelligence, as well as in numerous areas connected with the modelling processes.

In numerous implementations, the information containing the temporal component of the phenomenon is of significant importance. It takes place, for

ISBN 978–83–7431–231–8 ISSN 0860-150X

example, when we want to predict the future behaviour of the system, when we want to draft a plan basing it on our knowledge about the future, when we want to discover what is the basis of changes taking place in the system, and when we analyse data based on previous information and attempt to explain why events are as they are.

In recent years, much research related to time has emerged. This research, in principle, has focused on analysing the temporal context in topics associated with artificial intelligence and management of databases. There are, however, significant differences between a temporal approach to the representation of knowledge and the temporal approach to database management. The differences in the implementation of temporal information become clear when we analyse the articles contained in this volume.

Temporal processing of knowledge is connected with the processing of knowledge in general, whereas in the process of forming a system for database processing, efforts are focused mainly on issues connected with storing knowledge, searching for information, maintaining data, and constructing a communication interface with the system user which would allow efficient access to the knowledge contained in the database. Systems of knowledge representation consider one time perspective, whereas database management systems may contain numerous time perspectives and require both linear time models as well as accurate dates. Information in knowledge processing systems is represented by sentences. In database systems, information represented in "classical" form is enriched by the temporal component.

Temporal logic, as an apparatus of formal description, should be employed when we are performing the description of knowledge undergoing a change over time. Time and change are therefore strictly connected with each other. The mutual interdependence of change and time are the basis of two diverse approaches to temporal reasoning:

– An approach based on the concept of change, and
– An approach based on the concept of time.

The approach based on the concept of change has a number of limitations. The limitations of models based on the concept of change are, for example, immediate actions (actions should not be extended in time) or immediate results (results extended in time are not expressible in this approach). Furthermore, implementing this approach, it is not possible to denote continuous processes, simultaneous events or events overlapping each other in time.

In the approach based on the concept of time, time is considered as a separate entity or as a one-dimensional space. Time may have a point structure (where time points are the basic concept) or interval structure (where a interval is the basic concept). The majority of the discussion is based on linear and discrete time.

Point time differs from segment time not only by the type of assumed primary concepts. Differences between these approaches take place on many levels. Consequently, conceptual differences occur. In the interval approach, the completion is assumed only above segments, whereas in the point approach, completion is accepted both in points as well in segments. Structural differences also occur: i.e. in the interval approach, ordered linear time is considered; whereas in the point approach, branching time may be considered. In order to avoid technical difficulties, in the interval approach discrete time is considered. In the point approach it is possible to consider compact time, unlimited time, and real timeline. There are also, among the above mentioned time concepts, differences in logical approaches. In the interval models, logic based on time segments and also first order predicate calculus is implemented. In point models, logic based on modal logic with dissimilar time moments which perform the role of possible worlds is employed.

Classical methods of modelling databases concentrate on the problem of processing static information using database systems. Methods of constructing databases that take into consideration the temporal aspect are, however, in their essence unifying methods, allowing the processing of both static information as well as dynamic information which changes over time.

Models which are implemented in the temporal representation of knowledge are usually based on certain temporal logic. Prior to the commencement of the construction of temporal logic, a number of significant matters essential for the construction of this logic should be considered and a number of important questions answered. Issues which should be taken into account are as follows:

Conceptual issues, i.e.,

- whether the considered logic will be based on point time or on segment time?
- whether theorems will be recognized in reference only to time points or in reference to segments?
- whether zero measure sections which would be identical with points are permitted?
- whether from the truthfulness of a sentence in reference to a segment may be inferred the truthfulness of this sentence in reference to another segment?

Structural issues should also be considered, i.e. issues connected with mathematical objects used in representing time structures. Is time ordered, is it discreet or compact, limited or limitless, etc.? Finally, the problem of **which** logical formalism will be implemented should be considered. Will it be first-order logic or one of the versions of temporal logic?

An important quality of the language of temporal logic is the possibility of implementing it in the specification of a wide spectrum of information systems. Computer science adopts temporal logic to its own applications. From the point of view of the informational applications of temporal logic, the logic of discrete time is of significant importance. The language of temporal logic employed in computer science omits operators of past time or assigns them a supplementary character. Temporal logic implemented in computer science is, on the one hand, poorer due to the lack of philosophical inspiration, but on the other, it is richer due to the tasks it is to perform in computer science and its applications.

It seems that time is an immanent component of our daily life. Utilizing time, making use of time, or referring to time, usually do not cause problems. It is a different matter when we refer to time in information systems. Implementing temporal data into information systems causes many difficulties. These difficulties are the result of several factors: the lack of precision of certain expressions of the natural language referring to time (for example: "now" may refer to this moment in time, to today, to the current year, the current decade, etc.) or different interpretations of time elements (for example, historians perceive a year as not a large time interval, whereas a second is perceived by engineers designing microprocessors as a very long time.) There are various actions which may differ; for example, the time of the occurrence of the effects of actions, for there exist both actions with immediate effects as well as those whose occurrence will be extended in time. Certain actions may occur simultaneously, others may occur sequentially, still others may start and end in different time moments, but a particular temporal interval may be common to both actions.

The information system should be inspected prior to its use to determine if it is performing the tasks for which it was designed, and if these tasks are being carried out properly. The information system should also be tested. Testing however will not give sufficient certainty as to the infallibility of the system. In the case of reaction systems, methods based on temporal logic turn out to be most appropriate for conducting testing procedures. The are two types of such methods: proof theory and model theory. The basic drawback of model theory methods is however the problem of finding proof. Proof theory, on the other hand, does not give a negative answer: the proof is found when the system is correct, but when it is not correct, we do not find the proof. Lack of proof however, should not be treated as a statement of the faultiness of a system. For the lack of proof may be the result of the inability to find it. The model theory method does not possess the above mentioned flaws, for it allows the finding of errors in a system even when the system is erroneous.

In his article, Kazimierz Trzęsicki considers methods for verifying software as well as implemented information systems supporting 'model theory' verification of software.

The article of F. Samaranche deals with the extension of Knuth's 16 Boolean binary logic operators for the need of fuzzy logic. The author provides Venn diagrams for the introduced operators. The introduced operators are improved versions of the Boolean operators described in 2003.

A. Schumann considers an abstract machine of Belousov-Zhabotinsky and proposes a proof for theoretical interpretation of process calculus within the framework of considered formalism.

A. Krasuski, T. Maciak, and K. Kreński describe a national project for a decision support system for the fire service based on a distributed database.

S. Zadrozny presents a construction of a logical system which, in the intent of the author, may serve as a means for formal description of a mental process. The author supplies and analyses a collection of specific sentence conjunctions and considers the advantages of supplied formalism in reference to modelling of human intelligence by utilising information techniques.

M. Giero describes the application of bi–temporal databases for the representation and exploration of medical data. The article describes the semantics of the system, gives preferable qualities of semantic time, and presents examples of queries conducted utilizing temporal operators.

In another article, M. Giero and R. Milewski describe the application of the formalism of temporal logic and bi-temporal databases for storing and retrieving information containing medical data on the treatment of infertility.

In the last article, R. Milewski, J. Jamiołkowski, A. Milewska, J. Domitrz and S. Wołczyński describe a unique case of representation and exploration of knowledge on the basis of an implemented application assisting electronic registration of patients treated for infertility.

BIBLIOGRAPHY

1. Allen, J. F. "Maintaining Knowledge about Temporal Intervals." Communication of the ACM. 26 (11), 1983.

2. Bolour, A., T. L. Anderson, L. J. Dekeyser and H. K. T. Wong. "The Role of Time in Information Processing: a survey." SIGMOD. Record (12), 1982.
3. McKenzie, E. and R. Snodgrass. "Extending the Relational Algebra to Support Transaction Time." in proceedings of SIGMOD Conference.
4. Pratt, V. R. "Semantical Considerations on Floyd-Hoare Logic." in proceedings of 17th FOCS. IEEE, New York.
5. Renato Barrera and Khaled K. AI-Taha, Models in Temporal Knowledge Representation and Temporal DBMS, Report, 1990.
6. Sernadas, A. "Temporal Aspects of Logical Procedure Definition." Information Systems. 5: 1980.
7. Worboys, M. F. "Reasoning about GIS Using Temporal and Dynamic Logics." in proceedings of The Temporal GIS Workshp, University of Maine, Orono, Maine, 1990.

Dariusz Surowik
Chair of Logic, Informatics and Philisophy of Science
University of Białystok
ul. Sosnowa 64,
15–887 Białystok, Poland
surowik@uwb.edu.pl

Kazimierz Trzęsicki
University of Białystok

TEMPORAL LOGIC MODEL CHECKERS AS APPLIED IN COMPUTER SCIENCE*

Abstract: Various logics are applied to specification and verification of both hardware and software systems. Since systems are operating in time, temporal logic is a proper tool. The problem with finding the proof is the most important disadvantage of the proof-theoretical method. The proof-theoretical method presupposes the axiomatization of logic. Proprieties of a system can also be checked using a model of the system. A model is constructed with the specification language and checked using automatic model checkers. The model checking application presupposes the decidability of the task. The explosion of the cases that have to be explored is the main disadvantage of this method.
Temporal logic model checking is an algorithmic method that can be used to check whether a given model (representing a system) satisfies certain properties (expressed as temporal logic formulas).

Key Words: Temporal Logic, Model Checking.

1. Introduction

Connections between logic and **C**omputer **S**cience, CS, are wide-spread and varied. Notions and methods from logic can successfully be applied within CS. Logic plays the same role in CS as the calculus plays in physics. Logic is "the calculus of computer science" (Manna & Waldinger 1985, Cengarle & Haeberer 2000, Connelly, Gousie, Hadimioglu, Ivanov & Hoffman 2004).

Logics of programs play more practical role in CS than that of the logic in "pure" mathematics. On the one hand, logic permeates more and more the main areas of CS. On the other hand, we may notice that (Kaufmann & Moore 2004, p. 181):

> Until the invention of the digital computer, there were few applications of formal mathematical logic outside the study of logic itself. In particular, while

* The work was supported by the KBN grant 3 T11F 01130.

ISBN 978–83–7431–231–8 ISSN 0860-150X

> many logicians investigated alternative proof systems, studied the power of various logics, and formalized the foundations of mathematics, few people used formal logic and formal proofs to analyze the properties of other systems. The lack of applications can be attributed to two considerations: (i) the very formality of formal logic detracts from its clarity as a tool of communication and understanding, and (ii) the "natural" applications of mathematical logic in the pre-digital world were in pure mathematics and there was little interest in the added value of formalization. Both of these considerations changed with the invention of the digital computer. The tedious and precise manipulation of formulas in a formal syntax can be carried out by software operating under the guidance of a user who is generally concerned more with the strategic direction of the proof.

The logical methods are applicable for the design, specification,[1] verification[2] and optimization of programs, program systems and circuits. Logic has a significant role in computer programming. While the connections between modal logic[3] and CS may be viewed as nothing more than specific instances, there is something special to them. The dynamic character of phenomena of CS has its counterpart in modal logics. In fact, **T**emporal **L**ogic, TL, is a multi-modal logic with a time dependent interpretation of modalities.

---

[1] The specification prescribes what the systems has to do and what not.

[2] Some authors distinguish between ***V**alidation* and ***V**erification* and refer to the overall checking process as V&V. Validation is answering the question: *Are we trying to make the right thing? (are we building the right thing?).* Verification (functional correctness) answers the question: *Have we made what we were trying to make? (are we building the thing right?)* Verification methods aim at establishing that an implementation satisfies a specification (Baier & Katoen 2008, p. 13). The different characterizations of verification and validation originate from Boehm (1981). In general methodology of sciences the term "verification" denotes establishing correctness. The term "falsification" (or "refutation") is used in meaning: to detect an error. In CS "verification" covers both meanings and refers to the two-sided process of determining whether the system is correct or erroneous.

For Dijkstra (1989) the verification problem is distinct from the pleasantness problem which concerns having a specification capturing a system that is truly needed and wanted. Emerson observes that (2008, p. 28):

> The pleasantness problem is inherently pre-formal. Nonetheless, it has been found that carefully writing a formal specification (which may be the conjunction of many sub-specifications) is an excellent way to illuminate the murk associated with the pleasantness problem.

[3] The traditional modal logic deals with three 'modes' or 'moods' or 'modalities' of the copula 'to be', namely, *possibility*, *impossibility*, and *necessity*. Related terms, such as *eventually*, *formerly*, *can*, *could*, *might*, *may*, *must*, are treated in a similar way, hence by extension, logics that deals with these terms are also called modal logics.

The basic modal operator $\Box$ (necessarily) is not rigidly defined. Different logics are obtained from its different definitions of it. Here we are interested in temporal logic that is the modal logic of temporal modalities such as: *always, eventually.*

In 1974, the British computer scientist Rod M. Burstall first remarked on the possibility of application of modal logic to solve problems of CS[4]. The Dynamic Logic of Programs has been invented by Vaughan R. Pratt (1980):

> In the spring of 1974 I was teaching a class on the semantics and axiomatics of programming languages. At the suggestion of one of the students, R. Moore, I considered applying modal logic to a formal treatment of a construct due to C. A. R. Hoare, "$p\{a\}q$", which expresses the notion that if p holds before executing program a, then q holds afterwards. Although I was skeptical at first, a weekend with Hughes and Cresswell[5] convinced me that a most harmonious union between modal logic and programs was possible. The union promised to be of interest to computer scientists because of the power and mathematical elegance of the treatment. It also seemed likely to interest modal logicians because it made a well-motivated and potentially very fruitful connection between modal logic and Tarski's calculus of binary relations.

This approach was a substantial improvement over the existing approach based on the pre-condition/post-condition mechanism provided by Hoare's logic.[6] Kripke models, the standard semantic structure on which modal languages are interpreted, are nothing but graphs. Graphs are ubiquitous in CS.

The connection between the possible worlds of the logician and the internal states of a computer is easily described. In possible world semantics, ϕ is possible in some world w if and only if ϕ is true in some world w' accessible to w. Depending on the properties of the accessibility relation (reflexive, symmetric, and so on), there will be different theorems about possibility and necessity. The accessibility relation of modal logic semantics can thus be understood as the relation between states of a computer under the control of a program such that, beginning in one state, the machine will (in a finite time) be in one of the accessible states. In some programs, for instance, one cannot return from one state to an earlier state; hence state accessibility here is not symmetric.

---

[4] Charles Leonard Hamblin a pioneer computer scientist, a prominent philosopher and logician is considered as one the founders of the modern temporal logic (and modern logic). His contributions to applied and theoretical computing are multifold. Hamblin was first to propose an axiomatic account of time based on intervals (Hamblin 1969). This idea has been influential in AI, both as a basis for reasoning about time, and, when extended to multiple dimensions, as a basis for reasoning about space (Allen 1984). For more see (Barton 1970, Allen 1985, Williams 1985).

[5] The book Pratt is talking about is (Hughes & Cresswell 1968).

[6] Hoare's logic views a program as a transformation from an initial state to a final state. Such view cannot be applied successfully, where the computation does not bring to a final state. Thus it is not eligible to tackle problems of reactive or non-terminating systems, such as operating systems, protocols, concurrent programs, and hardware systems.

The question of using TL to software engineering was undertaken by Kröger (1977, 1987, 1991, 2008). The development of TL as applied to CS is due to Amir Pnueli[7]. He was inspired by "Temporal Logic", a book written by Rescher and Urquhart (1971).[8] "The Temporal Logic of Programs" (1977), a paper by Pnueli, is the classical source of TL for specification and verification of programs. This work is commonly seen as a breakthrough in using of TL in CS. Amir Pnueli argues that temporal logic can be used as a formalism to reason about the behavior of computer programs and, in particular, of non-terminating concurrent systems.[9] According to Bochmann (Clarke 2008, p. 5):

> temporal logic has brought a more elegant way to talk about liveness and eventuality; in the protocol verification community we were talking about reachable deadlock states (easy to characterize) or undesirable loops (difficult to characterize).

In general, properties are mostly describing correctness or safety of the system's operation. For Clarke (2008, p. 1) works of Pnueli (1977), Owicki and Lamport (1982):

> demonstrated convincingly that Temporal Logic was ideal for expressing concepts like mutual exclusion, absence of deadlock, and absence of starvation.

There is a difference between logician and computer scientists approach to systems of logics (Bradfield & Stirling 2001, p. 315):

> Decidability and axiomatization are standard questions for logicians; but for practitioner, the important question is model-checking.

In the opinion of Dijkstra:[10]

> The situation of programmer is similar to the situation of mathematician, who develops a theory and proves results. [...] One can never guarantee that

---

[7] Pnueli received the Turing Award in 1996:

> for seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.

[8] See (Hasle & Øhrstrøm 2004, p. 222).

[9] A system is said to be concurrent when its behavior is the result of the interaction and evolution of multiple computing agents. Interacting processes do not known about the internal state of the others. The initial interest in concurrent systems was motivated by the speed improvements brought forth by multi-processor computers.

[10] See Dijkstra E. W., *Programming Considered as a Human Activity*, `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD117.html`.

> a proof is correct, the best one can say, is: "I have not discovered any mistakes". [...] So extremely plausible, that the analogy may serve as a great source of inspiration. [...]
>
> Even under the assumption of flawlessly working machines we should ask ourselves the questions: "When an automatic computer produces results, why do we trust them, if we do so?" and after that; "What measures can we take to increase our confidence that the results produced are indeed the results intended?"

In another work (1968, p. 6), Dijkstra says:

> Program testing can be used to show the presence of bugs, but never to show their absence.

Although the theory spoke of verification, if we apply it, we soon realize that its real value lay in falsification.

The verification problem can be said to have been born at the same time as computer science itself. Formulated in terms of Turing Machines, the verification problem was already considered by Turing in the work on the Halting problem (1936–37). He demonstrated that there is no general method of proving the correctness of any program. Therefore, that paper was, in a way, the death of the problem, since it was shown that the task is theoretically unmechanizable. So without knowing anything about the particulars of a given product, one thing is generally safe to assume: it has bugs. Nevertheless as mathematicians did not stop proving theorems as a result of Gödel's theorem, computer scientists did not leave recognizing verification to be a fundamental problem of their subject. It is a millions of programmers' daydream: a compiler that automatically detects all the bugs in the code (Hoare 2003).

Our reliance on the functioning of **I**nformation and **C**ommunication **T**echnology, ICT, systems is growing rapidly. Our society is increasingly dependent on ICT in almost every aspect of daily life. Computer controls everything possible: from entertainment to work, from children's toys to nuclear weapons and from car to space-rocket systems. Often we are not even aware that computers and software are involved. ICT-based solutions are becoming ubiquitous and are to be found in several safety-critical systems. In such cases the reliability cannot be compromised. Defects can be fatal and extremely costly. Errors in ICT systems may cause not only material losses, e.g., in e-banking, but also may be dangerous for life, e.g., in health care, transportation, especially air and space flights.[11] Correctness of design

[11] Due to a design error in the control software of the radiation therapy machine

is a very important factor of systems for preventing economical and human losses caused by minor errors. The need for reliable hardware and software systems is critical. The reduction of errors in *ICT* systems is one of the most important challenges of *CS* (Kröger & Merz 2008, p. V). It has long been known that (Emerson 2008, p. 27):

> computer software programs, computer hardware designs, and computer systems in general exhibit errors. Working programmers may devote more than half of their time on testing and debugging in order to increase reliability. A great deal of research effort has been and is devoted to developing improved testing methods. Testing successfully identifies many significant errors. Yet, serious errors still afflict many computer systems including systems that are safety critical, mission critical, or economically vital. The US National Institute of Standards and Technology has estimated that programming errors cost the US economy $60B annually.[12]

The reliability of *ICT* systems is no longer a luxury but an urgent necessity.

Despite efforts of *ICT* engineers errors are inevitable. Though they are generally unpredictable (Holzmann 2002)

> . . . within fixed domain we can often predict fairly accurately just how many mistakes will be made. For programmers in industrial software development, the residual software defect ratio (the number of latent faults that remain in the code at the end of the development process) is normally somewhere between 0.5 and 5 defects per one thousand lines of non-comment source code (Holzmann 2001). . . . Even at a low residual defect density of 0.1 defect per one

---

Therac-25 in the period 1985–1987 at least 6 cases of overdosis (100-times dosis) took place and three cancer patients died. A famous example: the Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value. Costs: more than 500 million US$. In the report (Lions, J. L. et al. 1996) we read:

> The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane-5 software design. The Board is in favor of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.

In 2008 it was announced that the Royal Navy was ahead of schedule for switching their nuclear submarines to a customized Microsoft `Windows XP` solution dubbed *Submarine Command System Next Generation*. In this case any error may have an unimaginable aftermath. Let us add that US Navy computer solutions are based on Linux.

[12] See: National Institute of Standards and Technology, US Department of Commerce, "Software Errors Cost U.S. Economy $59.5 Billion Annually", NIST News Release, June 28, 2002.

> thousand lines of code, a ten million line source package will have an expected 103 latent defects.

Therefore, the main challenge for CS is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity. Summarizing, we may repeat after Henk Barendregt:

> It is fair to state, that in this digital era correct systems for information processing are more valuable than gold.

ICT systems are more and more complicated. Their complexity grows rapidly. Large scale software is generally written by different groups of programmers, and integration testing becomes a major problem. Verification of digital hardware designs has become one of the most expensive and time-consuming components of the current product development cycle. The verification which requires exhaustive search is hard to deal with bigger state spaces. To establish confidence in ICT systems, traditional testing and simulation are used. However, testing on a representative set of scenarios is often inadequate. Simulation and testing, unlike verification, can only indicate errors and can never show the absence of errors. These methods usually involve providing certain inputs and observing corresponding outputs. Testing and simulation can be a cost efficient way of minimizing errors. However, covering all possible interactions and potential errors is rarely feasible. Moreover, usually empirical testing and simulation is expensive, not ultimately decisive and sometimes excluded for economical or ethical reasons. Formal methods are the most notable efforts to guarantee the correctness of system design and behaviors. Thus, the formal specification and computer aided validation and verification are more and more indispensable.[13] There is an increasing interest in formal methods. Formal methods have gained popularity in industry since the advent of the famous Intel Pentium $FDIV$ bug in 1994, which caused *Intel* to recall faulty chips and take a loss of $475 million (Coe, Mathisen, Moler & Pratt 1995). Digital computers are intended to be abstract discrete state machines and such machines and their software are naturally formalized in mathematical logic.

---

[13] Computer Aided Verification, CAV, a subdiscipline of CS, is concerned with ensuring that software and hardware systems operate correctly and reliably. The International Conference on Computer Aided Verification is dedicated to the theory and practice of computer aided formal analysis methods for hardware and software systems, `http://www.cav-conference.org`.

> Given the formal descriptions of such systems, it is then natural to reason about the systems by formal means. And with the aid of software to take care of the myriad details, the approach can be made practical. Indeed, given the cost of bugs and the complexity of modern hardware and software, these applications cry out for mechanical analysis by formal mathematical means. (Kaufmann & Moore 2004, p. 181–182)

The reliability of *ITC* systems is a key issue in the design processes. Errors should already be detected at the design stage. It is very important to specify the correctness property of system design and behavior, and an appropriate property must be specified to represent a correct requirement. More time and effort is spent on validation than on construction. It is estimated that 70% of design-time is spent to minimize the risk of errors (Schneider 2003), see (Miller, Donaldson & Calder 2006). 30% to 50% of software project costs is devoted to testing. The problem of verification is the subject of interdisciplinary investigations. "Verification", a special interest group of the Gigascale Systems Research Center declares, `http://www.gigascale.org/sig/sig_verification/`:

> We welcome interactions with groups in our sponsor companies involved in all aspects of design verification – both pre-silicon verification using simulation, formal verification and emulation, as well as post-silicon debug.

Such concerns have motivated the industry to consider alternative techniques for verification, especially those based on formal methods.

Formal methods, model checkers as well as theorem provers and proof assistants, are proposed as efficient, safe and less expensive tools. Effective mechanized reasoning could improve the quality of computer aided design in many fields of human activity, giving enormous economic benefits and making automated devices safer. According to Emerson (2008, pp. 27–28):

> Given the incomplete coverage of testing, alternative approaches have been sought. The most promising approach depends on the fact that programs and more generally computer systems may be viewed as mathematical objects with behavior that is in principle well-determined. This makes it possible to specify using mathematical logic what constitutes the intended (correct) behavior. Then one can try to give a formal proof or otherwise establish that the program meets its specification. This line of study has been active for about four decades now. It is often referred to as *formal methods*.

2. Formal methods of *ICT* systems verification

2.1. Formal methods

Formal methods are the applied logic and mathematics for modeling and analyzing of *ICT* systems. Formal methods include: formal specification, specification analysis and proof, transformational development, and program verification. In the case of formal methods, implementation is understood as an abstract model of the system to be verified and specification refers to some property of the system expressed in a suitable formula of specification language. The principal benefits of formal methods are in reducing the number of faults in systems. Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area. The use of formal methods is most likely to be cost-effective because high system failure costs must be avoided. Formal methods are highly recommended by *IEC* (**I**nternational **E**lectrotechnical **C**ommission), *ESA* (**E**uropean **S**pace **A**gency) for safety-critical software. As it is remarked by Emerson (2008, p. 37), formal verification is becoming a staple of *CS* and electrical engineering education. At the same time there is ever growing research interest in model checking. Nevertheless, these methods have not become the only stream software development techniques. Other software engineering techniques have been successful at increasing system quality. Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market. Moreover, the scope of formal methods is limited. They are not well-suited to specifying and analyzing user interfaces and user interaction. Formal methods are still hard to scale up to large systems. In spite of this, the need for formal methods has not been reduced. The toil of several decades, and the ingenuity of researchers working in the area of formal methods has led to the discovery of powerful and efficient techniques for *ICT* systems analysis and verification. These, combined with the raw computing horsepower available today, have enabled researchers to devise very efficient formal tools.

The formal methods to be appropriate need to be properly adapted. For building automated tools for verifying systems, one aims at simpler formalisms to specify the behavior of the system to be analyzed. Every program, at its lowest level, can be described as a set of states and a binary transition relation. The set of states corresponds to the states the program can be in, and the two states are in the transition relation if the first can transition to the second in one step during the execution of the program. Classical methods mirror the static nature of mathematical notions. Dynamic behavior of programs requires another approach. Temporal logic and its language are of particular interest in the case of reactive[14], in particular concurrent systems.

> On the one hand, such specification languages should be simple and easy to understand, such that non-experts also are able to use them. On the other hand, they should be expressive enough to formalize the stepwise behavior of the processes and their interactions. Furthermore, they have to be equipped with a formal semantics which renders the intuitive meaning of the language commands in an unambiguous manner. (Baier & Katoen 2008, p. 63)

The language of TL is one that fulfills three important criteria. It:

- has the ability to express all sorts of specification (expressiveness) independently of the programming language to be used for implementation purposes;
- has reasonable complexity to evaluate the specified rules (complexity);
- due to its resemblance to natural language is easy to learn (pragmatics).

In many areas of CS understanding, TL has become as useful and profitable as understanding the algorithms themselves. The knowledge of TL is indispensable in practice, though, as it is remarked by Schnoebelen (2002):

> In today's curricula, thousands of programmers first learn about temporal logic in a course on model checking!

A raport on the use of formal methods prepared by FAA (**F**ederal **A**viation **A**uthority), and $NASA$ (**N**orth-**A**tlantic **S**pace **A**gency) concludes:

> Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers.

TL languages can be used for specification of a wide spectrum of ICT systems. Methods of TL can be applied to verification (Manna & Pnueli 1992, 1995a). In the case of reactive systems, TL is more useful than

[14] Systems can be divided into two categories: transformational programs (data intensive) and reactive systems (control intensive). The systems of the second type maintain an ongoing interaction with their environment (external and/or internal stimuli) and which ideally never terminate. Their specifications are typically expressed as constraints on their behavior over time. Examples of reactive systems include microprocessors, operating systems, banking networks, communication protocols, on-board avionics systems, automotive electronics, and many modern medical devices.

Floyd-Hoare logic that is better in the case of "input-output" programs. TL languages (Kröger & Merz 2008, p. 181):

> provide general linguistic and deductive frameworks for *state systems* in the same manner as classical logics do for mathematical systems.

Logic may be axiomatically or semantically presented. The former presentation is concerned with the axioms and rules defining the notion of a *proof*. The latter presentation is oriented to building a model. The duality in the presentation has led to proof-theoretical and model-theoretical approaches (Clarke, Wing, Alur, Cleaveland, Dill, Emerson, Garland, German, Guttag, Hall, Henzinger, Holzmann, Jones, Kurshan, Leveson, McMillan, Moore, Peled, Pnueli, Rushby, Shankar, Sifakis, Sistla, Steffen, Wolper, Woodcock & Zave 1996). Formal methods can be divided into two basic categories of

1. theorem proving and proof checking,

and

2. model-checking.

2.2. Proof-theoretical approach

Already in the works of Turing, the mathematical methods were applied to check correctness of programs (Randell 1973). By the end of the sixties of the last century, Floyd (1967), Hoare (1969) and Naur (1966) proposed axiomatic proving sequential programs with respect to their specification. E. G. Dijkstra extended Hoare's ideas (Dijkstra 1975). Proof-theoretical or deductive method based on TL was proposed by Pnueli and Manna (1992, 1995a). The benefits were achieved in the hardware sector (Kaufmann & Moore 2004, Brock & Hunt 1997). Correctness of ICT system can be demonstrated through logical proving about system constraints or requirement for safe system behavior. The verification problem is represented as a theorem in a formal deductive theory. Such a theory consists of formal language in which formulas are written and a set of axioms and a set of inference rules. Propositions specifying the ICT system are joined as premisses to the thesis of the deduction system. Proofs can be "described" a variety of ways, e.g., by giving the inference steps, by specifying tactics or strategies to try, by stating the "landmark" subgoals or lemmas to establish, etc. Often, combinations of these styles are used within a single large proof project. Verification is positive if the proposition expressing the desired property is proved. Correctness of formal derivations could be "mechanically" checked, but finding a proof needs some experience and insight. It requires highly skilled mathematical logician's intervention.

At the time of its introduction in the early 1980's, a "manual" proof-theoretic approach was a prevailing paradigm for verification. Nowadays, proofs are supported by semi-automatic means,[15] *provers* and *proof checkers*. Interactive provers are used to partially automate the process of proving, nevertheless (Kaufmann & Moore 2004, pp. 182–183):

> all proofs of commercially interesting theorems completed with mechanical theorem proving systems have one thing in common: they require a great deal of user expertise and effort.

For example (Kaufmann & Moore 2004, p. 182):

> The proof, constructed under the direction of this paper's authors and Tom Lynch, a member of the design team for the floating point unit, was completed 9 weeks after the effort commenced. About 1200 definitions and theorems were written by the authors and accepted, after appropriate proofs were completed by the *ACL2*[16] theorem prover.

Among the mechanical theorem proving systems used to prove commercially interesting theorems about hardware designs are Acl2 ("Acl2" stands for "A Computational Logic for Applicative Common Lisp")[17], Coq[18], Hol[19], Hol Light[20], Isabelle[21], and Pvs[22]. The proof assistant approach is a subject of research projects, e.g. *BRICKS*, `http://www.bsik-bricks.nl/researchprojects_afm4.shtml`.

The proof-theoretic framework is one-sided, if the program is really incorrect, the proof systems do not cater for proving incorrectness. It is possible only to prove that a proposition is a thesis. If we do not have a proof, we are entitled only to say that we could not find a proof, and nothing more. Theorem provers do not provide concrete counterexamples. However, theorem proving can deal with an infinite state space, i.e., system with infinitely many configurations. It is not the only advantage of the method.

---

[15] Until the artificial intelligence problem is solved, human interaction will be important in theorem proving. For this reason it is more realistic to think about theorem provers as proof assistants and refer to them as proof checkers (Kaufmann & Moore 2004).

[16] See (M. Kaufmann, Manolios & Moore 2000, Boyer & Moore 1979).

[17] See `http://www.cs.utexas.edu/~moore/acl2/`, (Kaufmann & Moore 2004).

[18] See `http://coq.inria.fr/`.

[19] See `http://www.cl.cam.ac.uk/research/hvg/HOL/`, (Gordon & Melham 1993).

[20] See `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[21] See `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[22] See `http://pvs.csl.sri.com/`.

This method is also indispensable in some intractable cases of finite state systems. Though today's model checkers are able to handle very large state spaces, eg. 10^{120} [(Kaufmann & Moore 2004, p. 183), (Clarke, Grumberg, Jha, Lu & Veith 2001)] but it does not mean that these states are explored explicitly. The above discussed theorem about $FDIV$ (see p. 19) could be checked by running the microcode on about 10^{30} examples. Since in this case there are no reduction techniques, if it is assumed that one example could be checked in one femtosecond (10^{-15} seconds – the cycle time of a petahertz processor), the checking of the theorem will take more than 10^{7} years (Kaufmann & Moore 2004, p. 183).

For Emerson (2008, p. 28):

> The need to encompass concurrent programs, and the desire to avoid the difficulties with manual deductive proofs, motivated the development of model checking. In my experience, constructing proofs was sufficiently difficult that it did seem there ought to be an easier alternative.

Though the proof-theoretic approach is more elegant, the best results have been obtained by model-theoretic or hybrid[23] approaches. Already C. A. R. Hoare, argued that formal proofs of correctness represent only one of many possible options on the road to reliable ICT systems (Hoare 1996).

2.3. Model-theoretical approach

The basic papers on the use of Temporal Logic Model Checking were written in the early 1980's. Both, the idea of automatic verification of concurrent programs based on model-theoretic approach and the term "model checking" were introduced by Clarke and Emerson in (1982a),[24] and independently the use of model checking as a device of CS was conceived by Quille and Sifakis (1982).[25] The idea was developed in works by Clarke, Emerson, Sistla and others (Clarke, Emerson & Sistla 1983, Clarke, Emerson

[23] Hybrid methods combine both the proof- and model-theoretical approaches.

[24] See (Emerson 2008, p. 9). Allen Emerson wrote his PhD dissertation under Ed Clarke's guidance.

[25] E. M. Clarke and E. A. Emerson interpreted concurrent system as finite Kripke structure/transition system and properties were expressed in CTL (**C**omputational **T**ree **L**ogic) language. J.-P. Queille and J. Sifakis based on Petri nets and properties were expressed in language of branching time logic. Edmund M. Clarke jr. (CMU, USA), Allen E. Emerson (Texas at Austin, USA), Joseph Sifakis (IMAG Grenoble, France) were granted with ACM Turing Award 2007. Jury justification:

> For their roles in developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

& Sistla 1986, Clarke, Grumberg & Peled 1999, Bidoit, Finkel, Laroussinie, Petit, Petrucci & Schnoebelen 2001, Clarke & Wing 1996). Model checking was given crucial impetus by Kenneth L. McMillan who, using the technique of BDDs (**B**inary **D**ecision **D**iagrams) introduced and popularized by Randy Bryant, developed the highly efficient Symbolic Model Checking approach (McMillan 1993).

Model checking is a verification technique that is preferred to theorem proving technique. This method, similarly as it is in the case of logical calculi, is more effective comparatively to proof-theoretic method. It is one of the most active research areas because its procedures are automatic and easy to understand. This automated technique for verification and debugging has developed into a mature and widely used approach with many applications (Baier & Katoen 2008, p. xv).

According to Edmund M. Clarke (2008, p. 1):

> Model Checking did not arise in a historical vacuum. There was an important problem that needed to be solved, namely concurrent program verification.

In another place, he continues:[26]

> Existing techniques for solving the problem were based on manual proof construction from program axioms. They did not scale to examples longer than a page and were extremely tedious to use. By 1981 the time was ripe for a new approach to the problem, and most of necessary ideas were already in place.

Model checking bridges the gap between theoretical computer science and hardware and software engineering. Model checking does not exclude the use of proof-theoretical methods, and conversely, the proof-theoretical methods do not exclude using of model checking. The hybrid approach has many advantages. In practice, one of theses methods is complementary to the other, at least at the heuristic level.

On the one hand, failed proofs can guide to the discovery of counterexamples. Any attempt of proving may be forego by looking for counterexamples. Counterexamples of consequences of a theorem can help to reformulate it. Examples may aid comprehension and invention of ideas and can be used as a basis for generalization being expressed by a theorem. The role of decision procedures is often essential in theorem proving. There has been considerable interest in developing theorem provers that integrate SAT solving algorithms. The efficient and flexible incorporating of decision procedures

[26] See `http://events.berkeley.edu/index.php/calendar/sn/coe.html?event`.

into theorem provers is very important for their successful use. There are several approaches for combining and augmenting of decision procedures.

On the other hand, the combination of model checking with deductive methods allows the verification of a broad class of systems and, as it is in the case of e.g. STeP,[27] not restricted to finite-state systems. The question of combining proof-theoretical and model checking methods and the general problem of how to flexibly integrate decision procedures into heuristic theorem provers are subjects of many works (Boyer & Moore 1988).

Any verification using model-based techniques is only as good as the model of the system. It is one of the caveat of the model checking paradigm that the results of verification are only as reliable as the accuracy of the model which has been constructed for analysis. Thus, in model checking the first task is to convert a system to a formal model accepted by a model checker. It is a model in the form of a Kripke structure[28] or labeled graph of state transitions[29] – that has to accurately describe the behavior of the checked system. To do this, formal languages defined by formal semantics must be used. To draw an abstract model many techniques are applied. Many methods are used to reduce states of a system. In practice, this process is not automated.

The second task is to specify properties that must be satisfied by the real system. Mechanically assisted verification of properties of a complex system requires an accurate formulation of these properties in a formal language with defined semantics. The specification usually is given in some logical formalism. Generally, temporal logics are used to represent a temporal characteristic of systems.

A model checker is a device which is to decide whether the system satisfies its properties as expressed by temporal logic formulas. The answer is positive only if all runs are models of the given temporal logic formula. The

[27] See p. 83.

[28] Kripke or relational semantics of modal logics has been conceived in the fifties of the last century. This semantics was philosophically inspired, nevertheless, it has found application in CS. In CS Kripke structure is associated with a transition system. Because of the graphical nature of the state-space, it is sometimes referred to as the state graph associated with the system.

Here we will discuss languages with semantics based on Kripke structure (frame). Similarly as in modal logics this role may be played by Hintikka frames (Ben-Ari, Manna & Pnueli 1981). A Kripke frame consists of non-empty set and a binary relation defined on this set. In modal logics elements of the set are called possible worlds and the relation is understood as accessibility of one world from another. In the case of TL as applied in CS the Kripke semantics is based on computational time.

[29] A Kripke model is a directed graph where vertices are labeled by sets of atomic propositions and called states. Edges are called transitions.

technique is based on the idea of exhaustive exploration of the reachable state space of a system. For this reason, it can only be applied to systems with a finite state space, i.e., systems with finitely many configurations, and – for practical limitations (tractability) – with not too many states. The verification is completely automatic with the abstract model and properties. Thus, it is possible to verify the correctness of very complicated and very large systems, manual checking of which is almost not possible. We can verify a complex system as a hardware circuit or communication protocol automatically. The verification results are correct and easy to analyze. However, it does need human assistance to analyze the result of model checking. If logic is complete with respect to the model and is decidable, then in the case of any proposition that specifies the behavior of the system the procedure of checking is finite. But if the model is too detailed the verification becomes intractable. A model checker verifies the model and generates verification results, "True" or counterexample if the result is "False". If the proposition is satisfied, the system is verified. If the proposition is not valid, the construction results in a counterexample – this is one of the important advantages of model checking. The counterexample provides information about an error (bug) in the system. The model checker can produce a counterexample for the checked property, and it can help the designer in tracking down where the error occurred.

The counterexample gives us a new precondition or a negative result in the following way: when we obtain a counterexample, we analyze it and as far as this trace could not occur in real system, we add new preconditions to the formula. We may obtain a counterexample again, which often results to many preconditions. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking process. The verification model is traditionally constructed by hand, and is therefore subject to human mistakes. It means that error can also result from incorrect modeling of the system or from an incorrect specification. The error trace can also be useful in identifying and fixing these two problems.

Model checking comes in two varieties, depending on the way the proprieties are expressed. If theory of automata is employed, the system as well as its specification are described by automaton. Questions concerning system and its specification are reduced to the question about the behavior of automaton. In other words, automata theoretic approach means:

- specifying systems using automata
- reducing model checking to automata theory.

In the case of TL model checking, the system is modeled as a finite-state automaton, while the specification is described in temporal language. A model checking algorithm is used to verify whether the automaton has proper temporal-logical proprieties. In other words, if TL is applied (Miller et al. 2006, p. 2–3):

> Model checking involves checking the truth of a set of specifications defined using a temporal logic. Generally, the temporal logic that is used is either CTL^* or one of its sublogics, CTL [...] (Clarke et al. 1986) or LTL [...] (Pnueli 1981).

Various model checkers are developed. They are applied to verification of large models, to real-time systems, probabilistic systems, etc. (Holzmann 1991, Kurshan 1995, Clarke et al. 1999, Bérard, Bidoit, Finkel, Laroussinie, Petit, Petrucci & Schnoebelen 2001) – see (Schnoebelen 2002). Despite being hampered by state explosion, since its beginning model checking has had a substantive impact on program verification efforts.

Verification by model checking has gained popularity in industry for two reasons:

- the procedure can be fully automated and
- counterexamples are automatically generated if the property being verified does not hold.

Software is usually less structured than hardware and, especially in the case of concurrency, asynchronous. Thus, the state space is bigger in the case of software than in hardware. For this reason, in the case of hardware, the state explosion problem is less serious (Clarke 2008, p. 18).

It is worth mentioning some of the applications of model checking elsewhere. These include understanding and analyzing legal contracts, which are after all, prescriptions for behavior (Daskalopulu 2000); analyzing processes in living organisms for systems biology (Heath, Kwiatowska, Norman, Parker & Tymchysyn 2006); e-business processes, such as accounting and workflow systems (Wang, Hidvegi, Bailey & Whinston 2000). Model checking has also been employed for tasks in artificial intelligence, such as planning (Giunchig lia & Traverso 1999). Conversely, techniques from artificial intelligence related to SAT-based planning (Kautz & Selman 1992) are relevant to (bounded) model checking.

Let us repeat after Emerson some interesting remarks concerning model checking (Emerson 2008, p. 42):

> Edsger W. Dijkstra commented to me that it was an “acceptable crutch” if one was going to do after-the-fact verification. When I had the pleasure of meeting Saul Kripke and explaining model checking over Kripke structures to him, he

> commented that he never thought of that. Daniel Jackson has remarked that model checking has "saved the reputation" of formal methods.

It should be emphasized that a model checker caters for both verification and refutation of correctness properties. An important strength of model checkers is that they can readily provide a counter-example for most errors (Emerson 2008, p. 40). The real value of model checking is that it is an uncommonly good debugger – *reductio ad bug*[30].

As model checking verifies models and not realizations, testing is an essential complementary technique. Moreover, even if correctness of a system is checked it is not guaranteed to yield correct results: as with any tool, a model checker may contain software defects. Despite all the mentioned limitations, model checking is an effective technique to expose potential design errors.

In this article we survey and classify temporal logic model checking methods and model checkers. According to Emerson (2008, p. 31):

> . . . temporal logic has been a crucial factor in the success of model checking. We have one logical framework with a few basic temporal operators permitting the expression of limitless specifications. The connection with natural language is often significant as well. Temporal logic made it possible, by and large, to express the correctness properties that needed to be expressed. Without that ability, there would be no reason to use model checking. Alternative temporal formalisms in some cases may be used as they can be more expressive or succinct than temporal logic. But historically it was temporal logic that was the driving force.

2.4. State explosion problem

Model checking has some practical disadvantages. The most serious (and obvious) drawback of model checking is the combinatorial explosion of system states, commonly known as the *state explosion problem* (Clarke et al. 1999, Clarke & E. 1982b, McMillan 1993). Model checking algorithms rely upon the construction of models representing all system states. Finite state representation increases exponentially with the number of variables and concurrent components, therefore the number of states that have to be explored is growing exponentially with respect to the number of states of processes in the system. A concurrent program with k processes can

[30] For more about strengths and weaknesses of model checking see (Baier & Katoen 2008, pp. 14–16).

have a state graph of size $exp(k)$. For instance, in a banking network with 100 automatic teller machines each controlled by a finite state machine with 10 states, we can have 10^{100} global states. Thus, even for small sized examples, the state space becomes infeasible. This problem is caused when the size of the state space generated becomes so large that it is impossible to represent it in the computer memory given current memory configuration of computers. Systems with infinite state spaces, in general, cannot be handled.

Typical hardware and software systems contain many more states than can be practically explored by exhaustive methods. Nevertheless, model checking is an effective technique to expose potential design errors. Since the size of the state space grows exponentially with the number of processes, model checking techniques based on explicit state enumeration can only handle relatively small examples. Detailed application-specific formal analysis is required to invent appropriate abstraction and reduction techniques. These techniques allow model checkers to detect and avoid exploring many "equivalent" behaviors. Most current research in model checking is therefore devoted to combating state explosion. To cope with the problem, several techniques of model abstraction and state-space reduction have been developed. Various approaches and techniques (e.g., compression, reduction, aggregation, decomposition) are employed to reduce computational complexity (Schnoebelen 2002). Significant progress was made around 1990: both Symbolic Model Checking and the Partial Order Reduction were developed about this time.

To reduce the memory required to store each state, e.g., symbolic state representation is used:[31] if we can describe the set of states with a smaller number of symbolic states, we can verify larger systems. Symbolic methods are based on **B**inary **D**ecision **D**iagram (BDD) or its variants (Clarke, Fujita, McGeer, McMillan, Yang & Zhao 1993). With sophisticated implementations and refinements of symbolic model checking, it became possible to verify systems of industrial size, and to detect errors that can hardly be found using simulation. To reduce the number of states or paths explored, the methods of on-the-fly (Bouajjani, Tripakis & Yovine 1997), abstraction (Clarke, Grumberg & Long 1994, Daws & Tripakis 1998), partitioning (Burch, Clarke & Long 1991), partial order reduction (Willems & Wolper 1996, Gerth, Kuiper, Peled & Penczek 1995) or symmetry reduction are applied (Clarke et al. 1999, Miller et al. 2006).

[31] In the case of enumerative approach, each state of the system is represented explicitly.

As it is stated by Emerson (2008, p. 39):

> A remaining significant factor in ameliorating state explosion is the exponential growth in computer power, speed and especially memory size, expressed in Moore's law[32] which has obtained over the past quarter century. For example, in 1981 the IBM PC had less than 1M (random access) memory while today many PC's have 1G or more memory. Such a 1000-fold or larger increase in memory permits significantly larger programs to be handled.

The limits of models checking are pushed by employing work-station clusters and *GRID*s, e.g. the *VeriGEM* project aims at using the storage and processing capacity of clusters of workstations on a nation-wide scale, `http://www.bsik-bricks.nl/research_projects_afm6.shtml`.

Despite numerous improvements in model checking techniques, the state explosion problem remains their main obstacle. Recent development in manufacturing and design of *ICT* systems pose new challenges for functional verification methods. Hence, there is a growing need to investigate and develop more robust and scalable verification methods based on novel and alternative technologies.

3. Temporal logic

Formal verification requires a precise and unambiguous statement of the properties to be examined; this is typically done in temporal logic. *TL* provides a formalism which allows us to reason about how the truth value of certain assertions changes over time. *TL* is a form of logic especially appropriate for statements and reasoning about behaviors of order in time. *TL* is also precisely defined while limited in expressive power especially in comparison to formalisms such as First Order Arithmetic plus Monadic Predicates; yet, it seems ideally suited to describe synchronization and coordination behavior of concurrent systems. It possesses a nice combination of expressiveness and decidability. Although the language of first-order logic can express events of order in time, it is not intuitive since it explicitly uses variables to refer to objects. Modal temporal adverbs enable suppressing of explicit use of first-order variables denoting the time. Natural language sentences about events in time could be more intuitively expressed in a formal

[32] Moore's law describes a long-term trend in the history of computing hardware. See `http://en.wikipedia.org/wiki/Moore's_law`.

language if the language is enriched with modal adverbs. TL is a formalism for reasoning about time without introducing time explicitly.

In contrast to first-order representations, modal temporal logic makes a fundamental distinction between variability over time (as expressed by modal temporal operators) and variability in a state (as expressed using propositional or first-order languages). The language of TL also reflects the temporally indefinite structure of language in a way that is more natural than the method of using state variables and constants in a first-order logic.

TL has been developed for philosophical reasons and for long was a subject of philosophy rather than computer science. Nevertheless, Arthur Norman Prior, the founder of this logic, already was aware of practical gains to be had from this study in the representation of time-delay in computer circuits (Prior 1996, p. 46). Though there are no serious metaphysical assumptions that time is discrete, he justified the consideration (1967, p. 67):

> they are applicable in limited fields of discourse in which we are concerned only with what happens next in a sequence of discrete states, e.g. in the workings of a digital computer.

Rescher and Urquhart (1971) also pointed to the application of TL as a tool in consideration of:

> a programmed sequence of states, deterministic or stochastic [. . .].

The restriction to finite-state systems means that model checking procedures are in principle algorithmic and in practice efficient for many systems of considerable size.

A broad range of relevant system properties could be expressed in propositional temporal language. Among these, the safety (*something bad never happens*, e.g. the program never divides by zero), the liveness (*something good will eventually happen*, e.g. the program eventually terminates), reachability (*is it possible to end up in a deadlock state?*), fairness (*does, under certain conditions, an event occur repeatedly?*), and real-time properties (*is the system acting in time?*) properties are most useful when analyzing systems.

Temporal logics applied in CS include **P**ropositional **D**ynamic **L**ogic (PDL), **L**inear **T**emporal **L**ogic (LTL), and **P**ropositional **L**inear **T**emporal **L**ogic ($PLTL$) as its sublogic, CTL, and CTL^* as its generalization, Hennessy-Milner logic, and the logic T. Generally, they come in two varieties according to two common views, when describing discrete-state systems: linear and branching. The **L**inear-**T**ime (LT) view gives a full behavioral description. There is a single time line, and the system can be described by a trace of the states it can visit along time. If the notion of linear time is assumed, a single path at a time is considered. In the case of **B**ranching-**T**ime (BT) view the possible paths of the system form a computation tree. In the case of branching notion of time we can reason about multiple paths at a time. The difference between linear and branching time approach may be illustrated by the following systems.

$M1$ $M2$

From LT viewpoint there is no difference between $M1$ and $M2$. From the viewpoint of BT there is a difference: in the case of the system $M1$ there is no choice, but in the system $M2$ different runs are possible.

In LTL, formulas are interpreted over infinite words (Pnueli 1977, Vardi 1995), while in **B**ranching **T**emporal **L**ogics (BTL), for example, CTL, formulas are interpreted over infinite trees (Clarke & E. 1982a). According to linear or branching logic, the algorithms of model checking come into LTL and CTL model checking, respectively. BTL allows expression of some useful properties like 'reset'. CTL, a limited fragment of the more complete BT logic CTL^*, can be model checked in time linear in the formula size (as well as in the transition system). But formulas are usually far smaller than system models, so this is not as important as it may first seem. Some BTL, like μ-calculus and CTL, are well-suited for the kind of fixed-point computation scheme used in symbolic model checking.

3.1. Time

An important point of the temporal logic formalism is the view of time. A temporal expression is an expression for its interpretation a temporal structure is indispensable. Usually, the time is conceived as a set of time points with *earlier-later* relation on the set. Formally time $\mathfrak{T} = \langle T, \lhd \rangle$ consists of non-empty set T (usually of time points) and a binary relation $\lhd \subseteq T \times T$. In the case of minimal temporal logic, no property of $\lhd$ is assumed. The minimal temporal logic is the common part of different logics that "express" different properties of *earlier-later* relation. The *earlier-later* as intuitively conceived is asymmetric, transitive,[33] and infinite in both direction.

An important choice regards the shape of the future. Is the future unique or do we have different possible futures? By choosing linear time, we answer saying that future is unique, a linear path with no choice points. This is the approach taken by LTL (Pnueli 1977). To capture the idea that the future is open, but the past is determined, we allow branching into the future. With branching time, the future becomes a tree, adding some uncertainty to the model. At some points we may have different alternative futures, each as likely to occur as the other. Examples of BTL are CTL (Clarke & E. 1982b) and CTL^* (Clar et al. 1986). In circular time past, present and future coalesce. This option is useful in the treatment of certain repetitive processes.

There are several possible structures that one could reasonably imagine over states in time. TL as applied in CS is based on computational time, i.e. the time determined by the succession of states of a working ICT system. Moments of such a time are identified with states of system that are determined by values of variables (a set of propositional letters AP – **A**tomic **P**roposition) at a moment.[34] Each state is a set of propositions which are true of that state. States are related by an immediate predecessor-successor relation. The relation $\leq$ between states is determined by reflexive and transitive closing of system of transition,[35] hence states are linearly ordered and the relation $\leq$ is transitive. Since the set S of states is denumerable, the order is discrete, i.e. states form a sequence.

> The digital computers [...] may be classified amongst the 'discrete state machines'. These are the machines which move by sudden jumps or clicks from one quite definite state to another. (Turing 1950, p. 439)

---

[33] Irreflexivity of a relation is a consequence of its asymmetry.

[34] A computer operating under a program begins in an initial state with a given input. Various momentary states of the machine are characterized by the momentary values of all the variables in the program.

[35] By a system of transition we understand a tuple $\langle S, \{\xmapsto{a}\}_{a\in A}\rangle$, where S is a non-empty enumerable set of states, A is a non-empty set of actions (transitions, processes). With any action $a \in A$ a relation of transition $\xmapsto{a} \subseteq S \times S$ is associated. The relation of transition is total, i.e., for any state s there is state t such that s is in this relation with t. In other words: every state has a valid successor and each finite path is a prefix of some infinite path.

Discrete time allows us to talk about the next and previous moment in time.[36] The elements of S are being indexed with natural numbers. A computation (run, path) π is thus a sequence of subsets of AP. It is natural for our purposes to have a bound on the past, for instance for anchored semantics (Manna & Pnueli 1989). Traditionally the time domain is unbounded. For instance in the case of reactive systems (operational and control systems) it is reasonable to suppose that the sequence of states does not have an end. But bounded variations have been studied (Manna & Pnueli 1989), too. Any run π forms a sequence even though the branching structure of time is considered, e.g. in the case of concurrent systems any state of the system may have more than one possible successor state and runs may "branch" in such states following different successors. Some states are distinguished as initial. Let I be the set of all initial states.

The discrete option is more common, however the continuous option has been shown to have interesting applications in the field of real-time systems. **M**etric **T**emporal **L**ogic (MTL) extends TL with real time constraints. It was suggested by Chang, Pnueli, and Manna (1994) as a vehicle for the verification of real time systems. An example of a temporal logic over a dense domain is $MITL$ (**M**etric **I**nterval **T**emporal **L**ogic) (Alur, Feder & Henzinger 1991).

3.2. Temporal language

Propositional temporal language consists of the language of classical propositional logic enriched with additional temporal operators.

Philosophical temporal logic provided operators to reason both about the future and about the past. In the usual applications of temporal logic to verification the future temporal operators are preferred. Nevertheless, it has been pointed out that past-tense operators could provide a better abstraction for the formulation of certain specifications (Lichtenstein, Pnueli & Zuck 1985, Zuck 1986).

The propositional language consists of AP – an enumerable set of propositional letters (usually letters p, q, r, and if necessary, with indices are used). Symbols of classical propositional connectives are: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence). The symbols are read as in classical logic. We could have specified a smaller set of Boolean connectives, as long as they form a functionally complete basis, e.g., $\neg$, $\Rightarrow$.

[36] This is very usefully in the field of program specification and verification. It should be stressed that this decision is not motivated by the belief that time itself is discrete.

Other symbols may be used as abbreviations: instead of

$$(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$$

we may write

$$\phi \Leftrightarrow \psi.$$

The usual conventions concerning omitting superfluous parenthesis are accepted. The syntactic rules of forming sentences by means of classical connectives are the same as classical propositional logic. Meanings of the connectives are the same as in the classical logic but the definitions differ with respect to considered semantics.

Temporal logic formulas describe assertions about temporal relationships in state sequences. With respect to past and future, two kinds of temporal operators are distinguished. Future operators relate assertions on future states (including possibly the present one) to the reference state. In the case of past operators, assertions are related to past states (including possibly the present one). To any future temporal operator – if for the respective property of time the property symmetrical to it holds – a symmetrical past temporal operator can be defined (by changing the direction of the relation *earlier-later*), and conversely – to any past temporal operator a corresponding future temporal operator may be defined.

Connectives of classical logic are defined "within states". Temporal operators are defined by relation "between states" on a path.

In the case of linear temporal logic of computational time, formulas are evaluated over sequences of "time points", with $\mathbb{N}$ as index set and the linear order $<$ on $\mathbb{N}$. Temporal logics may be based on non-linear time models, too. The most popular is the branching time model. The idea of branching time was considered for philosophical reasons (omniscience and free will) by A. N. Prior. His solutions were inspired by ideas of Ockham and Peirce.

Unary operators G (*it will always be such that ...*) and H (*it has been always such that ...*) are most basic. The binary operators S (**S**ince) and U (**U**ntil)[37] were introduced by Kamp (1968). U and S are expressively complete for Dedekind's complete flows of time.[38] If time is discrete, two unary

[37] In temporal logic as applied in computer science, the discussed operators G, H, S and U are used in slightly different meaning than in philosophical logic. In computer science for simplicity (Lamport 1983) the reflexive meaning is preferred and often modal logic instead of typical temporal logic symbols are employed.

[38] If U and S are taken in strong and very strict meaning.

operators $\bigcirc$ (*next*) and $\ominus$ (*previous*) are definable. In the case of branching time, the language can be augmented with two unary path-operators: $\boldsymbol{E}$ and $\boldsymbol{A}$. In temporal logic as applied in computer science, first of all we are concerned with future temporal operators.[39]

Sentences (formulas) of propositional language are denoted with small Greek letters: $\phi, \psi, \ldots$, and if necessary, with indices.

Temporal operators are semantically defined with respect to Kripke frame. By the Kripke frame we understand a tuple $\langle S, I, \leq \rangle$, where S is non-empty set of states, $I(\subseteq S)$ – non-empty set of initial states and $\leq$ the relation obtained by reflexive and transitive closure of the system of transition. The structure is also known as a state graph or state transition graph or transition system.[40]

Though the semantical definitions of temporal operators could be done generally without any assumptions concerning the computational time, they are defined separately for linear and branching times. Let us start with the language of logic of linear computational time.

3.3. Linear temporal logics

Application of linear temporal logics LTL[41] in CS was originated by Pnueli (1977). Invented by him, the logic DUX was applied to concurrent systems. This logic has been axiomatize and developed in (Gabbay, Pnueli, Shelah & Stavi 1980) and (Manna & Pnueli 1981). The expressiveness of the language of LTL has been studied (Thomas 1997, Emerson 1990). Since the work of Pnueli, a variety of variants and extensions of LTL have been investigated. Among them, worth mentioning are Lamport's Temporal Logic of Actions (TLA) (Lamport 1994), and LTL with past operators (Gabbay et al. 1980, Lichtenstein et al. 1985, Laroussinie, Markay & Schnoebelen 2002).[42]

In the case of LTL, it is assumed that the computational time is isomorphic to the set of natural numbers $\mathbb{N}$, i.e., it is linear, discrete and bounded in the past. The language with two temporal operators G and $\bigcirc$ is the most basic. The operator G is called *always* and informally read as: *henceforth* or *always from now on*. The *next time* is denoted by $\bigcirc$ and informally read as:

[39] It has been proved that in the case of initial semantics for any formula there is an equivalent formula in that no past time operators occur (Gabbay 1981).

[40] See (Emerson 1990) for details.

[41] The term LTL refers mostly to propositional LTL ($PLTL$), as opposed to **Q**uantified **P**ropositional LTL ($QPLTL$).

[42] Adding the past operators does not change the expressiveness of LTL, but can be helpful for specification convenience and modular reasoning.

at the next instant. Formula $G\phi$ is read "*always* ϕ" and $\bigcirc\phi$ is read "*next* ϕ". Let LTL^{-} denote as well this language and its logic.

Definition 1

The set of LTL^{-} of well formed formulas is the smallest set such that

- $AP \subseteq LTL^{-}$
- $\neg\phi,\ (\phi \vee \psi),\ (\phi \wedge \psi),\ (\phi \Rightarrow \psi)\ \in LTL^{-},$ if $\phi,\ \psi\ \in LTL^{-}$
- $G\phi,\ \bigcirc\phi\ \in LTL^{-},$ if $\phi \in LTL^{-}$.

Labelling L is a mapping

$$L\colon S \rightarrow 2^{AP}$$

that labels each state in S with a set of propositional letters (that are satisfied in that state).

Let $\pi(i)$ be the set of propositional letters assigned by L to the state s_i, $s_i \in S; i \in \mathbb{N}$. A run (a path) π is a sequence: $\pi(0), \pi(1), \pi(2), \ldots$. In other words, a path (a run) is based on linearly ordered maximal subset of S. Let Π be the class of all runs (paths). Let π^i be the suffix of sequence π that starts from $\pi(i)$. π^0 is equal to π. It is supposed that $s_0 \in I$.

Kripke model is a 3-tuple $\langle S, <, L\rangle$ or more concise π.

The notion of satisfaction $\models$ is defined inductively.

Definition 2

For any $i\ \in\ \mathbb{N}$:

- $\pi(i) \models \phi$ iff $\phi \in \pi(i)$, for $\phi \in AP$
- $\pi(i) \models \neg\phi$ iff it is not true that $\pi(i) \models \phi$
- $\pi(i) \models \phi \vee \psi$ iff $\pi(i) \models \phi$ or $\pi(i) \models \psi$
- $\pi(i) \models \phi \wedge \psi$ iff $\pi(i) \models \phi$ and $\pi(i) \models \psi$
- $\pi(i) \models \phi \Rightarrow \psi$ iff not $\pi(i) \models \phi$ or $\pi(i) \models \psi$
- $\pi(i) \models G\phi$ iff for all $j,\ i \leq j\colon \pi(j) \models \phi$
- $\pi(i) \models \bigcirc\phi$ iff $\pi(i+1) \models \phi$.

$\pi^j(i)\ \models\ \phi$ will mean: $\pi(j+i)\ \models\ \phi$. In particular $\pi(i)\ \models\ \phi$ means: $\pi^0(i) \models \phi$.

The operator G is taken in reflexive meaning.[43] $G\phi$ informally means: *ϕ holds in all forthcoming states including the present one.*

---

[43] It is typical in writings devoted to applications of TL in CS. The practice started with (Gabbay et al. 1980).

The operator F is dual to the operator G, i.e. $F\phi$ means the same as $\neg G\neg\phi$ or $G\phi$ means the same as $\neg F\neg\phi$. The operator F may be defined semantically as follows:

- $\pi(i) \models F\phi$ iff exists $j,\ i \leq j\colon \pi(j) \models \phi$.

F is called *sometime* or *eventuality* and the formula $F\phi$ informally is read as: *sometime* ϕ, *now or in the future* ϕ or *eventually from now on.*

The language LTL^- enriched with a temporal operator U is denoted LTL. In the clauses for LTL^- instead of LTL^- we write LTL and a new clause is added:

- if $\phi, \psi \in LTL$, then $(\phi U \psi) \in LTL$.

The operator U is called *until.*[44] The formula $\phi U \psi$ is read: ϕ *until* ψ.

$\phi U \psi$ can be understand as follows: *there is a subsequent state (possibly the present one) in which* ψ *holds, and* ϕ *holds until that state.*

Such a meaning of U is defined as follows:

- $\pi(i) \models \phi U \psi$ iff exists $j,\ i \leq j\colon \pi(j) \models \psi$ and for all $k,\ i \leq k < j$: $\pi(k) \models \phi$.

It can be observed that the word *until* may be used in various meanings. First of all, strong and weak meanings may be discerned. The difference is based on the existence of the point in that the second argument is satisfied. In the case of weak meaning, conversely to the strong one (as in above definition), the existence of this point is omitted. The operator W renders the weak meaning of the word *until.*

- $\pi(i) \models \phi W \psi$ iff exists $j,\ i \leq j\colon \pi(j) \models \psi$ and for all $k,\ i \leq k < j$: $\pi(k) \models \phi$ or for all $j,\ i \leq j\colon \pi(j) \models \phi$.

$\phi W \psi$ is read: ϕ *is waiting for* ψ or ϕ *unless* ψ.

The operators U and W are taken in the reflexive meaning. Two other types of meanings may be pointed out. In definitions of U and W there are two possibilities of changing of $\leq$ for $<$. Besides the reflexive meaning, the strict U^s and W^s (in the formal definitions of U and W the first occurrence of $\leq$ is replaced by $<$) and very strict U^{vs} and W^{vs} (all the occurrences of $\leq$ are replaced by $<$) meanings are distinguished.[45]

- $\pi(i) \models \phi U^s \psi$ iff exists $j,\ i < j\colon \pi(j) \models \psi$ and for all $k,\ i \leq k < j$: $\pi(k) \models \phi$

---

[44] The operator was originally investigated by Kamp (1968) and introduced into the context of program analysis in (Gabbay et al. 1980).

[45] The strong and very strict notion of "until" is preferred in philosophical logic.

- $\pi(i) \models \phi W^s \psi$ iff exists $j,\ i < j\colon \pi(j) \models \psi$ and for all $k,\ i \leq k < j$: $\pi(k) \models \phi$ or for all $j,\ i \leq j\colon \pi(j) \models \phi$
- $\pi(i) \models \phi U^{vs} \psi$ iff exists $j,\ i < j\colon \pi(j) \models \psi$ and for all $k,\ i < k < j$: $\pi(k) \models \phi$
- $\pi(i) \models \phi W^{vs} \psi$ iff exists $j,\ i < j\colon \pi(j) \models \psi$ and for all $k,\ i < k < j$: $\pi(k) \models \phi$ or for all $j,\ i < j\colon \pi(j) \models \phi$.

It could be proved that all the other future temporal operators of LTL can be defined by means of U and $\bigcirc$.

Figuratively speaking, states do not have a past. For the present state, if it is not initial, there are past states, but the future in no way is affect by the past states. It depends only on the present state. This fact does not cancel the possibility of usefulness of a language with past temporal operators (Lichtenstein et al. 1985). Moreover, such languages are not more complex (Sistla & Clarke 1985).

LTL is typically presented with temporal operators meant to express the truth values of formulas in the future. This need not to be the case, it is indeed possible to introduce temporal operators regarding the past, or ever consider a logic with only past temporal operators. Though adding past-tense operators to the future fragment of LTL does not increase its expressive power,[46] the past-tense operators allow to express some interesting properties in a more natural manner (Lichtenstein et al. 1985). It has been argued that past temporal operators can be particularly useful in specifying the intended behavior of certain systems (Lichtenstein et al. 1985, Zuck 1986).

It has been shown that LTL, both with past-tense or future-tense operators, cannot express certain properties which involve counting (Wolper 1983). To overcome these shortcomings, the introduction of special grammar operators has been suggested (Wolper 1983) or the addition of counting operators as done in the ATG Temporal Rover (Drusinsky 2000). The problem is solved in μ-calculus (Goranko 2000, p. 67).

The language $LTLP$ is the LTL language extended by past time operators. $\ominus$ (weak previous operator) is an unary operator such that $\ominus\phi$ means: *ϕ held in the previous state*. H (has-always-been operator) is the operator symmetrical to G. $G\phi$ means: *ϕ held in all past states* (*including the present one*). S is a two-argument operator and $\phi S \psi$ is read: *ϕ since ψ*. Both the operators $\ominus$ and S suffice to define all the other past temporal operators of $LTLP$.

[46] This is however misleading in a certain sense.

- $\pi(i) \models \ominus\phi$ iff $\pi(i-1) \models \phi$, if $i > 0$
- $\pi(i) \models H\phi$ iff for any $j,\ j \leq i\colon \pi(j) \models \phi$
- $\pi(i) \models \phi S\psi$ iff exists $j,\ j \leq i\colon \pi(j) \models \psi$ and for all $k,\ j < k\ \leq i$: $\pi(k) \models \phi$.

In the case of *since*, similar consideration concerning its meaning may be repeated, as in the case of *until*: there are strong and weak, and for both the reflexive, stricte and very stricte meanings.

$\bigcirc$ and $\ominus$ are symmetrical one to another. The same is true about U and S (and other respective variants of *until* and *since*), G nad H. The symmetry does not result in mirror image rule even for the law of linear logic, because of asymmetry of past and future. The computational linear and branching time has a beginning but does not have an end. Past is limited by the initial state but it is supposed that there are infinitely many future states.

The alphabet of $TLTP$ is augmented by symbols $\ominus, H, S$. The language $LTLP$ is defined by adding to the clauses of the definition of the language LTL (in the clauses LTL is changed for $LTLP$) the following clauses:

- if $\phi \in LTLP$, then $\ominus\phi \in LTLP$
- if $\phi \in LTLP$, then $H\phi \in LTLP$
- if $\phi, \psi \in LTLP$, then $\phi S\psi \in LTLP$.

The operator P (*once*) as counterpart of F may be defined as abbreviation for $\neg H \neg$.

In TL, the satisfiability and validity could be defined in different ways. There are two view points: *classical* or *floating viewpoint* and *initial* or *anchored* (Emerson 1990, Lichtenstein & Pnueli 2000). Thus, two semantics are distinguished: normal and initial (Szałas 1995, p. 2). Initial semantics is typical and usual in the case of software development (Manna & A. Pnueli 1992, 1995a). Both versions do not differ in the set of valid formulas. They differ, however, when one considers properties of temporal theories.

Let us start with classical notions. In any of the discussed languages the definitions of satisfiability and validity have the following schemata.

Definition 3

- A formula ϕ is satisfiable (valid) in π, $\pi \models \phi$, iff for any $i,\ 0 \leq i\colon \pi(i) \models \phi$
- a formula ϕ follows from a set Σ of formulas, $\Sigma \models \phi$, iff for any $\pi(\in \Pi)$: if $\pi \models \Sigma$, then $\pi \models \phi$
- ϕ is (universally) valid, $\models \phi$, if and only if $\emptyset \models \phi$.

$\pi(i) \models \Sigma$ means: *for any* $\phi \in \Sigma$: $\pi(i) \models \phi$.

$\pi \models \Sigma$ means: *for any* $\phi \in \Sigma$: $\pi \models \phi$.

$\pi^i \models \phi$ will mean: *for all* $j,\ i \leq j$: $\pi(j) \models \phi$. In particular, $\pi \models \phi$ means: $\pi^0 \models \phi$.

The anchored viewpoint is rendered in the following definitions.

Definition 4

- ϕ is initially satisfiable (valid) in π, $\pi \models^0 \phi$, iff $\pi(0) \models \phi$
- ϕ initially follows from a set Σ, $\Sigma \models^0 \phi$, iff for any $\pi(\in \Pi)$: if $\pi \models^0 \Sigma$, then $\pi \models^0 \phi$
- ϕ (universally) initially valid, $\models^0 \phi$, if and only if $\emptyset \models^0 \phi$.

Linear temporal logic equipped with the initial validity semantics will be denoted by LTL_0^-, LTL_0, $LTLP_0$.

$\pi^i \models^0 \phi$ will mean: $\pi(i) \models^0 \phi$ or $\pi(i) \models \phi$. In particular $\pi \models^0 \phi$ means: $\pi(0) \models^0 \phi$ or $\pi(0) \models \phi$.

Validity and satisfiability are "dual" in the following sense: ϕ *is valid iff* $\neg\phi$ *is not satisfiable.*

The following theorems are consequences of the above definitions.

Theorem 1

If $\Sigma \models \phi$ and $\Sigma \models \phi \Rightarrow \psi$, then $\Sigma \models \psi$.

Proof. Let $\Sigma \models \phi \Rightarrow \psi$ and $\Sigma \models \phi$ and not $\Sigma \models \psi$. Hence for some π: $\pi \models \Sigma$ and for some i: $\pi(i) \not\models \psi$. From the supposition $\pi(i) \models \phi$. Thus not $\pi(i) \models \phi \Rightarrow \psi$. Therefore $\Sigma \not\models \phi \Rightarrow \psi$. This is in contradiction with the assumption. $\square$

Theorem 2

If $\Sigma \models \phi$, then

- $\Sigma \models G\phi$
- $\Sigma \models \bigcirc\phi$.

Proof. Let us prove only the second fact. Let us suppose that $\Sigma \models \bigcirc\phi$ is not valid. Thus for some π: $\pi \models \Sigma$ and for some i: $\pi(i) \not\models \bigcirc\phi$. By definition of $\bigcirc$ we have that $\pi(i+1) \not\models \phi$. This fact is in contradiction with the assumption that $\Sigma \models \phi$. $\square$

Let us consider some connections between LTL and LTL_0.

Lemma 3

- If $\pi \models \phi$, then $\pi \models^0 \phi$
- $\pi \models \phi$ iff $\pi \models^0 G\phi$.

Proof. From the definition $\pi \models \phi$ means that for any $i,\ 0 \leq i{:}\, \pi(i) \models \phi$. In particular thus $\pi(0) \models \phi$. Hence $\pi \models^0 \phi$.

From the definition of $\models$ for any $i,\ 0 \leq i{:}\, \pi(i) \models \phi$. This thus means that $\pi(0) \models G\phi$. By the definition of $\models^0$ this is equivalent to $\pi \models^0 G\phi$. □

Let $G\Sigma = \{G\phi{:}\, \phi \in \Sigma\}$.

Theorem 4

- If $\Sigma \models^0 \phi$, then $\Sigma \models \phi$
- $\Sigma \models \phi$ iff $G\Sigma \models^0 \phi$.

Proof. Let $\Sigma \models^0 \phi$ and for some $\pi{:}\, \pi \models \Sigma$. Hence for all $i,\ 0 \leq i{:}\, \pi(i) \models^0 \phi$. Thus for all $i,\ 0 \leq i{:}\, \pi(i) \models \phi$. From it follows $\pi \models \phi$ and finally, because of arbitrariness of π, $\Sigma \models \phi$.

Let $\Sigma \models \phi$. Thus by definition for any π: if $\pi \models \Sigma$, then $\pi \models \phi$. Let for some $\pi{:}\, \pi \models \Sigma$. Let $\psi \in \Sigma$, thus $\pi \models \psi$. It means that for all i, $0 \leq i{:}\, \pi(i) \models \psi$. Hence for all $i{:}\, \psi(i) \models^0 \psi$. Thus $\pi \models^0 G\psi$. From $\pi \models \phi$ follows that $\pi \models^0 \phi$. For arbitrariness of π and ψ we obtain $G\Sigma \models^0 \phi$.

Let $\Sigma \not\models \phi$. Thus for some $\pi{:}\, \pi \models \Sigma$ and for some $i{:}\, \pi(i) \not\models \phi$. Hence for any $\psi,\ \psi \in \Sigma{:}\, \pi \models \psi$. From it follows that for any $i,\ 0 \leq i{:}\, \pi(i) \models \psi$. By definition of $\models^0$ and G we obtain that $\pi \models^0 G\psi$. For some $i{:}\, \pi^i \not\models \phi$ since for some $i{:}\, \pi(i) \not\models \phi$. Hence $\pi^i \not\models^0 \phi$. Thus $G\Sigma \not\models^0 \phi$. □

In LTL the relationship between implication and semantical consequence (the semantical counterpart of deduction theorem) is stated in the following theorem.

Theorem 5

$\Sigma \cup \{\phi\} \models \psi$ iff $\Sigma \models G\phi \Rightarrow \psi$.

Proof. Let us suppose that for some π: $\pi \models \Sigma$, $\pi \not\models G\phi \Rightarrow \psi$. Thus for some $i{:}\, \pi(i) \not\models G\phi \Rightarrow \psi$. Hence $\pi(i) \models G\phi$ and $\pi(i) \not\models \psi$. From $\pi(i) \models G\phi$ it follows that $\pi^i \models \phi$. Next we have that $\pi^i \models \Sigma \cup \{\phi\}$. We get a contradiction with $\Sigma \cup \{\phi\} \models \psi$, because $\pi^i(0) \not\models \psi$.

Let now for some $\pi\colon \pi \models \Sigma \cup \{\phi\}$ and for some $i\colon \pi(i) \not\models \psi$. In consequence we have $\pi \models \phi$. In particular thus for any j, $i \leq j\colon \pi(j) \models \phi$. Hence $\pi^i(0) \models G\phi$. From $\pi \models \Sigma$ it follows that $\pi^i \models \Sigma$. We get a contradiction, because $\pi^i(0) \not\models \psi$. □

In the case of initial semantics the semantical counterpart of deduction theorem is formulated as in classical propositional logic.

Theorem 6

$\Sigma \cup \{\phi\} \models^0 \psi$ iff $\Sigma \models^0 \phi \Rightarrow \psi$.

Proof. Let $\Sigma \models^0 \phi \Rightarrow \psi$ does not hold. Thus for some $\pi\colon \pi \models^0 \Sigma$ and $\pi \not\models^0 \phi \Rightarrow \psi$. Hence $\pi \models^0 \phi$ and $\pi \not\models^0 \psi$. By definition of initial semantics we have that $\pi(0) \models \phi$ and $\pi(0) \not\models \psi$. Then $\pi(0) \models \Sigma \cup \{\phi\}$ and therefore $\Sigma \cup \{\phi\} \not\models^0 \psi$.

Let now $\Sigma \cup \{\phi\} \not\models^0 \psi$. Thus for some $\pi\colon \pi \models^0 \Sigma \cup \{\phi\}$ and $\pi \not\models^0 \psi$. By definition of initial semantics, then $\pi(0) \models \Sigma \cup \{\phi\}$ and $\pi(0) \not\models \psi$. Therefore $\pi(0) \models \Sigma$ and $\pi(0) \not\models \phi \Rightarrow \psi$. Hence $\Sigma\{\phi\} \not\models^0 \psi$. □

Despite all the differences between LTL and LTL_0 it is remarkable that in both cases LTL and LTL_0 universally valid are the same formulas.

Theorem 7

$\models \phi$ iff $\models^0 \phi$.

From the angle of initial semantics, the languages LTL and $LTLP$ have the same expressibility (Gabbay et al. 1980, Gabbay 1989). It means that for any formula ϕ of the language $LTLP$ there is a formula ψ of the LTL such that both the formulas are initially equivalent.

With the help of the language of LTL the property of fairness and some other correctness properties could be expressed. As an example, let us consider 2-processes-resource-manager.

Let reg_i and $owns_i$ be propositional letters.

- reg_i is true iff the i-process is requested the resource.
- $owns_i$ is true exactly then the i-process owns the resource.

The **safety** property is expressed by:

$$G\neg(owns_1 \wedge owns_2)$$

– it is excluded that both the processes at the same time own the resource. The **liveness** property is expressed by:

$$G(req_1 \Rightarrow Fowns_1)$$

– if the 1-process is requested the resource, then the process eventually will own it.

The **strong fairness** is expressed by:

$$GF(req_1 \wedge \neg(owns_1 \vee owns_2)) \Rightarrow GF owns_1$$

– if endless often the free resource is requested, then endless often the resource will be own.

The **priority**:

$$G(req_1 \wedge req_2 \Rightarrow (\neg owns_2 W(owns_2 W(\neg owns_2 W owns_1))))$$

– if both the processes compete, the 2-process will win (has the priority)

There is a strong connection between TL models and automata (Goranko 2000, p. 55–62). A proposition of TL is satisfied if the language accepted by a proper automaton is not empty. Automata checking non-emptiness are constructive, i.e. automaton produces the model if the proposition is satisfied. This fact is important for the theory of automata and for fruitfulness of application of TL in CS (Vardi & Wolper 1986b, Thomas 1990, Wolper 1995).

3.4. Temporal logics of branching time

Lamport was the first to investigate the expressive power of various TLs (Clarke 2008, p. 11). Already in 1980, he remarked that languages of BTLs and LTLs differ in expressivity. Lamport (1980) discussed two logics: a simple linear-time logic and a simple branching-time logic. Emerson and Halpern (1983, 1986) fixed these problems. BTLs cannot express certain natural fairness properties that can be easily expressed in the LTLs. The language of LTLs cannot express the possibility of an event occurring at sometime in the future along with some computation path.

The development of BTLs for specification and verification of ICT systems began in eighties of the last century (Pnueli 1985b, Clarke et al. 1986, Emerson & Halpern 1986). Semantics of these languages gives more information (Glabbeek 2001). Various branching temporal logics have been proposed.

As in the case of linear temporal logics, our discussion will be restricted to propositional logics.

The operators of the language of BTL Basic **B**ranching Time **T**emporal **L**ogic [or UB – **U**nified System of **B**ranching Time, (Ben-Ari et al. 1981)] combine the expressibility of the existence of branches in the state structure with the possibility of speaking about states on the branches. This is enabled by path modalities:

$\boldsymbol{E} \bigcirc \phi$ – *there is a successor state (starting from the present state) in which ϕ holds,*

$\boldsymbol{E}G\phi$ – *there is a branch (starting from the present state) on which ϕ holds in all subsequent states,*

$\boldsymbol{E}F\phi$ – *there is a branch (starting from the present state) on which ϕ holds in some subsequent state,*

$\boldsymbol{A} \bigcirc \phi$ – *ϕ holds in all successor states (starting from the present state),*

$\boldsymbol{A}G\phi$ – *ϕ holds on all branches (starting from the present state) in all subsequent states,*

$\boldsymbol{A}F\phi$ – *on all branches (starting from the present state), ϕ holds in some subsequent state.*

The language of BTL is obtained by extension of LTL^- by the above modalities.

These modalities are integral, i.e. the paths operators can only be used before $\bigcirc$, G, F. E.g., $\boldsymbol{E}\neg \bigcirc \phi$ is not a formula.

Let $\Pi(i)$ denote all the paths such that the state $\pi(i)$ is their element, i.e. $\pi_1 \in \Pi(\pi(i))$ if and only if $\pi_1(i) = \pi(i)$. Because of the linearity in the past for any path of $\Pi(\pi(i))$ the number of former states is the same. In other words, members of $\Pi(\pi(i))$ are paths with the same common initial sequence: $\pi_0, \ldots, \pi_i$. They may differ in the case of $(i + 1)$-members.

To define the notion of satisfaction in BTL the following clauses are added to the clauses of LTL^-:

Definition 5 (Satisfiability)

- $\pi(i) \models \boldsymbol{E} \bigcirc \phi$ iff exists $\pi_1 \in \Pi(\pi(i))$: $\pi_1(i+1) \models \phi$,
- $\pi(i) \models \boldsymbol{A} \bigcirc \phi$ iff for all $\pi_1 \in \Pi(\pi(i))$: $\pi_1(i+1) \models \phi$,
- $\pi(i) \models \boldsymbol{E}G\phi$ iff exists $\pi_1 \in \Pi(\pi(i))$ and for all $j,\ j \in \mathbb{N}$: $\pi_1(i+j) \models \phi$,
- $\pi(i) \models \boldsymbol{A}G\phi$ iff for all $\pi_1 \in \Pi(\pi(i))$ and for all $j,\ j \in \mathbb{N}$: $\pi_1(i+j) \models \phi$,
- $\pi(i) \models \boldsymbol{E}F\phi$ iff exists $\pi_1 \in \Pi(\pi(i))$ and there is $j,\ j \in \mathbb{N}$: $\pi_1(i+j) \models \phi$,
- $\pi(i) \models \boldsymbol{A}F\phi$ iff for all $\pi_1 \in \Pi(\pi(i))$ there is $j,\ j \in \mathbb{N}$: $\pi_1(i+j) \models \phi$.

The notions of satisfiability, consequence and validity are formulated as in the case of LTL.

The path modalities are dual in the following sense:

- $\models \boldsymbol{A} \bigcirc \phi \Leftrightarrow \neg \boldsymbol{E} \bigcirc \neg \phi$,
- $\models \boldsymbol{A} G\phi \Leftrightarrow \neg \boldsymbol{E} F \neg \phi$,
- $\models \boldsymbol{A} F\phi \Leftrightarrow \neg \boldsymbol{E} G \neg \phi$.

Let us remark that $\boldsymbol{E}F\phi$ means that ϕ is possible in some future (along some future, ϕ eventually holds and is thus possible). The inevitability of ϕ is expressed by $\boldsymbol{A}F\phi$ (along all futures, ϕ eventually holds and is thus inevitable) (Ben-Ari, Manna & Pnueli 1983).

Extension of the language BTL by binary operator U yields the language CTL (Clarke & E. 1982b, Queille & Sifakis 1982).

In BTL, as well as in CTL, the path quantifiers $\boldsymbol{A}$ and $\boldsymbol{E}$ are used only as an “attachment” to temporal operators. They are used as “standalone” in the language CTL^*, where the path quantifiers $\boldsymbol{A}$ and $\boldsymbol{E}$ are used to denote *for all paths* and *for some path*, respectively.

There are two kinds of formulas of the language of CTL^*: p-formulas (**p**ath formulas) and s-formulas (**s**tate formulas).

Definition 6

The language of CTL^* is the smallest set of formulas that satisfies the following conditions:

- ϕ is a s-formula, if $\phi \in AP$;
- if ϕ and ψ are s-formulas, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \Rightarrow \psi$;
- if ϕ is a p-formula, then $\boldsymbol{A}\phi$ and $\boldsymbol{E}\phi$ are s-formulas;
- any s-formula is also a p-formula;
- if ϕ and ψ are p-formulas, then so are $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$, $\bigcirc\phi$, $(\phi U \psi)$, $G\phi$, $F\phi$.

The formulas of CTL[47] are formulas of CTL^* in that the temporal operators $\bigcirc$, U, F and G are immediately preceded by a path quantifier. The language of branching time may also be extended be adding past temporal operators.

Two notions of satisfiability have to be defined. In any case the symbol $\models$ will be used, however the meaning of $\models$ will depend on the connective or operator taken into account. In the case of s-formulas, satisfiability will be defined for states but in the case of p-formulas it will be done for paths.

[47] The language CTL was conceived earlier than CTL^*.

The operator $\bigcirc$ and temporal operators, in particular U are operating on path. The next or the future state is on a path. In other words, there is no next or future state independently on the path. To define a next or a future state we have to point a path on that the state is next or future.

Definition 7 (satisfiability in a state)

For any $\pi \in \Pi$, any $i \in \mathbb{N}$:

- $\pi(i) \models \phi$ iff $\phi \in \pi(i)$, if $\phi \in AP$;
- $\pi(i) \models \neg\phi$ iff not $\pi(i) \models \phi$;
- $\pi(i) \models \phi \vee \psi$ iff $\pi(i) \models \phi$ or $\pi(i) \models \psi$;
- $\pi(i) \models \phi \wedge \psi$ iff $\pi(i) \models \phi$ and $\pi(i) \models \psi$;
- $\pi(i) \models \phi \Rightarrow \psi$ iff if $\pi(i) \models \phi$, then $\pi(i) \models \psi$;
- $\pi(i) \models \boldsymbol{E}\phi$ iff $\exists \pi_1 \in \Pi(\pi(i))\colon \pi_1^i \models \phi$;
- $\pi(i) \models \boldsymbol{A}\phi$ iff $\forall \pi_1 \in \Pi(\pi(i))\colon \pi_1^i \models \phi$.

Let us remark that though $\pi(i)$ and $\pi_1(i)$ are equal for any $\pi, \pi_1 \in \Pi$, in the last two clauses the indication of a path could not be omitted if in ϕ operators occur for that the satisfiability is defined on a path.

All the future temporal operators that are definable in CTL^* are definable by means of $\bigcirc$ and U. Thus it is enough to define the notion of satisfiability for both these operators.

Definition 8 (satisfiability on a path)

- $\pi^i \models \bigcirc\phi$ iff $\pi^{i+1} \models \phi$;
- $\pi^i \models \phi U \psi$ iff exists $j,\ j \geq i$: ($\pi^j \models \psi$ and for any $k,\ i \leq k\ < j\colon \pi^k \models \phi$).

The notions of satisfiability (validity), consequence and (universal) validity as defined in the same way as for earlier logics.

It can be seen that all valid formulas of BTL are valid formulas of CTL and these are valid formulas of CTL^*. The conversely does not hold.

Some semantical facts are worth to mention.

- $\models \boldsymbol{E}F\phi \Leftrightarrow \boldsymbol{E}\top U\phi$.
- $\models \boldsymbol{E}\phi W\psi \Leftrightarrow \boldsymbol{E}\phi U\psi \vee \boldsymbol{E}G\phi$.
- $\models \boldsymbol{A}\phi W\psi \Leftrightarrow \neg\boldsymbol{E}(\neg\psi)U(\neg\phi \wedge \neg\psi)$.
- $\models \boldsymbol{A}\phi U\psi \Leftrightarrow \boldsymbol{A}\phi W\psi \wedge \boldsymbol{A}F\phi$.

By means of $\boldsymbol{E}U$, $\boldsymbol{A}F$, $\boldsymbol{E}\bigcirc$ all the other modalities can be defined (Laroussinie 1995).

CTL^* is more expressive than CTL, but is more computationally complex. CTL is decidable in exponential time. CTL^* is decidable in double exponential time. The lower bound was described by Vardi and Stockmeyer (1985), the upper bound was estimated by Emerson and Jutla (1988, 1999).

The logic CTL^*P has been defined by Laroussinie and Schnoebelen (1994) as a complete logic of computational trees with linear past temporal operators. This logic has been investigated in (Zanardo & Carmo 1993, Kupferman & Pnueli 1995, Laroussinie & Schnoebelen 2000). CTL^*P is more useful than CTL^*. The axiomatization of this logic is not so complicated as axiomatization of CTL^*.

The language of CTL^*P is obtained by adding past temporal operators $\ominus$ and S to the alphabet of CTL^*. The formation rules regarding the operators $\ominus$ and S are analogical to the rules for $\bigcirc$ and U, respectively.

To the notion of satisfiability for CTL^* the following clauses are added:

Definition 9

- $\pi^i \models \ominus\phi$ iff $\pi^{i-1} \models \phi$, if $i > 0$,
- $\pi^i \models \phi S\psi$ iff exists $j,\ j \leq i$: ($\pi^j \models \psi$ and for any $k,\ j \leq k\ < i$: $\pi^k \models \phi$).

In principle structures with time linear in the past are considered, i.e., structures typical for Ockham logic [in Prior sense (Prior 1967, Burgess 1984)]. It does not mean that logic of time branching in the past are not taken into account (Reisig 1989, Kaminski 1994, Kupferman & Pnueli 1995). CTL^*P has been axiomatized and the completeness has been proved by Reynolds (2005).

The branching time logics BTL, CTL and CTL^* can be extended or modified in various ways.[48]

The languages of linear and branching temporal logics differ in expressiveness (Emerson 1996). Some proprieties are expressible in the language of linear time and not expressible in the language of branching temporal language, and conversely. LTL and CTL are of incomparable expressive power, while CTL^* is strictly more expressive than either of the others. Eliminating the existential path quantifier from CTL and CTL^* does not affect the relative expressive power of the logics. $\boldsymbol{A}CTL^*$ trivially encompasses LTL and $\boldsymbol{A}CTL$. On the one hand, the formula $\boldsymbol{A}F\boldsymbol{A}Gp$ is a formula of $\boldsymbol{A}CTL$ that does not have an equivalent LTL formula. On the other hand, there is no $\boldsymbol{A}CTL$ formula that is equivalent to LTL formula $\boldsymbol{A}FGp$. Thus,

[48] For more see e.g. (Kroger & Merz 2008, p. 368–373).

LTL and $\boldsymbol{A}CTL$ are incomparable, and both are strictly less expressive than $\boldsymbol{A}CTL^*$.

Languages of branching TLs, in particular CTL and its variants are most widely used in connection with verification. The languages CTL and CTL^* suffice to express many important proprieties (Emerson & Halpern 1983). CTL has the flexibility and expressiveness to capture many important correctness properties. In addition, the CTL model checking algorithm is of reasonable efficiency, polynomial in the structure and specification sizes.

Linear TLs and branching TLs differ in the methods of model checking though common methods are developed (Vardi 1989, Kupferman, Vardi & Wolper 2000). In some cases the complexity of methods of linear TLs is greater than branching TLs, and conversely. CTL^* (Emerson & Halpern 1986) provides a uniform framework that subsumes both LTL and CTL, but at the higher cost of deciding satisfiability.

Some of TLs in increasing expressive power are: Hennessy-Milner logic HML (Hennessy & Milner 1985), Unified System of Branching-Time Logic (Ben-Ari et al. 1981, Ben-Ari et al. 1983), Computation Tree Logic CTL (Clarke & E. 1982a), Extended Computation Tree Logic CTL^* (Clarke & E. 1982a), and modal μ-calculus (Kozen 1982, Kozen 1983).

There has been an ongoing debate as to whether linear TL or branching TL is better for program reasoning (Lamport 1980, Emerson & Halpern 1983, Vardi 2001).

3.5. The μ-calculus

The μ-calculus may be viewed as a particular but very general temporal logic. Some formulations go back to the work of de Bakker and Scott (1969). It can be introduced as an extension of the temporal logic for transition systems with operators for a least fixpoint μ and a greatest fixpoint ν (Emerson & Clarke 1980, Pratt 1981, Kozen 1983). The μ-calculus provides operators for defining correctness properties using recursive definitions plus least fixpoint and greatest fixpoint operators. Many μ-calculus characterizations of correctness properties are elegant due to its simple underlying mathematical organization. It turns out to be an extremely powerful formalism for specification and verification of temporal properties.

The idea of model checking instead of testing satisfiability was introduced in (Emerson & Lei 1986). μ-calculus subsumes most temporal logics, including LTL, CTL and more powerful logics such as CTL^*, i.e., in μ-calculus most of the known temporal logics can be translated (Dam 1994, Emerson 1990, Goranko 2000).

The μ-calculus is a rich and powerful formalism; its formulas are really representations of alternating finite state automata on infinite trees (Emerson & Jutla 1991). The μ-calculus is very expressive and very flexible. Model checking is essentially a problem of fixedpoint computation.

The power of the μ-calculus comes from its fixpoint operators. Least fixpoints correspond to well-founded or terminating recursion, and are used to capture liveness or progress properties asserting that something does happen. Greatest fixpoints permit infinite recursion. They can be used to capture safety or invariance properties.

Definition 10

An operator $\tau\colon 2^S \to 2^S$ is monotonic, if $X_1 \subseteq X_2$ implies $\tau(X_1) \subseteq \tau(X_2)$, where $X_1, X_2 \subseteq S$.

Definition 11

A set $X \subseteq S$ is a

- fixpoint of τ, if $\tau(X) = X$;
- pre-fixpoint of τ, if $\tau(X) \subseteq X$;
- post-fixpoint of τ, if $\tau(X) \supseteq X$;
- least fixpoint, $\mu\tau$, if X is a fixpoint (pre-fixpoint) and for every fixpoint (pre-fixpoint) $Y, X \subseteq Y$.
- greatest fixpoint, $\nu\tau$, if X is a fixpoint (post-fixpoint) and for every fixpoint (post-fixpoint) $Y, X \supseteq Y$.

Let us remark that if τ have a least $\mu\tau$ or greatest $\nu\tau$ fixpoints, they are unique.

Theorem 8 (Knaster-Tarski Theorem)

Let $\tau\colon 2^S \to 2^S$ be a monotone operator.

1. $\mu\tau = \bigcap\{X \subseteq S\colon \tau(X) = X\} = \bigcap\{X \subseteq S\colon \tau(X) \subseteq X\}$;
2. $\nu\tau = \bigcup\{X \subseteq S\colon \tau(X) = X\} = \bigcup\{X \subseteq S\colon \tau(X) \supseteq S\}$;
3. $\mu\tau = \bigcup_{\alpha \leq |S|} \tau^{\alpha}(\emptyset)$;
4. $\nu\tau = \bigcap_{\alpha \leq |S|} \tau_{\alpha}(S)$.

Definition 12

Let $|S|$ be the cardinality of S. Let α be an ordinal taking all values not greater than $|S|$.

τ^{α} is defined as follows:

- $\tau^0(X) = X,\ \tau^{\beta+1}(X) = \tau(\tau^{\beta}(X))$, and

- $\tau^\gamma(X) = \bigcup_{\beta<\gamma} \tau^\beta(X)$ for limit ordinals γ.

The meaning of τ_α is defined as follows:

- $\tau_0(X) = X, \tau_{\beta+1}(X) = \tau(\tau_\beta(X))$, and
- $\tau_\gamma(X) = \bigcap_{\beta<\gamma} \tau_\beta(X)$ for limit ordinals γ.

Models $\mathfrak{M}$ for μ-calculus are labeled transition systems $\langle S, R, L\rangle$, where S is non-empty set (of states) and $R(\subseteq S \times S)$ is such that $(s, s_1) \in R$ means that s_1 is an immediate successor of s, L is labelling function. For given L with every formula ϕ its extensional $\|\phi\|$ can be associated, i.e. the set of states where ϕ is true. If p is an atomic proposition occurring in ϕ, then ϕ can be regarded as an operator $\lambda p.\phi(p) : 2^S \to 2^S$, defined by $\lambda p.\phi(p)(\|p\|) = \|\phi\|$.

Definition 13

A formula ϕ is positive in the propositional variable z iff every occurrence of z in ϕ is positive, i.e. in the scope of even number of negations.

Theorem 9

If ϕ is positive in z, then $\lambda z.\phi(z)$ is monotone.

By the theorem 9 with every formula $\phi(z)$ monotone in z formulas expressing the least fixpoint $\mu z.\phi(z)$ and the greatest fixpoint $\nu z.\phi(z)$ of the $\lambda z.\phi(z)$ can be associated.

The basic language of μ-calculus extends the language of propositional calculus with a countable set of propositional variables $PV = \{z_0, z_1, \ldots\}$ $(AP \cap PV = \emptyset)$, operator $\bigcirc$ and operator of the least fixpoint μ. $\bigcirc\phi$ is true at s means that ϕ is true at all s_1 such that $(s, s_1) \in R$.

Each fixpoint formula such as $\mu z.\phi$ should be positive in the propositional variable z meaning z occurs under an even number of negations

The greatest fixpoint operator ν is dual to μ and could be defined:

$$\nu z.\phi ::= \neg\mu z.\neg\phi[\neg z/z].$$

Let us remark that the formula $\nu z.\phi$ is allowed only if propositional variable z occurs positively in ϕ.

A propositional variable z is bound in a formula ϕ if it occurs in $\mu z.\psi$, a subformula of ϕ. An occurrence of a variable is free, if it is not bound. A formula is a sentence if no variable is free in it.

Each (closed) formula ϕ may be identified with the set $\|\phi\|$ of states of S where it is true. *False* corresponds to the empty set, *true* corresponds to S. Implication $\phi \Rightarrow \psi$ corresponds to simple set-theoretic containment $\|\phi\| \subseteq \|\psi\|$.

$\|\phi\|_V$, the extensional of a formula ϕ relative to the valuation $V\colon PV \to 2^S$, is defined inductively:

- $\|\phi\|_V = \{s\colon \phi \in L(s)\}$ for $\phi \in AP$;
- $\|\neg\phi\|_V = S \setminus \|\phi\|_V$;
- $\|\phi \Rightarrow \psi\|_V = (S \setminus \|\phi\|_V) \cup \|\psi\|_V$;
- $\|X\phi\|_V = \{s\colon R(s) \subseteq \|\phi\|_V\}$;
- $\|\mu z.\phi\|_V = \bigcap\{X \subseteq S\colon \|\phi\|_{V[z::=X]} \subseteq X\}$,
  where $V[z ::= X]$ is the valuation obtained from V be redefining to take a value X at z.

The basic semantic notion of μ-calculus is truth of a formula at a state of a model $\mathfrak{M}$ relative to valuation V:

$$\mathfrak{M}, V, s \models \phi.$$

Note the following facts (Goranko 2000, p. 65):

- the truth definition coincides with the usual one for the formulas with no propositional variables and μ-operators;
- $\|\phi\|_V$, the extensional of a formula ϕ, hence its truth in a model, only depends on the valuation of the free occurrences of propositional variables. In particular, the extensional of a sentence does not depend on the valuation.

The language LTL is definable in the μ-calculus, i.e. LTL can be regarded as a fragment of μ-calculus. To embed CTL in μ-calculus as a primitive is taken $\boldsymbol{A}\bigcirc$. CTL^* is also embeddable, but the translation is far more complicated (Emerson & Lei 1986a, Dam 1994a, Goranko 2000, p. 69).

The μ-calculus has been axiomatized by Kozen (1983).

Axiom 1

$\phi(\mu z.\phi(z)) \Rightarrow \mu z.\phi(z)$

Park's rule:

$$\frac{\phi(\theta/z) \Rightarrow \theta}{\mu z.\phi(z) \Rightarrow \theta}.$$

The rule says that $\mu z.\phi(z)$ is a least pre-fixpoint.

The completeness was proved by Walukiewicz (1995, 1996).

4. Methods of model checking

The fundamental accomplishment of model checking is enabling broad scale formal verification (Emerson 2008, p. 40). Model checking is a widely used formal method of determining whether or not a given model satisfies properties, and for producing counterexamples if the model does not satisfy properties. Properties are given in some form of temporal logic, either LTL or CTL and CTL^*. The properties are expressed by modeling languages, including (pseudo) programming languages such as PROMELA (Holzmann 2003) or SMV (McMillan 1993), Petri nets (Girault & Valk 2003) or LOTOS (Bolognesi & Brinksma 1987).

Model checker is an algorithm to perform verification tasks which exploits various optimization strategies to find a counterexample for violated specifications. Fixed point algorithms traverse state space of systems and compute set of states related to the property.

The model checking problem can be stated as follows:

> Given a model $\mathfrak{M}$ and a logic formula ϕ, determine the set of (initial) states of $\mathfrak{M}$ that satisfy ϕ.

Generally, we say that the model $\mathfrak{M}$ satisfies the specification ϕ if all of the (initial) states of $\mathfrak{M}$ satisfy ϕ.

Emerson about his idea of model checking writes (2008, p. 9):

> ... *given any finite* **model** M *and* CTL *specification* f *one can algorithmically* **check** *that* M *is a genuine model of* f *by evaluating (verifying) the basic temporal modalities over* M *based on the Tarski-Knaster theorem.* This was the second key ingredient of model checking. Composite temporal formulae comprised of nested subformulae and boolean combinations of subformulae could be verified by recursive descent. Thus, fixpoint characterizations, the Tarski-Knaster theorem, and recursion yielded *model checking.*

It is worth to mention that software is usually more difficult to verify than hardware. It typically has less regular organization. It may involve significant use of recursion, and complex, dynamic data structures on the heap. It can also be extremely large.

Model checking is one of the most powerful "test acceleration technologies" that has been invented. To improve and extend the practical applicability of model checking method many advanced techniques are elaborated. Decision procedures, program analysis and type systems, and a shift of focus to partial specifications common to several systems (e.g. memory safety and race freedom) have resulted in several practical verification methods. Several interesting methodologies have been explored to avoid the state space explosion problem.

A brief survey of approaches to circumvent the state-explosion problem will be presented below.

4.1. Binary decision diagrams

In the early nineties of the last century it was observed (Bryant 1986) that finite sets can be efficiently represented by means of BDDs (**B**inary **D**ecision **D**iagrams). BDDs are a canonical normal form for propositional logic formulas. Any boolean formula may be represented as a BDD. Any assignment of truth values to the propositional letters of the formula corresponds to a path down the tree from the root node to a terminal node, which is labeled either true or false. The value of this label determines the value of the formula for this assignment of propositional letters.

Bryant popularized BDDs and developed a set of efficient algorithms for manipulating the data structure introduced in the work of (Lee 1959, Akers 1978) by placing ordering on them. An **O**rdered **B**inary **D**ecision **D**iagram OBDD (ROBDD often called BDD for short) is a BDD which has a total ordering applied to the variables labeling the vertices of the diagram. The size of the OBDD can vary greatly, depending on the ordering used. Heuristics have been developed to find efficient orderings for a given formula (when such an ordering exists). However, finding the optimal ordering is NP-complete (Bollig & Wegener 1996).

The use of BDDs was made popular by the work of Ken McMillan (1993).

BDDs are data structures used for symbolic representation of the program's states and state transitions. BDD is obtained from a binary decision tree by merging isomorphic subtrees and identical terminals. Any set of states can be encoded as a BDD. If S is a set of states encoded as a set of Boolean tuples (on a set X), then for any fixed ordering of the elements of X, there is a unique BDD representing S (Bryant 1992). A BDD is essentially an acyclic deterministic finite state automaton.

The development of BDDs was a cornerstone for symbolic model checking procedures based on fixpoint computations (Coudert, Berthet & Madre 1990).[49] The method was developed in (Burch, Clarke, McMillan, Dill & Hwang 1990, Burch, Clarke, McMillan, Dill & Hwang 1992, McMillan 1993).

[49] See (Clarke et al. 1999, Schneider 2003) for more details.

BDDs tend to blow up in size for large systems. Conventional BDDs have topped out for systems with a few hundred state variables. BDD based algorithms are very sensitive to the variable ordering.

4.2. The Boolean satisfiability

The Boolean **SAT**isfiability (SAT) problem is the problem of finding an assignment to the set of propositional letters such that a boolean formula will have the value 'true' under this assignment. In other words, SAT problem posed on a formula is to determine whether there exists a variable assignment under which the formula evaluates to true.[50] The SAT problem is known to be NP-Complete. However, over the years there has been tremendous progress in SAT solvers technology.[51] Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland ($DPLL$) algorithm (Davis, Logemann & Loveland 1962, Davis & Putnam 1960). Improvements in SAT solver technology have led to the development of several powerful SAT solvers (Prasad et at. 2005). If the formula is satisfiable, a counterexample can be extracted from the output of the SAT procedure. If the formula is not satisfiable, the system and its specification can be unwound more to determine if a longer counterexample exists.

SAT-based bounded model checking is an alternative to the BDD approach (Biere, Cimatti, Clarke & Zhu 1999). This approach can accommodate larger systems than the BDD approach. The efficiency of SAT procedures has made it possible to handle ICT systems much larger than any BDD-based model checker is able to do.

BDD-based symbolic verification algorithms are typically limited by memory resources, while SAT-based verification algorithms are limited by time resources (Grumberg, Heyman, Ifergan & Schuster 2005). It is widely acknowledged that the strength of SAT-based algorithms lies primarily in falsification, while BDD-based model checking continues to be the *de facto*

[50] The annual award, which recognizes a specific fundamental contribution or series of outstanding contributions to the CAV field was established in 2008 by the steering committee of the International Conference on Computer Aided Verification. In 2009 the award was given to Conor F. Madigan, Kateeva, Inc.; Sharad Malik, Princeton University; Joao P. Marques-Silva, University College Dublin, Ireland; Matthew W. Moskewicz, University of California, Berkeley; Karem A. Sakallah, University of Michigan; Lintao Zhang, Microsoft Research and Ying Zhao, Wuxi Capital Group. The award includes a $10,000 prize and was presented with the citation:

> For fundamental contributions to the development of high-performance Boolean satisfiability solvers.

[51] The progress is summarized in a survey (Zhang & Malik 2002, Prasad, Biere & Gupta 2005).

standard for verifying properties (Prasad et al. 2005, Biere, Cimatti, Clarke, Strichman & Zhu 2003).

However the SAT approach only explores for "close" errors at depth bounded by k where typically k ranges from a few tens to hundreds of steps. In general, it cannot find "deep" errors and provide verification of correctness.

4.3. Partial order reduction

Concurrency is a major practical obstacle to model checking: the interleaving of concurrent threads causes an exponential explosion, and if threads can be dynamically created, the number of control states is unbounded. **P**artial **O**rder **R**eduction (*PRO*, for short) helps improve the performance of a model-checker by eliminating the interleaving of independent actions by reducing the number of interleavings of independent concurrent transitions.

The earliest identifiable connection between parallel processing and the central idea behind *POR* (long before that term was coined) appears in Lipton's work of (1975) on optimizing `P/V` programs for the purpose of efficient reasoning. In that work, Lipton identifies left and right movers – actions that can be postponed without affecting the correctness of reasoning. Additionally, in the parallel compilation literature (e.g., (Rinard & Diniz 1997)), it has been observed that by identifying computations that "commute," one can schedule these computations for parallel evaluation without interference.

The explosion of states and transitions in a model results from the interleaving of actions of distinct processes in all possible orders. In general, the consideration of all such interleavings is crucial – bugs in concurrent systems often correspond to the unexpected ordering of actions. However, if a set of transitions is entirely independent and invisible with respect to the property being verified,[52] the order in which transitions are executed does not affect the overall behavior of the system. *POR* (Emerson, Jha & Peled 1997, Godefroid 1996, Peled 1996a) exploits this fact, and considers only one representative ordering for any set of concurrently enabled, independent, invisible transitions.

POR methods rely on determining a suitable subset of transitions to be considered at every state. As a result, rather than exploring a structure $\mathfrak{M}$, an equivalent (usually smaller) structure $\mathfrak{M}'$ is explored, with fewer transitions and states.

[52] A transition is invisible with respect to a property ϕ if the truth of ϕ is unaffected by the transition.

The particular subset (and correspondingly, equivalence relation) depends upon the strategy being used. A common strategy, for example, is the ample sets method (Peled 1996b). This is the method chosen for the *POR* implementation in SPIN (Holzmann 2003, Holzmann & Peled 1994). To describe this method let us suppose that a property ϕ is to be verified. For any state s reached along a search path, rather than considering all of the transitions enabled at s [$enabled(s)$], an ample set [$ample(s)$] of transitions is chosen in such a way to ensure that

- any transition $t \in enabled(s)$ which is not in $ample(s)$ is independent of all transitions in $ample(s)$. That is, the execution of t does not affect the enabledness of any of the transitionsnin $ample(s)$, and vice versa.
- All transitions in $ample(s)$ are invisible.
- If $a \in ample(s)$, then the state resulting from taking transition a from s has not been reached along the current search path.

In the discussed case, the equivalence relation is a *trace* equivalence. Two transition sequences are said to be trace equivalent if one can be obtained from the other by repeatedly commuting the order of adjacent, independent transitions. Using the ample sets method, every transition sequence in the original structure $\mathfrak{M}$ is trace equivalent to a transition sequence in the reduced structure $\mathfrak{M}'$. It follows that for any stuttering-closed (Peled 1996b) LTL formula ϕ: $\mathfrak{M} \models \phi$ iff $\mathfrak{M}' \models \phi$.

Other strategies for determining suitable subsets of transitions include the stubborn sets method (Valmari 1992) or the sleep sets and persistent sets method (Godefroid 1996b) which is implemented in VERISOFT (Godefroid 1997).

For some systems wherein all actions are independent, *POR* cannot offer any improvement in verification space or time. In many realistic cases, however, *POR* can be extremely effective. For example, for some systems the growth of the state-space as the number of processes increases is reduced from exponential to polynomial when *POR* methods are used. In others, the global state-space may increase with the growth of a parameter, whereas the size of the reduced state-space remains unchanged (Godefroid 1996a).

4.4. Abstraction

Abstraction is certainly the most important technique for handling the state explosion problem. It plays an important role in the verification of infinite-state systems.

The abstraction is a method of suppressing details of a given model $\mathfrak{M}$ to obtain a smaller model $\overline{\mathfrak{M}}$ that is equivalent to $\mathfrak{M}$ for purposes of verification. $\mathfrak{M}$ and $\overline{\mathfrak{M}}$ can differ considerably.

Abstractions should preserve the (non-)validity of the properties that need to be checked.

The abstraction is exact if and only if the correctness of $\overline{\mathfrak{M}}$ is equivalent to correctness of $\mathfrak{M}$. An abstraction is conservative if correctness of $\overline{\mathfrak{M}}$ implies correctness of $\mathfrak{M}$.

For the purpose of abstraction bisimulation and simulation relations can be exploiting. In the case of bisimulation (Park 1981) $\mathfrak{M}$ and $\overline{\mathfrak{M}}$ cannot be distinguished by any "reasonable" temporal logic (Emerson 2008, p. 38). If systems are bisimilar, the abstraction is exact. Using bisimulation as a pre-processing phase to model checking does reduce the resource requirements of model checking. Bisimulation may also be applied in symbolic model checking, because bisimulation can be computed as a fixpoint of a simple boolean expression. An abstraction obtained from $\mathfrak{M}$ by partitioning and clustering states in the natural way will be conservative. A simulation from $\mathfrak{M}$ to $\overline{\mathfrak{M}}$ yields a conservative abstraction.

If $\overline{\mathfrak{M}}$ is incorrect the abstraction may be too coarse. Repeatedly refining it as needed and as resources permit typically leads to determination of correctness vs. incorrectness for $\mathfrak{M}$ (Kurshan 1995).

If state spaces are too large abstractions can be made with respect to a single property. In such a case, different abstractions have to be considered for any particular property.

If only a part of state space is explored, the verification result is not precise and correctness of a system is only probable.[53]

The symmetry abstraction is based on the fact that subcomponents of $\mathfrak{M}$ may be symmetrical and for this reasons as redundant can be identifying.

A key technique for the verification of programs is **C**ounter-**E**xample **G**uided **A**bstraction **R**efinement (*CEGAR*).[54] *CEGAR* begins checking with a coarse (imprecise) abstraction and iteratively refines it. When a counter-example is found, the question raises if the violation is genuine or the result of an incomplete abstraction. If the violation is not feasible, the proof of infeasibility is used to refine the abstraction and checking begins again (Clarke, Grumberg, Jha, Lu & Veith 2000).

Abstraction is employed already at the design phase: a representative system is abstracted out which captures the higher level behavior of the system. Abstractions are often applied as a pre-processing phase in model checking.

---

[53] Probabilistic verification, a model checking technique that is based on a partial state-space exploration should not be confused with verifying probabilistic systems.

[54] A prominent safety-checking tool based on *CEGAR* is Blast, see p. 87.

Abstraction is useful for reasoning about concurrent systems that contain data paths. The specification of systems that include data paths usually involves fairly simple relationship among the data values of the system.

The reduction due to abstraction can result in exponentially smaller number of states. For example (Emerson 2008, p. 38), a resource controller with 150 homogeneous processes of size about 10^{47} states can be model checked over the abstract $\overline{M}$ in a few tens of minutes (Emerson & Sistla 1997).

Abstraction is typically a manual process, often requiring considerable creativity. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions.

4.5. Lazy abstraction

The common methods of model checking are either precise or they are scalable, depending on whether they track the values of variables or only a fixed small set of data flow facts (e.g. types), and are usually insufficient for precisely verifying large programs. **L**azy **A**bstraction (*LA*) achieves both precision and scalability by localizing the use of precise information.

The model checking method is based on the abstract-check-refine paradigm:[55]. *LA* tightly integrates and optimizes the three steps of the loop: abstract-check-refine.

The lazy abstraction is built on-the-fly and is parsimonious: different parts of the state space use different abstractions, namely they are only as precise as is required to verify that part of the system. The refinement is on-demand, i.e., the model used in the previous iteration may be re-used, and it is local, i.e., only the abstract states which comprises a spurious counterexample are refined, so that only the small part of the abstraction through which the abstract counterexample path passes is refined, and re-analyzed. Model checking is not repeated for those parts of the system that are known to be error-free.

Instead of iteratively refining an abstraction, single abstract model is built and refined on demand, so that different parts of the model may exhibit different degrees of precision, namely just enough to verify the desired property (Henzinger, Jhala, Majumdar & Sutre 2002, McMillan 2002).

The lazy abstraction methodology reduces the space and time requirements considerably.

The abstract-check-refine loop is not guaranteed to terminate in general. *LA* is implemented in model checker Blast.

[55] Abstract-check-refine approach follows three steps: build an abstract model, then check the desired property, and if the check fails, refine the model and start over.

4.6. Local and global model checking

The model-checking algorithms can be classified into (Petcu 2003, Schneider 2003):
- global
- local.

The global algorithm will compute all states that satisfy a given formula. This method is based on generating the whole state-space whereas the local algorithm will only check a formula for one state (typically the initial system state). Local model checking partially constructs the state-space one step at a time until a solution is reached.

Global and local model checking procedures follow radically different paradigms (Schuele & Schneider 2004): while global model checking aims at computing all states where a given specification holds by means of fixpoint iterations, in contrast, local model checking directly answers the question whether a given set of states satisfies the specification by means of deduction and induction. Local model checking algorithms based on logics like the μ-calculus and use a tableau-based procedure. For the verification of finite state systems, this may result in different runtimes. Nevertheless, both algorithms run in time linear in the size of the transition system and the length of the formula. For the verification of infinite state systems, however, the differences are far more important. Since most problems are undecidable for such systems, it may be the case that one of the procedures does not terminate.

In global model checking, the syntax tree of a specification is traversed in a bottom-up manner, whereas in local model checking, a specification is evaluated top-down. A major advantage of the latter approach is that subformulas can be checked by need, i.e., in the spirit of lazy evaluation.

Global procedure requires the a priori generation of all system states, and storing these states may consume large quantities of memory. In the case of local procedure, there is no need for the state space to be known in advance, it can be constructed incrementally, as the model checking computation proceeds. The size of the state space, actually has to be explored depends on how much of it turns out to be relevant to establishing satisfaction of the formula to be verified. The main drawback of the local procedure is that often the entire state space is generated (like checking that a property holds globally). In the worst case, algorithms will exhibit the same storage requirements as those of global algorithms, in practice they generally use less.

4.7. Checking of finite and infinite states systems

Model checking (Clarke et al. 1999) is a technique used for verifying the correctness of finite state systems and usually restricted to propositional logics. The restriction to propositional logic is not so important from the point of view of practice as it may seem at first glance (Kroger & Merz 2008, p. 376). In the case of finite domain, first-order logic formulas are reducible to finite conjunction – if the formula says about all the elements of the domain – or finite disjunction – if the formula says about some elements of the domain – of propositions each of which is about exactly one element of the domain. Thus, systems properties of finite state transition system may be encoded in propositional temporal logic.

In many areas one has to deal with infinite state spaces (Schuele & Schneider 2004, Esparza 2003). There are different "sources of infinity" and at least five of them could be pointed out:

- Data manipulation: *data structures over infinite domains – natural numbers, integers, etc.*
- Control structures: *unbounded call stacks or dynamic creation of processes.*
- Asynchronous communication: *unbounded queues for process communication.*
- Parametrization: *infinite families of distributed systems.*
- Real-time constraints: *timing constraints based on real-valued clocks.*

For finite state systems, termination of the model checking algorithms is guaranteed. Unfortunately, this does not hold for systems with infinite state spaces. Consequently, techniques are required to achieve termination, e.g., by using additional information such as invariants and well-founded orderings. As another problem, propositional logic and hence BDDs are naturally limited to the representation of finite sets and do not allow us to reason about systems with infinitely many states. Hence, we need more powerful representations that enable us to deal with infinite sets. To this end, e.g., Presburger arithmetic (Presburger 1929, Enderton 1972) has been proposed which can be translated to finite automata to obtain efficient tools (Burkart, Caucal, Moller & Steffen 2001).

Initial work on the verification of infinite state systems was done by Bradfield & Stirling in (1991), where the authors established proof rules for a tableau calculus. Some years later, Bultan, Gerber, and Pugh (1997, 1999) proposed a method for global model checking of infinite state systems by means of Presburger arithmetic.

4.8. Explicit and symbolic checking

Originally model checking used an explicit representation of states. State spaces of systems increase exponentially with the number of variables and concurrent components. Hence, a naive implementation of the explicit state enumeration is infeasible. Instead of explicite state enumeration the method of symbolic representation may be applied. Thus model checking algorithms can be classified into (Clarke & Veith 2003, p. 211):

- *explicit*,
- *symbolic*.

Symbolic representations are mathematical objects with semantics corresponding to sets of states. The algorithm is symbolic in the sense that it manipulates sets of states, instead of states. Sets of states are represented implicitly by means of predicates. Symbolic algorithms employ data structures such as BDD to describe sets of states and transitions. Model checking is performed directly on the BDD representations. BDD made symbolic model checking popular (Burch et al. 1992).

Symbolic model checking is first introduced with the work of McMillan (Burch et al. 1990, McMillan 1993), in which BDD data structure is used to implicitly (as opposed to explicitly) represent the set of states and the transition relation between states (Bryant 1986). Since then, symbolic model checking is used to refer to a technique used in model checking to implicitly represent and manipulate the states and the transition relation of a system. A particular symbolic approach (namely BDD-based encoding) has proved especially successful for the verification of CTL properties for very large systems (McMillan 1993). Although the techniques of model checking were oriented toward the verification of hardware circuits (Bryant & Chen 1995, Clarke, Grumberg, Hiraishi, Jha, Long, McMillan & Ness 1993, Burch, Clarke, Long, MacMillan & Dill 1994), they have been extended and applied to probabilistic systems and timed systems, for which corresponding symbolic data structures have also been developed.

Symbolic algorithm avoids ever building the graph, where the Kripke structure is not described in extension (by a description having size $|S|$, as in enumerative methods); instead it represents the structure implicitly via more succinct data structures, most often some kind of restricted logical formula for which efficient constraint-solving techniques apply. Symbolic algorithms verify systems that defy enumerative methods (Burch et al. 1992, McMillan 1993). There are also systems on which they do not perform better than the naive non-symbolic approach (an approach that can be defined as "build the structure enumeratively and then use the best model checking algorithm at hand").

Explicit algorithms are algorithms which work directly on the Kripke structure, and construct necessary parts of the Kripke structure on-the-fly, using methods such as partial order reduction to prune the search space. Techniques adopted to reduce the state space may cause some loss of information. This may be unacceptable, for example when safety properties are verified. One of the most successful approaches that can allow a larger state space to be explored is symbolic model checking.

Fixpoints corresponding to truth sets of checked temporal formulas can be computed by aggregating sets of states iteratively. Properties which require more than one fixpoint computation can be computed recursively starting from the inner fixpoints and propagating the partial results to the outer fixpoints.

Symbolic model checking is one of the most successful approaches to reducing the space requirements that have been investigated (Burch et al. 1992). By introducing symbolic representations for sets of states and transition relations and using of a symbolic model checking, systems with very large state spaces (10^{100} or more states) can be verified (Burch et al. 1990, Coudert et al. 1990). Further, the time and space requirements with these techniques may in practice be polynomial in the number of components of the system. In many cases, symbolic algorithms achieve great reductions in the size of the data structures, and thus help to alleviate the state explosion problem. Unfortunately, the symbolic procedures still have limits, and many realistic problems are not tractable due to their size.

Symbolic Model Checking (Burch et al. 1990) uses Boolean formulas to represent sets of states and transition relations. Traditionally, symbolic model checking has become identified with BDD (Bryant 1986), a canonical form of representing Boolean formulas. Some other representations, like **C**onjunctive **N**ormal **F**orm (CNF) using SAT and polynomial algebra have been demonstrated to be quite powerful in practice. Most Symbolic Model Checkers exploit Tarski's Lemma that every monotonic functional on a complete lattice has a fixpoint.

Symbolic algorithms have often been proved useful in practice in tackling the state-explosion problem (Holzmann 1991, McMillan 1993, Bouajjani et al. 1997).

Traditionally, explicit methods are typical of LTL, while symbolic methods have been used primarily for CTL model checking. At the beginning, the techniques of symbolic model checking was oriented at verification of hardware systems (Mishra & Clarke 1985, Bryant & Chen 1995, Clarke, Grumberg, Hiraishi, Jha, Long, McMillan & Ness 1993, Burch et al. 1994). In order to improve the efficiency a hybrid approach it combines features of both the symbolic and explicit implementations.

4.9. Bounded model checking

Bounded **M**odel **C**hecking (BMC) is an attractive alternative to symbolic model checking, since it often allows a more efficient verification (Schuele & Schneider 2007). The idea of BMC is to reduce the model checking problem to a satisfiability problem of the underlying base logic, so that sophisticated decision procedures can be utilized to check the resulting formula (Biere, Cimatti, Clarke, Fujita & Zhu 1999, Biere, Cimatti, Clarke & Zhu 1999). This technique is based on SAT-method. It is a method without BDDs which uses much less space than BDD-based approaches.

The basic idea is to consider counter examples of a particular length and generate a propositional formula that is satisfiable iff such a counter example exists. In BMC, if the checked formula is satisfiable, a counterexample can be extracted from the output of the SAT procedure. If for a bounded length k the formula is not satisfiable, it can be unwound more to determine if a longer counterexample exists, k is increased. The process can be repeated with larger and larger values of k until all possible violations have been ruled out. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., k is sufficiently large to ensure that no counterexample exists) or when the SAT procedure exceeds its time or memory bounds.

The bounded model checking for LTL can be reduced to propositional satisfiability in polynomial time (Biere, Cimatti, Clarke & Zhu 1999).

The disadvantage of BMC is that it is an incomplete SAT-based formal verification method. BMC is effective for showing the presence of errors. Thus, it is typically only applicable for refutation; the completeness threshold is too large for most practical instances. It is not at all effective for showing that a specification is true unless the diameter of the state space is known.

4.10. LTL and CTL model checking

In temporal logic, model checking can be divided into:

- LTL model checking
  – if verified system is specified using linear time model,
- CTL model checking
  – if verified system is specified with branching time model.

The model checking problem for LTL can be restated as:

given $\mathfrak{M}$ and ϕ, does there exists a path of $\mathfrak{M}$ that does not satisfy ϕ?

One approach to LTL model checking is the tableau approach described in e.g. (Müller-Olm, Schmidt & Steffen 1999). The other approach, the automata-theoretic approach is more efficient (Lichtenstein & Pnueli 1985) and (Vardi & Wolper 1986a).

The model checking algorithm for CTL (Clarke et al. 1986, Quielle & Sifakis 1982) works by successively marking the states which satisfy subformulas of the formula to be checked. The particular form of algorithm used depends on the formula.

CTL^* model checking was first introduced in (Clarke et al. 1986). A method for checking CTL^* properties (Emerson & Lei 1987) involves the use of an LTL model checker on the subformulas of the property to be checked.

Most modal checkers are used to verify either CTL or LTL properties, but not both.

4.11. Büchi automaton

Automata theory plays a central role in formal methods. These alternative techniques for verifying ICT systems in some cases may be used as they can be more expressive or succinct than temporal logic. In automata-theoretic approach, both the system and the specification are described in automata. And questions about systems and their specifications can be reduced to questions about emptiness and containment of automata. Temporal logic and automata provide different means to describe temporal structures. TL formulas are more "declarative", whereas automata are more "operational" in nature. For the purpose of model checking a kind of automaton that is suited for accepting ω-regular languages is needed. Automata-theoretic approach includes: (finite state) automata (on infinite strings) which accept infinite inputs by infinitely often entering a designated set of automaton states. The use of ω-automata for automated verification was first proposed by Vardi and Wolper (1986b, 1986a). ω-automata are finite automata operating on infinite words. Nondeterministic Büchi automata (NBAs) are the simplest ω-automata. They look exactly like finite automata. However, they operate on infinite words, and they have a different acceptance condition. Büchi automata are FSAs defined over infinite runs. The syntax of non-deterministic finite automata, NBAs is exactly the same as for NFAs. The automata differ in their semantics: the accepted language of NBA is an ω-language, whereas the accepted language of an NFA is a language of finite words. The intuitive meaning of the acceptance criterion is that the set of accepted states has to be visited infinitely often. Thus, the accepted language consists of all infinite words that have a run in which some accept state is visited infinitely often. ω-automata is a formalism widely used by researchers and engineers alike.

Definition 14

Nondeterministic Büchi Automaton (Buchi 1960) $\mathcal{A}$ is a tuple:[56]

$$\mathcal{A} = \langle S, \Sigma, \delta, I, F\rangle,$$

where

- S is a finite set of states,
- Σ is an alphabet,
- $\delta\colon S \times \Sigma \to 2^S$ is a transition function,
- $I \subseteq S$ is a set of initial states, and
- $F \subseteq S$ is a set of accept (or: final) states, called the acceptance set.

A run for $\delta = A_0A_1A_2\ldots \in \Sigma^\omega$ denotes an infinite sequence $s_0s_1s_2\ldots$ of states in $\mathcal{A}$ such that $s_0 \in I$ and $s_i \xrightarrow{A_i} s_{i+1}$ for $i \leq 0$. Run $s_0s_1s_2\ldots$ is accepting if $s_i \in F$ for infinitely many indices $i \in \mathbb{N}$. The accepted language of $\mathcal{A}$ is

$$\mathcal{L}_\omega(\mathcal{A}) = \{\sigma \in \Sigma^\omega |\text{ there exists an accepting run for } \sigma \text{ in } \mathcal{A}\}.$$

The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is defined as the number of states and transitions in $\mathcal{A}$.

As for an NFA, the transition function δ is identified with the induced transition relation $\rightarrow \subseteq S \times \Sigma \times S$ which is given by

$$s \xrightarrow{A} p \text{ if and only if } p \in \delta(s, A).$$

Since the state space S of an NBA $\mathcal{A}$ is finite, each run for an infinite word $\sigma \in \Sigma^\omega$ is infinite, and hence visits some state $s \in S$ infinitely often. Acceptance of a run depends on whether or not the set of all states that appear infinitely often in the given run contains an accept state. The definition of an NBA allows for the special case where $F = \emptyset$, which means that there are no accept states. Clearly, in this case, no run is accepting. Thus $\mathcal{L}_\omega(\mathcal{A}) = \emptyset$ if $F = \emptyset$. There are also no accepting runs whenever, $I = \emptyset$ as in this case, no word has a run.

[56] See eg. (Baier & Katoen 2008, p. 174).

Definition 15

Let $\mathcal{A} = \langle S, \Sigma, \delta, I, F \rangle$, be an NBA. $\mathcal{A}$ is a Deterministic Büchi Automaton (DBA), if $|I| \leq 1$ and $|\delta(s, A)| \leq 1$, for all $s \in S$ and $A \in \Sigma$. $\mathcal{A}$ is total if $|I| = 1$ and $|\delta(s, A)| = 1$, for all $s \in S$ and $A \in \Sigma$.

In the case of finite automata, each non-deterministic automaton can be converted into a language-equivalent deterministic automaton. This is not the case with Büchi automata. It means that NBAs are strictly more expressive than DBAs.

Definition 16

A Generalized Nondeterministic Büchi Automata $GNBA$ is a tuple:

$$\mathcal{G} = \langle S, \Sigma, \delta, I, \mathcal{F} \rangle,$$

where S, Σ, δ, I are defined as for an NBA and $\mathcal{F}$ is a (possibly empty) subset of 2^S.

The elements $F \in \mathcal{F}$ are called acceptance sets. Runs in a $GNBA$ are defined as for an NBA. That is, a run in $\mathcal{G}$ for the infinite word $A_0 A_1 \ldots \in \Sigma^\omega$ is an infinite state sequence $s_0 s_1 s_2 \ldots \in S^\omega$ such that $s_0 \in I$ and $s_{i+1} \in \delta(s_i, A_i)$, for all $i \leq 0$.

Every LTL formula can be represented as a Büchi automaton (Wolper, Vardi & Sistla 1983, Vardi & Wolper 1994, Holzmann 2004). Büchi automaton applied in LTL has been generalized for branching temporal logic (Vardi & Wolper 1986). The idea is the same as in LTL. Three automata are distinguished: local (that in essence does not differ from the automaton for LTL); existential eventuality (that correspond to eventuality expressed by the formula $\boldsymbol{E}\phi U \psi$) and universal eventuality (that correspond to the eventuality expressed by the formula $\boldsymbol{A}\phi U \psi$). These three automata are reduced to one that accepts the intersection of their languages (Wolper 1995).

4.12. On-the-fly model checking

It is not always necessary to build the entire state-space in order to determine whether or not a system satisfies a given property. The algorithm is *on-the-fly* in the sense that the state-space is generated dynamically and only the minimal amount of information required by the verification procedure is stored in memory (Bouajjani et al. 1997). It means that states are computed and stored on demand, while in other algorithms, the whole state-space has to be generated *a priori*.

If the property to be checked is false, only part of the state-space needs to be constructed; it can be stopped as soon as an error state or violating cycle is found. This means that although debugging can be performed relatively easily, property verification very quickly becomes prohibitive.

On-the-fly methods are most suitable for model checking algorithms based on a depth-first traversal of the state-space (i.e., explicit state methods) and have been developed to check specifications in LTL, CTL and CTL^* (Vardi & Wolper 1986a, Vergauwen & Lewi 1993, Bhat, Cleaveland & Grumbero 1995).

On-the-fly method is applied for explicit model checking. Nevertheless, there are some approaches for combining on-the-fly techniques with symbolic model checking exist (Ben-David & Heyman 2000). Usually they are restricted to checking safety properties.

On-the-fly algorithms have often been proved useful in practice in tackling the state-explosion problem (Holzmann 1991, McMillan 1993, Bouajjani et al. 1997).

4.13. Symmetry reduction

The use of symmetry reduction to increase the efficiency of model checking has inspired a wealth of activity in the area of model checking research. The earliest use of symmetry reduction in automatic verification was in the context of high-level (colored) Petri nets (Huber, Jenson, Jepson & Jenson 1985), where reduction by equivalent markings was used to construct finite reachablility trees. These ideas were later extended for deadlock detection and the checking of liveness properties in place/transition nets (Starke 1991).

Concurrent systems often contain many replicated components and as a consequence, model checking may involve making a redundant search over equivalent areas of the state-space. Inherent symmetry of the original system will be reflected in the state-space. Therefore, knowledge of the symmetry of the system can be used to avoid searching areas of the state-space which are symmetrically equivalent to areas that have been searched previously (Miller et al. 2006). Most symmetry reduction techniques exploit this type of symmetry by restricting the state-space search to equivalence class representatives, and often result in significant savings in memory and verification time (Bosnacki, Dams & Holenderski 2002, Clarke, Enders, Filkhorn & Jha 1996, Emerson & Sistla 1996, Ip & Dill 1996).

4.14. Modular verification

Modular verification is another possible way to combat the state-space explosion problem. Efforts to develop modular verification frameworks were undertaken in the mid 1980s (Pnueli 1985a). The verified system is decomposed into subproblems of manageable complexity. The decomposition reflects the modularity in the design.

In modular verification the specification includes two parts. One part describes the desired behavior of the module. The other part describes the assumed behavior of the system within which the module is interacting [Kupferman & Vardi (1995, 1996, 1997, 1998)].

Grumberg and Long (1994) have proposed a method for modular verification in assume-guarantee style, where both assumptions and guarantees are expressed in BTL. In the context of modular verification, it is advantageous to use only $\boldsymbol{A}CTL$ or $\boldsymbol{A}CTL^*$, that is universal fragments of CTL and CTL^* without $\boldsymbol{E}$, existential path quantifiers. The $\boldsymbol{A}CTL$ and $\boldsymbol{A}CTL^*$ formulas have the helpful property that once they are satisfied in a module, they are also satisfied in a system that contains this module (Shurek & Grumberg 1990).

For the LTL paradigm, in the assume-guarantee specification, both the assumption and guarantee are specified as LTL formulas. The model checking under assume-guarantee specification with respect to the LTL formula formed as assumption implies guarantee. In another approach, the assume-guarantee pair consists of a linear temporal assumption and a branching temporal guarantee.

The modular analysis is applied to such programs as, e.g, `Linux`, which contains thousands of device drivers that are developed independently by many developers. Though each individual driver source code is relatively small – ≈10k lines of code – the whole operating system contains a few million lines of code (Post & Küchlin 2006).

4.15. Model checking of timed systems

So far there are discussed systems that describe how a system may evolve from one state to another. Timing aspects are, however, not covered. There are some applications where it is desirable to consider aspects of timing behavior. Systems such as device drivers from ABS braking technology in cars to avionics control, coffee machines, communication protocols, and automatic teller machines, to mention a few, must react in time, they can be viewed as timed systems, in that their correct behavior depends crucially on their meeting various timing constraints (Laroussinie, Markey & Schnoebelen 2004) (Laroussinie, Markey & Schnoebelen 2005) (Lasota

& Walukiewicz 2005). For a train crossing it is essential that on detecting the approach of a train, the gate is closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. For a radiation machine the time period during which a cancer patient is subjected to a high dose of radiation is extremely important; a small extension of this period is dangerous and can cause the patient's death. Correctness in time-critical systems not only depends on the logical result of the computation but also on the time at which the results are produced. (Baier & Katoen 2008, Ch. 9).

For some applications discrete time domains are appropriate: one time unit corresponds to one clock pulse. The next-step operator can be used to "measure" the discrete elapse of time and temporal logics LTL and CTL are eligible to express timing constraints. A continuous time model is more adequate and more intuitive for systems in which components may proceed at distinct speeds (Baier & Katoen 2008, Ch. 9).

In order to express timing constraints, the logical formalisms have to be extended to allow expression of the ordering of states, with a notion of quantitative time. There are different formalisms for modeling timed systems, among which, timed automata[57] which were proposed by Alur and Dill (1994), are one of the most successful formalisms for the description of timed systems. Timed specifications can be based on metric temporal logic `http://www.comlab.ox.ac.uk/projects/timedsystems/`.

Formal analysis methods for timed systems are more difficult compare to untimed systems. There are three factors which affect the size of the state space. The state space under consideration grows exponentially with (Daws & Tripakis 1998):

- the number of concurrent components,
- the number of clocks

and

- the length of the clock constraints used in the model and the specification.

Unlike traditional model checking which is performed on finite state automata, timed automata have infinite state space because of the real value of the clocks. Since clocks are real-valued, the state space of timed automata is infinite. Much of the work on model checking timed automata is focused on using a finite representation for the infinite state space.

[57] Timed automata are automata extended with clocks that progress synchronously with time.

Model checking has been successfully implemented for real-time systems which are modeled as timed automata (Alur & Dill 1994) and a number of tools for automatic verification of systems have emerged:

- UPPAAL `www.docs.uu.se/docs/rtmv/uppaal/`,
- KRONOS `www.verimag.imag.fr/TEMPORISE/kronos/`,
- HYTECH `www.cad.eecs.berkeley.edu/~tah/HyTech/`.

These tools have reached a state, where they are mature enough for application on realistic case studies (Bengtsson, Griffioen, Kristoffersen, Larsen, Larsson, Pettersson & Yi 1996).

5. Model checkers

By a model checker we mean a procedure which checks if a transition system is a model for a formula expressing a certain property of this system (Clarke et al. 1986).

The late eighties and the nineties have produced a number of "checkers" verifying complex aspects of industrial designs. Since then, model checkers are common in many industrial settings where applications are safety critical or economically vital. There is a wide variety of these tools available, with a number of different capabilities suited to different kinds of problems. The variety is of great benefit to practitioners. They have to know which tools are available and which tools to chose for a particular problem.

An analysis in performing property checking is sound "if every true error is reported by the analysis". The analysis is complete "if every reported error is a true error". The third claim concerning the analysis is its usefulness: "if it finds error someone cares about".

Model checking tools typically include a modeling language for representing the program corresponding to a verified structure, a specification logic such as CTL or LTL for capturing correctness properties, a model checking algorithm that is often fixpoint based. Some of these are academic tools, others are industrial internal tools, and some are for sale by CAD vendors. Anything with a finite state structure (e.g., decision processes, reliability models, planning in AI) can be approached through model checking. Even some systems with an infinite number of states can be amenable to model checking, if there is a suitable finite representation of infinite sets of states in terms of symbolic constraints. Today, software, hardware and CAD companies employ several kinds of model checkers. In software, $Bell$ $Labs$, JPL, and $Microsoft$, government agencies such as $NASA$ in USA, in hardware and CAD, IBM, $Intel$ (to name a few) have had tremendous success using model checking for verifying switch software, flight control software, and device drivers.

Some programs are grouped as it is in the case of Model-Checking Kit `http://www.fmi.uni-stuttgart.de/szs/tools/mckit/overview.shtml`. This is a collection of programs which allow to model a finite-state system using a variety of modeling languages, and verify it using a variety of checkers, including deadlock-checkers, reachability-checkers, and model-checkers for the temporal logics CTL and LTL. The most interesting feature of the Kit is that:

> Independently of the description language chosen by the user, (almost) all checkers can be applied to the same model.

The counterexamples produced by the checker are presented to the user in terms of the description language used to model the system.

The Kit is an open system: new description languages and checkers can be added to it.

The description languages and the checkers have been provided by research groups at the *Carnegie-Mellon University*, the *University of Newcastle upon Tyne*, *Helsinki University of Technology*, *Bell Labs*, the *Brandenburg Technical University at Cottbus*, the *Technical University of Munich*, the *University of Stuttgart*, and the *Humboldt-Universität zu Berlin*.

Below, we give a few examples of model checkers. Usually their description will be taken from their website's home pages.

5.1. Explicit State-Based Model Checkers

Two of the most popular on-the-fly, explicit-state-based model checkers are Spin (**S**imple **P**romela **IN**terpreter) and Murϕ or Murphi (Dill, Drexler, Hu & Yang 1992, Dill 1996).

Other state-based verifiers include branching time temporal logic CTL Prod. It supports verification of CTL properties from the reachability graph, and on-the-fly verification of LTL-properties (Varpaaniemi, Halme, Hiekkanen & Pyssysalo 1995).

Prod is an advanced tool for efficient reachability analysis.[58] It implements different advanced reachability techniques for palliating the state explosion problem, including partial-order techniques like stubborn sets and sleep sets, and techniques exploiting symmetries. Prod is distributed by the

[58] Reachability analysis asks whether a system can evolve from legitimate initial states to unsafe states. It is thus a fundamental tool in the validation of the ICT systems.

Formal Methods Group of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology `www.fortunecity.com/banners/interstitial.html` `www.tcs.hut.fi/prod/proddescription.html`.

PEP (**P**rogramming **E**nvironment based on **P**etri nets) `http://parsys.informatik.uni-oldenburg.de/~pep/`; `http://sourceforge.net/projects/peptool` (Best &' Grahlmann 1996), in which systems are specified using Petri nets.

The on-the-fly verification of various temporal or μ-calculus properties of LOTOS (**L**anguage **O**f **T**emporal **O**rdering **S**pecification) `http://www.cs.stir.ac.uk/ kjt/research/well/well.html` specifications is achieved by translation into state-spaces using CÆSAR `http://www.inrialpes.fr/vasy/cadp.html` (Garavel & Sifakis 1990), which are then checked using the model checkers XTL (e**X**ecutable **T**emporal **L**anguage) `http://www.inrialpes.fr/vasy/cadp/man/xtl.html` (Mateescu & Garavel 1998) or EVALUATOR (Mateescu 2003), respectively.

COSPAN (**CO**ordinated **SP**ecification **AN**alysis), (Kurshan 1995, Alur & Kurshan 1995, Hardin, Harel & Kurshan 1996) is an ω-automata-based tool. The system to be verified is modeled as a collection of coordinating processes described in the selection/resolution modeling language. The verifier supports both on-the-fly enumerative search and symbolic search using BDDs. COSPAN is the core engine of the commercial verification tool from Lucent Technologies Inc.

5.1.1. SPIN

SPIN is a popular open-source software tool. It is one of the most powerful standard model checking tools. It was conceived by Gerard J. Holzmann (Bell Laboratories in Murray Hill, New York and afterwards NASA's Jet Propulsion Laboratory in Pasadena, California) in 1980 for verifying communications protocols. At the beginning of the website's home page `http://spinroot.com/spin/whatispin.html` we read:

> SPIN is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at *Bell Labs* in the original UNIX group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious *System Software Award* for 2001 by the *ACM*.

SPIN has two principle modes of operation: simulation and verification. Properties to be verified are expressed as *LTL* formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm. SPIN verifies the model and generates verification results, "true" or counterexample if the result is "false". Verification is subdivided into two aspects: safety and liveness. Some basic safety and liveness properties, such as deadlock, invalid end state and non-progress cycle are verified. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

In SPIN, specifications are described using the high-level state-based description language PROMELA (**Pro**cess/**Pro**tocol **Me**ta **La**nguage), which is loosely based on Dijkstra's guarded command language (Dijkstra 1976). The language is intended to make it easier to find good abstractions of system designs. Emphasis in this language is on the modeling of process synchronization and coordination, not on computation. PROMELA allows for the expression of non-determinism, asynchronous and synchronous communication, dynamic process creation, and mobile communications (communication channels can contain references to other communication channels).

To optimize verification runs, SPIN uses efficient partial order reduction techniques, and also employs statement merging (Holzmann 1999), a special case of partial order reduction that merges internal, invisible process statements to reduce the number of reachable system states. For efficient state-storage, SPIN offers state compression (a form of byte-sharing) or, alternatively, BDD-like storage techniques based on minimized automata (Visser & Barringer 1996). In addition, approximate hashing methods are available, namely, hash-compact methods (Wolper & Leroy 1993) and bit-state hashing (Holzmann 1998).

SPIN generates C sources for a problem-specific model checker. This technique saves memory and improves performance, while also allowing the direct insertion of chunks of C code into the model.

SPIN reports on deadlocks, unspecified receptions, unexecutable code, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles and acceptance cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formula.

SPIN uses a depth-first search algorithm (breadth-first search is also possible) and can be used as a full LTL model checking system supporting all correctness requirements expressible in linear-time temporal logic (or Büuchi automata, directly). It can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties (e.g., progress and lack of deadlock), which can often be expressed and verified without the use of LTL. XSPIN is its graphical interface (Holzmann & Peled 1996, Holzmann 1997, Holzmann 2003).

SPIN has been used to trace logical errors in distributed systems designs, such as operating systems (Cattel 1994, Kumar & Li 2002), computer networks (Yuen & Tloe 2001), and railway signaling systems (Cimatti, Giunchig lia, Mingardi, Romano, Torielli & Traverso 1997), and for the feature interaction analysis of telecommunications and email systems (Calder & Miller 2001, Calder & Miller 2003, Holzmann & Smith 1999b).

SPIN is one of the most widely used model checkers and has a fairly broad group of users in both academia and industry. It is designed for analyzing the logical consistency of concurrent or distributed asynchronous software systems, and is specially focused on proving the correctness of process interactions. SPIN has been used to detect design errors in distributed applications such as operating systems, data communications protocols, switching systems, concurrent software, railway signaling protocols, etc., ranging from high-level abstract descriptions to low-level detailed codes.

Ben-Ari writes about SPIN (2008, p. viii):

> ...I found that SPIN is a very rare artifact: Although it is an industrial-strength tool, it can be easily used by students. The software is simple to install and to run, and models are written in PROMELA, which looks like a familiar programming language.

SPIN continues to evolve to keep pace with new developments in the field. The DSPIN tool (Iosif & Sisto 1999) is an extension of SPIN, which has been designed for modeling and verifying object-oriented software (JAVA programs, in particular). In addition to the usual features available with SPIN, the DSPIN model checker allows for the dynamic creation of heap objects and the representation of garbage collection.

5.1.2. Murϕ

$Mur\phi$ is a system description high-level language and model checker developed by software engineers to formally evaluate behavioral requirements for finite-state asynchronous concurrent systems (Dill et al. 1992, Dill 1996).

The $Mur\phi$ description language was inspired by Misra and Chandy's Unity formalism (Chandy & Jayadev 1988).

$Mur\phi$ is high-level in the sense that many features found in common high-level programming languages such as $Pascal$ or C are part of $Mur\phi$.

A $Mur\phi$ description consists of a collection of declarations of constants, data types such as subranges, records, and arrays, global variables, transition rules written in a Pascal-like language, which are guarded commands, a description of the initial states, and a set of invariants. Each transition rule consists of the followings:

1. a condition (a Boolean expression on the global variables)
2. an action (a statement that can modify the values of the variables).

The $Mur\phi$ Compiler generates a special purpose verifier from a $Mur\phi$ description. This verifier performs a depth- or breadth-first search over the state-space to check the properties of the system, such as deadlock or assertion or invariance violations. More complex temporal properties cannot be verified.

An execution of a $Mur\phi$ program is any sequence of states that can be generated by starting in one of the states generated by a start rule, then repeatedly selecting a rule and executing it. $Mur\phi$ is non-deterministic: there can be many executions, varying according to which rule was selected at each step of the execution. As states are generated by the verifier, various conditions are checked.

The $Mur\phi$ verifier is appropriate for protocols and finite-state systems which can reasonably be modeled as a collection of processes that run at arbitrary speeds, where the steps of the processes interleave (only one process takes a step at any time), and where the processes interact by reading and writing shared variables (asynchronous systems). It has been applied to several problems, e.g. multiprocessor cache coherence problems, link-level protocols, a hybrid byzantine agreement algorithm, mutual exclusion algorithms, memory model specifications.

An interesting but quite out-of-date information about $Mur\phi$ is available at original $Mur\phi$ web page: `http://sprout.stanford.edu/dill/murphi.html`. $Mur\phi$ is developed by prof. Ganesh Gopalakrishnan's research group at the University of Utah `http://www.cs.utah.edu/formal_verification/`.

5.2. Symbolic Model Checkers

5.2.1. Smv

Model checker Smv `http://www.cs.cmu.edu/~modelcheck/smv.html` (**S**ymbolic **m**odel **v**erifier) accepts both the temporal logics LTL and CTL. It is the first and the most successful $OBDD$-based symbolic model checker (McMillan 1993). Smv has been developed by The Model Checking Group that is a part of Specification and Verification Center, Carnegie Mellon University `http://www-2.cs.cmu.edu/~modelcheck/index.html`.

Systems are described using the SMV language, which has been developed with a precise semantics that relates programs to their expressions as Boolean formulas.

SMV is aimed at reliable verification of industrially sized designs. It supports both synchronous and asynchronous communication, and provides for modular hierarchical descriptions and the definition of reusable components. SMV has been used to verify various hardware systems, including an avionics triple sensor voter (Danjani-Brown, Cofer, Hartmann & Pratt 2003), the GIGAMAX cache coherence protocol (McMillan & Schwalbe 1992) and the *t*9000 virtual channel processor (Barrett 1995). The technique has been applied to several complex industrial systems such as the FUTUREBUS+ and the PCI local bus protocols. Extremely large state-spaces can often be traversed in minutes.

CADENCE SMV `http://www.kenmcmil.com/smv.html` is a symbolic model checking tool released by Cadence Berkeley Labs. CADENCE SMV is provided for formal verification of temporal logic properties of finite state systems, such as computer hardware designs. It is an extension of SMV. It has a more expressive mode description language, and also supports synthesizable VERILOG as a modeling language. CADENCE SMV supports a variety of techniques for compositional verification, allowing it to be applied to large designs, with user guidance. It allows several forms of specification, including the temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications. It also includes an easy-to-use graphical user interface and source level debugging capabilities.

The free research version of CADENCE SMV is available.

NUSMV (Cimatti, Clarke, Giunchig lia & Roveri 1999, Cimatti, Clarke, Giunchig lia, Giunchig lia, Pistore, Roveri, Sebastiani & Tacchella 2002) `http://nusmv.irst.itc.it`, `http://nusmv.fbk.eu/` is an updated version of SMV. The additional features contained in NUSMV include a textual interaction shell and graphical interface, extended model partitioning techniques, and facilities for LTL model checking. NUSMV (Cimatti, Clarke, Giunchig lia & Roveri 2000) has been developed as a joint project between Formal Methods group in the Automated Reasoning System division at Istituto Trentino di Cultura, Istituto per la Ricerca Scientifica e Tecnologica in Trento, Italy), the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at the University of Genoa and the Mechanized Reasoning Group at the University of Trento.

NUSMV 2 is open source software. It combines BDD-based model checking with SAT-based model checking. It has been designed as an open architecture for model checking. NUSMV 2 exploits the CUDD library developed by Fabio Somenzi at Colorado University and SAT-based model checking component that includes an RBC-based Bounded Model Checker, connected to the SIM SAT library developed by the University of Genova. It is aimed at reliable verification of industrially sized designs, for use as a back-end for other verification tools and as a research tool for formal verification techniques.

An enhanced version of Smv, RuleBase `http://www.haifa.ibm.com/projects/verification/RB_Homepage/` (Beer, Ben-David, Eisner & Landver 1996) is an industry-oriented tool for the verification of hardware designs, developed by the IBM Haifa Research Laboratory. In an effort to make the specification of CTL properties easier for the non-expert, RuleBase supports its own language, Sugar. In addition, RuleBase supports standard hardware description languages such as VHDL and Verilog. RuleBase is especially applicable for verifying the control logic of large hardware designs. Based on years of experience in practical formal verification, RuleBase offers this advanced technology to designers and verification engineers and not only to formal verification experts.

5.2.2. Vereofy

Vereofy `http://www.vereofy.de/` was written by Prof. Christel Baier at Technische Universität Dresden. It is developed in the context of the EU project CREDO. Vereofy is a formal verification tool of checking of component-based systems for operational correctness.

It uses two input languages: a scripting language called RSL (**R**eo **S**cripting **L**anguage), and a guarded command language, called CARML (**C**onstraint **A**utomata **R**eactive **M**odule **L**anguage). RSL and CARML are equally powerful and rely on the same semantic model. However, due to the nature of RSL and CARML, in most cases a hybrid approach, where CARML is used to provide the interface specifications of components and RSL for specifying the network is preferable.

Vereofy allows for linear and branching time model checking. To tackle the state space explosion problem, Vereofy generates an internal symbolic representation of the constraint automata for components, the network, and finally the composite system.

The model checker can be used as a stand-alone tool or via a graphical user interface (the Reo GUI developed at the Centrum Wiskunde & Informatica, Amsterdam as a part of the Eclipse Coordination Tools) as a plug-in.

5.3. Real-Time Model Checkers

Model checking tools were initially developed to reason about the logical correctness of discrete state systems, but have since been extended to deal with real-time and limited forms of hybrid systems.

When modeling certain critical systems, it is essential to include some notion of time. If time is considered to increase in discrete steps (discrete-time), then existing model checkers can be readily extended (Alur & Henzinger 1992, Emerson 1992). Real-time systems are systems that must perform a task within strict time deadlines. Embedded controllers, circuits and communication protocols are examples of such time-dependent systems. Real-time systems need to be rigorously modeled and specified in order to be able to formally prove their correctness with respect to the desired requirements. A real-time extension to Cospan (Alur & Kurshan 1995, Alur & Kurshan 1996) allows real-time constraints to be expressed by associating lower and upper bounds on the time spent by a process in a local state. An execution is said to be timing-consistent if its steps can be assigned real-valued time-stamps that satisfy all the specified bounds.

The hybrid model checker HyTech (Henzinger, Ho & Wong-Toi 1997) is used to analyze dynamical systems whose behavior exhibits both discrete and continuous change. HyTech automatically computes the conditions on the parameters under which the system satisfies its safety and timing requirements.

5.3.1. Uppaal

The most widely used dense real-time model checker (in which time is viewed as increasing continuously) is Uppaal `http://www.uppaal.com/` (Larson, Pettersson & Yi 1997). Models are expressed as timed automata (Alur & Dill 1993) and properties defined in Uppaal logic, a subset of Timed Computational Tree Logic ($TCTL$) (Alur, Courcoubetis & Dill 1990). Uppaal uses a combination of on-the-fly and symbolic techniques (Larson, Pettersson & Yi 1995, Yi, Pettersson & Daniels 1994), so as to reduce the verification problem to that of manipulating and solving simple constraints. *Uppaal* is a tool suite for validation and verification of real-time systems modeled as networks of timed automata extended with data variables. *Uppaal* consists of three main parts: a graphical user interface, a simulator and a model-checker engine. Modeling can be done in the graphical user interface. The simulator is helpful when debugging design errors because it can run interactively to check whether the system works as intended and generate traces. The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using test automata or systems decorated with debugging information (Larsen, Pettersson & Yi 1997). Uppaal implements the forward search algorithm in which the state space is explored in a breadth-first manner. It also uses on-the-fly verification combined with a symbolic technique, reducing the verification problem to that of solving simple constraints systems. The computation of clock constraints is aided with the data structure known as **D**ifference **B**ound **M**atrice**s** (Dsms) (Bengtsson & Yi 2004). The non-convex zones are stored and manipulated in the data structure called **C**lock **D**ifference **D**iagrams (Cdds) (Behrmann, Larsen, Pearson, Weise & Yi 1999).

Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

5.3.2. Kronos

Another real-time model checker is Kronos `http://www-verimag.imag.fr/TEMPORISE/kronos/` (Yovine 1997). Kronos is developed by Sergio Yovine at VERIMAG, a leading research center in embedded systems in France. Kronos checks whether a real-time system modeled by a timed automaton satisfies a timing property specified by a formula of the **T**imed **C**omputational **T**ree **L**ogic $TCTL$, a timed extension of CTL. Kronos implements a symbolic model-checking algorithm, where sets of states are symbolically represented by linear constraints over the clocks of the timed automaton. The correctness requirements are expressed in the real-time temporal logic $TCTL$.

Kronos is a tool developed with the aim to assist the user to validate complex real-time systems and is used to analyze systems modeled in several timed process description formalisms, such as Atp (Nicollin & Sifakis 1994) and ET-Lotos (L'eonrad & Leduc 1997, L'eonrad & Leduc 1998). Krons is a tool which implements a model checking algorithm for the $TCTL$ (Alur, Courcoubetis & Dill 1993, Clarke et al. 1999, Huth & Ryan 2004).

Kronos implements both the forward and backward algorithms (Daws & Yovine 1995). It allows one to express and verify not only reachability properties but liveness properties as well. The system is modeled as a set of concurrently operating timed automata.

Kronos supports verification based on both the region and simulation graphs (Bouajjani et al. 1997). Kronos has been used to verify real-time systems including the classical CSMA/CD protocol.

To improve the exploration of the state space, KRONOS also implements an on-the-fly technique. In this approach, a symbolic graph called a simulation graph is constructed. The computation of clock constraints is also aided with the DBM data structure.

KRONOS checks whether a timed automaton satisfies a $TCTL$-formula. The model-checking algorithm is based upon a symbolic representation of the infinite state space by sets of linear constraints.

Since KRONOS works on timed automata, potentially many more functionalities can be checked compared to SPIN.

KRONOS is freely distributed through the web for academic non-profit use.

5.3.3. STEP

Stanford **Te**mporal **P**rover, STEP `http://www-step.stanford.edu/`, STEP, is developed at Stanford University by the $REACT$ research group (National Science Foundation, Grant No. 9804100) (Manna, Bjørner, Browne, Chang, Alfaro, Devarajan, Kapur, Lee & Sipma 1994, Bjørner, Browne, Chang, Colón, Kapur, Manna, Sipma & Uribe 1996).

STEP is a system for reasoning about reactive, real-time and hybrid systems based on their temporal specification.

Unlike most systems for temporal verification, STEP is not restricted to finite-state systems, but combines model checking with deductive methods (Sipma, Uribe & Manna 1996) to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains. STEP is being extended with modular verification diagrams (Browne, Manna & Sipma 1996).

STEP integrates model-checking and theorem-proving methods for proving that a temporal logic formula ϕ is valid for a program $\mathcal{P}$. The model checker is based on the construction of the product automaton for $\mathcal{P}$ and $\neg\phi$ and checking the emptiness of its language.

An educational version of the system, which accompanies the textbook (Manna & A. Pnueli 1995b), is available: `step-request@cs.stanford.edu`.

5.4. Tools of direct model checking

Model checking requires the manual construction of a model, via a modeling language, which is then converted to a Kripke structure or an automaton for model checking. Model checking starts with translation to model checker language. Structures have to be “described” in this language. Since

1980 through mid 1990s, it was a hand-translation with ad-hoc abstractions. Semi-automated, table-driven translations begin in 1998. Automated translations still with ad hoc abstractions are characteristic for the period 1997–1999. State-less model checking for C VeriSoft has been applied in 1997.

In model checking considerable gains can be made by finding ways to extract models directly from program source code. There have been several promising attempts to do so.

Model checking may be applied directly to program source code written in languages such as Java or C. Early approaches to model checking Java software, like JCAT (Demartini, Iosif & Sisto 1999) and Java PathFinder (JPF1) (Havelund & Pressburger 2000), involved the direct translation of Java code into Promela, and subsequent verification via Spin. Although both of these systems were successful, direct translation meant that programs were only able to contain features that were supported by both Java and Promela (this is not true for floating point numbers, for example). The Bandera `http://santos.cis.ksu.edu/bandera/` tool (Corbett, Dwyer, Hatcliff, Laubach, Păsrăeanu, Robby & Zheng 2000) avoids direct translation, instead by extracting an abstracted finite-state model from Java source code. This model is then translated into a suitable modeling language (Promela or Smv) and model checked accordingly. Meanwhile, a second-generation of the Java PathFinder tool ($JPF2$) (Visser, Havelund, Brat & Park 2000), which makes extensive use of Bandera abstraction tools, has been developed to model check Java byte code directly.

5.4.1. VeriSoft

VeriSoft `http://cm.bell-labs.com/who/god/verisoft/` is the first model checker that could handle programs directly. It relies on partial-order reductions to limit the number of times a state is revisited.

VeriSoft is a tool for Systematic Software Testing.

- Customers. VeriSoft is a tool for software developers and testers of concurrent/reactive/real-time systems.
- Description. VeriSoft automatically searches for coordination problems (deadlocks, etc.) and assertion violations in a software system by generating, controlling, and observing the possible executions and interactions of all its components. It integrates automatic test generation, execution and evaluation in a single framework. VeriSoft includes an interactive graphical simulator that can drive existing debuggers for examining precisely the concurrent execution of multiple processes.

- Benefits. VeriSoft can quickly reveal behaviors that are virtually impossible to detect using conventional testing techniques, and hence reduces the cost of testing and debugging, while increasing reliability.
- Scope. VeriSoft can test software applications developed in any language (C, C++, Tcl, etc.). VeriSoft is optimized for analyzing multi-process applications. It can analyze systems composed of processes described by hundreds of thousands of lines of code. Source code for all the components is not required.
- Technology. The key technology used in VeriSoft is a new form of systematic state-space exploration (also called "model checking" in the research literature). With its first prototype developed in 1996 and its design first presented at POPL'97, VeriSoft is the first software model checker using a run-time scheduler for systematically driving the executions of an application through its state space.

The VeriSoft model checker (Godefroid 1997) is used to verify concurrent processes executing *C* code. Unlike traditional model checking techniques, the use of VeriSoft does not rely on states being expressed as sequences of bits. Systematic search of the state-space allows the user to check for deadlock and assertion violations, as well as for timeouts and live locks. A stateless search is used, whereby only states along the current path are stored, together with as many states as possible in the remaining available memory. As a result, state-space explosion is not a problem – it is theoretically possible to verify systems of any size. However, as as result, the same path may be explored many times, and so the search can be very slow.

5.4.2. Java PathFinder

On the home page of **J**ava **P**ath**F**inder (Jpf) `http://javapathfinder.sourceforge.net/` we read that it is a system to verify executable Java bytecode programs. In its basic form, it is a Java Virtual Machine that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. Unlike traditional debuggers, Jpf reports the entire execution path that leads to a defect. Jpf is especially well-suited to finding hard-to-test concurrency defects in multithreaded programs. Jpf integrates model checking, program analysis and testing.

While software model checking in theory sounds like a safe and robust verification method, reality shows that it does not scale well. To make it practical, a model checker has to employ flexible heuristics and state abstractions. Jpf is unique in terms of its configurability and extensibility, and hence is a good platform to explore new ways to improve scalability. JPF uses state compression to handle big states, and partial order and symmetry reduction, slicing, abstraction, and runtime analysis techniques to reduce the state space.

JPF is a pure Java application that can be run either as a standalone command line tool, or embedded into systems like development environments. It was mostly developed – and is still used – at the NASA Ames Research Center. Started in 1999 as a feasibility study for software model checking, JPF has found its way into academia and industry, and has even helped detect defects in real spacecraft.

5.4.3. BOGOR

`http://bogor.projects.cis.ksu.edu/` is the address of the BOGOR-Website-Home. BOGOR is an extensible software model checking framework with state of the art software model checking algorithms, visualizations, and user interface designed to support both general purpose and domain-specific software model checking. Although there are many model checkers available, BOGOR provides a number novel capabilities that make it especially well-suited for checking properties of a variety modern software artifacts, for building your own domain-specific engine, and for using it to teach model checking concepts.

- Direct support of features found concurrent object-oriented languages such as dynamic creation of threads and objects, object inheritance, virtual methods, exceptions, garbage collection, etc.
- BOGOR's modeling language can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g, multi-agent systems, avionics, security protocols, etc.) and a particular level of abstraction (e.g., design models, source code, byte code, etc.).
- BOGOR's open architecture well-organized module facility allows new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.) to be easily swapped in to replace Bogor's default model checking algorithms.
- BOGOR has a robust feature-rich graphical interface implemented as a plug-in for Eclipse – an open source and extensible universal tool platform from IBM. This user interface provides mechanisms for collecting and naming different BOGOR configurations, specification property collections, and a variety of visualization and navigation facilities.
- BOGOR is an excellent pedagogical vehicle for teaching foundations and applications of model checking because it allows students to see clean

implementations of basic model checking algorithms and to easily enhance and extend these algorithms in course projects (read more and see available course materials).

In short, Bogor aims to be not only a robust and feature-rich software model checking tool that handles the language constructs found in modern large-scale software system designs and implementations, it also aims to be a model checking framework that enables researchers and engineers to create families of domain-specific model checking engines.

The Bogor model checking framework (Robby & Hatcliff 2003) is used to check sequential and concurrent programs. The behavioral aspects of the program are first specified in Java modeling language, which, together with the original Java program, is then translated into a lower-level specification for verification. Bogor exploits the canonical heap representation of dSpin and is implemented as an Eclipse (Clayberg & Rubel 2004) plug-in.

5.4.4. Blast

For the direct model checking for C programs there are various tools available, e.g. Blast (**B**erkeley **L**azy **A**bstraction **S**oftware verification **T**ool) `http://mtc.epfl.ch/software-tools/blast/`, `http://www.sosy-lab.org/~dbeyer/blast_doc/blast001.html`, `http://www.sosy-lab.org/~dbeyer/blast_doc/blast.pdf` (Henzinger, Jhala, Majumdar & Sutre 2003). The first version of Blast was developed for checking safety properties in C programs at University of California, Berkeley by Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. The Blast project is supported by the National Science Foundation. The task addressed by Blast is the need to check whether a system satisfies the behavioral requirements of its associated interfaces.

Blast employs *CEGAR*[59] framework to construct an abstract model that is then model-checked. The abstraction is constructed on-the-fly, and only to the requested precision. The refinement is applied locally, i.e., it uses lazy abstraction to reduce unnecessary abstraction refinement.

The Blast specification language has a very C-like syntax. This makes it easier to learn, especially for C programmers, compared to learning a new specification language. The specification essentially looks for certain patterns in the original program and inserts some checks and actions to be performed when these patterns are matched. Blast claims to handle all syntactic constructs of C, including pointers, structures, and procedures.

[59] See p. 60.

Blast uses Simplify (Detlefs, Nelson & Saxe 2003, Detlefs, Nelson & Saxe 2005) and Vampyre `http://www.cs.ucla.edu/~rupak/Vampyre/` as theorem provers.

Blast is a popular software model checker for revealing errors in `Linux` kernel code.

Blast is relatively independent of the underlying machine and operating system. Blast is free software, released under the Modified *BSD* license `http://www.oss-watch.ac.uk/resources/modbsd.xml`. It is targeted at the general programmers in the software industry. In order to encourage programmers to use Blast for verification, an eclipse plug-in has been developed for Blast.

Blast is based on similar concepts as Slam. Blast and Slam are relatively new. Slam was developed by Microsoft Research around 2000, i.e., earlier than Blast, which was developed around 2002. Both checkers have many characteristics in common.

5.4.5. Slam

Slam `http://research.microsoft.com/en-us/projects/slam/` was originally developed to check C programs (system software) for temporal safety properties. These properties remain its main goal. Slam is a project developed by Microsoft Research for addressing critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning. The main application domain is device drivers in `Windows`.

Slam has been customized for the `Windows` product `Static Driver Verifier`, Sdv, a tool in the `Windows Driver Development Kit`. Sdv uses the Slam verification engine to analyze the source code of `Windows` device drivers (Ball, Cook, Levin & Rajamani 2004). Sdv involves a similar abstraction, verification, and refinement loop to that of Blast and exploits the Bebop model checker during the verification stage.

The specification language used for Slam is Slic (**S**pecification **L**anguage for **I**nterface **C**hecking). The concept behind Slam has been used for a tool called Beacon (Ball, Chaki & Rajamani 2001), which checks for interface usage rules in multithreaded software libraries.

Slam and Blast work on a C program and take the specification of the property to be checked as its input. With respect to the specification language, Blast has an advantage over Slam. Slic does not support type-state properties. It monitors only function calls and returns and so, is limited to the specification of interfaces.

Slam and Blast either verify that the system is safe, i.e. the program satisfies the specified property or give an error trace that violates that property. SLAM is comparable to BLAST in scalability and precision. Both SLAM and BLAST perform static analysis and use *CEGAR* paradigm to extract a finite state model from the C program. Both tools handle C language constructs (like pointers, structures, and procedures) and assume a logical model of the memory.

One key difference between SLAM and BLAST is the use of lazy abstraction in BLAST.

SLAM and BLAST differ from other model checking tools in many ways. First of all, the traditional approach to model-checking (followed by SPIN and KRONOS) has been to first create a model of a system, and once the model has been verified, move on to the actual implementation. SLAM and BLAST fall in the category of the "modern" approach in model checking. The user has already completed the implementation and wishes to verify the software. The objective then is to create a model from the existing program and apply model checking principles, such that the original program is verified.

SLAM and BLAST are used for checking safety properties only, whereas SPIN is more developed and also checks liveness properties.

5.4.6. CHIC

Checker for **I**nterface **C**ompatibility CHIC `http://www.eecs.berkeley.edu/~arindam/chic/` (de Alfaro & Henzinger 2001) is a modular verifier for behavioral compatibility checking of hardware and software systems. The goal of CHIC is to be able to check that the interfaces for software or hardware components provide guarantees that satisfy the assumptions they make about each other. CHIC supports a variety of interface property specification formalisms for a variety of application domains, such as resource-usage analysis for embedded systems, behavioral compatibility of web service applications, etc.

CHIC is a modular verifier for behavioral compatibility checking of software and hardware components. The goal of CHIC is to be able to check that the interfaces for software or hardware components provide guarantees that satisfy the assumptions they make about each other. CHIC supports a variety of interface property specification formalisms.

5.4.7. CHESS

CHESS `http://research.microsoft.com/en-us/projects/CHESS/` model checker is a software model checker for finding and reproducing Heisenbugs[60] in multithreaded software by systematic exploration of thread schedules. It finds errors, such as data-races, deadlocks, livelocks, and data-corruption induced access violations, that are extremely hard to find with other testing tools. Chess can be used for testing concurrent software, as an alternative to stress testing.

Once Chess locates an error, it provides a fully repeatable execution of the program leading to the error, thus greatly aiding the debugging process. Chess is available for both managed and native programs.

5.4.8. FeaVer

The FeaVer (**Fea**ture **Ver**ification system) `http://cm.bell-labs.com/cm/cs/what/feaver/` tool grew out of an attempt to come up with a thorough method to check the call processing software for a commercial switching product, called the PathStar® access server (Holzmann & Smith 1999b, Holzmann 2002). It allows models to be extracted mechanically from the source of software applications, and checked using Spin. Spin allows C code to be embedded directly within a Promela specification (Holzmann & Smith 1999a, Holzmann & Smith 2002).

In the application of FeaVer, abstraction functions are recorded in a lookup table that acts as a filter for the source code. Abstraction is only applied to basic statements and conditionals; the control-flow structure of the source code is preserved. To apply the abstraction and generate the system model, the source code is first parsed, with a standard compiler frontend.

5.4.9. Time Rover

The Time Rover `http://www.time-rover.com/` is a specification based verification tool for applications written in C, C++, Java, Verilog and VHDL. The tool combines formal specification, using LTL and MTL, with conventional simulation/execution based testing. The Temporal Rover is tailored for the verification of complex protocols and reactive systems where behavior is time dependent. The Temporal Rover generates executable code from LTL and MTL assertions written as comments in the source code. This executable source code is compiled and linked as part of the application under test. During application execution the generated code validates the executing program against the formal temporal specification requirements. Using MTL, real time and relative time constraints can be validated. A spe-

[60] It is a bug that disappears or alters its characteristics when an attempt is made to study it. Named after the Heisenberg Uncertainty Principle.

cial code generator support s validation of such constraints in the field, on an embedded target.

On the website `http://www.time-rover.com/company.html` we read:

> Time Rover Software specializes in the entire validation and verification process for safety critical software. We provide both experts and tools that will help your team to:
> - create the right product (validation).
> - create the product right (verification).
>
> Our methodology and technology are based on the Unified Modeling Language (UML) and are currently in active use by NASA IV and V center and the national Missile Defense development team.

5.5 Probabilistic model checker

Since Pnueli introduced temporal logic to computer science, logic has been extended in various ways to include probability. Probabilistic techniques have proved successful in the specification and verification of systems that exhibit uncertainty. The behavior of many real-life processes is inherently stochastic. Probability is an important component in the design and analysis of complex systems across a broad spectrum of application domains, including communication and multimedia protocols, randomized distributed algorithms, security protocols, dynamic power management and biological systems. This leads to the study of probabilistic model checking of probabilistic models based on Markov chains[61] or Markov decision processes. This formal tool provides efficient and rigorous methods for evaluating a wide range of properties, from performance and reliability to security and anonymity.

> Whereas model-checking techniques focus on the absolute guarantee of correctness – “it is impossible that the system fails” – in practice such rigid notions are hard, or even impossible, to guarantee. Instead, systems are subject to various phenomena of a stochastic nature, such as message loss or garbling and the like, and correctness – “with 99% chance the system will not fail” – is becoming less absolute. (Baier & Katoen 2008, p. 745).

Probabilistic aspects are essential for, among others:

- Randomized algorithms.
- Modeling unreliable and unpredictable system behavior.
- Model-based performance evaluation.

[61] The basic concepts of continuous-time Markov chains were introduced by Markov (1907) for state spaces and Kolmogorov (1936) for denumerable and continuous spaces.

5.5.1. Model Checking Probabilistic Systems

Probabilistic model checking concerns verification of probabilistic systems. Randomization is frequently used in real-world distributed coordination protocols, fault-tolerant algorithms and in adaptive schemes. Early work has concentrated on discrete-time models. Formal verification based on temporal logic has been successfully extended to the verification problems of probabilistic systems. In order to model random phenomena, transition systems are enriched with probabilities. The transitions between states are labeled with information about the likelihood that they will occur. As in the non-probabilistic case, the principal challenge when developing probabilistic model checker is to overcome the state explosion problem.

The verification of probabilistic systems can be focused on either quantitative properties or qualitative properties (or both).

Quantitative properties typically put constraints on the probability or expectation of certain events. Instances of quantitative properties are, e.g., the requirement that the probability for delivering a message within the next t time units is at least 0.98, or that the expected number of unsuccessful attempts to find a leader in a concurrent system is at most seven.

Qualitative properties, on the other hand, typically assert that a certain (good) event will happen almost surely, i.e., with probability 1, or dually, that a certain (bad) event almost never occurs, i.e., with 0 probability. That is, qualitative properties arise as a special case of quantitative properties where the probability bounds are the trivial bounds 0 or 1. Typical qualitative properties for Markov models are reachability, persistence (does eventually an event always hold?), and repeated reachability (can certain states be repeatedly reached?) (Baier & Katoen 2008, p. 746).

Early works in this field were focusing on the verification of qualitative properties. These included work of (Courcoubetis & M. Yannakakis 1988) which considered models of two types, **D**iscrete-**T**ime **M**arkov **C**hains (DTMCs) and **M**arkov **D**ecision **P**rocesses (MDPs).

The verification of quantitative properties is more involved than that of qualitative properties. Typical qualitative properties require that the probability of reaching a bad state is 0, or dually, that a certain desired system behavior appears with probability 1 whereas in the case of the quantitative properties the exact probability has to be computed for a given property in addition to the satisfaction of that property. In the work of (Hansson & Jonsson 1994), the **P**robabilistic **C**omputation **T**ree **L**ogic ($PCTL$) was introduced for the verification of DTMCs. The verification of quantitative properties for MDPs was considered in (Courcoubetis & Yannakakis 1990, Bianco & Alfaro 1995, Baier & Kwiatkowska 1998).

Methods to verify DTMCa or MDPs against a linear-time have been considered, e.g., (Courcoubetis & Yannakakis 1995, Pnueli & Zuck 1993, Vardi 1985). Probabilistic branching time logic model checking is studied in, e.g., (Hansson & Jonsson 1994, Aziz, Singhal & Balarin 1995, Baier & Kwiatkowska 1998, Bianco & De Alfaro 1995).

Tools concerning model checking probabilistic systems such as PRISM (**PR**obabilistic **S**ymbolic **M**odel Checker) `http://www.cs.bham.ac.uk/~dxp/prism/`, (Kwiatkowska, Norman & Parker 2001, Kwiatkowska, Norman & Parker 2002b, Kwiatkowska, Norman & Parker 2002a) have been developed and applied to several real-world case studies. Other tools include ETMCC (Hermanns, Katoen, Meyer-Kayser & Siegle 2000), CASPA (Kuntz, Siegle & Werner 2004) and MRMC (**M**arkov **R**eward **M**odel **C**hecker) (Katoen, Khattri & Zapreev 2005).

5.5.2. ETMCC

Probabilistic Model Checker ETMCC (**E**rlangen-**T**wente **M**arkov **C**hain **C**hecker) (Hermanns et al. 2000) is developed jointly by the Stochastic Modeling and Verification group at the University of Erlangen-Nürnberg, Germany, and the Formal Methods group at the University of Twente, the Netherlands. ETMCC is the first implementation of a model checker for **D**iscrete-**T**ime **M**arkov **C**hains (DTMs) and **C**ontinuous-**T**ime **M**arkov **C**hains (CTMCs). It uses numerical methods to model check $PCTL$ (Hansson & Jonsson 1994) and **C**ontinuous **S**tochastic **L**ogic (CSL)[62] formulas respectively for DTMCs and CTMCs. The current version of ETMCC comes along with an experimental model checking engine supporting verification techniques to check action based CSL (aCSL) (Vaandrager F. W. and De Nicola 1990) requirements against action-labeled continuous time Markov chains.

5.5.3. Markov Reward Model Checker

Markov **R**eward **M**odel **C**hecker (MRMC) `http://www.mrmc-tool.org/trac/` has been developed by the Formal Methods & Tools group at the University of Twente, The Netherlands and the Software Modeling and Verification group at RWTH Aachen University, Germany under the guidance of Joost-Pieter Katoen (Baier & Katoen 2008, Ch. 10 Probabilistic systems). MRMC is a successor of ETMCC, which is a prototype implementation of a model checker for continuous-time Markov chains.

[62] A branching-time temporal logic a'la CTL with state and path formulas (Aziz, Sanwal, Singhal & Brayton 1996, Baier, Katoen & Hermanns 1999, Aziz, Sanwal, Singhal & Brayton 2000).

Mrmc is a tool (back-end) for performing model checking on Markov reward models, i.e. it is a model checker for:

- Discrete time Markov chains,
- Continuous time Markov chains,
- Discrete time Markov Reward models,
- Continuous time Markov Reward models,
- Continuous time Markov decision processes.

The tool supports verification of:

- Probabilistic Computation Tree Logic,
- Continuous Stochastic Logic,
- Probabilistic Reward Computation Tree Logic,
- Continuous Stochastic Reward Logic.

Mrmc allows for for the automated verification of properties concerning long-run and instantaneous rewards as well as cumulative rewards. It supports:

- Numerical model checking on all types of input models,
- Model checking by Discrete Event Simulation on CTMCs,
- Formula-dependent and formula-independent bisimulation.

Mrmc is a command-line tool, written in C. It is available for:

- `Windows`,
- `Linux`, and
- `Mac OS X`

platforms. The tool is distributed under the GNU Public License.

5.5.4. Prism

Prism stands for Probabilistic Symbolic Model Checker `http://www.prismmodelchecker.org/`. It is the internationally leading probabilistic model checker being implemented at the University of Birmingham (Kwiatkowska et al. 2001, Kwiatkowska et al. 2002), `http://www.cs.bham.ac.uk/~dxp/prism/`. First public release: September 2001.

There are three types of probabilistic models that Prism can support directly: **D**iscrete-**T**ime **M**arkov **C**hains (Dtmcs), Markov decision processes (Mdps) and **C**ontinuous-**T**ime **M**arkov **C**hains (Ctmcs).

Prism (Kwiatkowska et al. 2002, Rutten, Kwiatkowska, Norman & Parker 2004) allows time to be considered as increasing either in discrete steps or continuously. Models are expressed in Prism own modeling language and converted to a variant of the Markov chain (either discrete- or continuous-time). Properties are written in terms of $PCTL$ or CSL, respectively. Models can also be expressed using Pepa (**P**erformance **E**valuation **P**rocess **A**lgebra) (Hillston 1996) and converted to Prism.

The user interface and parsers are written in JAVA; the core algorithms are mostly implemented in $C++$. For state space representation, PRISM uses a modified version of the *CUDD* package (Somenzi 1997).

PRISM offers a choice between three engines: one symbolic using MTBDDS (**M**ulti-**T**erminal **B**inary **D**ecision **D**iagrams); one based on sparse matrix techniques; and one hybrid engine PRISMH which combines both symbolic and sparse approaches. It is expected that PRISM is faster, whereas PRISMH consumes less memory.

The current version of PRISM is 3.2 (first released 15 Jun 2008). Notable improvements and additions since the last main release (3.1.1) include:

- Support for 64-bit architectures and Mac OS X v10.5 (Leopard)
- Additions to property specification language
- Redesign of the simulator *GUI*
- New graph plotting engine using *JFreeChart*
- Prototype *SBML*-to-PRISM translator
- Extra reward model checking algorithms for some engines

PRISM has been used to analyze several real-world case studies `http://www.cs.bham.ac.uk/~dxp/prism/`. PRISM enables quantitative analysis of properties such as expected time, average power consumption, and probability of delivery by deadline. It can be used to analyze systems from a wide range of application domains, including communication and multimedia protocols, randomized distributed algorithms, security protocols, biological systems and many others. In particular, it has been used to model and analyze over 30 real-world protocols, which included anonymity protocols for the Internet, Bluetooth device discovery, dynamic power management, nanotechnology designs and biochemical reactions. For example, it was discovered with PRISM that the Crowds anonymity protocol does not in fact guarantee anonymity, and that the worst case time to hear one message during Bluetooth device discovery is 2.5 seconds. PRISM was also used to analyze the IEEE 1394 FIREWIRE root contention protocol, a randomized leader election protocol which uses an electronic coin. Analysis with PRISM confirmed that a biased coin gives an advantage.

PRISM is free and open source, released under the *GNU* General Public License (*GPL*), available freely for research an teaching. There are 16459 downloads of PRISM to date.

5.5.5. APMC

APMC, **A**pproximate **P**robabilistic **M**odel **C**hecker `http://apmc.berbiqui.org/index.php/Accueil` is an approximate distributed model checker for fully probabilistic systems. APMC uses a randomized algorithm to approximate the probability that a temporal formula is true, by using sampling of execution paths of the system. APMC uses a distributed computation model to distribute path generation and formula verification on a cluster of workstations. The implementation of the tool started in 2003 and was originally done using C programming language together with LEX and YACC. APMC was rewritten recently in JAVA for its version 3.0.

Publications concerning APMC are available at: `http://apmc.berbiqui.org/index.php/Publications`.

6. Conclusions

Since the early nineties of the last century when in the verification of finite state systems the breakthrough was achieved to present time, a success story of modern computer science was written. Formal methods of *ICT*, in particular methods of model checking are still developing. The old ones are still improved to be more efficient and more flexible. The plenitude of new ones reveals the potentiality of formal methods of *ICT* systems verification. Due to human ingenuity, in formal methods as well as in technology, the potentiality of automatic formal verification increased enormously, but the horizon is still ahead of us, there is so much more and there will always be, since there are still more advanced and more complicated *ICT* systems conceived that need to be verified.

References

Akers, S. (1978), 'Binary decision diagrams', *IEEE Transactions on Computers* **C-27**(6), 509–516.

Allen, J. F. (1984), 'Towards a general theory of action and time', *Artificial Intelligence* **23**, 123–154.

Allen, J. F. (1985), 'Charles hamblin (1922–1985)', *The Australian Computer Journal* **17**, 194–195.

Alur, R., Courcoubetis, C. & Dill, D. L. (1990), Model-checking for real-time systems, *in* 'Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, Philadelphia, PA, pp. 414–425.

Alur, R., Courcoubetis, C. & Dill, D. L. (1993), 'Model-checking in dense real-time', *Information and Computation* **104**(1), 2–34.

Alur, R. & Dill, D. (1993), 'A theory of timed automata', *Inf. Comput.* **194**, 2–34.

Alur, R. & Dill, D. L. (1994), 'A theory of timed automata', *Theoretical Computer Science* **126**(2), 183–235.

Alur, R., Feder, T. & Henzinger, T. A. (1991), The benefits of relaxing punctuality, *in* 'Symposium on Principles of Distributed Computing', pp. 139–152.

Alur, R. & Henzinger, T. (1992), Logics and models of real time: A survey, *in* I. W. de Bakker et al., ed., 'Proceedings of the REX Workshop on Real-Time: Theory and Practice', Vol. 600 of *Lecture Notes in Computer Science*, Springer-Verlag, Mook, the Netherlands, pp. 74–106.

Alur, R. & Kurshan, R. (1995), Timing analysis in Cospan, *in* R. A. et al., ed., 'Proceedings of the 3rd DIMACS/SYCON Workshop on Hybrid Systems: Verification and Control', Vol. 1066 of *Lecture Notes in Computer Science*, Springer-Verlag, New Brunswick, NJ, pp. 220–231.

Alur, R. & Kurshan, P. (1996), Timing analysis in Cospan, *in* 'Hybrid System-III, Control and Verification', Vol. 1066 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 220–231.

Aziz, A., Sanwal, K., Singhal, V. & Brayton, R. (2000), 'Model checking continuous time Markov chains', *ACM Trans. Computational Logic* **1**(1), 162–170.

Aziz, A., Sanwal, K., Singhal, V. & Brayton, R. K. (1996), Verifying continuous time Markov chains, *in* R. Alur & T. A. Henzinger, eds, 'Eighth International Conference on Computer Aided Verification CAV 1996', Vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, New Brunswick, NJ, USA, pp. 269–276.

Aziz, A., Singhal, V. & Balarin, F. (1995), It usually works: The temporal logic of stochastic systems, *in* 'Proceedings of the 7th International Conference on Computer Aided Verification', Springer-Verlag, London, UK, pp. 155–165.

Baier, C. & Katoen, J.-P. (2008), *Principles of Model Checking*, The MIT Press. Foreword by Kim Guldstrand Larsen.

Baier, C., Katoen, J.-P. & Hermanns, H. (1999), Approximate symbolic model checking of continuous-time Markov chains, *in* 'International Conference on Concurrency Theory', pp. 146–161.

Baier, C. & Kwiatkowska, M. (1998), 'Model checking for a probabilistic branching time logic with fairness', *Distributed Computing* **11**(3), 125–155.

Ball, T., Chaki, S. & Rajamani, S. K. (2001), Parameterized verification of multithreaded software libraries, *in* 'TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer-Verlag, London, UK, pp. 158–173.

Ball, T., Cook, B., Levin, V. & Rajamani, K. (2004), Slam and static driver verifier: Technology transfer of formal methods inside microsoft, *in* E. B. et al, ed., 'Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)', Vol. 2999 of *Lecture Notes in Computer Science*, Springer-Verlag, Canterbury, UK, pp. 1–20.

Barrett, G. (1995), 'Model checking in practice: The $t9000$ virtual channel processor', *IEEE Trans. Softw. Eng.* **21**(2), 69–78.

Barton, R. S. (1970), Ideas for computer systems organization: a personal survey, *in* J. S. Jou, ed., 'Software Engineering', Vol. 1 of *Proceedings of the Third Symposium on Computer and Information Sciences held in Miami Beach, Florida, December 1969*, Academic Press, New York, NY, USA, pp. 7–16.

Beer, I., Ben-David, S., Eisner, C. & Landver, A. (1996), Rulebase: An industry-oriented formal verification tool, *in* 'Proceedings of the 33rd Conference on Design Automation (DAC'96)', ACM Press, Las Vegas, NV, pp. 655–660.

Behrmann, G., Larsen, K. G., Pearson, J., Weise, C. & Yi, W. (1999), Efficient timed reachability analysis using clock difference diagrams, *in* 'Computer Aided Verification', pp. 341–353.

Ben-Ari, M. (2008), *Principles of the Spin Model Checker*, Springer, London.

Ben-Ari, M., Manna, Z. & Pnueli, A. (1981), The temporal logic of branching time, *in* 'Proc. 8th ACM Symposium on Principles of Programming Languages', ACM Press, New York, pp. 164–176. Por. (Ben-Ari et al. 1983).

Ben-Ari, M., Manna, Z. & Pnueli, A. (1983), 'The temporal logic of branching time', *Acta Informatica* **20**, 207–226. Por. (Ben-Ari et al. 1981).

Ben-David, S. & Heyman, T. (2000), Scalable distributed on-the-fly symbolic model checking, *in* W. A. Hunt Jr. & S. D. Johnson, eds, 'Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)', Vol. 1954 of *Lecture Notes in Computer Science*, Springer-Verlag, Austin, TX., pp. 390–404.

Bengtsson, J., Griffioen, W. O. D., Kristoffersen, K. J., Larsen, K. G., Larsson, F., Pettersson, P. & Yi, W. (1996), Verification of an audio protocol with bus collision using UPPAAL, *in* R. Alur & T. A. Henzinger, eds, 'Proceedings of the Eighth International Conference on Computer Aided Verification CAV', Vol. 1102, Springer Verlag, New Brunswick, NJ, USA, pp. 244–256.

Bengtsson, J. & Yi, W. (2004), Timed automata: Semantics, algorithms and tools, *in* G. R. Jörg Desel, Wolfgang Reisig, ed., 'Lectures on Concurrency and Petri Nets', Vol. 3098 of *Lecture Notes in Computer Science*, Springer, pp. 87–124.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. & Schnoebelen, P. (2001), *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer.

Best, B. & Grahlmann, B. (1996), PEP – More than a Petri net tool, *in* T. Margaria & B. Steffen, eds, 'Proceedings of the 2nd International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '96)', Vol. 1055 of *Lecture Notes in Computer Science*, Springer-Verlag, Passau, Germany, pp. 397–401.

Bhat, G., Cleaveland, R. & Grumbero, O. (1995), Efficient on-the-fly model checking for CTL^*, *in* 'Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, San Diego, CA, pp. 388–397.

Bianco, A. & Alfaro, L. d. (1995), Model checking of probabilistic and nondeterministic systems, *in* 'Foundations of Software Technology and Theoretical Computer Science', Vol. 1026 of *Lectures Notes in Computer Science*, pp. 499–512.

Bianco, A. & De Alfaro, L. (1995), Model checking of probabilistic and nondeterministic systems, *in* 'Temporal Logies and Verification Theory. Foundations of Software Technology and Theoretical Computer

Science', Vol. 1026 of *Lecture Notes in Computer Science*, Springer, Berlin/Heidelberg, pp. 499–513.

Bidoit, B. M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. & Schnoebelen, P. (2001), *Systems and Software Verification: Model-checking Techniques and Tools*, Springer.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M. & Zhu, Y. (1999), Symbolic model checking using SAT procedures instead of BDDs, *in* 'Proceedings of the Design Automation Conference (DAC'99)', pp. 317–320.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O. & Zhu, Y. (2003), 'Bounded model checking', *Advances in Computers* **58**.

Biere, A., Cimatti, A., Clarke, E. M. & Zhu, Y. (1999), Symbolic model checking without BDDs, *in* 'Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)', number 1579 *in* 'LNCS'.

Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H. B. & Uribe, T. E. (1996), STeP: Deductive-algorithmic verification of reactive and real-time systems, *in* 'International Conference on Computer Aided Verification', Vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 415–418.

Boehm, B. W. (1981), *Software Engineering Economics*, Prentice-Hall.

Bollig, B. & Wegener, I. (1996), 'Improving the variable ordering of *obdd*s is *np*-complete', *IEEE Trans. Comput.* **45**(9), 993–1002.

Bolognesi, T. & Brinksma, E. (1987), 'Introduction to the ISO specification language Lotos', *Comput. Netw. ISDN Syst.* **14**, 25–59.

Bosnacki, D., Dams, D. & Holenderski, L. (2002), 'Symmetric spin', *Int. J. Soft. Tools Technol. Transfer* **4**(1), 65–80.

Bouajjani, A., Tripakis, S. & Yovine, S. (1997), On-the-fly symbolic model checking for real-time systems, *in* 'Proceedings of 18th IEEE Real-Time Systems Symposium (RTSS '97)', IEEE CS Press, Los Alamitos, pp. 25–34.

Boyer, R. S. & Moore, J. S. (1979), *A Computational Logic*, Academic Press, New York.

Boyer, R. S. & Moore, J. S. (1988), 'Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic', *Machine Intelligence* **11**, 83–124.

Bradfield, J. C. & Stirling, C. (2001), Modal logics and μ-calculi: An introduction, *in* J. A. Bergstra, A. Ponse & S. A. Smolka, eds, 'Handbook of Process Algebra', Elsevier Science, chapter 4, pp. 293–330.

Bradfield, J. & Stirling, C. (1991), Local model checking for infinite state spaces, *in* K. Larsen & A. Skou, eds, 'Workshop on Computer Aided Verification (CAV)'.

Brock, B. & Hunt, W. (1997), Formally specifying and mechanically verifying programs for the motorola complex arithmetic processor dsp, *in* 'Proceedings of the IEEE International Conference on Computer Design (ICCD'97)', pp. 31–36.

Browne, A., Manna, Z. & Sipma, H. (1996), Modular verification diagrams, Technical report, Computer Science Department, Stanford University.

Bryant, R. E. (1986), 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers* **C-35**(8), 677–691.

Bryant, R. E. (1992), 'Symbolic boolean manipulation with ordered binary-decision diagrams', *ACM Computing Surveys* **24**(3), 293–318.

Bryant, R. E. & Chen, Y.-A. (1995), Verification of arithmetic circuits with binary moment diagrams, *in* 'Design Automation Conference', pp. 535–541.

Büchi, J. R. (1960), 'On a decision method in restricted second order arithmetic', *Z. Math. Logik Grundlagen Math.* **6**, 66–92.

Bultan, T., Gerber, R. & Pugh, W. (1997), Symbolic model checking of infinite state systems using presburger arithmetic, *in* 'CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification', Springer-Verlag, London, UK, pp. 400–411.

Bultan, T., Gerber, R. & Pugh, W. (1999), 'Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results', *ACM Trans. Program. Lang. Syst.* **21**(4), 747–789.

Burch, J., Clarke, E., McMillan, K., Dill, D. & Hwang, L. (1992), 'Symbolic model checking: 10^{20} states and beyond', *Inf. Comput.* **2**, 142–170.

Burch, J. R., Clarke, E. M. & Long, D. E. (1991), Symbolic model checking with partitioned transition relations, *in* A. Halaas & P. B. Denyer, eds, 'International Conference on Very Large Scale Integration', North-Holland, Edinburgh, Scotland, pp. 49–58.

Burch, J. R., Clarke, E. M., Long, D. E., MacMillan, K. L. & Dill1, D. L. (1994), 'Symbolic model checking for sequential circuit verification', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **13**(4), 401–4124.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L. & Hwang, L. J. (1990), Symbolic model checking: 10^{20} states and beyond, *in* 'Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)', IEEE Computer Society Press, pp. 428–439.

Burgess, J. P. (1984), Basic tense logic, *in* D. Gabbay & F. Guenthner, eds, 'Handbook of Philosophical Logic', Vol. II, D. Reidel, Dordrecht, pp. 89–133.

Burkart, O., Caucal, D., Moller, F. & Steffen, B. (2001), Verification of infinite structures, *in* S. S. J. Bergstra, A. Ponse, ed., 'Handbook of Process Algebra', Amsterdam, pp. 545–623.

Calder, M. & Miller, A. (2001), Using Spin for feature interaction analysis – A case study, *in* M. Dwyer, ed., 'Proceedings of the 8th International Spin Workshop (Spin 2001)', Vol. 2057 of *Lecture Notes in Computer Science*, Springer-Verlag, Toronto, Canada, pp. 143–162.

Calder, M. & Miller, A. (2003), Generalizing feature interactions in email, *in* D. Amyot & L. Logrippo, eds, 'Feature Interactions in Telecommunications and Software Systems VII', IOS Press, Ottawa, Canada, pp. 187–205.

Cattel, T. (1994), Modeling and verification of a multiprocessor real-time *os* kernel, *in* D. Hogrefe & S. Leue, eds, 'Proceedings of the 7th WG6.1 International Conference on Formal Description Technigues (FORTE '94)', Vol. 6, Berne, Switzerland. International Federation for Information Processing, Chapman and Hall, London, UK, pp. 55–70.

Cengarle, M. V. & Haeberer, A. M. (2000), Towards an epistemology-based methodology for verification and validation testing, Technical report 0001, Ludwig-Maximilian's Universität, Institut für Informatik, München, Oettingenstr. 67. 71 pages.

Chandy, M. K. & Jayadev, M. (1988), *Parallel Program Design – a Foundation*, Addison-Wesley.

Chang, E., Pnueli, A. & Manna, Z. (1994), Compositional verification of real-time systems, *in* 'Proc. 9'th IEEE Symp. On Logic In Computer Science', pp. 458–465.

Cimatti, A., Clarke, E., Giunchig lia, E., Giunchig lia, F., Pistore, M., Roveri, M., Sebastiani, R. & Tacchella, A. (2002), NuSMV2: A new opensource tool for symbolic model checking, *in* E. Brinksma & K. Larsen, eds, 'Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)', Vol. 2404 of *Lecture Notes in Computer Science*, Springer-Verlag, Copenhagen, Denmark, pp. 359–364.

Cimatti, A., Clarke, E., Giunchig lia, F. & Roveri, M. (1999), NuSMV2: A new symbolic model verifier, *in* N. Halbwachs & D. Peled, eds, 'Proceedings of the 11th International Conference on Computer-Aided Verification (CAV '99)', Vol. 1633 of *Lecture Notes in Computer Science*, Springer-Verlag, Trento, Italy, pp. 495–499.

Cimatti, A., Clarke, E. M., Giunchig lia, F. & Roveri, M. (2000), 'NuSmv: A new symbolic model checker', *International Journal on Software Tools for Technology Transfer* **2**(4), 410–425.

Cimatti, A., Giunchig lia, F., Mingardi, G., Romano, D., Torielli, F. & Traverso, P. (1997), Model checking safety critical software with Spin: An application to a railway interlocking system, *in* R. Langerak, ed., 'Proceedings of the 3rd SPIN Workshop (SPIN '97)', Twente University, The Netherlands, pp. 5–17.

Clarke, E., Enders, R., Filkhorn, T. & Jha, S. (1996), 'Exploiting symmetry in temporal logic model checking', *Formal Methods Syst. Desi.* **9**(1/2), 77–104.

Clarke, E., Fujita, M., McGeer, P., McMillan, K., Yang, J. & Zhao, X. (1993), Multiterminal binary decision diagrams: An efficient data structure for matrix representation, *in* 'Proc. IWLS'93', pp. 1–15.

Clarke, E., Grumberg, O., Jha, o., Lu, Y. & Veith, H. (2000), Counterexample-guided abstraction refinement, *in* E. Emerson & A. Sistla, eds, 'Computer Aided Verification', Vol. 1855 of *LNCS*, Springer, Berlin/Heidelberg, pp. 154–169.

Clarke, E. M. (2008), The birth of model checking, *in* Grumberg & Veith (2008), pp. 1–26.

Clarke, E. M. & E., E. A. (1982*a*), Design and synthesis of synchronization skeletons using branching-time temporal logic, *in* 'Logic of Programs, Workshop', Vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 52–71.

Clarke, E. M. & E., E. A. (1982*b*), Synthesis of synchronization skeletons for branching time temporal logic, *in* 'Logic of Programs, Workshop', Vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, Yorktown Heights, NY.

Clarke, E. M., Emerson, E. A. & Sistla, A. P. (1983), Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach, *in* 'Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages', Austin, Texas, pp. 117–126.

Clarke, E. M., Emerson, E. A. & Sistla, A. P. (1986), 'Automatic verification of finite-state concurrent systems using temporal logic specifications', *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

Clarke, E. M., Grumberg, J. O. & Peled, D. A. (1999), *Model Checking*, The MIT Press.

Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L. & Ness, L. A. (1993), Verification of the futurebus+ cache coherence protocol, *in* D. Agnew, L. Claesen & R. Camposano, eds, 'The Eleventh International Symposium on Computer Hardware Description Languages and their Applications, Ottawa, Canada, 1993', Elsevier Science Publishers B.V., Amsterdam, pp. 5–20.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. & Veith, H. (2001), Progress on the state explosion problem in model checking, *in* 'Informatics – 10 Years Back. 10 Years Ahead.', Vol. 2000 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 176–194.

Clarke, E. M., Grumberg, O. & Long, D. E. (1994), 'Model checking and abstraction', *ACM Transactions on Programming Languages and Systems* **16**(5), 1512–1542.

Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J. & Zave, P. (1996), 'Formal methods: state of the art and future directions', *ACM Computing Surveys* **28**(4), 626–643.

Clarke, E. & Veith, H. (2003), Counterexamples revisited: Principles, algorithms, applications, *in* N. Derschowitz, ed., 'Verification: Theory and

Practice. Essays Delivered to Zohar Manna on the Occasion of His 64th Birthday', Springer, Berlin Heidelberg, pp. 208–224.

Clarke, E. & Wing, J. M. (1996), 'Formal methods: State-of-the-art and future directions', *ACM Comput. Surv.* **28**(4), 626–643. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

Clayberg, E. & Rubel, D. (2004), *Eclipse: Building Commercial-Quality Plug-Ins*, Addison Wesley, Reading, MA.

Coe, T., Mathisen, T., Moler, C. & Pratt, V. (1995), 'Computational aspects of the pentium affair', *IEEE Comput. Sci. Eng.* **2**(1), 18–31.

Connelly, R., Gousie, M. B., Hadimioglu, H., Ivanov, L. & Hoffman, M. (2004), 'The role of digital logic in the computer science curriculum', *Journal of Computing Sciences in Colleges* **19**, 5–8.

Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Păsrăeanu, C., Robby & Zheng, H. (2000), Bandera: Extracting finite-state models from java source code, *in* 'Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland', ACM Press, New York, pp. 439–448.

Coudert, O., Berthet, C. & Madre, J. C. (1990), Verifying temporal properties of sequential machines without building their state diagrams, *in* R. P. Kurshan & E. M. Clarke, eds, 'Proceedings of the 1990 Workshop on Computer-Aided Verification'.

Courcoubetis, C. & M. Yannakakis, M. (1988), Verifying temporal properties of finite state probabilistic programs, *in* 'Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS'88)', IEEE Computer Society Press, pp. 338–345.

Courcoubetis, C. & Yannakakis, M. (1990), Markov decision processes and regular events, *in* M. Paterson, ed., 'Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)', Vol. 443 of *Lecture Notes in Computer Science*, Springer, pp. 336–349.

Courcoubetis, C. & Yannakakis, M. (1995), 'The complexity of probabilistic verification', *J. ACM* **42**(4), 857–907.

Dam, M. (1994), 'CTL^* and $ECTL^*$ as fragments of the modal μ-calculus', *Theoretical Computer Science* **126**(1), 77–96.

Danjani-Brown, S., Cofer, D., Hartmann, G. & Pratt, S. (2003), Formal modeling and analysis of an avionics triplex sensor voter, *in* T. Ball

& S. Rajamani, eds, 'Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)', Vol. 2648 of *Lecture Notes in Computer Science*, Springer-Verlag, Portland, OR, pp. 34–48.

Daskalopulu, A. (2000), Model checking contractual protocols, *in* J. Breuker, R. Leenes & R. Winkels, eds, 'Legal Knowledge and Information Systems', JURIX 2000: The 13th Annual Conference, IOS Press, Amsterdam, pp. 35–47.

Davis, M., Logemann, G. & Loveland, D. (1962), 'A machine program for theorem-proving', *Communications of the ACM* **5**(7), 394–397.

Davis, M. & Putnam, H. (1960), 'A computing procedure for quantification theory', *Journal of the ACM* **7**(3), 201–215.

Daws, C. & Tripakis, S. (1998), Model checking of real-time reachability properties using abstractions, *in* 'Proc. of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems', Vol. 1384 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 313–329.

Daws, C. & Yovine, S. (1995), Two examples of verification of multirate timed automata with Kronos, *in* 'Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)', Pisa, Italy, pp. 66–75.

de Alfaro, L. & Henzinger, T. A. (2001), Interface theories for component-based design, *in* 'Proceedings of the First International Workshop on Embedded Software (EMSOFT)', Vol. 2211 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 148–165.

deBakker, J. & Scott, D. (1969), A theory of programs. Unpublished manuscript.

Demartini, C., Iosif, R. & Sisto, R. (1999), 'A deadlock detection tool for concurrent java programs', *Softw. Pract. Exper.* **29**(7), 577–603.

Detlefs, D., Nelson, G. & Saxe, J. B. (2003), Simplify: A theorem prover for program checking, Technical report, Systems Research Center HP Laboratories Palo Alto. HPL-2003-148.

Detlefs, D., Nelson, G. & Saxe, J. B. (2005), 'Simplify: a theorem prover for program checking', *J. ACM* **52**(3), 365–473.

Dijkstra, E. (1976), *A Discipline of Programming*, Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ.

Dijkstra, E. W. (1968), Notes on structured programming, *in* E. W. D. O.-J. Dahl & C. A. R. Hoare, eds, 'Structured Programming', Academic Press, London, pp. 1–82.

Dijkstra, E. W. (1975), 'Guarded commands, nondeterminacy and formal derivation of programs', *Comm. of the ACM* **18**(8), 453–457.

Dijkstra, E. W. (1989), In reply to comments. EWD1058.

Dill, D. L. (1996), The murϕ verification system, *in* R. Alur & T. Henzinger, eds, 'Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, New Brunswick, NJ, pp. 390–393.

Dill, D. L., Drexler, A. L., Hu, A. J. & Yang, C. H. (1992), Protocol verification as a hardware design aid, *in* 'Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computer and Processors (ICCD'92)', IEEE Computer Society, Cambridge, MA, pp. 522–525.

Drusinsky, D. (2000), The temporal rover and the ATG rover, *in* 'Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification', Vol. 1885 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 323–330.

Emerson, E. (1992), Real time and the μ-calculus, *in* I. W. de Bakker et al., ed., 'Proceedings of the REXWorkshop on Real-Time: Theory and Practice', Vol. 600 of *Lecture Notes in Computer Science*, Springer-Verlag, Mook, the Netherlands, pp. 176–194.

Emerson, E. A. (1990), Temporal and modal logic, *in* J. Leeuwen, ed., 'Handbook od Theoretical Computer Science', Vol. A, Elsvier, chapter 16, pp. 995–1072.

Emerson, E. A. (1996), Automated temporal reasoning about reactive systems, *in* 'Logics for Concurency: Structure versus Automata', Springer, pp. 41–101.

Emerson, E. A. (2008), The beginning of model checking: A personal perspective, *in* Grumberg & Veith (2008), pp. 27–45.

Emerson, E. A. & Clarke, E. M. (1980), Characterizing correctness properties of parallel programs using fixpoints, *in* 'Proceedings of the 7th Colloquium on Automata, Languages and Programming', Springer-Verlag, London, UK, pp. 169–181.

Emerson, E. A. & Halpern, J. Y. (1983), 'sometimes' and 'not never' revisited: On branching versus linear time temporal logic, *in* '10th ACM Symposium on Theory of Computing', ACM Press, New York, pp. 127–140. Por. (Emerson & Halpern 1986).

Emerson, E. A. & Halpern, J. Y. (1986), "sometimes' and 'not never' revisited: On branching versus linear time temporal logic', *Journal of ACM* **33**(1), 151–178. Por. (Emerson & Halpern 1983).

Emerson, E. A. & Jutla, C. S. (1988), The complexity of tree automata and logics of programs, *in* 'Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science', pp. 328–337.

Emerson, E. A. & Jutla, C. S. (1991), Tree automata, mu-calculus and determinacy, *in* 'SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science', IEEE Computer Society, Washington, DC, USA, pp. 368–377.

Emerson, E. A. & Jutla, C. S. (1999), 'The complexity of tree automata and logics of programs', *SIAM Journal on Computing* **29**, 132–158.

Emerson, E. A. & Lei, C.-L. (1986*a*), Efficient model checking in fragments of the propositional mu-calculus (extended abstract), *in* 'LICS', IEEE Computer Society, pp. 267–278.

Emerson, E. A. & Lei, C.-L. (1986*b*), Temporal reasoning under generalized fairness constraints, *in* '3rd Symposium on Theoretical Aspects of Computer Science', Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 21–36.

Emerson, E. A. & Lei, C.-L. (1987), 'Modalities for model checking: Branching time logic strikes back', *Sci. of Comput. Program.* **8**(3), 275–306.

Emerson, E. A. & Sistla, A. P. (1997), 'Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach', *ACM Trans. Program. Lang. Syst.* **19**(4), 617–638.

Emerson, E., Jha, S. & Peled, D. (1997), Combining partial order and symmetry to reductions, *in* E. Brinksma, ed., 'Proceedings of the 3rd International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'97)', Vol. 1217 of *Lecture Notes in Computer Science*, Springer-Verlag, Enschede, the Netherlands, pp. 19–34.

Emerson, E. & Sistla, A. P. (1996), 'Symmetry and model checking', *Formal Methods Syst. Des.* **9**(1–2), 105–131.

Enderton, H. (1972), *A Mathematical Introduction to Logic*, Academic, New York.

Esparza, J. (2003), An automata-theoretic approach to software verification, *in* Z. Ésik & Z. Fülöp, eds, 'Developments in Language Theory', Vol. 2710 of *Lecture Notes in Computer Science*, Springer, p. 21.

Floyd, R. W. (1967), Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics', Vol. 19, American Mathematical Society, Providence, pp. 19–32.

Gabbay, D. M. (1981), Expressive functional completness in tense logic, *in* U. Mönnich, ed., 'Aspects of Philosophical Logic', Dordrecht, pp. 91–117. Preliminary Report.

Gabbay, D. M. (1989), The declarative past and imperative future: Executable temporal logic for interactive systems, *in* B. Banieqbal, H. Barringer & A. Pnueli, eds, 'Proceedings of Colloquium on Temporal Logic in Specification', Vol. 398 of *Lecture Notes in Computer Science*, Springer-Verlag, Altrincham, pp. 67–89.

Gabbay, D. M., Pnueli, A., Shelah, S. & Stavi, J. (1980), On the temporal analysis of fairness, *in* '7th Annual ACM Symposium on Principles of Programming Languages (POPL'80)', Vol. 47 of *Applications of Modal Logic in Linguistics*, ACM Press, pp. 163–173.

Garavel, H. & Sifakis, J. (1990), Compilation and verification of LOTOS specifications, *in* L. L. et al., ed., 'Proceedings of the IFIP WG6.1 10th International Symposium on Protocol Specification, Testing and Verification (PSTV '90)', Ottawa, Canada, pp. 379–394.

Gerth, R., Kuiper, R., Peled, D. & Penczek, W. (1995), A partial order approach to branching time logic model checking, *in* 'Proceedings of the Third Israel Symposium on the Theory of Computing and Systems (ISTCS'95), Tel Aviv, Israel, January 4–6, 1995'.

Girault, C. & Valk, R., eds (2003). *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*, Springer-Verlag, New York.

Giunchig lia, F. & Traverso, P. (1999), Planning as model checking, *in* 'Proceedings of the Fifth European Workshop on Planning, (ECP'99)', Springer, pp. 1–20.

Glabbeek, R. J. v. (2001), The linear time – branching time spectrum I, *in* J. A. Bergstra, A. Ponse & S. A. Smolka, eds, 'Handbook of Process Algebra', Elsevier Science, chapter 4, pp. 3–99.

Godefroid, P. (1996*a*), On the costs and benefits of using partial-order methods for the verification of concurrent systems, *in* D. P. et al., ed., 'Proceedings of the DIMACS Workshop on Partial Order Methods in

Verification (POMIV'96)', Vol. 29 of *Princeton, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Boston, MA, pp. 289–303.

Godefroid, P. (1996*b*), *Partial Order Methods for the Verification of Concurrent Systems*, Vol. 1032 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Godefroid, P. (1997), Model checking for programming languages using verisoft, *in* 'Proceedings of the 24th Symposium on Principles of Programming Languages (POPL'97)', ACM Press, New York, pp. 174–186.

Goranko, V. (2000), Temporal logics of computations.

Gordon, M. & Melham, T. (1993), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press.

Grumberg, O., Heyman, T., Ifergan, N. & Schuster, A. (2005), Achieving speedups in distributed symbolic reachability analysis through asynchronous computation, *in* D. Borrione & W. Paul, eds, 'Correct Hardware Design and Verification Methods', Vol. 3725 of *Lecture Notes in Computer Science*, Berlin-Heidelberg, pp. 129–145.

Grumberg, O. & Long, D. E. (1994), 'Model checking and modular verification', *ACM Transactions on Programming Languages and Systems* **16**(3), 843–871.

Grumberg, O. & Veith, H., eds (2008), *25 Years of Model Checking - History, Achievements, Perspectives*, Vol. 5000 of *Lecture Notes in Computer Science*, Springer.

Hamblin, C. L. (1969), 'Starting and stopping', *The Monist* **53**, 410–425.

Hansson, H. & Jonsson, B. (1994), 'A logic for reasoning about time and reliability', *Formal Aspects of Computing* **6**, 512–535.

Hardin, R. H., Harel, Z. & Kurshan, R. P. (1996), COSPAN, *in* R. Alur & T. A. Henzinger, eds, 'Proceedings of the eighth International Conference on Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 423–427.

Hasle, P. F. V. & Øhrstrøm, P. (2004), Foundations of temporal logic. The WWW-site for Arthur Prior, `http://www.kommunikation.aau.dk/prior/index2.htm`.

Havelund, K. & Pressburger, T. (2000), 'Model checking java programs using java pathfinder', *Inte. J. Softw. Tools Technol. Transfer* **2**(4), 366–381.

Heath, J., Kwiatowska, M., Norman, G., Parker, D. & Tymchysyn, O. (2006), Probabalistic model checking of complex biological pathways, *in* C. Priami, ed., 'Proc. Comp. Methods in Systems Biology, (CSMB'06)', Vol. 4210 of *Lecture Notes in Bioinformatics*, Springer, pp. 32–47.

Hennessy, M. & Milner, R. (1985), 'Algebraic laws for nondeterminism and concurrency', *Journal of the Association for Computing Machinery* **32**(1), 137–161.

Henzinger, T., Ho, P. & Wong-Toi, H. (1997), 'A model checker for hybrid systems', *Int. J. Softw. Tools Technol. Transfer* **1**(1/2), 110–122.

Henzinger, T., Jhala, R., Majumdar, R. & Sutre, G. (2002), Lazy abstraction, *in* 'Proceedings of the 29th Annual Symposium on Principles of Programming Languages', ACM Press, pp. 58–70.

Henzinger, T., Jhala, R., Majumdar, R. & Sutre, G. (2003), Software verification with Blast, *in* T. Ball & S. Rajamani, eds, 'Model Checking Software: Proceedings of the 10th International Spin Workshop (Spin 2003)', Vol. 2648 of *Lecture Notes in Computer Science*, Springer-Verlag, Portland, OR, pp. 235–239.

Hermanns, H., Katoen, J.-P., Meyer-Kayser, J. & Siegle, M. (2000), A Markov chain model checker, *in* 'Tools and Algorithms for Construction and Analysis of Systems', pp. 347–362.

Hillston, J. (1996), *A Compositional Approach to Performance Modeling*, Distinguished Dissertations in Computer Science, Cambridge University Press, Cambridge, UK.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**(10), 576–580,583. Również w: (Hoare & Jones 1989, 45–58).

Hoare, C. A. R. (1996), How did software get so reliable without proof?, *in* 'Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods (FME'96)', Vol. 1051 of *LNCS*, Springer-Verlag, London, UK, pp. 1–17.

Hoare, C. A. R. & Jones, C. B. (1989), *Essays in Computing Science*, Prentice Hall.

Hoare, T. (2003), 'The verifying compiler: A grand challenge for computing research', *J. ACM* **50**, 63–69.

Holzmann, G. (1991), *Design and validation of computer protocols*, Prentice Hall, New Jersey.

Holzmann, G. (1998), 'An analysis of bitstate hashing', *Formal Methods Syst. Des.* **13**(3), 289–307.

Holzmann, G. (1999), The engineering of a model checker: The gnu i-protocol case study revisited, *in* D. D. et al., ed., 'Proceedings of the 5th and 6th International SPIN Workshops', Vol. 1680 of *Lecture Notes in Computer Science*, Springer-Verlag, Trento, Italy and Toulouse, France, pp. 232–244.

Holzmann, G. (2003), *The* SPIN *Model Model Checker: Primer and Reference Manual*, Addison Wesley, Boston, MA.

Holzmann, G. J. (1997), 'The model checker SPIN', *IEEE Transactions on Software Engineering* **23**(5), 279–295.

Holzmann, G. J. (2001), Economics of software verification, *in* 'Proc. Workshop on Program Analysis for Software Tools and Engineering', ACM, Snowbird, Utah, USA.

Holzmann, G. J. (2002), Software analysis and model checking, *in* 'CAV', pp. 1–16.

Holzmann, G. J. (2004), *The* SPIN *Model Checker: Primer and Reference Manual*, Addison-Wesley.

Holzmann, G. J. & Peled, D. (1996), The state of *SPIN*, *in* R. Alur & T. A. Henzinger, eds, 'Proceedings of the eighth International Conference on Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 385–389.

Holzmann, G. J. & Smith, M. H. (2002), FEAVER 1.0 user guide, Technical report, Bell Labs. 64 pgs.

Holzmann, G. & Peled, D. (1994), An improvement in formal verification, *in* D. Hogrefe & S. Leuse, eds, 'Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE'94)', Vol. 6, Berne, Switzerland. International Federation for Information Processing, Chapman and Hall, London, UK, pp. 197–211.

Holzmann, G. & Smith, M. (1999*a*), A practical method for the verification of event-driven software, *in* 'Proceedings of the 21st International Conference on Software engineering (ICSE '99), Los Angeles, CA', ACM Press, New York, pp. 597–607.

Holzmann, G. & Smith, M. (1999*b*), Software model checking. Extracting verification models from source code, *in* J. W. et al., ed., 'Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)', Vol. 156, International Federation for Information Processing, Kluwer, Beijing, China, pp. 481–497.

Huber, P., Jenson, A., Jepson, L. & Jenson, K. (1985), Towards reachability trees for high-level Petri nets, *in* G. R. et al., ed., 'Proceedings of the European Workshop on Applications and Theory in Petri Nets', Vol. 188 of *Lecture Notes in Computer Science*, Springer-Verlag, Aarhus, Denmark, pp. 215–233.

Hughes, G. E. & Cresswell, M. J. (1968), *An Introduction to Modal Logic*, Methuen and Co., London.

Huth, M. & Ryan, M. (2004), *Logic in Computer Science. Modelling and Reasoning about Systems*, 2 edn, Cambridge University Press, Cambridge. I ed. 1999.

Iosif, R. & Sisto, R. (1999), dspin: A dynamic extension of spin, *in* D. D. et al., ed., 'Proceedings of the 5th and 6th International SPIN Workshops', Vol. 1680 of *Lecture Notes in Computer Science*, Springer-Verlag, Trento, Italy and Toulouse, France, pp. 20–33.

Ip, C. & Dill, D. (1996), 'Better verification through symmetry', *Formal Methods in Syst. Des.* **9**, 41–75.

Kaminski, M. (1994), 'A branching time logic with past operators', *Journal of Computer and System Sciences* **49**(2), 223–246.

Kamp, J. A. W. (1968), Tense Logic and the Theory of Linear Order, Phd thesis, University of California, Los Angeles.

Katoen, J.-P., Khattri, M. & Zapreev, I. S. (2005), A Markov reward model checker, *in* 'Quantitative Evaluation of Systems (QEST)', pp. 243–244.

Kaufmann, M. & Moore, J. S. (2004), 'Some key research problems in automated theorem proving for hardware and software verification', *Rev. R. Acad. Cien. Serie A. Mat.* **98**(1), 181–196.

Kautz, H. & Selman, B. (1992), Planning as satisfiability, *in* 'ECAI '92: Proceedings of the 10th European conference on Artificial intelligence', John Wiley & Sons, Inc., New York, NY, USA, pp. 359–363.

Kolmogorov, A. N. (1936), 'Anfangsgründe der Theorie der Markoffschen Ketten mit unendlichen vielen möglichen Zuständen', *Matematicheskii Sbornik* pp. 607–610.

Kozen, D. (1982), Results on the propositional μ-calculus, *in* M. Nielsen & E. M. Schmidt, eds, 'Automata, Languages and Programming. Ninth Colloquium 1982', Vol. 140 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 348–359.

Kozen, D. (1983), 'Results on the propositional μ-calculus', *Theoretical Computer Science* **27**(3), 333–354.

Kröger, F. (1977), 'A logic of algorithmic reasoning', *Acta Informatica* **8**(3), 243–266.

Kröger, F. (1987), *Temporal Logic of Programs*, Springer-Verlag New York, Inc., New York, NY, USA.

Kröger, F. & Merz, S. (1991), 'Temporal logic and recursion', *Fundam. Inform.* **14**(2), 261–281.

Kröger, F. & Merz, S. (2008), *Temporal Logic and State Systems*, Springer.

Kumar, S. & Li, K. (2002), Using model checking to debug device firmware, *in* 'Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002)', USENIX, Boston, MA.

Kuntz, M., Siegle, M. & Werner, E. (2004), *Symbolic performance and dependability evaluation with the tool CASPA*.

Kupferman, O. & Pnueli, A. (1995), Once and for all, *in* 'Proc. 10th IEEE Symposium on Logic in Computer Science', San Diego, pp. 25–35.

Kupferman, O. & Vardi, M. Y. (1995), On the complexity of branching modular model checking, *in* 'Proceedings of 6th International Conference on Concurrency Theory (CONCUR '95)', Vol. 962 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 408–422.

Kupferman, O. & Vardi, M. Y. (1996), On the complexity of branching modular model checking, *in* 'Proceedings of the eighth International Conference on Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 75–86.

Kupferman, O. & Vardi, M. Y. (1997), Module checking revisited, *in* 'Proceedings of the ninth International Conference on Computer Aided Verification (CAV '97)', Vol. 1254 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 36–47.

Kupferman, O. & Vardi, M. Y. (1998), Modular model checking, *in* 'Compositionality: the significant difference.', Vol. 1536 of *Lecture Notes in Computer Science*, International symposium, Bad Malente, Springer-Verlag, pp. 381–401.

Kupferman, O., Vardi, M. Y. & Wolper, P. (2000), 'An automata-theoretic approach to branching-time model checking', *Journal of the ACM* **47**(2), 312–360.

Kurshan, R. (1995), *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Series in Computer Science, Princeton University Press, Princeton, NJ.

Kwiatkowska, M., Norman, G. & Parker, D. (2001), *PRISM*: Probabilistic symbolic model checker, *in* P. Kemper, ed., 'Proc. Tools Session of Aachen 2001', International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, Dortmund, pp. 7–12. Available as Technical Report 760/2001, University of Dortmund.

Kwiatkowska, M., Norman, G. & Parker, D. (2002*a*), Probabilistic symbolic model checking with *PRISM*: A hybrid approach, *in* J.-P. Katoen & P. Stevens, eds, 'Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)', Vol. 2280 of *Lecture Notes in Computer Science*, Springer, pp. 56–66.

Kwiatkowska, M., Norman, G. & Parker, D. (2002*b*), Probabilistic symbolic model checking with PRISM, *in* J. Katoen & P. Stevens, eds, 'Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)', Vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag, Grenoble, France, pp. 52–66. Held as part of the Joint European Conference on Theory and Practice of Software (ETAPS 2002).

Lamport, L. (1980), 'Sometimes' is sometimes 'not never': on the temporal logic of programs, *in* 'Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, New York, pp. 174–185.

Lamport, L. (1983), What good is temporal logic?, *in* 'Information Processing'83. Proc. IFIP 9th World Computer Congress', North-Holland, pp. 657–668.

Lamport, L. (1994), 'The temporal logic of actions', *ACM Transactions on Programming Languages and Systems* **16**(3), 872–923.

Laroussinie, F. (1995), 'About the expressive power of *CTL* combinators', *Information Processing Letters* **54**(6), 343–345.

Laroussinie, F., Markay, N. & Schnoebelen, P. (2002), Temporal logic with forgettable past, *in* '17th IEEE Symposium on Logic in Computer Science (LICS)', IEEE Computer Society Press, pp. 383–392.

Laroussinie, F., Markey, N. & Schnoebelen, P. (2004), Model checking timed automata with one or two clocks, *in* 'Proc. of the 15th Int. Conf. on Concurrency Theory (CONCUR'04)', Vol. 3170 of *Lecture Notes in Computer Science*, Springer, pp. 387–401.

Laroussinie, F., Markey, N. & Schnoebelen, P. (2005), 'Efficient timed model checking for discrete time systems', *Theo. Comp. Sci.* **353**(1–3), 249–271.

Laroussinie, F. & Schnoebelen, P. (1994), A hierarchy of temporal logics with past, *in* 'Proc. STACS'94, Caen, France', Vol. 775 of *LNCS*, Springer-Verlag, pp. 47–58.

Laroussinie, F. & Schnoebelen, P. (2000), 'Specification in $CTL + Past$ for verificatin in *CTL*', *Information and Computation* **156**, 236–263].

Larsen, K. G., Pettersson, P. & Yi, W. (1997), 'Uppaal in a nutshell', *International Journal on Software Tools for Technology Transfer* **1**(1–2), 134–152.

Larson, K., Pettersson, P. & Yi, W. (1995), Model-checking for real-time systems, *in* H. Reichel, ed., 'Proceedings of the 10th International Symposium on the Fundamentals of Computation Theory (FCT'95)', Vol. 965 of *Lecture Notes in Computer Science*, Springer-Verlag, Dresden, Germany, pp. 62–88.

Larson, K., Pettersson, P. & Yi, W. (1997), 'Uppaal in a nutshell', *Int. J. Softw. Tools. Technol. Transfer* **1**(1/2), 134–152.

Lasota, S. & Walukiewicz, I. (2005), Alternating timed automata, *in* 'Proc. of the 8th Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'05)', Vol. 3441 of *LNCS*, Springer, pp. 299–314.

Lee, C. (1959), 'Representation of switching circuits by binary-decision programs', *Bell System Technical Journal* **38**, 985–999.

L'eonard, L. & Leduc, G. (1997), 'An introduction to et-lotos for the description of time-sensitive systems', *Comput. Netw. ISDN Syst.* **29**(3), 271–292.

L'eonard, L. & Leduc, G. (1998), 'A formal definition of time in lotos', *Formal Aspects Comput.* **10**(3), 248–266.

Lichtenstein, O. & Pnueli, A. (1985), Checking that finite state concurrent programs satisfy their linear specification, *in* 'Proc. 12th ACM Symp. Principles of Programming Languages (POPL'85)', ACM Press, New York, pp. 97–107.

Lichtenstein, O. & Pnueli, A. (2000), 'Propositional temporal logics: Decidability and completeness', *Journal of the IGPL* **8**(1), 55–85.

Lichtenstein, O., Pnueli, A. & Zuck, L. D. (1985), The glory of the past, *in* R. Parikh, ed., 'Proceedings 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985', Vol. 193 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 196–218.

Lions, J. L. et al. (1996), ARIANE-5; Flight 501 Failure, Technical report, Report by the Inquiry Board. The Chairman of the Board: Prof. J. L. LIONS,
`http://www.cs.unibo.it/~laneve/papers/ariane5rep.html`.

Lipton, R. J. (1975), 'Reduction: A method of proving properties of parallel programs', *Communications of the ACM* **18**(12), 717–721.

M. Kaufmann, M., Manolios, P. & Moore, J. S. (2000), *Computer-Aided Reasoning: An Approach*, Kluwer Academic Press, Boston.

Manna, Z. & A. Pnueli, A. (1992, 1995*a*), *The Temporal Logic of Reactive and Concurrent Systems*, Vol. 1: Specification, 2: Safety, Springer-Verlag, New York.

Manna, Z. & A. Pnueli, A. (1995*b*), *The Temporal Logic of Reactive and Concurrent Systems: Safety*, Springer-Verlag, New York.

Manna, Z., Bjørner, N., Browne, A., Chang, E., Alfaro, L. D., Devarajan, H., Kapur, A., Lee, J. & Sipma, H. (1994), STeP: The stanford temporal prover, Technical report, Computer Science Department, Stanford University Stanford, CA.

Manna, Z. & Pnueli, A. (1981), Verification of concurrent programs: The temporal framework, *in* R. Boyer & J. Moore, eds, 'The Correctness Problem in Computer Science', Academic Press, London, pp. 215–273.

Manna, Z. & Pnueli, A. (1989), The anchored version of the temporal framework, *in* 'Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency', Vol. 354 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 201–284.

Manna, Z. & Waldinger, R. (1985), *The Logical Basis for Computer Programming*, Addison-Wesley.

Markov, A. A. (1907), 'Investigations of an important case of dependent trials', *Izvestia Acad., Nauk VI, Series I* **61**.

Mateescu, R. (2003), On-the-fly verification using Cadp, *in* T. Arts & W. Fokkink, eds, 'Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003)', Vol. 80 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Trondheim, Norway, pp. 1–5.

Mateescu, R. & Garavel, H. (1998), Xtl: A metalanguage and tool for temporal logic model-checking, *in* T. Margaria & B. Steffen, eds, 'Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT '98)', Aalborg, Denmark.

McMillan, K. (2002), Lazy abstraction with interpolants, *in* E. Brinksma & K. G. Larsen, eds, 'Computer Aided Verification. Abstraction/Refinement', Vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 123–136.

McMillan, K. L. (1993), *Symbolic Model Checking: An approach to the State Explosion Problem*, Kluwer Academic, Hingham, MA.

McMillan, K. L. & Schwalbe, J. (1992), Formal specification of the gigamax cache consistency protocols, *in* N. Suzuki, ed., 'Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors', Information Processing Society of Japan, MIT Press, Tokyo, pp. 242–251.

Miller, A., Donaldson, A. & Calder, M. (2006), 'Symmetry in temporal logic model checking', *ACM Computing Surveys* **38**(3).

Mishra, B. & Clarke, E. M. (1985), 'Automatic and hierarchical verification of asynchronous circuits using temporal logic', *Theoretical Computer Science* **38**, 269–291.

Müller-Olm, M., Schmidt, D. & Steffen, B. (1999), Model-checking: A tutorial introduction, *in* A. Cortesi & G. File, eds, 'Proceedings of the 6th International Static Analysis Symposium (SAS'99)', Vol. 1694 of *Lecture Notes in Computer Science*, Springer-Verlag, Venice, pp. 330–354.

Naur, P. (1966), 'Proof of algorithms by general snapshots', *BIT* **6**(4), 310–316.

Nicollin, X. & Sifakis, J. (1994), 'Atp: Theory and application', *Inf. Comput.* **114**, 131–178.

Owicki, S. S. & Lamport, L. (1982), 'Proving liveness properties of concurrent programs', *ACM Trans. Program. Lang. Syst.* **4**(3), 455–495.

Park, D. (1981), Concurrency and automata on infinite sequences, *in* P. Deussen, ed., 'Theoretical Computer Science', Vol. 104 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 167–183.

Peled, D. (1996*a*), 'Combining partial order reductions with on-the-fly model checking', *Formal Methods Syst. Des.* **8**, 39–64.

Peled, D. (1996*b*), Partial order reduction: Linear and branching temporal logics and process algebras, *in* D. P. et al., ed., 'Proceedings of the DIMACS Workshop on Partial Order Methods in Verification (POMIV'96)', Vol. 29 of *Princeton, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Boston, MA, pp. 233–257.

Petcu, D. (2003), Parallel explicit state reachability analysis and state space construction, *in* 'Second International Symposium on Parallel and Distributed Computing', pp. 207–214.

Pnueli, A. (1977), The temporal logic of programs, *in* 'Proceedings of the 18th IEEE-CS Symposium on Foundation of Computer Science (FOCS-77)', IEEE Computer Society Press, pp. 46–57.

Pnueli, A. (1981), 'The temporal semantics of concurrent programs', *Theoretical Comput. Sci.* **13**, 45–60.

Pnueli, A. (1985*a*), In transition from global to modular temporal reasoning about programs, *in* 'Logic and Models of Concurrent Systems', Vol. F-13, NATO Advanced Summer Institutes, Springer Verlag, pp. 123–144.

Pnueli, A. (1985*b*), Linear and branching structures in the semantics and logics of reactive systems, *in* 'Proceedings of the 12th ICALP', pp. 15–32.

Pnueli, A. & Zuck, L. (1993), 'Probabilistic verification', *Information and Computation* **103**, 1–29.

Post, H. & Küchlin, W. (2006), Automatic data environment construction for static device drivers analysis, *in* 'SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems', ACM, New York, NY, USA, pp. 89–92.

Prasad, M. R., Biere, A. & Gupta, A. (2005), 'A survey of recent advances in *sat*-based formal verification', *International Journal on Software Tools for Technology Transfer (STTT)* **7**(2), 156–173.

Pratt, V. R. (1980), 'Applications of modal logic to programming', *Studia Logica* **9**, 257–274.

Pratt, V. R. (1981), A decidable μ-calculus: Preliminary report, *in* '22nd Annual Symposium on Foundations of Computer Science', IEEE, Nashville, Tennessee, pp. 421–427.

Presburger, M. (1929), Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt, *in* F. Leja, ed., 'Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich', Skład Głowny, Warszawa, pp. 92–101.

Prior, A. N. (1967), *Past, Present and Future*, Oxford University Press.

Prior, A. N. (1996), A statement of temporal realism, *in* B. J. Copeland, ed., 'Logic and Reality: Essays on the Legacy of Arthur Prior', Oxford University Press.

Queille, J.-P. & Sifakis, J. (1982), Specification and verification of concurrent systems in CESAR, *in* 'Proceedings 5th International Symposium on Programming', Vol. 137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 337–351.

Quielle, J. & Sifakis, J. (1982), Specification and verification of concurrent systems in CÆSAR, *in* Grumberg & Veith (2008), pp. 195–220.

Randell, B. (1973), *The Origin of Digital Computers*, Springer Verlag.

Reisig, W. (1989), Towards a temporal logic for causality and choice in distributed systems, *in* 'Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency', Vol. 354 of *Lect. Notes in Comp. Sci.*, Springer, pp. 603–627.

Rescher, N. & Urquhart, A. (1971), *Temporal Logic*, Springer, Wien, New York.

Reynolds, M. (2005), 'An axiomatization of PCTL*', *Information and Computation* **201**(1), 72–119.

Rinard, M. & Diniz, P. (1997), 'Commutativity analysis: A new analysis technique for parallelizing compilers', *ACM Transactions on Programming Languages and Systems* **19**(6), 1–47.

Robby, Dwyer, M. & Hatcliff, J. (2003), Bogor: An extensible and highly-modular model checking framework, *in* 'Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, Helsinki, Finland', ACM Press, New York, pp. 267–276. Held jointly with the 9th European Software Engineering Conference (ESEC/FSE 2003).

Rutten, J., Kwiatkowska, M., Norman, G. & Parker, D. (2004), *Mathematical Techniques for Analysing Concurrent and Probabilisitic Systems*, Vol. 23 of *American Mathematical Society, CRM Monograph Series*, Centre de Recherches Mathématiques, Université de Montréal.

Schneider, K. (2003), *Verification of Reactive Systems. Formal Methods and Algorithms*, Texts in Theoretical Computer Science (EATCS Series), Springer-Verlag.

Schnoebelen, P. (2002), 'The complexity of temporal logic model checking', *Advances in Modal Logic* **4**, 1–44.

Schuele, T. & Schneider, K. (2004), Global vs. local model checking: A comparison of verification techniques for infinite state systems, *in* 'SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference', IEEE Computer Society, Washington, DC, USA, pp. 67–76.

Schuele, T. & Schneider, K. (2007), 'Bounded model checking of infinite state systems', *Form. Methods Syst. Des.* **30**(1), 51–81.

Shurek, G. & Grumberg, O. (1990), The modular framework of computer-aided verification: Motivation, solution and evaluation criteria, *in* 'Proceedings of the 1990 Workshop on Computer-Aided Verification'.

Sipma, O., Uribe, H. & Manna, Z. (1996), Deductive model checking, *in* 'International Conference on Computer Aided Verification', Vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219.

Sistla, A. P. & Clarke, E. M. (1985), 'The complexity of propositional linear temporal logics', *Journal of the ACM* **32**(3), 733–749.

Somenzi, F. (1997), Cudd: Cu decision diagram package, Technical report, Public software, `http://vlsi.colorado.edu/`.

Starke, P. H. (1991), 'Reachability analysis of Petri nets using symmetries', *Syst. Anal. Model. Simul.* **8**(5/5), 293–303.

Szałas, A. (1995), 'Temporal logic of programs: a standard approach', pp. 1–50.

Thomas, W. (1990), Automata on infinite objects, *in* J. Leeuwen, ed., 'Handbook of Theoretical Computer Science', Vol. B: Formal Models and Semantics, Elsevier, pp. 165–191.

Thomas, W. (1997), Languages, automata, and logic, *in* 'Handbook of Formal Languages', Springer, pp. 389–455.

Turing, A. M. (1936–37), 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society* **42**(Series 2), 230–265. Received May 25, 1936; Appendix added August 28; read November 12, 1936; corrections Ibid. vol. 43(1937), pp. 544–546. Turing's paper appeared in Part 2 of vol. 42 which was issued in December 1936 (Reprint in: (Turing 1965); 151–154). Online version: `http://www.abelard.org/turpap2/tp2-ie.asp`.

Turing, A. M. (1950), 'Computing machinery and intelligence', *Mind* **49**, 433–460. Available online: `http://cogprints.org/499/00/turing.html`. Reprinted in (Turing 1992).

Turing, A. M. (1965), On computable numbers, with an application to the Entscheidungsproblem, *in* M. Davis, ed., 'The Undecidable', Raven Press, Hewlett, NY, pp. 116–151.

Turing, A. M. (1992), *Collected Works of A.M. Turing: Mechanical Intelligence*, North Holland, Amsterdam.

Vaandrager F. W.and De Nicola, R. (1990), Actions versus state based logics for transition systems, *in* 'Proc. Ecole de Printemps on Semantics of Concurrency', Vol. 469 of *Lecture Notes in Computer Science*, Springer, pp. 407–419.

Valmari, A. (1992), 'A stubborn attack on state explosion', *Formal Methods Syst. Des.* **1**, 297–322.

Vardi, M. & Wolper, P. (1986*a*), An automata-theoretic approach to automatic program verification (preliminary report), *in* 'Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, Cambridge, MA, pp. 332–344.

Vardi, M. & Wolper, P. (1994), 'Reasoning about infinite computations', *Inf. Comput.* **115**, 1–37.

Vardi, M. Y. (1985), Automatic verification of probabilistic concurrent finite-state programs, *in* 'Proc. 26th IEEE Symp. on Foundations of Computer Science', Portland, pp. 327–338.

Vardi, M. Y. (1989), Unified verification theory, *in* 'Temporal Logic in Specification', Vol. 398 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 202–212.

Vardi, M. Y. (1995), An automata-theoretic approach to linear temporal logic, *in* 'Banff Higher Order Workshop', pp. 238–266.

Vardi, M. Y. (2001), Branching vs. linear time: Final showdown, *in* 'Proceedings of the 7th International Conference on Tools and Algorithm for the Construction and Analysis of Systems TACAS'01', pp. 1–22.

Vardi, M. Y. & Stockmeyer, L. (1985), Improved upper and lower bounds for modal logics of programs, *in* 'Proceedings of the 17th Annual ACM Symposium on the Theory of Computing', ACM, pp. 240–251.

Vardi, M. Y. & Wolper, P. (1986*b*), 'Automata-theoretic techniques for modal logics of programms', *Journal of Computer and System Science* **32**(2), 183–221.

Varpaaniemi, K., Halme, J., Hiekkanen, K. & Pyssysalo, T. (1995), PROD reference manual, Tech. Rep. B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland.

Vergauwen, B. & Lewi, J. (1993), A linear local model checking algorithm for *ctl*, *in* E. Best, ed., 'Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)', Vol. 715 of *Lecture Notes in Computer Science*, Hildesheim, Germany, pp. 447–461.

Visser, W. & Barringer, H. (1996), Memory efficient state storage in SPIN, *in* J.-C. G. et al., ed., 'Proceedings of the 2nd Workshop on the SPIN Verification System, Rutgers University, NJ', Vol. 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Boston, MA, pp. 185–203.

Visser, W., Havelund, K., Brat, G. & Park, S. (2000), Model checking programs, *in* P. Alexander & P. Flener, eds, 'Proceedings of the 15th IEEE Conference on Automated Software Engineering (ASE-2000)', IEEE Computer Society Press, Grenoble, France, pp. 3–12.

Walukiewicz, I. (1995), Completeness of kozen's axiomatisation of the propositional μ-calculus, *in* 'Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society, San Diego, California, pp. 14–24.

Walukiewicz, I. (1996), 'A note on the completeness of kozen's axiomatisation of the propositional μ-calculus', *The Bulletin of Symbolic Logic* **2**(3), 349–366.

Wang, W., Hidvegi, Z., Bailey, A. & Whinston, A. (2000), 'E-process design and assurance using model checking', *IEEE Computer* **33**(10), 48–53.

Willems, B. & Wolper, P. (1996), Partial-order methods for model checking: From linear time to branching time, *in* 'Logic in Computer Science', pp. 294–303.

Williams, G. (1992), 'A shy blend of logic, maths and languages (Obituary of Charles Hamblin)', *Sydney Morning Herald* .

Wolper, P. (1983), 'Temporal logic can be more expressive', *Information and Control* **56**(1–2), 72–93.

Wolper, P. (1995), On the relation of program and computation to models of temporal logic, *in* L. Bolc & A. Szałas, eds, 'Time and Logic. A Computational Approach', UCL Press Limited, pp. 131–178.

Wolper, P. & Leroy, D. (1993), Reliable hashing without collision detection, *in* C. Courcoubetis, ed., 'Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)', Vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, Elounda, Greece, pp. 59–70.

Wolper, P., Vardi, M. Y. & Sistla, A. P. (1983), Reasoning about infinite computation paths, *in* 'Proc. 24th IEEE Symposium on Foundations of Computer Science', IEEE Press, Tucson, AZ, pp. 185–194.

Yi, W., Pettersson, P. & Daniels, M. (1994), Automatic verification of real-time communicating systems by constraint-solving, *in* D. Hogrefe & S. Leue, eds, 'Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)', Vol. 6, International Federation for Information Processing, Berne, Switzerland, Chapman and Hall, London, UK, pp. 243–258.

Yovine, S. (1997), 'Kronos: A verification tool for real-time systems', *Int. J. Softw. Tools Technol. Transfer* **1**(1/2), 123–133.

Yuen, C. & Tloe (2001), Modeling and verifying a price model for congestion control in computer networks using *promela*/SPIN, *in* M. Dwyer, ed., 'Proceedings of the 8th International SPIN Workshop (SPIN 2001)', Vol. 2057 of *Lecture Notes in Computer Science*, Springer-Verlag, Toronto, Canada, pp. 272–287.

Zanardo, A. & Carmo, J. (1993), 'Ockhamist computational logic: Past-sensitive necessitation in *CTL*', *J. Logic Computat.* **3**(3), 249–268.

Zhang, L. & Malik, S. (2002), The quest for efficient boolean satisfiability solvers, *in* E. Brinksma & K. G. Larsen, eds, 'Proceedings of the 14th International Conference on Computer Aided Verification (CAV)', Vol. 2404 of *Lecture Notes in Computer Science*, Springer, pp. 17–36.

Zuck, L. (1986), Past Temporal Logic, PhD thesis, Weizmann Institute.

Kazimierz Trzęsicki
Chair of Logic, Informatics and Philisophy of Science
University of Białystok
ul. Sosnowa 64,
15–887 Białystok, Poland
kasimir@ii.uwb.edu.pl

Florentin Smarandache
University of New Mexico, USA

Vic Christianto
SciPrint.org administrator

n-ARY FUZZY LOGIC AND NEUTROSOPHIC LOGIC OPERATORS

We extend Knuth's 16 Boolean binary logic operators to fuzzy logic and neutrosophic logic binary operators. Then we generalize them to n–ary fuzzy logic and neutrosophic logic operators using the smarandache codification of the Venn diagram and a defined vector neutrosophic law. In such way, new operators in neutrosophic logic/set/probability are built.

Keywords: binary/trinary/n-ary fuzzy logic operators, T-norm, T-conorm, binary/trinary/n-ary neutrosophic logic operators, N-norm, N-conorm

Introduction

For the beginning let's consider the Venn Diagram of two variables x and y, for each possible operator, as in Knuth's table, but we adjust this table to the Fuzzy Logic (FL).

Let's denote the fuzzy logic values of these variables as

$$FL(x) = (t_1, f_1)$$

where

t_1 = truth value of variable x,

f_1 = falsehood value of variable x,

with $0 \le t_1,\ f_1 \le 1$ and $t_1 + f_1 = 1$;

and similarly for y:

$$FL(y) = (t_2, f_2)$$

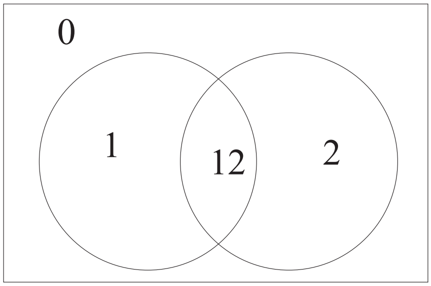
with the same $0 \le t_2,\ f_2 \le 1$ and $t_2 + f_2 = 1$.

We can define all 16 Fuzzy Logical Operators with respect to two FL operators: FL conjunction (FLC) and FL negation (FLN).

Since in FL the falsehood value is equal to 1-truth value, we can deal with only one component: the truth value.

ISBN 978–83–7431–231–8 ISSN 0860-150X

The Venn Diagram for two sets X and Y



has $2^2 = 4$ disjoint parts:

$0 =$ the part that does not belong to any set (the complement or negation)
$1 =$ the part that belongs to 1^{st} set only;
$2 =$ the part that belongs to 2^{nd} set only;
$12 =$ the part that belongs to 1^{st} and 2^{nd} set only;
{called Smarandache's codification [1]}.

Shading none, one, two, three, or four parts in all possible combinations will make $2^4 = 2^{2^2} = 16$ possible binary operators.

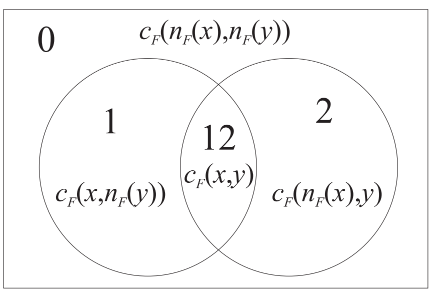
We can start using a $T - norm$ and the negation operator.

Let's take the binary conjunction or intersection (which is a $T - norm$) denoted as $c_F(x, y)$:

$$c_F : ([0,1] \times [0,1])^2 \to [0,1] \times [0,1]$$

and unary negation operator denoted as $n_F(x)$, with:

$$n_F : [0,1] \times [0,1] \to [0,1] \times [0,1]$$



The fuzzy logic value of each part is:

$P12 = part12 =$ intersection of x and y; so $FL(P12) = c_F(x, y)$.

$P1 = part1 =$ intersection of x and negation of y; $FL(P1) = c_F(x, n_F(y))$.

$P2 = part2 =$ intersection of negation of x and y; $FL(P2) = c_F(n_F(x), y)$.

$P0 = part0 =$ intersection of negation of x and the negation of y; $FL(P0) = c_F(n_F(x), n_F(y))$.

and for normalization we set the condition:

$$c_F(x, y) + c_F((n_F(x), y) + c_F(x, n_F(y)) + c_F(n_F(x), n_F(y)) = (1, 0).$$

Then consider a binary $T - conorm$ (disjunction or union), denoted by $d_F(x, y)$:

$$d_F : ([0, 1] \times [0, 1])^2 \to [0, 1] \times [0, 1]$$

$$d_F(x, y) = (t_1 + t_2, f_1 + f_2 - 1)$$

if x and y are disjoint and $t_1 + t_2 \leq 1$.

This fuzzy disjunction operator of disjoint variables allows us to add the fuzzy truth-values of disjoint parts of a shaded area in the below table. When the truth-value increases, the false value decreases. More general, $d_F^k(x_1, x_2, \ldots, x_k)$, as a k-ary disjunction (or union), for $K \geq 2$, is defined as:

$$d_F^k : ([0, 1] \times [0, 1])^k \to [0, 1] \times [0, 1]$$

$$d_F^k(x_1, x_2, \ldots, x_k) = (t_1 + t_2 + \ldots + t_k, f_1 + f_2 + \ldots + f_k - k - 1)$$

if all x_i are disjoint two by two and $t_1 + t_2 + \ldots + t_k \leq 1$.

As a particular case let's take as a binary fuzzy conjunction:

$$c_F(x, y) = (t_1 t_2, f_1 + f_2 - f_1 f_2)$$

and as unary fuzzy negation:

$$n_F(x) = (1 - t_1, 1 - f_1) = (f_1, t_1),$$

where

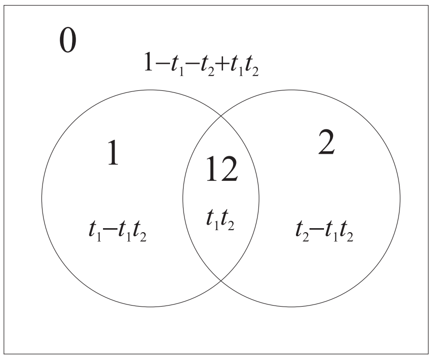
$FL(x) = (t_1, f_1)$, with $t_1 + f_1 = 1$, and $0 \leq t_1,\ f_1 \leq 1$;

$FL(y) = (t_2, f_2)$, with $t_2 + f_2 = 1$, and $0 \leq t_2,\ f_2 \leq 1$;

whence:

$$FL(P12) = (t_1t_2, f_1 + f_2 - f_1f_2)$$
$$FL(P1) = (t_1f_2, f_1 + t_2 - f_1t_2)$$
$$FL(P2) = (f_1t_2, t_1 + f_2 - t_1f_2)$$
$$FL(P0) = (f_1f_2, t_1 + t_2 - t_1t_2)$$

The Venn Diagram for $n = 2$ and considering only the truth values, becomes:



since

$$t_1f_2 = t_1(1 - t_2) = t_1 - t_1t_2$$
$$f_1t_2 = (1 - t_1)t_2 = t_2 - t_1t_2$$
$$f_1f_2 = (1 - t_1)(1 - t_2) = 1 - t_1 - t_2 + t_1t_2$$

We now use:

$$d_F^k(P12, P1, P2, P0) = ((t_1t_2) + (t_1 - t_1t_2) + (t_2 - t_1t_2) + (1 - t_1 - t_2 + t_1t_2),$$
$$(f_1 + f_2 - f_1f_2) + (f_1 + t_2 - f_1t_2) + (t_1 + f_2 - t_1f_2) + (t_1 + t_2 - t_1t_2) - 3) = (1, 0).$$

So, the whole fuzzy space is normalized under $FL(\cdot)$.

For the neurosophic logic, we consider

$$NL(x) = (T_1, I_1, F_1), \text{ with } 0 \leq T_1, I_1, F_1 \leq 1;$$
$$NL(y) = (T_2, I_2, F_2), \text{ with } 0 \leq T_2, I_2, F_2 \leq 1;$$

if the sum of components is 1 as in Atanassov's intuitionist fuzzy logic, i.e. $T_i + I_i + F_i = 1$, they are considered *normalized*; otherwise *non-normalized*, i.e. the sum of the components is <1 (*sub-normalized*) or >1 (*over-normalized*).

We define a binary neutrosophic conjunction (intersection) operator, which is a particular case of an $N - norm$ (neutrosophic norm, a generalization of the fuzzy $t - norm$):

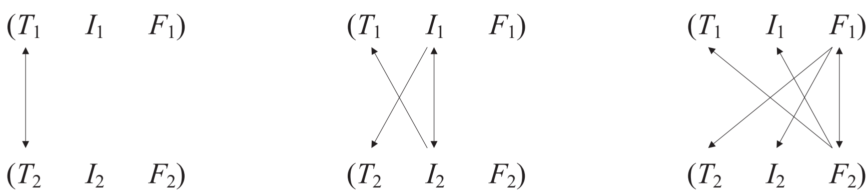
$$c_N : ([0,1] \times [0,1] \times [0,1])^2 \to [0,1] \times [0,1] \times [0,1]$$

$$c_N(x, y) = (T_1T_2, I_1I_2 + I_1T_2 + T_1I_2, F_1F_2 + F_1I_2 + F_1T_2 + F_2T_1 + F_2I_1).$$

The neutrosophic conjunction (intersection) operator $x \wedge_N y$ component truth, indeterminacy, and falsehood values result from the multiplication

$$(T_1 + I_1 + F_1) \cdot (T_2 + I_2 + F_2)$$

since we consider in a prudent way $T \prec I \prec F$, where "$\prec$" means "weaker", i.e. the products T_iI_j will go to I, T_iF_j will go to F, and I_iF_j will go to F (or reciprocally we can say that F prevails in front of I and of T,



So, the truth value is T_1T_2, the indeterminacy value is $I_1I_2 + I_1T_2 + T_1I_2$ and the false value is $F_1F_2 + F_1I_2 + F_1T_2 + F_2T_1 + F_2I_1$. The norm of $x \wedge_N y$ is $(T_1+I_1+F_1)\cdot(T_2+I_2+F_2)$. Thus, if x and y are normalized, then $x \wedge_N y$ is also normalized. Of course, the reader can redefine the neutrosophic conjunction operator, depending on application, in a different way, for example in a more optimistic way, i.e. $I \prec T \prec F$ or T prevails with respect to I, then we get:

$$c_N^{ITF}(x, y) = (T_1T_2 + T_1I_2 + T_2I_1, I_1I_2, F_1F_2 + F_1I_2 + F_1T_2 + F_2T_1 + F_2I_1).$$

Or, the reader can consider the order $T \prec F \prec I$, etc.

Let's also define the unary neutrosophic negation operator:

$$n_N : [0,1] \times [0,1] \times [0,1] \to [0,1] \times [0,1] \times [0,1]$$

$$n_N(T, I, F) = (F, I, T)$$

by interchanging the truth T and falsehood F vector components. Then:

$$NL(P12) = (T_1T_2, I_1I_2 + I_1T_2 + I_2T_1, F_1F_2 + F_1I_2 + F_1T_2 + F_2T_1 + F_2I_1)$$

$$NL(P1) \ = (T_1F_2, I_1I_2 + I_1F_2 + I_2T_1, F_1T_2 + F_1I_2 + F_1F_2 + T_2T_1 + T_2I_1)$$

$$NL(P2) \ = (F_1T_2, I_1I_2 + I_1T_2 + I_2F_1, T_1F_2 + T_1I_2 + T_1T_2 + F_2F_1 + F_2I_1)$$

$$NL(P0) \ = (F_1F_2, I_1I_2 + I_1F_2 + I_2F_1, T_1T_2 + T_1I_2 + T_1F_2 + T_2F_1 + T_2I_1)$$

Similarly as in our above fuzzy logic work, we now define a binary $N-$ *conorm* (disjunction or union), i.e. neutrosophic conform.

$$d_N : ([0,1] \times [0,1] \times [0,1])^2 \to [0,1] \times [0,1] \times [0,1]$$

$$d_N(x, y) = \left(T_1 + T_2, (I_1 + I_2) \cdot \frac{\tau - T_1 - T_2}{I_1 + I_2 + F_1 + F_2}, (F_1 + F_2) \cdot \frac{\tau - T_1 - T_2}{I_1 + I_2 + F_1 + F_2}\right)$$

if x and y are disjoint, and $T_1 + T_2 \leq 1$ where τ is the neutrosophic norm of $x \vee_N y$, i.e.

$$\tau = (T_1 + I_1 + F_1) \cdot (T_2 + I_2 + F_2).$$

We consider as neutrosophic norm of x, where $NL(x) = T_1 + I_1 + F_1$, the sum of its components: $T_1 + I_1 + F_1$, which in many cases is 1, but can also be positive < 1 or > 1.

When the truth value increases $(T_1 + T_2)$ is the above definition, the indeterminacy and falsehood values decrease proportionally with respect to their sums $I_1 + I_2$ and respectively $F_1 + F_2$.

This neutrosophic disjunction operator of disjoint variables allows us to add neutrosophic truth values of disjoint parts of a shaded area in a Venn Diagram.

Now, we complete Donald E. Knuth's Table of the Sixteen Logical Operators on two variables with Fuzzy Logical operators on two variables with Fuzzy Logic truth values, and Neutrosophic Logic truth/indeterminacy/false values (for the case $T \prec I \prec F$).

Table 1

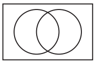
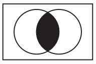
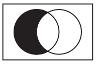
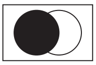
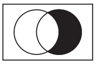
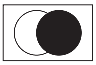
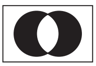
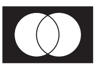
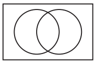
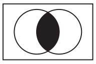
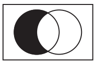
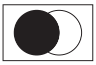
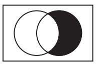
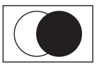
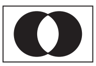
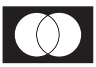
Fuzzy Logic Truth Values	Venn Diagram	Notations	Operator symbol	Name(s)
0		0	$\perp$	Contradiction, falsehood; constant 0
t_1t_2		$xy, x\wedge y, x\& y$	$\wedge$	Conjunction; and
$t_1-t_1t_2$		$x\wedge\overline{y}, x\not\supset y, [x>y], x-y$	$\overline{\supset}$	Nonimplication; difference, but not
t_1		x	L	Left projection
$t_2-t_1t_2$		$\overline{x}\wedge y, x\not\subset y, [x<y], y-x$	$\overline{\subset}$	Converse nonimplication; not...but
t_2		y	R	Right projection
$t_1+t_2-2t_1t_2$		$x\oplus y, x\not\equiv y, x\,\hat{}\,y$	$\oplus$	Exclusive disjunction; nonequivalence; "xor"
$t_1+t_2-t_1t_2$		$x\vee y, x\vert y$	$\vee$	(Inclusive) disjunction; or; and/or
$1-t_1-t_2+t_1t_2$		$\overline{x}\wedge\overline{y}, \overline{x\vee y}, x\overline{\vee}y, x\uparrow y$	$\overline{\vee}$	Nondisjunction, joint denial, neither...nor
$1-t_1-t_2+2t_1t_2$		$x\equiv y, x\leftrightarrow y, x\Leftrightarrow y$	$\equiv$	Equivalence; if and only if
$1-t_2$		$\overline{y}, \neg y, !y, \sim y$	$\overline{R}$	Right complementation
$1-t_2+t_1t_2$		$x\vee\overline{y}, x\subset y, x\Leftarrow y, [x\geq y], x^y$	$\subset$	Converse implication if
$1-t_1$		$\overline{x}, \neg x, !x, \sim x$	$\overline{L}$	Left complementation
$1-t_1+t_1t_2$		$\overline{x}\vee y, x\supset y, x\Rightarrow y, [x\leq y], y^x$	$\supset$	Implication; only if; if..then
$1-t_1t_2$		$\overline{x}\vee\overline{y}, \overline{x\wedge y}, x\overline{\wedge}y, x\vert y$	$\overline{\wedge}$	Nonconjunction, not both...and; "nand"
1		1	T	Affirmation; validity; tautology; constant 1

Table 2

Venn Diagram	Neutrosophic Logic Values
	(0,0,1)
	$(T_1T_2, I_1I_2+IT, F_1F_2+FI+FT)$, where $IT=I_1T_2+I_2T_1$ similarly FI,FT
	$(T_1F_2, \underbrace{I_1I_2+IT_{\bar{y}}}_{I_{P1}}, \underbrace{F_{\bar{y}}F_{\bar{y}}+F_{\bar{y}}I+F_{\bar{y}}T}_{F_{P2}})$
	(T_1,I_1,F_1)
	$(F_1T_2, \underbrace{I_1I_2+IT_{\bar{x}}}_{I_{P2}}, \underbrace{F_{\bar{x}}F_{\bar{x}}+F_{\bar{x}}I+F_{\bar{x}}T}_{F_{P2}})$
	(T_2,I_2,F_2)
	$\left(TF,(I_{P1}+I_{P2})\cdot\frac{\tau-TF}{I_{P1}+I_{P2}+F_{P1}+F_{P2}},(F_{P1}+F_{P2})\cdot\frac{\tau-TF}{I_{P1}+I_{P2}+F_{P1}+F_{P2}}\right)$ where $\tau=(T_1+I_1+F_1)\cdot(T_2+I_2+F_2)$ which is the neutrosophic norm
	$(T_1T_2+TI+TF, I_1I_2+IF, F_1F_2)$
	$(F_1F_2, I_1I_2+IF, T_1T_2+TI+TF)$
	$\left((F_{P1}+F_{P2})\cdot\frac{\tau-TF}{I_{P1}+I_{P2}+F_{P1}+F_{P2}},(I_{P1}+I_{P2})\cdot\frac{\tau-TF}{I_{P1}+I_{P2}+F_{P1}+F_{P2}},TF\right)$
	(F_2,I_2,T_2)
	$(F_{\bar{x}}F_{\bar{x}}+F_{\bar{x}}I+F_{\bar{x}}T, I_1I_2+IT_{\bar{x}}, F_1T_2)$
	(F_1,I_1,T_1)
	$(F_{\bar{y}}F_{\bar{y}}+F_{\bar{y}}I+F_{\bar{y}}T, I_1I_2+IT_{\bar{y}}, T_1F_2)$
	$(F_1F_2+FI+FT, I_1I_2+IT, T_1T_2)$
	(1,0,0)

These 16 neutrosophic binary operators are approximated, since the binary N-conorm gives an approximation because of 'indeterminacy' component.

Tri-nary Fuzzy Logic and Neutrosophic Logic Operators

In a more general way, for $k \geq 2$:

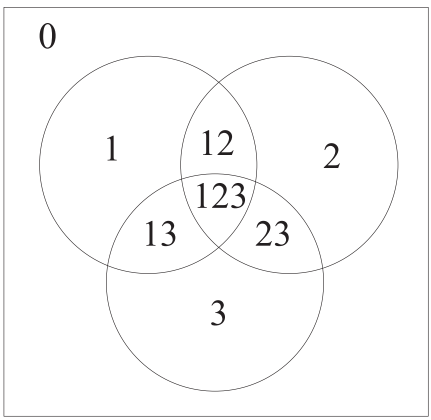
$$d_N^k : ([0,1] \times [0,1] \times [0,1])^k \to [0,1] \times [0,1] \times [0,1],$$

$$d_N^k(x_1, x_2, \ldots, x_k) = \left(\sum_{i=1}^{k} T_i, \left(\sum_{i=1}^{k} I_i \right) \cdot \frac{\tau_k - \sum\limits_{i=1}^{k} T_i}{\sum\limits_{i=1}^{k} (I_i + F_i)}, \left(\sum_{i=1}^{k} F_i \right) \cdot \frac{\tau_k - \sum\limits_{i=1}^{k} T_i}{\sum\limits_{i=1}^{k} (I_i + F_i)} \right)$$

if all x_i are disjoint two by two, and $\sum\limits_{i=1}^{k} T_i \leq 1$.

We can extend Knuth's Table from binary operators to tri-nary operators (and we get $2^{2^3} = 256$ tri-nary operators) and in general to n–ary operators (and we get 2^{2^n} n-ary operators).

Let's present the tri-nary Venn Diagram, with 3 variables x, y, z



using the name Smarandache codification.

This has $2^3 = 8$ disjoint parts, and if we shade none, one, two, ..., or eight of them and consider all possible combinations we get $2^8 = 256$ tri-nary operators in the above tri-nary Venn Diagram.

For $n = 3$ we have:

$$\begin{aligned}
P123 &= c_F(x, y, z) \\
P12 &= c_F(x, y, n_F(z)) \\
P13 &= c_F(x, n_F(y), z) \\
P23 &= c_F(n_F(x), y, z) \\
P1 &= c_F(x, n_F(y), n_F(z)) \\
P2 &= c_F(n_F(x), y, n_F(z)) \\
P3 &= c_F(n_F(x), n_F(y), z) \\
P0 &= c_F(n_F(x), n_F(y), n_F(z))
\end{aligned}$$

Let

$$\begin{aligned}
FL(x) &= (t_1, f_1), \text{ with } t_1 + f_1 = 1, \quad 0 \leq t_1, \quad f_1 \leq 1, \\
FL(y) &= (t_2, f_2), \text{ with } t_2 + f_2 = 1, \quad 0 \leq t_2, \quad f_2 \leq 1, \\
FL(u) &= (t_3, f_3), \text{ with } t_3 + f_3 = 1, \quad 0 \leq t_3, \quad f_3 \leq 1,
\end{aligned}$$

We consider the particular case defined by tri-nary conjunction fuzzy operator:

$$c_F : ([0,1] \times [0,1])^3 \to [0,1] \times [0,1]$$

$$c_F(x, y, z) = (t_1 t_2 t_3, f_1 + f_2 + f_3 - f_1 f_2 - f_2 f_3 - f_3 f_1 + f_1 f_2 f_3)$$

because

$$\begin{aligned}
((t_1, f_1) \wedge_F (t_2, f_2)) \wedge_F (t_3, f_3) &= (t_1 t_2, f_1 + f_2 - f_1 f_2) \wedge_F (t_3, f_3) = \\
&= (t_1 t_2 t_3, f_1 + f_2 + f_3 - f_1 f_2 - f_2 f_3 - f_3 f_1 + f_1 f_2 f_3)
\end{aligned}$$

and the unary negation operator:

$$n_F : ([0,1] \times [0,1]) \to [0,1] \times [0,1]$$

$$n_F(x) = (1 - t_1, 1 - f_1) = (f_1, t_1).$$

We define the function:

$$L_1 : [0,1] \times [0,1] \times [0,1] \to [0,1]$$

$$L_1(\alpha, \beta, \gamma) = \alpha \cdot \beta \cdot \gamma$$

and the function

$$L_2 : [0,1] \times [0,1] \times [0,1] \to [0,1]$$

$$L_2(\alpha, \beta, \gamma) = \alpha + \beta + \gamma - \alpha\beta - \beta\gamma - \gamma\alpha + \alpha\beta\gamma$$

then:

$$\begin{aligned}
FL(P123) &= (L_1(t_1,t_2,t_3), L_2(f_1,f_2,f_3))\\
FL(P12) &= (L_1(t_1,t_2,f_3), L_2(f_1,f_2,t_3))\\
FL(P13) &= (L_1(t_1,f_2,t_3), L_2(f_1,t_2,f_3))\\
FL(P23) &= (L_1(f_1,t_2,t_3), L_2(t_1,f_2,f_3))\\
FL(P1) &= (L_1(t_1,f_2,f_3), L_2(f_1,t_2,t_3))\\
FL(P2) &= (L_1(f_1,t_2,f_3), L_2(t_1,f_2,t_3))\\
FL(P3) &= (L_1(f_1,f_2,t_3), L_2(t_1,t_2,f_3))\\
FL(P0) &= (L_1(f_1,f_2,f_3), L_2(t_1,t_2,t_3))
\end{aligned}$$

We thus get the fuzzy truth-values as follows:

$$\begin{aligned}
FL_t(P123) &= t_1t_2t_3\\
FL_t(P12) &= t_1t_2(1-t_3) = t_1t_2 - t_1t_2t_3\\
FL_t(P13) &= t_1(1-t_2)t_3 = t_1t_3 - t_1t_2t_3\\
FL_t(P23) &= (1-t_1)t_2t_3 = t_2t_3 - t_1t_2t_3\\
FL_t(P1) &= t_1(1-t_2)(1-t_3) = t_1 - t_1t_2 - t_1t_3 + t_1t_2t_3\\
FL_t(P2) &= (1-t_1)t_2(1-t_3) = t_2 - t_1t_2 - t_2t_3 + t_1t_2t_3\\
FL_t(P3) &= (1-t_1)(1-t_2)t_3 = t_3 - t_1t_3 - t_2t_3 + t_1t_2t_3\\
FL_t(P0) &= (1-t_1)(1-t_2)(1-t_3) = 1 - t_1 - t_2 - t_3 + t_1t_2 + t_1t_3 +\\
&\quad + t_2t_3 - t_1t_2t_3.
\end{aligned}$$

We, then, consider the same disjunction or union operator $d_F(x,y) = t_1+t_2$, $f_1 + f_2 - 1$, if x and y are disjoint, and $t_1 + t_2 \leq 1$ allowing us to add the fuzzy truth values of each part of a shaded area.

Neutrophic Composition Law

Let's consider $k \geq 2$ neutrophic variables, $x_i(T_i, I_i, F_I)$, for all $i \in \{1, 2, \ldots, k\}$. Let denote

$T = (T_1, \ldots, T_k)$

$I = (I_1, \ldots, I_k)$

$F = (F_1, \ldots, F_k)$.

We now define a neutrosophic composition law o_N in the following way:

$$o_N : \{T, I, F\} \to [0,1]$$

If $z \in \{T, I, F\}$ then $z_{o_N} z = \prod_{i=1}^{k} z_i$.

If $z, w \in \{T, I, F\}$ then

$$z_{o_N} w = w_{o_N} z = \sum_{\substack{r=1 \\ \{i_1,\ldots,i_r,j_{r+1},\ldots,j_k\} \equiv \{1,2,\ldots,k\} \\ (i_1,\ldots,i_r) \in C^k(1,2,\ldots,k) \\ (j_{r+1},\ldots,j_k) \in C^{k-r}(1,2,\ldots,k)}}^{k-1} z_{i_1} \ldots z_{i_r} w_{j_{r+1}} \ldots w_{j_k}$$

where $C^r(1, 2, \ldots, k)$ means the set of combinations of the elements $\{1, 2, \ldots, k\}$ taken by r. [Similarly for $C^{k-r}(1, 2, \ldots, k)$].

In other words, $z_{o_N} w$ is the sum of all possible products of the components of vectors z and w, such that each product has at least a z_i factor and at least w_j factor, and each product has exactly k factors where each factor is a different vector component of z or of w. Similarly if we multiply three vectors:

$$T_{o_N} I_{o_N} F = \sum_{\substack{u,v,k-u-v=1 \\ \{i_1,\ldots,i_u,j_{u+1},\ldots,j_{u+v},l_{u+v+1},\ldots,l_k\} \equiv \{1,2,\ldots,k\} \\ (i_1,\ldots,i_u) \in C^u(1,2,\ldots,k) \\ (j_{u+1},\ldots,j_{u+v}) \in C^v(1,2,\ldots,k) \\ (l_{u+v+1},\ldots,l_k) \in C^{k-u-v}(1,2,\ldots,k)}}^{k-2} T_{i_1 \ldots i_u} I_{j_{u+1} \ldots j_{u+v}} F_{l_{u+v+1}} \ldots F_{l_k}$$

Let's see an example for $k = 3$.

$x_1(T_1, I_1, F_1)$

$x_2(T_2, I_2, F_2)$

$x_3(T_3, I_3, F_3)$

$T_{o_N} T = T_1 T_2 T_3, \quad I_{o_N} I = I_1 I_2 I_3, \quad F_{o_N} F = F_1 F_2 F_3$

$T_{o_N} I = T_1 I_2 I_3 + I_1 T_2 I_3 + I_1 I_2 T_3 + T_1 T_2 I_3 + T_1 I_2 T_3 + I_1 T_2 T_3$

$T_{o_N} F = T_1 F_2 F_3 + F_1 T_2 F_3 + F_1 F_2 T_3 + T_1 T_2 F_3 + T_1 F_2 T_3 + F_1 T_2 T_3$

$I_{o_N} F = I_1 F_2 F_3 + F_1 I_2 F_3 + F_1 F_2 I_3 + I_1 I_2 F_3 + I_1 F_2 I_3 + F_1 I_2 I_3$

$T_{o_n} I_{o_N} F = T_1 I_2 F_3 + T_1 F_2 I_3 + I_1 T_2 F_3 + I_1 F_2 T_3 + F_1 I_2 T_3 + F_1 T_2 I_3$

For the case when indeterminacy I is not decomposed in subcomponents {as for example $I = P \cup U$ where P = paradox (true and false simultaneously) and U = uncertainty (true or false, not sure which one)}, the previous formulas can be easily written using only three components as:

$$T_{o_N} I_{o_N} F = \sum_{i,j,r \in \mathcal{P}(1,2,3)} T_i I_j F_r$$

where $\mathcal{P}(1,2,3)$ means the set of permutations of $(1,2,3)$ i.e.

$$\{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$$

$$z_{o_N} w = \sum_{\substack{i=1 \\ (i,j,r)\equiv(1,2,3) \\ (j,r)\in\mathcal{P}^2(1,2,3)}}^{3} z_i w_j w_{j_r} + w_i z_j z_r$$

This neurotrophic law is associative and commutative.

Neutrophic Logic Operators

Let's consider the neutrophic logic cricy values of variables x, y, z (so, for $n = 3$)

$$NL(x) = (T_1, I_1, F_1) \text{ with } 0 \leq T_1, I_1, F_1 \leq 1$$
$$NL(y) = (T_2, I_2, F_2) \text{ with } 0 \leq T_2, I_2, F_2 \leq 1$$
$$NL(z) = (T_3, I_3, F_3) \text{ with } 0 \leq T_3, I_3, F_3 \leq 1$$

In neutrosophic logic it is not necessary to have the sum of components equals to 1, as in intuitionist fuzzy logic, i.e. $T_k + I_k + F_k$ is not necessary 1, for $1 \leq k \leq 3$.

As a particular case, we define the tri-nary conjunction neutrosophic operator:

$$c_N : ([0,1] \times [0,1] \times [0,1])^3 \to [0,1] \times [0,1] \times [0,1]$$

$$c_N(x,y) = (T_{o_N} T, I_{o_N} I + I_{o_N} T, F_{o_N} F + F_{o_N} I + F_{o_N} T)$$

If x or y are normalized, then $c_N(x,y)$ is also normalized.

If x or y are non-normalized then $|c_N(x,y)| = |x| \cdot |y|$ where $|\cdot|$ means norm.

c_N is an N-norm (neutrosophic norm, i.e. generalization of the fuzzy t-norm).

Again, as a particular case, we define the unary negation neutrosophic operator:

$$n_N : [0,1] \times [0,1] \times [0,1] \to [0,1] \times [0,1] \times [0,1]$$

$$n_N(x) = n_N(T_1, I_1, F_1) = (F_1, I_1, T_1).$$

We take the same Venn Diagram for $n = 3$.

So,

$NL(x) = (T_1, I_1, F_1)$

$NL(y) = (T_2, I_2, F_2)$

$NL(z) = (T_3, I_3, F_2)$.

Vectors

$$T = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix}, \quad I = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} \quad \text{and } F = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \end{pmatrix}.$$

We note

$$T_{\overline{x}} = \begin{pmatrix} F_1 \\ T_2 \\ T_3 \end{pmatrix}, \quad T_{\overline{y}} = \begin{pmatrix} T_1 \\ F_2 \\ T_3 \end{pmatrix}, \quad T_{\overline{z}} = \begin{pmatrix} T_1 \\ T_2 \\ F_3 \end{pmatrix}, \quad T_{\overline{xy}} = \begin{pmatrix} F_1 \\ F_2 \\ T_3 \end{pmatrix}, \quad \text{etc.}$$

and similarly

$$F_{\overline{x}} = \begin{pmatrix} T_1 \\ F_2 \\ F_3 \end{pmatrix}, \quad F_{\overline{y}} = \begin{pmatrix} F_1 \\ T_2 \\ F_3 \end{pmatrix}, \quad F_{\overline{xz}} = \begin{pmatrix} T_1 \\ F_2 \\ T_3 \end{pmatrix}, \quad \text{etc.}$$

For shorter and easier notations let's denote $z_{o_N} w = zw$ and respectively $z_{o_N} w_{o_N} v = zwv$ for the vector neutrosophic law defined previously. Then

$$\begin{aligned}
NL(P123) &= c_N(x, y) = (TT, II + IT, FF + FI + FT + FIT) = \\
&= (T_1T_2T_3, I_1I_2I_3 + I_1I_2T_3 + I_1T_2I_3 + T_1I_2I_3 + I_1T_2T_3 + T_1I_2T_3 + T_1T_2I_3, \\
&\quad F_1F_2F_3 + F_1F_2I_3 + F_1I_2F_3 + I_1F_2F_3 + F_1I_2I_3 + I_1F_2I_3 + I_1I_2F_3 + \\
&\quad + F_1F_2T_3 + F_1T_2F_3 + T_1F_2F_3 + F_1T_2T_3 + T_1F_2T_3 + T_1T_2F_3 + \\
&\quad + T_1I_2F_3 + T_1F_2I_3 + I_1F_2T_3 + I_1T_2F_3 + F_1I_2T_3 + F_1T_2I_3)
\end{aligned}$$

$$\begin{aligned}
NL(P12) &= c_N(x, y, n_N(z)) = \\
&= (T_{\overline{z}}T_{\overline{z}}, II + IT_{\overline{z}}, F_{\overline{z}}F_{\overline{z}} + F_{\overline{z}}I + F_{\overline{z}}T_{\overline{z}} + F_{\overline{z}}IT_{\overline{z}}) \\
NL(P13) &= c_N(x, n_N(y), z) = \\
&= (T_{\overline{y}}T_{\overline{y}}, II + IT_{\overline{y}}, F_{\overline{y}}F_{\overline{y}} + F_{\overline{y}}I + F_{\overline{y}}T_{\overline{y}} + F_{\overline{y}}IT_{\overline{y}}) \\
NL(P23) &= c_N(n_N(x), y, z) = \\
&= (T_{\overline{x}}T_{\overline{x}}, II + IT_{\overline{x}}, F_{\overline{x}}F_{\overline{x}} + F_{\overline{x}}I + F_{\overline{x}}T_{\overline{x}} + F_{\overline{x}}IT_{\overline{x}}) \\
NL(P1) &= c_N(x, n_N(y), n_N(z)) = \\
&= (T_{\overline{yz}}T_{\overline{yz}}, II + IT_{\overline{yz}}, F_{\overline{yz}}F_{\overline{yz}} + F_{\overline{yz}}I + F_{\overline{yz}}T_{\overline{yz}} + F_{\overline{yz}}IT_{\overline{yz}}) \\
NL(P2) &= c_N(n_N(x), y, n_N(z)) = \\
&= (T_{\overline{xz}}T_{\overline{xz}}, II + IT_{\overline{xz}}, F_{\overline{xz}}F_{\overline{xz}} + F_{\overline{xz}}I + F_{\overline{xz}}T_{\overline{xz}} + F_{\overline{xz}}IT_{\overline{xz}})
\end{aligned}$$

$$\begin{aligned} NL(P0) &= c_N(n_N(x), n_N(y), n_N(z)) = \\ &= (T_{\overline{xyz}}T_{\overline{xyz}}, II + IT_{\overline{xyz}}, F_{\overline{xyz}}F_{\overline{xyz}} + F_{\overline{xyz}}I + F_{\overline{xyz}}T_{\overline{xyz}} + \\ &\quad + F_{\overline{xyz}}IT_{\overline{xyz}}) = (FF, II + IF, TT + TI + TF + TIF) \end{aligned}$$

n-ary Fuzzy Logic and Neutrosophic Logic Operators

We can generalize for any integer $n \geq 2$.

The Venn Diagram has 2^{2^n} disjoint parts. Each part has the form $Pi_1 \dots i_k j_{k+1} \dots j_n$, where $0 \leq k \leq n$, and of course $0 \leq n-k \leq n$; $\{i_1, \dots, i_k\}$ is a combination of k elements of the set $\{1, 2, \dots, n\}$, while $\{j_{k+1}, \dots, j_n\}$ the $n-k$ elements left, i.e. $\{j_{k+1}, \dots, j_n\} = \{1, 2, \dots, n\} \setminus \{i_1, \dots, i_k\}$. $\{i_1, \dots, i_k\}$ are replaced by the corresponding numbers from $\{1, 2, \dots, n\}$, while $\{j_{k+1}, \dots, j_n\}$ are replaced by blanks.

For example, when $n = 3$,

$$Pi_1i_2j_3 = P13 \text{ if } \{i_1, i_2\} = \{1, 3\},$$

$$Pi_1j_2j_3 = P1 \text{ if } \{i_1\} = \{1\}.$$

Hence, for fuzzy logic we have:

$$Pi_1 \dots i_k j_{k+1} \dots j_n = c_F\left(x_{i_1}, \dots, x_{i_k}, n_F(x_{j_{k+1}}), \dots, n_F(x_{j_n})\right),$$

whence

$$FL(Pi_1 \dots i_k j_{k+1} \dots j_n) = \left(\left(\prod_{r=1}^{k} t_{i_r}\right)\left(\prod_{s=k+1}^{n} (1 - t_{j_s})\right), \varphi(f_1, f_2, \dots, f_n)\right)$$

where $\varphi : [0,1]^n \to [0,1]$,

$$\varphi(\alpha_1, \alpha_2, \dots, \alpha_n) = S_1 - S_2 + S_3 + \dots + (-1)^{n+1} S_n = \sum_{l=1}^{n} (-1)^{l+1} S_i$$

where

$$S_1 = \sum_{i=1}^{n} \alpha_i$$

$$S_2 = \sum_{1 \leq i < j \leq n} \alpha_i \alpha_j$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$

$$S_i = \sum_{1 \leq i_1 < i_2 < \dots < i_j \leq n} \alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_j}$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$

$$S_n = \alpha_1 \cdot \alpha_2 \cdot \dots \cdot \alpha_n$$

And for neutrosophic logic we have:

$$Pi_1 \ldots i_k j_{k+1} \ldots j_n = c_N \left(x_{i_1}, \ldots, x_{i_k}, n_N(x_{j_{k+1}}), \ldots, n_N(x_{j_n})\right),$$

whence:

$$NL(Pi_1 \ldots i_k j_{k+1} \ldots j_n) = (T_{12\ldots n}, I_{12\ldots n}, F_{12\ldots n}),$$

where

$$T_{12\ldots n} = T_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} T_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} = \left(\prod_{r=1}^{k} T_{i_r}\right) \cdot \prod_{s=k+1}^{n} F_{j_s}.$$

$$I_{12\ldots n} = II + IT_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}}$$

$$F_{12\ldots n} = F_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} F_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} + F_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} I + F_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} T_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} + \\ + F_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}} IT_{\overline{x}_{j_{k+1}}\ldots\overline{x}_{j_n}}$$

Conclusion

A generalization of Knuth's Boolean binary operations is presented in this paper, i.e. we present n-ary Fuzzy Logic Operators and Neutrosophic Logic Operators based on Smarandache's codification of the Venn Diagram and on a defined vector neutrosophic law which helps in calculating fuzzy and neutrosophic operators.

Better neutrosophic operators than in [2] are proposed herein.

References

[1] F. Smarandache & J. Dezert, Advances and Applications of DSmt for Information Fusion, Am. Res. Press, 2004.

[2] F. Smarandache, A unifying field in logics: Neutrosophic Logic, Neutrosophy, Neutrosophic Set, Neutrosophic Probability and Statistics, 1998, 2001, 2003.

[3] Knuth, Art of Computer Programming, The, Volumes 1–3 Boxed Set (2nd Edition), Addison-Wesley Professional, 1998.

[4] Zadeh, Fuzzy Sets, Information and Control, Vol. 8, 338–353, 1965.

[5] Driankov, Dimiter; Hellendoorn, Hans; and Reinfrank, Michael: *An Introduction to Fuzzy Control.* Springer, Berlin/Heidelberg, 1993.

[6] Atanassov, K., Stoyanova, D., Remarks on the intuitionistic fuzzy sets. II, Notes on Intuitionistic Fuzzy Sets, Vol. 1, No. 2, 85–86, 1995.

Florentin Smarandache
University of New Mexico
Department of Mathematics and Science
200 College Road
Gallup, NM87301, USA
smarand@unm.edu, fsmarandache@yahoo.com

Vic Christianto
Institute of Gravitation and Cosmology
PFUR, Moscow, 117198
Russia
e-mail: vxianto@yahoo.com

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17 (30) 2009

Andrew Schumann
Belarusian State University, Minsk, Belarus

CIRCULAR PROOFS IN PROOF-THEORETIC SIMULATION OF BELOUSOV-ZHABOTINSKY REACTION

In the paper I consider abstract machine of reaction-diffusion computing. This machine is constructed by using process algebra. Further, I propose proof-theoretic interpretation of process calculus within the framework of Belousov-Zhabotinsky reaction. I show that some proofs simulating the behaviour of Belousov-Zhabotinsky reaction are circular. This means that the derivation tree simulating the behaviour tree of Belousov-Zhabotinsky reaction has cycles, i.e. some derivable formulas occur among top premisses.

1. Introduction

The reaction-diffusion approach to computing was first proposed by Alan Turing in his famous paper titled "The Chemical Basis of Morphogenesis." In his version, design patterns were considered as a conceptual framework for transferring knowledge from biology to computing. The importance of this idea is that reaction-diffusion processes are observed throughout nature in all spatiotemporal pattern formation and self-organization far from equilibrium coupled with diffusion.

After Turing, the interest in controlling self-organized systems and in building up computational models based on them in order to achieve their specific desired behavior has been increasing. Control of self-organization in reaction-diffusion frames is an ambitious goal allowing to simulate all massive-parallel natural processes: chemical reactions, morphogenesis, population dynamics, biological, economic and social evolution and other phenomena that produce a complex network of interactions among the component species.

In the initial meaning, reaction-diffusion systems involve constituents locally transformed into each other by chemical reactions and transported in space by diffusion. The simplest kind of a chemical wave emerges when reactants are converted into products as the front propagates through the reac-

ISBN 978–83–7431–231–8 ISSN 0860-150X

tion mixture. Usually, chemical reaction-diffusion systems eventually evolve to the state of chemical equilibrium, but there is a possibility of thermodynamic equilibrium, like pattern formation in Belousov-Zhabotinsky reaction. In the latter case, equilibrium of self-organized patterns occur due to interaction of autocatalytic compensated by reaction step blocking the unbounded growth.

Wave is a key notion for analyzing reaction-diffusion processes. It is a form of propagating the spatial coupling of elements from the stable or the most stable state toward the unstable or the least stable one. Due to waves of reagent diffusions there is a chemical reaction. Furthermore, a wave can cause composite patterns in a regime of periodic or chaotic oscillations, like that we observe in Belousov-Zhabotinsky reaction. If a wave causes pattering and regulation of timing independently of input conditions and the letter are determined by the system itself, then this is called auto-wave. Its typical instance is described in Belousov-Zhabotinsky reaction. Auto-waves may be detected in any self-organized systems, some examples of auto-waves are given below. As a result, we can claim that reaction-diffusion nature is observed not just within chemical reactions, but also in different pattering and interactions like as follows:

- Auto-waves of morphogenesis consisting in the processes that generate tissue organization and shape and are usually the downstream response to the timing and patterning.
- Auto-waves in an ecological system that change the network of interacting populations.
- Auto-waves of epidemics, they are observed in the rapid increase of the size of infected population due to the interaction between infected and susceptible individuals.
- Auto-waves of opinion dynamics, i.e. changes in opinions caused by interaction of people.
- Auto-waves of aggregated individual amoebae of the cellular slime mold Dictyostelium discoideum to form a multicellular migrating slug, which moves toward a region suitable for culmination.

As we see, thanks to the above-mentioned examples, the majority of physical, chemical, biological and sociological processes can be described in terms of auto-waves, i.e. propagating fronts of one nature or another.

The dynamics of complex reaction-diffusion frames can be described in terms of activator and inhibitor. Activator, A, is a short-range autocatalytic substance, inhibitor, I, is its long-range antagonist. A autocatalytically promotes its own synthesis. I inhibits synthesis of A, and I can diffuse faster than A. The pattern of peaks of A is dependent upon the reaction-diffusion frame. The dynamics of a self-organizing system is typically non-linear, because of circular or feedback relations between the components. In other words, in complex reaction-diffusion systems there are circular cause-and-effect relations: each component affects the other components, but these components in turn affect the first component. Feedback can have two basic values: positive if activator diffuses faster or negative if inhibitor diffuses faster. In the positive feedback, the recurrent influence reinforces the initial change. In the negative feedback, the reaction is opposite to the initial action, so change is counteracted. The aim of negative feedback is to stabilize the system, by returning to its original states. On the other hand, positive feedback makes changes grow in an explosive manner. It leads to accelerated development, which ends when all components have been absorbed into the new configuration, leaving the system in a stable state.

Diffusion waves of activator and inhibitor interact according to reaction rules. Solutions of those rules are represented by multisets of waves: this accounts for the associativity and commutativity of parallel composition, that is the implicit stirring mechanism. Setting diffusion waves by rules allows us to define reaction-diffusion processes as a kind of computing. This computing is fundamentally similar to parallel computing in that it takes advantage of the many different waves to try many different possibilities of diffusion at once.

Setting up reaction-diffusion computing is a way of simulating physics by a universal computer. This aim to build up a "life computer" is the most ambitious task attempted in unconventional computing, e.g. see for more details [13]. Its difficulties consist in that in conventional approach to computing control is centralized. On the other hand, in self-organizing systems control of the organization is typically distributed over the whole of the system. Due to a centralized control, software is written for serial computation and a problem is broken into a discrete series of executed instructions so that only one instruction may execute at any moment in time. Due to a decentralized, distributed control in parallel computing, a computational problem is solved by the simultaneous use of multiple compute resources and a problem is broken into discrete parts that can be solved concurrently. We can add that reaction-diffusion computing is not just parallel, but massive-parallel. This means that parts of a problem are broken into can be indiscrete and infinite many.

The sequential model of computation called von Neumann paradigm is unapplied, broken in reaction-diffusion computing. The reason is that the latter computing may be presented as a massive-parallel locally-connected mathematical machine with circular and cyclic processes. These machines cannot have a single centralized source exercising precise control over vast numbers of heterogeneous devices. Interactive-computing paradigm is able to describe concurrent (parallel) computations whose configuration may change during the computation and is decentralized as well. Within the framework of this paradigm, one proposed a lot of so-called concurrency calculi also called process algebras. They are typically presented using systems of equations. These formalisms for concurrent systems are formal in the sense that they represent systems by expressions and then reason about systems by manipulating the corresponding expressions.

One of the unconventional, nature inspired models similar to reaction-diffusion computing is chemical machine in that molecules are viewed as computational processes supplemented with a minimal reaction kinetics. Berry and Boudol first built up a chemical abstract machine [6] as an example of how a chemical paradigm of the interactions between molecules can be utilized in concurrent computations (in algebraic process calculi). In the **second section** of this paper we will consider more general abstract machine of reaction-diffusion computing. This machine is constructed by using process algebra. In the **third section** we will propose proof-theoretic interpretation of process calculus for reaction-diffusion computing within the framework of Belousov-Zhabotinsky reaction. We will show that some proofs simulating the behaviour of Belousov-Zhabotinsky reaction are circular. This means that the derivation tree simulating the behaviour tree of Belousov-Zhabotinsky reaction has cycles (circles), i.e. some derivable formulas themselves occur among top premisses.

2. Process calculus of reaction-diffusion computing

A behaviour in reaction-diffusion systems is thought as the total of actions that a system can perform taking into account that we describe certain aspects of behaviour by introducing an abstraction or idealization of the 'real' behaviour. The simplest model of behaviour we will use is to consider behaviour as an input/output function. Evidently, it is a restriction, idealization we will deal with. Usually, such input/output function approach is implemented in automata theory, where a process is modeled as an automaton. An automaton has a number of states and a number of transitions, going from one state to another state. A behaviour is a run, i.e. a path from initial state to final state. There are different ways of modeling reaction-diffusion behaviour within this approach. The most popular is to simulate the behaviour by cellular automata.

In the present paper we will use more general approach presented by process algebra – the study of the behaviour of parallel or distributed systems by algebraic means. It offers tools to describe or specify transition systems, and thus it has tools to talk about parallel composition in general.

Assume that the computational domain Ω is partitioned into computational cells $c_j = \overline{1, K}$ such that $c_i \cap c_j = \emptyset, i \neq j$ and $\bigcup_{j=1}^{K} c_j = \Omega$. Further, suppose that in the K cells, there are N chemically active species and the state of species i in cell j is denoted by $p_{ij}, i = \overline{1, N},\ j = \overline{1, K}$. These states are time dependent and they are changed by reactions occurring between waves in the same cell and by diffusion where waves move to adjacent cells. In reaction-diffusion, the species interact concurrently and in a parallel manner.

Our process calculus contain the following basic operators: Nil (inaction), '$\bullet$' (prefix), '$|$' (cooperation), '$\backslash$' (hiding), '&' (reaction/fusion), '$\oplus$' (choice), a (constant or restriction to a stable state). Let $N = \{a, b, \ldots\}$ be a set of names and $L = \{a, \overline{a}\colon a \in N\}$, where a is considered as activator and $\overline{a}$ as inhibitor for a, be the set of labels built on N. We use the symbols α, β, etc., to range over labels, with $\alpha = \overline{\overline{\alpha}}$, and the symbols P, Q, etc., to range over states $p_{ij}, i = \overline{1, N}$, $j = \overline{1, K}$. The behaviour of states is given by the syntax:

$$P ::= Nil \mid \alpha \bullet P \mid (P|P) \mid P \backslash X \mid P \& P \mid P \oplus P \mid a$$

An operational semantics for this syntax is defined in Fig. 1. The informal meanings of basic operations are as follows:

Nil This is the empty process which does nothing. In other words, Nil represents the component which is not capable of performing any activities: a deadlocked component.

$\alpha \bullet P$ A process $\alpha \in L$ followed by the process P: P becomes activity only after the action α has been performed.

$P|Q$ This is a parallel composition (commutative and associative) of actions: P and Q are performed in parallel.

$P\backslash X$ This restriction operator allows us to force some of P's actions not to occur; all of the actions in the set X are prohibited, i.e. the component $P\backslash X$ behaves as P except that any activities of types within the set X are hidden, meaning that their type is not visible outside the component upon completion.

Prefix	$\dfrac{}{\alpha \bullet P \longrightarrow^{\alpha} P}$
Constant	$\dfrac{P \longrightarrow^{\alpha} P'}{a \longrightarrow^{\alpha} P'} \ (a := P)$
Choice	$\dfrac{P \longrightarrow^{\alpha} P'}{P \oplus Q \longrightarrow^{\alpha} P'}, \quad \dfrac{Q \longrightarrow^{\alpha} Q'}{P \oplus Q \longrightarrow^{\alpha} Q'}$
Cooperation	$\dfrac{P \longrightarrow^{\alpha} P'}{P\|Q \longrightarrow^{\alpha} P'\|Q}, \quad \dfrac{Q \longrightarrow^{\alpha} Q'}{P\|Q \longrightarrow^{\alpha} P\|Q'}$
Hiding	$\dfrac{P \longrightarrow^{\alpha} P'}{P\backslash X \longrightarrow^{\alpha} P'\backslash X} \ (\alpha \notin X), \quad \dfrac{P \longrightarrow^{\alpha} P'}{P\backslash X \longrightarrow^{\beta} P'\backslash X} \ (\alpha \in X)$
Fusion	$\dfrac{}{\alpha \bullet P\&\overline{P} \longrightarrow^{\alpha} Nil}, \quad \dfrac{P \longrightarrow^{\alpha} P' \quad Q \longrightarrow^{\alpha} P'}{P\&Q \longrightarrow^{\alpha} P'}, \quad \dfrac{P \longrightarrow^{\alpha} P'}{P\&Q \longrightarrow^{\alpha} P' \oplus Q'}$

Figure 1. Operational semantics: inference rules for basic operations. The ternary relation $P \longrightarrow^{\alpha} P'$ means that the initial state P is capable of engaging in action α and then behaving like P'

$P\&Q$ This is the fusion of P and Q; $P\&Q$ represents a system which may behave as both component P and Q. For instance, Nil behaves as $P\&\overline{P}$, where P is an activator and $\overline{P}$ an appropriate inhibitor respectively.

$P \oplus Q$ This is the choice between P and Q; $P \oplus Q$ represents a system which may behave either as component P or as Q. Thus the first activity to complete identifies one of the components which is selected as the component that continues to evolve; the other component is discarded.

a constants are components whose meaning is given by equations such as $a := P$. Here the constant a is given the behaviour of the component P. Constants can be used to describe infinite behaviours, via mutually recursive defining equations.

Thus, in this process calculus we have four kinds of transitions between states: (1) the internal transitions $p_{mj} \longrightarrow^{\alpha} p_{nj}$, i.e. a reaction α in a cell j is a transition from one state p_{mj} before the reaction to the state p_{nj} after the reaction, (2) the external transitions $p_{mk} \longrightarrow^{\alpha} p_{nl}$ (diffusion), i.e. a reaction α in a cell l is a transition from one state p_{mk} in a cell k before the reaction to the state p_{nl} in a cell l after the reaction.

3. Proof-theoretic simulation of behaviour in reaction-diffusion systems

A behaviour of reaction-diffusion systems can be viewed as a labelled transition system, which consists of a collection of states L and a collection Σ of transitions between them (actions over them). Assume $\Sigma\colon L \mapsto \mathcal{P}(L)$, where $\mathcal{P}(L) = \{T\colon T \subseteq L\}$, i.e. $\Sigma = \bigcup_{a\in L} \{\langle a, \alpha\rangle\colon \alpha \in T \subseteq L\}$. This means that $\Sigma(a)$ consists of all states that a reachable from a. The transition system is understood as a triple $\langle L, \Sigma, \longrightarrow\rangle$, where $\longrightarrow \subseteq L \times \Sigma \times L$ is a transition relation that models how a state $p \in L$ can evolve into another state $p' \in L$ due to an interaction $\sigma \in \Sigma$. Usually, $\langle p, \sigma, p'\rangle \in \longrightarrow$ is denoted by $p \longrightarrow^a p'$ under assumption that there exists a such that $\langle p, a\rangle \in \Sigma$ and $\langle a, p'\rangle \in \Sigma$.

$$\frac{Ce^{3+}\&HBrO_2\&BrO_3^-\&H^+}{Ce^{4+} \oplus HBrO_2 \oplus H_2O} \qquad (1)$$

$$\frac{Ce^{4+}\&BrCH(COOH)_2\&H_2O}{Br^- \oplus Ce^{3+} \oplus HCOOH \oplus CO_2 \oplus H^+} \qquad (2)$$

$$\frac{HBrO_2}{HOBr \oplus BrO_3^- \oplus H^+} \qquad (3)$$

$$\frac{BrO_3^-\&Br^-\&H^+}{HOBr \oplus HBrO_2} \qquad (4)$$

$$\frac{HBrO_2\&Br^-\&H^+}{HOBr} \qquad (5)$$

$$\frac{HOBr\&Br^-\&H^+}{Br_2 \oplus H_2O} \qquad (5)$$

$$\frac{Br_2\&CH_2(COOH)_2}{Br^- \oplus H^+ \oplus BrCH(COOH)_2} \qquad (6)$$

Figure 2. Axioms of proof-theoretic simulation of Belousov-Zhabotinsky reaction

The finite word $\alpha_1\alpha_2\ldots\alpha_n$ is a *finite trace of transition system* whenever there is a finite execution fragment of transition system

$$\varrho = p_0\alpha_1 p_1\alpha_2\ldots\alpha_n p_n \ \text{ such that } \ p_i \longrightarrow^{\alpha_{i+1}} p_{i+1} \ \text{ for all } \ 0 \le i < n.$$

Choice:	$\dfrac{P \oplus Q}{P}$ (CL), $\quad \dfrac{P \oplus Q}{Q}$ (CR),
Fusion:	$\dfrac{P}{P \& Q}$ (FL), $\quad \dfrac{Q}{P \& Q}$ (FR), $\quad \dfrac{P \& \overline{P}}{}$ (F),
Cooperation:	$\dfrac{P \mid Q}{P, Q}$ (C),
Hiding:	$\dfrac{P}{P'} \Rightarrow \dfrac{P \backslash X}{P' \backslash X}$ (H).

Figure 3. Inference rules of proof-theoretic simulation of Belousov-Zhabotinsky reaction. The variables P, Q run over the set $\{Ce^{3+}$, $HBrO_2$, BrO_3^-, H^+, Ce^{4+}, H_2O, $BrCH\ (COOH)_2$, Br^-, $HCOOH$, CO_2, $HOBr$, Br_2, $CH_2(COOH)_2\}$

The word $\alpha_1\alpha_2 \dots \alpha_n$ is denoted by $trace(\varrho)$. The infinite word $\alpha_1\alpha_2 \dots$ is an infinite trace whenever there is an infinite execution fragment of of transition system

$$\varrho = p_0\alpha_1 p_1\alpha_2 p_2\alpha_3 p_3 \dots \quad \text{such that} \quad p_i \longrightarrow^{\alpha_{i+1}} p_{i+1} \quad \text{for all} \quad 0 \leq i.$$

The word $\alpha_1\alpha_2 \dots$ is denoted by $trace(\varrho)$ too. An infinite word $trace(\varrho)$ is *circular* if

$$\varrho = p_0\alpha_1 p_1\alpha_2 p_2 \dots \alpha_n p_0\alpha_1 p_1\alpha_2 p_2 \dots \alpha_n p_0\alpha_1 p_1\alpha_2 p_2 \dots \alpha_n p_0 \dots$$

such that $p_i \longrightarrow^{\alpha_{i+1}} p_{i+1}$ for all $0 \leq i < n-1$ and $p_{n-1} \longrightarrow^{\alpha_n} p_0$.

Definition 1

An infinite (resp. finite) trace of state p denoted by $\varrho(p)$ is the trace of an infinite (resp. finite) execution fragment starting in p.

Each trace can be regarded as a graph, where nodes represent states and edges transitions. In this way, transition system is viewed as graph trees. On the other hand, in proof theory we can consider derivations as graph trees too. This allows us to simulate transition system within the framework of proof theory.

By a *derivation tree*, we mean a possibly infinite tree of formulas in which each parent formula is obtained as the conclusion of an inference rule with its children as premises.

Definition 2

Let For denote the set of all well-formed formulas in some language and Rul denote some set of rules. Let $n \in \mathbf{N}$ be the maximum number of premises of any $R \in Rul$. Then a derivation graph is given by $\langle V, s, r, \tau \rangle$, where: V is a set of nodes, $s\colon V \mapsto For$, r is a partial mapping from V to Rul, and τ is a partial mapping from V to V^n (we write $\tau_j(v)$ for the j-th component of $\tau(v)$); for all $v \in V$, $\tau_j(v)$ is defined just in case $r(v)$ is a rule with m premises, $1 \leq j \leq m$ and:

$$\frac{s(\tau_1(v)) \ldots s(\tau_m(v))}{s(v)}$$

is an instance of rule $r(v)$.

A derivation graph $\mathcal{G}$ consists of the set V of nodes and the set of edges $E = \{\langle v, \tau_j, v' \rangle \colon v, v' \in V, \tau_j(v) = v'\}$. The finite word $\tau_1\tau_2 \ldots \tau_n$ is a *finite trace of derivation* whenever there is a finite execution fragment of transition system

$$\pi = v_0\tau_1v_1\tau_2 \ldots \tau_nv_n \ \text{ such that } \ \tau_{i+1}(v_i) = v_{i+1} \ \text{ for all } \ 0 \leq i < n.$$

The word $\tau_1\tau_2 \ldots \tau_n$ is denoted by $trace(\pi)$. The infinite word $\tau_1\tau_2 \ldots$ is an *infinite trace of derivation* whenever there is an infinite execution fragment of of transition system

$$\pi = v_0\tau_1v_1\tau_2 \ldots \ \text{ such that } \ \tau_{i+1}(v_i) = v_{i+1} \ \text{ for all } \ 0 \leq i.$$

The word $\tau_1\tau_2 \ldots$ is denoted by $trace(\pi)$ too. An infinite word $trace(\pi)$ is *circular* if

$$\pi = v_0\tau_1v_1\tau_2 \ldots \tau_nv_0\tau_1v_1\tau_2 \ldots \tau_nv_0\tau_1v_1\tau_2 \ldots \tau_nv_0 \ldots$$

such that $\tau_{i+1}(v_i) = v_{i+1}$ for all $0 \leq i < n-1$ and $\tau_n(v_{n-1}) = v_0$. For more details about circular proofs see [9], [10], [11], [20].

Definition 3

A derivation graph $\mathcal{G}$ is a derivation tree if there is a distinguished node $v_0 \in V$ such that for all $v \in V$, there is a unique trace in $\mathcal{G}$ from v_0 to v. Then v_0 is called the root of the tree. It is denoted by $\pi(v_0)$.

Using the bisimulation principle [14], [16], [19], we can identify a transition system with a proof system, namely we can show that there exist

$$\frac{Ce^{3+}\&HBrO_2\&BrO_3^-\&H^+\ \dagger}{Ce^{4+}\oplus HBrO_2\oplus H_2O}\ Axiom\ (1),$$

$$\frac{Ce^{4+}\oplus HBrO_2\oplus H_2O}{Ce^{4+}}\ Inf.rule\ 2\times(CL),$$

$$\frac{Ce^{4+}}{Ce^{4+}\&BrCH(COOH)_2\&H_2O}\ Inf.rule\ 2\times(FL),$$

$$\frac{Ce^{4+}\&BrCH(COOH)_2\&H_2O}{Br^-\oplus Ce^{3+}\oplus HCOOH\oplus CO_2\oplus H^+}\ Axiom\ (2),$$

$$\frac{Br^-\oplus Ce^{3+}\oplus HCOOH\oplus CO_2\oplus H^+}{Ce^{3+}}\ Inf.rules\ (CR), 3\times(LR),$$

$$\frac{Ce^{3+}}{Ce^{3+}\&HBrO_2\&BrO_3^-\&H^+\ \dagger}\ Inf.rule\ 3\times(FL),$$

Figure 4. The derivation tree of proof-theoretic simulation of the Belousov-Zhabotinsky circular feedback $Ce^{3+} \longrightarrow Ce^{4+} \longrightarrow Ce^{3+} \longrightarrow \ldots$. We see that the pictured derivation tree has a *cycle*. This cycle presents the circular feedback in Belousov-Zhabotinsky reaction (more precisely temporal oscillations in a well-stirred solution): cerium(III) is colorless and cerium(IV) is yellow. Under some conditions, this cycle will repeat several times: in the beginning the solution is colorless, later it is yellow, later it is colorless, etc.

a transition system and a proof system such that for any ϱ and π (resp. p and v_0) $trace(\varrho)$ and $trace(\pi)$ (resp. $\varrho(p)$ and $\pi(v_0)$) are bisimilar.

Let us return to process calculus of reaction-diffusion computing. Due to the bisimulation between a transition system and a proof system, we can set up proof-theoretic frameworks of this process calculus. For example, we can define the proof-theoretic simulation of Belousov-Zhabotinsky reaction as follows:

Definition 4

Let $L = \{Ce^{3+}, HBrO_2, BrO_3^-, H^+, Ce^{4+}, H_2O, BrCH\ (COOH)_2, Br^-, HCOOH, CO_2, HOBr, Br_2, CH_2(COOH)_2\}$ be the set of states. The set of initial transitions Σ is defined by axioms in Fig. 2. The four basic operations $\oplus$ (choice), $\&$ (fusion), $|$ (parallel), $\backslash$ (hiding) are defined by inference rules in Fig. 3, which describe general properties of transitions. Then the proof-theoretic simulation of Belousov-Zhabotinsky reaction is a tuple $\langle L, \Sigma, \oplus, \&, |, \backslash\rangle$.

In this system, transitions between states are identified with derivations of states. The example is given in Fig. 4. Each step of derivation means a transition. As a result, the circular trace of state Ce^{3+} (resp. Ce^{4+}) has a meaning of circular proof, where the state Ce^{3+} (resp. Ce^{4+}) is unfolded infinitely often among premisses and at the same time among derivable expressions.

4. Conclusion

Self-organization phenomena in nature assume circularity and cause-and-effect feedback relations: each component affects the other components, but these components in turn affect the first component. The most popular example of such self-organization is presented by Belousov-Zhabotinsky reaction. In this system we observe circularity in the interchange of solution color: in the beginning the solution is colorless, then it becomes yellow, then it becomes colorless, etc. *In logical simulation of Belousov-Zhabotinsky reaction we obtain circular proofs. This shows that reaction-diffusion computing cannot dispense with logical circularity like cyclic proofs and feedback relations in state transitions.* We can suppose that logical circularity should be a key notion of “life computer”, i.e. of each self-organized reaction-diffusion system.

References

[1] Adamatzky A. Reaction-diffusion algorithm for constructing discrete generalized Voronoi diagram. Neural Network World 6, 1994, 635–643.

[2] Adamatzky A. Computing in Nonlinear Media and Automata Collectives. Institute of Physics Publishing, 2001.

[3] Adamatzky A., De Lacy Costello B., Asai T. Reaction-Diffusion Computers, Elsevier, 2005.

[4] Adamatzky A., A. Wuensche, and B. De Lacy Costello, Glider-based computation in reaction-diffusion hexagonal cellular automata, Chaos, Solitons & Fractals 27, 2006, 287–295.

[5] Adamatzky A. and Wuensche A. Computing in spiral rule reaction-diffusion hexagonal cellular automaton. Complex Systems 16, 2007.

[6] Berry G., Boudol G. The chemical abstract machine, Teor. Comput. Sci., 96, 1992, 217–248.

[7] Belousov V. Synergetics and biological morphogenesis, Self-organization: Autowaves and Structures Far From Equilibrium. Ed. V. I. Krinsky. Heidelberg-Springer, 1984, 204–208.

[8] Boerlijst M. C., Lamers M. E., Hogeweg P. Evolutionary consequences of spiral waves in a horst-parasitoid system, Proc. R. Soc., B 253, 1993, 15–18.

[9] Brotherston, J., Cyclic proofs for first-order logic with inductive definitions [in:] B. Beckert, editor, *TABLEAUX 2005*, volume 3702 of LNAI, Springer-Verlag, 2005, 78–92.

[10] Brotherston, J., *Sequent Calculus Proof Systems for Inductive Definitions.* PhD thesis, University of Edinburgh, November 2006.

[11] Brotherston, J., Simpson, A., Complete sequent calculi for induction and infinite descent. *LICS-22*, IEEE Computer Society, July 2007, 51–60.

[12] Drodland G. W., Clausi D. A. Embryonic tissue morphogenesis modeled by FEM, Journal Biomech. Eng., 116, 1994, 146–155.

[13] Feynman R. P. Simulating physics with computers, Int. J. of Theor. Phys., 21, 1982.

[14] Fiore, M.P., A coinduction principle for recursive data types based on bisimulation. *Proc. 8th Conf. Logic in Computer Science (LICS'93).* 1993, 110–119.

[15] Hafner G. S., Tokarski T. R. Morphogenesis and pattern formation in the retina of the crayfish, Procambarus clarkii Cell and Tissue Res., 293, 1998, 535–550.

[16] Howe, D. J., Proving congruence of bisimulation in functional programming languages, *Inf. and Comp.* 124, 1996.

[17] Ivanitsky G. R., Kunisky A. S., Tzyganov M. A. Study of 'target patterns' in a phage-bacterium system, Self-organization: Autowaves and Structures Far From Equilibrium. Ed. V. I. Krinsky. Heidelberg-Springer, 1984, 214–217.

[18] Prajer M., Fleury A., Laurent M. Dynamics of calcium regulation in Paramecium and possible morphogenetic implication, Journal Cell Sci., 110, 1997, 529–535.

[19] Rutten, J. J. M. M., Processes as terms: non-well-founded models for bisimulation. *Math. Struct. in Comp. Sci.*, 2 (3), 1992, 257–275.

[20] Santocanale L., A calculus of circular proofs and its categorical semantics [in:] M. Nielsen and U. Engberg, editors, *Proc. of FoSSaCS 2002*, Grenoble, Apr. 2002, Springer-Verlag LNCS 2303, 357–371.

[21] Vink E. P. de, and J. J. M. M. Rutten. Bisimulation for probabilistic transition systems: a coalgebraic approach. *Theor. Comp. Sci.*, 221, 1999, 271–293.

Andrew Schumann
Department of Philosophy and Science Methodology
Belarusian State University
Minsk, Belarus
Andrew.Schumann@gmail.com

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17 (30) 2009

Adam Krasuski
Szkoła Główna Służby Pożarniczej

Tadeusz Maciak
Szkoła Główna Służby Pożarniczej
Politechnika Białostocka

Karol Kreński
Szkoła Główna Służby Pożarniczej

DECISION SUPPORT SYSTEM FOR FIRE SERVICE BASED ON DISTRIBUTED DATABASE AND CASE-BASED REASONING

1. Introduction

Making decisions in complex situations is extremely difficult. Short time for analysing aspects of the incident and the psychological pressure, make the case even worse. Conditions in which decisions are made are common for Rescue Services. In order to overcome these difficulties, a number of research projects have been initiated, all aimed at finding the best way of making decisions in emergency situations.

In 1989, Klein, Calderwood, and Clinton-Cirocco presented a method of making decisions in emergency situations called the Recognition-primed Decision (RPD) [1]. In RPD, the decision maker (DM) considers the particular situation's similarity to the previous ones, and then takes a course of action based on the experience learned. Despite the fact that there are no two identical incidents, DM is able to tailor the strategy through understanding the goals, cues, and expectancies acquired through experience of similar incidents. The study conducted by Klein and co-workers concludes that 80% of non-routine decisions by Incident Commanders during the rescue action are made using RPD, with the assumption that routine incidents would show an even higher percentage of RPD.

There are also weaknesses of RPD. To make a good decision, DM has to have experience. Klein's group research proved that commanders with large experience take right decision at once. On the other hand, inexperienced commanders tend to take wrong decisions using RPD [2].

ISBN 978–83–7431–231–8 ISSN 0860-150X

There is a wide diversity of incidents that the National Fire Service in Poland must deal with. On the global scale, the similarity among the incidents is considerably high. However, at the fire stations' level, there is a very small probability that two similar rescue situations may occur. Furthermore, there is no system that gathers the experience gained during the actions, and makes it available to the other firemen. Based on the above, it can be concluded, that no Incident Commander possesses efficient knowledge to successfully use RPD. In connection with this, mistakes made by one Commander, are repeated by others. Therefore, there is no possibility to successfully use RPD by Commanders in the National Fire Service in Poland without a special system that gathers all experience gained by Commanders and makes it available to all firemen during the rescue actions.

Artificial intelligence research brought about methods that simulate human thinking processes. The most popular technique that closely simulates the RPD method is the Case-based Reasoning (CBR) [3].

This article describes Decision Support System (DSS) for the National Fire Service of Poland. The system is based on Directory Services and Case-based Reasoning techniques. Section 2 contains Case-based Reasoning technique principles. Section 3 contains basic information about the Directory Service. In section 4, description of the proposed Decision Support System is included. Conclusions are contained in the summary at the end of the paper.

2. Case-based Reasoning principles

Case-based Reasoning is an artificial intelligence technique that simulates the process by which humans solve problems [3, 4]. It's procedure rule is very similar to RPD. CBR systems solve problems by retrieving solutions to similar problems that have been solved in the past. Next, it adapts these solutions to the new circumstances, and it teaches itself during the process.

CBR systems store information or experience in cases. Cases are individual information entities that contain a problem description and a solution. Each case is independent of other cases and can be added to, and removed from the database without affecting other cases.

Each new problem is typically represented as a case without the solution. Attributes that describe the case are compared to the cases stored in the database, and the most comparable solution is retrieved. The retrieved solution is adapted to the new circumstances and reused as a potential solution. If this adapted solution is acceptable, the problem is considered to be solved by the system. If the suggested solution needs modification, it is revised, and the appropriate solution is prepared. The next time a user requests the solution to this problem, the newly learned case is available immediately. The learning capability allows CBR systems to continuously expand their knowledge of the operating domain. The process of solving and gathering new cases is presented on figure 1.

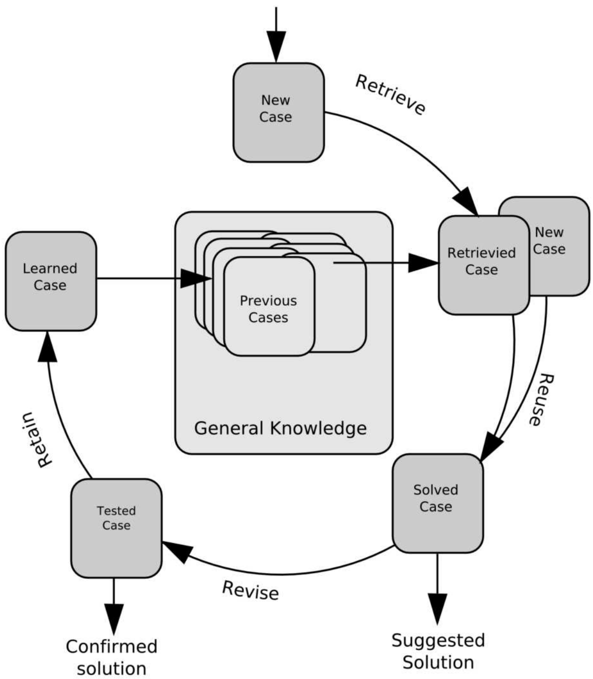


Figure 1. The Case-based Reasoning Cycle [5]

The quality of the case-based solution depends on four main factors:
- The representation of the domain;
- The similarity calculation;
- The adaptation capability;
- The quality of the cases.

Typical application areas of CBR include [5]:
- Troubleshooting and diagnosis of technical equipment;
- Customer support and help-desk;
- Engineering and design;
- Medical applications.

3. Directory Service principles

Directory Service is a simplified database [6]. Usually, it does not have typical database mechanisms, like transactions support or recovery protocol. Directories allow for both read and write operations, but are intended primarily for high-volume, efficient read operations by clients.

Lightweight Directory Access Protocol (LDAP) is a distributed directory service protocol. LDAP is based on a client-server model and runs over TCP/IP. It can be used to access stand-alone directory servers or X.500 directories [7].

Information is stored in the LDAP directory in the form of entries arranged in hierarchical tree-like structure. An LDAP entry is a collection of attributes. Entries are a representation of the real world object. For example, an entry can represent a person. Attributes that describe such an object could be: name, surname, email address. Each attribute has a name and one or many values. For example the attribute *email address* has a name *mail*. Attribute value *mail* could be *foo@foo.com*. An exemplary entry is presented on figure 2.

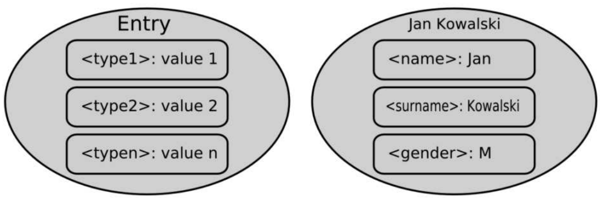


Figure 2. The idea and representation of an entry

LDAP defines operations for querying and updating a directory. There are operations provided for adding and deleting entries from the directory, changing the value of an existing entry, or changing the name of an entry. However, LDAP is mostly used to search for information in a directory. The LDAP search operation allows some portion of the directory to be searched for entries that match some criteria specified by the search filter. Information can be requested from each entry that matches the criteria.

4. Decision Support System for Fire Service

CBR technique was used by the author to create Decision Support System (DSS) for the State Fire Service in Poland. The CBR technique is based on descriptions of rescue actions called cases. Cases are created by defining special attribute types. The set of attributes depends on rescue actions kind. The cases are arranged in groups, creating a tree-like structure. All groups are stored in the Directory Service.

According to regulation [8], rescue actions can be divided into three main categories: fires, local threats, and false alarms. Each of these categories contains subcategory according to the type of fires or incidents. For example, fire category contains among others: woodland fires, house fires, and industrial fires. All the entries within the category use the same set of attributes that describes a rescue action. The set of attributes among categories is different.

The entries categorization described above was the base for creating directories in the Directory Service. Root directory contains three contexts called: fires, local threats, and false alarms. They were created as an object container in the directory name space. All of these containers have subcontainers for accidents types described by the regulation [8]. The fires group includes example subcategories:

- Entertainment buildings fires
- Residential building fires
- Transport fires
- Woodland fires

The local threats group includes example subcategories:

- Chemical accidents
- Ecological accidents
- Building collapse
- Road traffic accidents

The false alarm group contains among others:

- Detected by people
- From automatic fire detection system

Figure 3 illustrates structure of entry categorization.

All containers and subcontainers must have the attribute *name* defined. The corresponding value of the attribute must be one of the categories, presented on the figure 3, e.g. *name = Natural Calamity*.

Described subcontainers contain entries that represent real rescue actions. The selection of the attributes depends on the accident type. The attributes used and their values are the key to the efficiency of the CBR technique. It is very important to set accurate attributes. The sets of attributes currently used in the research are based on the regulation [8] and interviews with firemen who have taken part in rescue actions.

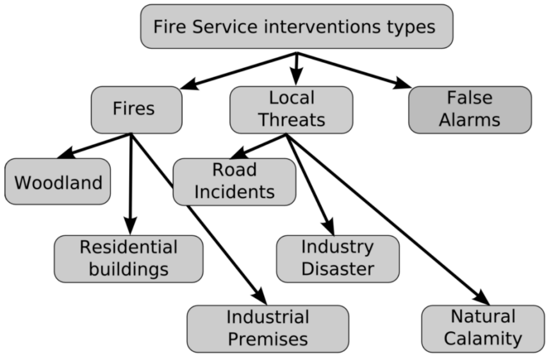


Figure 3. Incidents categorization tree

The abstract class *case* was created to represent object accident, as the effect of this study. The class includes three main attributes:

- Object type – identifies the type of an object in the accident. In the case of fires, the attribute values could be one of: hotels, industrial units, cars, etc. The list of valid values are defined by regulation [8];
- Accident size – identifies the size of an accident. The possible attribute values are: small, local, medium, large, very large. The classification of accidents is described in the regulation;
- Solution – contains a short and structured description of how the problem was fixed. This attribute contains the key information which is needed to manage the accident properly.

The abstract class *case* is extended by an inherited class which describes detailed type of the accident. For example, class *home_fire_case* describes home fires accidents and extends the abstract class *case*. This class includes a special type attributes, e.g.:

- Hazard category – danger category to which the object belongs;
- Water supply – type of water supply that the object contains. The possible values are: drencher, sprinklers, internal hydrant system;
- External hydrant system;
- Water storage tank;
- Object area;
- Cubic measure;
- Storeys number;
- Number of building's entries;
- Number of staircases;
- Exposed building;
- Fire zone;

- Largest fire zone;
- Electric system;
- Gas system;
- Building structure;
- Roof construction;
- Wall construction;
- Rescue equipment;
- etc.

The remaining special classes extend abstract class *case* analogically, defining the set of special attributes. The classification method and the list of obligatory attributes which belong to the detailed class are included in [9].

The structure described is a data deploying schema in the Directory Service. However, for better performance, LDAP is not a standalone service. Containers and included entities are distributed across nodes. The network architecture is composed of servers located in fire stations. The server located in the Headquarters of the State Fire Service contains RootDSE and three main contexts. Those are: fires, local threats, and false alarms. Containers which represent contexts, include objects of the class *referral* which are pointers to subcontainers stored on the servers located in the Districts Headquarters. Each server in the Districts Headquarters, stores two containers with detailed sets of accidents. For better reliability, each of the nodes is replicated by it's slave node.

The network architecture of distributed directory is presented on figure 4.

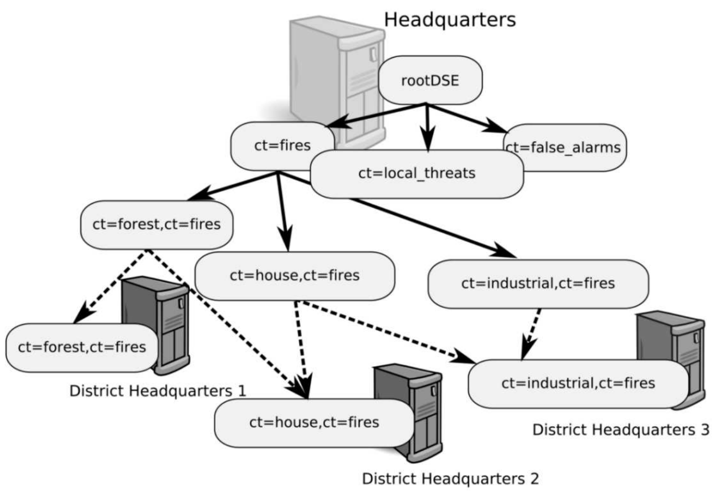


Figure 4. Proposed distributed database architecture for DSS

This solution has many advantages, such as:

- Data distribution causes that nodes have less amounts of data to process. That results in shorter response time;
- Increasing number of processors in the system shortens the response time;
- Data replication increases the reliability of the system.

The key issue in the Decision Support System is finding the solution for a new problem. The CBR technique is used for describing DSS. As mentioned above, the CBR finds solutions by comparing cases stored in the database, with the new case. In the proposed Decision Support System, the cases are described by the sets of attributes. The comparing algorithm proposed by Cortes is involved [5]. The similarity between a query (Q) and a case (C) is determined by calculating the weighted sum of the similarities between the values for each attribute.

$$\mathrm{SIM}(Q,C) = \frac{\sum_{i=1}^{n} w_i \,\mathrm{sim}_i(Q_i, C_i)}{n}$$

In the formula given above:

- Q is the query or the set of attributes describing a new case,
- C is the case stored in the directory,
- $\mathrm{SIM}(Q,C)$ is overall similarity between the query and the case,
- n is the number of attributes,
- w_i is the weighting factor for the attribute,
- i and sim_i is the similarity measure for the attribute i of the query and the case.

It is worth noticing, that to compare a query and a case, the weighted mean is used. On account of that, the detailed attributes have different influence for successfully solving the problem. For example, in the case of house fire where no external hydrant system is available, the key issue to fix the problem is the appropriately organized water supply. In the case an external hydrants system is present, the water supply organization is less important.

The presented formula for comparing queries and cases is of a general form. Depending on the situation, the particular algorithm should be involved. For example, during a traffic accident, a tank containing an explosive liquid was destroyed. One of the describing attributes of such an accident is *substance concentration*. Electronic detector indicated that the concentration is above lower explosion limit (LEL). The weight of the attribute *concentration* in the comparison algorithm is significant. There are two cases in the database that describe a similar situation. In the first case, the difference between the query and the case is very small but *concentration* is under LEL. In the second case the difference is significant, but the concentration is above LEL. It is obvious, that a better solution and possible danger are better described in the second case.

The example mentioned reveals that comparison of cases is not simple in some situations. The algorithms for such cases should be determined after detailed research, which have not been conducted. At the current stage of the study only a general comparison algorithm was tested.

Another relevant CBR issue is solution adaptation. At the current stage of research the adaptation is calculated by the Incident Commander. He receives only the solution and plausible dangers included in the most similar case from the database. Based on this and on the incident observation he adapts the solution.

The system works according to the following scenario:

The Control Room Commander (CRC) receives an emergency call about a fire or another accident. Information received from the caller is entered to the system via special forms. These forms allow for searching for the cases against the database. It allows for pre-selection of the cases stored in the database. For example, category *fires → woodland fires*. Hierarchical structure of the directory database narrows the number of directories being searched, and speeds up the response. The CRC next sends an appliance crew into the accident ground. After arriving at the fire ground, the Incident Commander sends a report to the CRC about the situation. It is a base for the CRC to look for a similar case in the database. The system finds the most similar case, then it shows the solution and plausible dangers on the screen based on the algorithms described above. The CRC sends this information to the Incident Commander. The Incident Commander using the proposed solution and it's own accident valuation, adapts the solution to the case.

The situation may change throughout the rescue action. Every time the Incident Commander sends the details of the action to the CRC, the CRC can search for new solutions.

After fixing the problem, attributes of the accident and the solution are entered to the database, and a new case is created.

The system will become useful in real rescue actions, as soon as the database is complemented with a considerable number of consecutive cases. At the final stage of the process, the cases are being gathered directly from the Incident Commanders reports. To start the system, a minimal number of cases in the database is needed. In order to overcome this difficulty, a special module has been created. The aim of this module is to fill the database with rescue actions from the past. The data will be acquired from the filing system EWID, and from text documents called Accident Analyses.

EWID provides the basic information, e.g.: category of the accident, object type, and its size. The rest of the attributes and solutions will be gathered from Accident Analyses. Accident Analyses are unstructured text documents. Special methodology, like text mining is necessary to create high quality information. The methodology of the data acquisition is described in [10].

5. Summary

According to Klein's group research, RPD is the most optimal way for taking decisions in emergency situations. CBR is a technique for simulating RPD in computers systems. The basis for CBR is a special database containing cases. In the proposed Decision Support System, these cases are the representations of real rescue actions. Cases are described by special type attributes, grouped in sets which depend on the type of the accident. The cases also contain solutions to the described problems. All the cases are divided into special groups which form a hierarchical structure. This structure is the base for the creation of the directory information tree in the Directory Service. The Directory Service is extremely efficient in data searching. To find the solution for a new accident, attributes of the accidents are compared against cases stored in databases. Special comparing algorithms are involved. The solution found is sent to the Incident Commander on the fire ground. Using this information and the Incident Commander's own situation evaluation, he adapts a solution to given conditions. Successful solution and the accident attributes are entered to the database, which results in creating a new case.

Bibliography

1. Klein G.: Sources of Power: How People Make Decisions. MIT Press. Massachusetts 1999.
2. Ross K. G.: The Recognition-Primed Decision Model. "Military Review" Nr 4, Lipiec – Sierpień 2004. str. 6–10.
3. Aamodt A., Plaza W.: Case-Based Reasoning. "IOComm – Artificial Intelligence Communications" Nr 1, Marzec 1994. str. 39–59.

4. Shiu S. C. K., Pal S. K.: Case-Base Reasoning: Concepts, Features and Soft Computing. "Applied Intelligence" Nr 3, Listopad 2004. str. 233–238.
5. Cortes U. et. al.: Deliverable 4.1. Report on current reasoning engine practice and integration strategies. W: RIMSAT 2003.
6. Arkills B.: LDAP Directories Explained. An Introduction and Analysis. Addison Wesley. Reading 2004.
7. Yeong W., Howes T., Kille S.: X.500 Lightweight Directory Access Protocol. W: RFC1487 Reston 1993.
8. Rozporządzenie Ministra Spraw Wewnętrznych i Administracji z dnia 29 grudnia 1999 r. w sprawie szczegółowych zasad organizacji Krajowego Systemu Ratowniczo-Gaśniczego.
9. Maciak T., Kreński K.: Dobór atrybutów bazy przeciwpożarowej budynków systemu informacji przestrzennej służb ratowniczych. "Zeszyty Naukowe SGSP" Nr 32, Kwiecień 2005. str. 59–67.
10. Loh S., Oliveira J. P. M., Gammeiro M. A.: Knowledge Discovery in Texts for Constructing Decision Support System. "Applied Intelligence" Nr 3, Maj 2003. str. 357–366.

Adam Krasuski
The Main School of Fire Service
Faculty of Fire Safety Engineering
Slowackiego Street 52/54, 01–629 Warsaw
krasus@inf.sgsp.edu.pl

Tadeusz Maciak
The Main School of Fire Service
Faculty of Fire Safety Engineering
Slowackiego Street 52/54, 01–629 Warsaw

Białystok Technical University
Computer Science Faculty
Wiejska Street 45A, 15–351 Białystok
maciak@sgsp.edu.pl

Karol Kreński
The Main School of Fire Service
Faculty of Fire Safety Engineering
Slowackiego Street 52/54, 01–629 Warsaw
mimooh@inf.sgsp.edu.pl

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17 (30) 2009

Stanislaw Zadrozny

UNIQUE LOGIC OF THOUGHT

Part I

Introduction

The present world is very different from the world of previous centuries. As never before in history, human activity today depends upon the efficient flow of information from newspapers, radio, TV, and, most importantly, from Internet access by PC users. Although people are not particularly conscious of it, thinking could not exist without information. Thinking is a process done individually, but it could not have developed without intelligent social interaction. It is compounded by the contributions of other people who create, communicate, and gather the results of perceived and cognizant reality. Thinking is impossible without socially evolved languages, numbers describing quantities, and graphs or pictures delineating qualities, all of which allows information to be distributed throughout society.

Our thesis, briefly stated, is that, by utilizing the concept of spaciousness, which may later be simulated by the computer, one will be able to imitate, replace, and extend the process of thinking. This concept of thought builds upon ideas previously discovered and described by Henri Bergson: "We express ourselves by words because of necessity but we think most often in space." Through language, the results of thinking are presented in a linear fashion, while the entire process occurs spaciously within the mind utilizing information as a medium. The premise underlying our thesis is that the construction of a spacious model of the mind on the proper level will enable a transfer of the thought process to a computer, which, in turn, would provide meaningful results.

Actual progress in the field of computer science and information technology has left the philosophical concept of information in its wake. It lacks a clear description of what is or is not information per se. On the side of

ISBN 978–83–7431–231–8 ISSN 0860-150X

computer science, there are mega, giga, and tera bytes, along with everything that can be done by them; on the side of the theory of knowledge, however, a conceptual apparatus that would allow for the full utilization of such potential does not yet exist.

The paramount concern of information technology is to provide speed, accuracy, and flow of the volume of data being reworked. The theory of knowledge should step down from its ivory tower and essentially provide keys to the interpretation and reinterpretation of these data. It can be assumed that closing the gap between computer science and the theory of knowledge will open up a new path to artificial thinking. Likewise, there is a strong belief that replacing language with information in the basic logical functions is the first step in this direction, since information, not language, is the medium through which thinking is realized. In order to specify the conditions under which thinking and its attributes are to be realized, our first question should be: What is information?

Information

The following definition is proposed: **"Information is a representation of reality objectified in a given code system"**. **Representation** may be described by synonyms, such as, presentation, reflection, projection, description, depiction, imitation, etc. The meaning of all of these words is similar, and it refers to the relationship between an object and its representation. With the word **reality**, one understands everything that can become the object of perception, inquiry, cognition, reasoning, investigation, experimentation, presentation, etc. Reality refers to physical objects, processes, and also to abstract ideas, notions, concepts, etc. **Objectified** means presented according to commonly accepted rules of operation and supported by a verified method and a proper technique. With the expression **given code system**, one understands a particular form of signs, symbols, and graphs selected for a presentation of the shape of information on a given carrier. An item that represents a single distinguished part of reality will be defined as an element of information (ei).

Levels of Analysis (Vertical Projection)

The acceptance of the concept that an element of information (ei) relates, on the one hand, to reality and, on the other hand, to the categories of cognition enables one to view particular information in the abstract. However, the result of representation, i.e., an element of information (ei), also has a vertical projection that can be examined on four levels, which we will call *levels of analysis*.

The level of analysis that describes an element of information (ei) within the framework of particular information is marked as 0 (zero). This represents the relationship between reality and information, as well as categories of cognition; and it also demonstrates that information, when it is free from particulars has a universal character. At this point, the structure of information is examined. The levels of analysis are presented as follows:

REALITY $\leftrightarrow$ elements of inform.	level 0 $\leftrightarrow$ categories of cognition $\downarrow$ [down]
words/express./figures	level 1 $\leftrightarrow$ code of vocabulary
letters/numbers/graphs	level 2 $\leftrightarrow$ code of signs
conf./sequen. of signals	level 3 $\leftrightarrow$ code of signals

These three (1, 2, 3) levels represent the shape of information, i.e., something material. The zero (0) level of elements of information has a virtual character. The existence of the zero (0) level implies a necessity for an interpretation requiring the possession of pertinent knowledge.

The level dealing with lingual description, parametrical expression, or graphical presentation is accepted as the next level and labeled (1). The level of analysis that consists of letters in written form or syllables in a spoken language, as well as numbers, digital symbols, and visual representation, such as graphs, ideograms, etc., are accepted as the next level and labeled (2).

The last distinguished level of analysis is called level of signals[1] and labeled (3). Because the shape of information is transmitted by signals in the telecommunication channel, the most popular concept of information pertains to a signal as a unit of information. This leads to the conclusion that more signals equal more information.

Regarding the issue of the quantity of information, our thesis claims that this can be clarified with respect to the level of analysis. Analysis of a certain structure of information on the 0 (zero) level enables the distinguishing of a certain number of elements of information within the scope of horizontal projection.

[1] SIGNAL, electrical impulse, light electromagnetic wave. Attention! Because a signal is a specific physical phenomenon that: 1) is generated by a certain system or transmitter for the purpose of conveying information; 2) is generated spontaneously by physical objects/processes from the surrounding world. It should be mentioned that the object of our interest in this work is the signal that is generated intentionally.

The fact that many different codes exist in relation to the three (1, 2, 3) distinguished levels gives rise to situations in which a given number of words/expressions/figures or a quantity of letters/numbers/graphs or a number of sequence of signals will be different, although the number of elements of information will remain the same.

The basic software utilized in each personal computer manifests a state close to the operational perfection of conversion between the third (3), second (2), and first (1) levels, i.e., from signals into signs, and signs into expression. Herein arises the problem that is still awaiting a solution – namely, the interpretation and reinterpretation of this meaning during the passage from the first (1) to the zero (0), and from the zero (0) to the first (1), levels of analysis.

Basic concepts

When issues of information in the most general sense are considered, the following situation thus occurs: There may or may not be information about a given event (x). This can be described as follows:

$\mathrm{I}\ (\mathrm{x}) \rightarrow \mathrm{e}\ (\mathrm{x})$

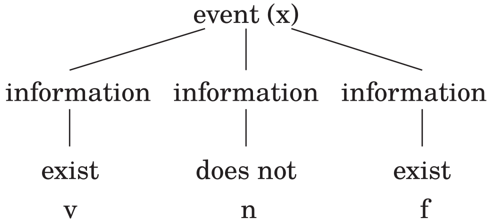
$\mathrm{n} \rightarrow \mathrm{e}\ (\mathrm{x})$ n (nullus)

In an instance in which some information about an event x does exist, it could be true or false information. This would be expressed as follows:

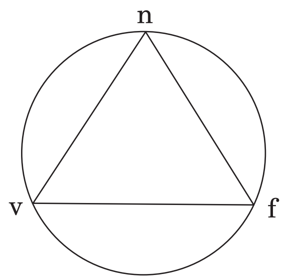
$\mathrm{I}\ (\mathrm{x}) \rightarrow \mathrm{e}\ (\mathrm{x})$ than I (x) could be true (verum)

or I (x) could be false (falsum)

Existence of information is superior, with regard to either truthfulness or falsity; thus, nonexistence is the negation of either true or false information.



It is assumed that truthfulness (v), nonexistence (n), and falsity (f) are states that make up a universal base (foundation) for analyzing information from the valued point of view. By analogy, in physics one has voltage (+), lack of voltage (o), or negative voltage (–). Similarly, in mathematics there are positive numbers (+), zero (o), and negative numbers (–). The Wankel engine would represent a similar mechanical analogy. The above relation could be presented by utilizing an equilateral triangle inscribed in a circle.



The above analysis provides a basis for replacing a proposition (sentence) with a segment of information in logical matrices, and likewise extending these for three values instead of two.

Truth-Functional Connective

The truth-functional connective for the three values (v, n, f)[2] – could be presented as follows. If there is one argument (p), then the matrix of negation will contain the following possibilities:

p	np^1	np^2
v	f	n
n	v	f
f	n	v

If p is v, then its negation is f and n; if p is n, then its negation is v and f; if p is f, then its negation is n and v. In other words, as this illustration makes clear, the assumption of one value excludes the other two.

If there are two arguments (p, q), then the combined matrix for alternative "$\vee$",[3] disjunction "/", conjunction "$\wedge$", equivalence "$\equiv$", and implication "$\rightarrow$" could be presented as follows:

[2] For a presentation of other concepts of multi-valued logic, see: [6] and [4] There, the author presents the concept and matrix of Lukasiewicz's 3-valued logic.

[3] The functions of alternative and disjunction are presented here separately, see [8] and [3].

p	q	p∨q	p/q	p∧q	p≡q	p→q
v	v	v	f	v	v	v
v	f	v	v	f	f	f
v	n	v	f	n	n	v
n	v	v	f	n	n	v
n	n	n	n	n	v	v
n	f	f	v	n	n	v
f	n	f	v	n	n	v
f	v	v	v	f	f	v
f	f	f	v	f	v	v

During the realization of a certain task, if **choice**, **elimination**, **coordination**, **estimation**, and **inference** were applied, then the above dependencies indicate what the value of these functions would be for a certain combination of two arguments. From this premise, logic in the most general manner provides guidance concerning what would be expected as a result of thought, i.e., the dynamics of the process of thinking.

ALTERNATIVE

If there are two arguments (p, q), then the matrix for the alternative would be presented as follows:

	p	q	p ∨ q	comments
1	v	v	v	suffices → for a choice of true argument
2	v	f	v	does not exclude → a choice of true argument
3	v	n	v	does not exclude → a choice of true argument
4	n	v	v	does not exclude → a choice of true argument
5	n	n	n	excludes → a choice
6	n	f	f	excludes → a choice of true argument
7	f	n	f	excludes → a choice of true argument
8	f	v	v	does not exclude → a choice of true argument
9	f	f	f	excludes → a choice of true argument

If **choice** is applied during the realization of a certain task, then the above dependencies show what the value of the alternative would be for a certain combination of two arguments.

Alternative presents a situation dealing with choice, and it exists when the choice is between two arguments. The alternative is true when at least one argument is true, although both could be true. The first variant v/v fulfills the alternative and represents a sufficient condition. Which argument is chosen is inconsequential, whereas the result will be a choice between true arguments. Four variants, namely, v/f, v/n, n/v, and f/v, also lead to a true alternative, with each representing a possible condition. One variant – n/n – leads to a lack of alternative and represents a necessary condition. Three variants – n/f, f/n, and f/f – lead to a false alternative, with each representing a necessary condition.

DISJUNCTION

If there are two arguments (p, q), then the matrix for disjunction would be presented as follows:

	p	q	p/q	comments
1	v	v	f	excludes → elimination of false argument
2	v	f	v	does not exclude → elimination of false argument
3	v	n	f	excludes → elimination of false argument
4	n	v	f	excludes → elimination of false argument
5	n	n	n	excludes → elimination
6	n	f	v	does not exclude → elimination of false argument
7	f	n	v	does not exclude → elimination of false argument
8	f	v	v	does not exclude → elimination of false argument
9	f	f	v	suffices → for elimination of false argument

If **elimination** is needed during the realization of a certain task, then the above dependencies indicate what the value of disjunction would be for a certain combination of two arguments.

Disjunction presents a situation related to elimination, and it exists when one of two arguments must be eliminated. Disjunction is true when at least one argument is false, although both could be false. Three variants – v/v, v/n, and n/v – lead to a false disjunction, with each representing a necessary condition. One variant – n/n – excludes an elimination, which also represents a necessary condition. Four variants – v/f, n/f, f/n, and f/v – lead to a true disjunction in which each represents a possible condition.

Variant f/f also leads to a true disjunction, and it represents a sufficient condition. Moreover, it is immaterial which argument would be eliminated, as the false argument would be excluded in any case.

CONJUNCTION

If there are two arguments (p, q), then the matrix for conjunction would be presented as follows:

	p	q	p∧q	comments
1	v	v	v	suffices → for true coordination
2	v	f	f	excludes → true coordination
3	v	n	n	excludes → coordination
4	n	v	n	excludes → coordination
5	n	n	n	excludes → coordination
6	n	f	n	excludes → coordination
7	f	n	n	excludes → coordination
8	f	v	f	excludes → true coordination
9	f	f	f	excludes → true coordination

If **coordination** must be applied during the realization of a certain task, then the above dependencies represent the value of conjunction for a certain combination of two arguments.

Conjunction represents a situation involving coordination, and it exists for the coordination of two arguments. Conjunction is true only when both arguments are true. The first variant – v/v – fulfills conjunction and represents a sufficient condition. Five variants – v/n, n/v, n/n, n/f, and f/n – lead to a lack of conjunction, representing in each instance a necessary condition. Here, something – which could be either true or false – could not be coordinated with nothing. Three variants – v/f, f/v, and f/f – lead to a false conjunction, and in each case they represent a necessary condition.

EQUIVALENCE

If there are two arguments (p, q), then the matrix for equivalence would be presented as follows:

	p	q	p≡q	comments
1	v	v	v	suffices → for true estimation
2	v	f	f	excludes → true estimation
3	v	n	n	excludes → estimation
4	n	v	n	excludes → estimation
5	n	n	v	suffices → for true estimation
6	n	f	n	excludes → estimation
7	f	n	n	excludes → estimation
8	f	v	f	excludes → true estimation
9	f	f	v	suffices → for true estimation

If **estimation** needs to be applied during the realization of a certain task, then the above dependencies illustrate what the value of equivalence would be for a certain combination of two arguments.

Equivalence represents the situation involving estimation. It exists when the value of the first argument is estimated by that of the second. Estimation is true whenever both arguments have the same value. Three variants – v/v, n/n, and f/f – lead to true equivalence, with each representing a sufficient condition. Two variants – v/f and f/v – lead to a false equivalence, and these represent a necessary condition. Four variants – v/n, n/v, n/f, and f/n – lead to a lack of equivalence, and these also represent a necessary condition.

IMPLICATION

If there are two arguments (p, q), then the matrix for implication would be presented as follows:

	p	q	p→q	comments
1	v	v	v	suffices → for true implication
2	v	f	f	excludes → true implication
3	v	n	v	does not exclude → true implication
4	n	v	v	does not exclude → true implication
5	n	n	v	suffices → for true implication
6	n	f	v	does not exclude → true implication
7	f	n	v	does not exclude → true implication
8	f	v	v	does not exclude → true implication
9	f	f	v	suffices → for true implication

If **inference** is applied during the realization of a certain task, then the above dependencies represent what the value of implication would be for a certain combination of two arguments.

Implication represents a situation dealing with inference, and it exists when the first argument represents a premise and the second, a conclusion. Implication is true in all cases, with the exception of variant 2 v/f, since it is not possible for a false conclusion to be inferred from a true premise. In other words, variant v/f represents a necessary condition because this excludes a true implication. However, the same combination of two arguments – v/f – would be false for equivalence and conjunction, yet true for alternative and disjunction. Three variants, namely, v/v, n/n, and f/f, assert inference, since a true premise leads to a true conclusion, the lack of a premise leads to the lack of a conclusion, and a false premise leads to a false conclusion, representing in each instance a sufficient condition (see Equivalence). Five variants – v/n, n/v, n/f, f/v and f/n – also fulfill an implication because they do not exclude true inference, representing in each instance a possible condition (see Alternative, Conjunction, Disjunction).

Truth-Functional Connective (Comments)

When one analyzes the issue of **lack** and its meaning in an abstract manner, two different situations arise: first, when such information does not exist **at all**; and, second, when a certain subject **does not have** the necessary information at a particular time and place, but nevertheless has to make a decision and move forward. Truth-functional matrices apply to both situations. This involves a different approach than the one provided by traditional logic. Two-valued logic is based on each one actual or universal relation between a subject and the object of examination, whereas 3-valued logic refers to the information about this examination. In such a situation, an examination is realized **by proxy**, as the subject deals exclusively with information.

The simplicity and beauty of dichotomous division has enabled it to maintain a prominent position in logic since the time of Aristotle. As a guide to thought, however, such a division appears to be insufficient in situations in which steps are undertaken that are partially based upon a lack of information or even upon obvious ignorance. Examples that come to mind are the construction of airplanes or electronic computing machines. What could have been predicted over 60 years ago about computer applications? The point to which technological progress in the field of electronics has advanced humankind was not only improbable at that time, but was even beyond one's wildest imagination. As never before in history, the actual virtual reality of information confronts a subject with a situation in which one has either true information, false information, or no information. Moreover, it is irrelevant if such information does not exist or if it is merely inaccessible.

Part II

Amount of Information

In the previous section, matrices of 3-valued logic were presented that depict an abstract situation in which, in addition to true and false information, the realization of a certain task takes into account a lack of information as a necessary component of logical relation. Now it would be constructive to return to one of the key questions related to the **quantity** or **amount** of information. Here we will refrain from discussing **Shannon's** concept of **quantity** of information, which is based upon transferring signals by wire, because this concept is commonly known[4]. We do wish to state, however, that, with the exception of the area of telecommunications, his concept has caused more confusion than enlightenment.

There is another aspect of the value of information that should be considered. Whenever an event x occurs, it is named an element of reality (er) that could be presented by one element of information (ei), and then the following possibilities occur: First (p^1), that the representation is true (v); second (p^2), that the representation has not been achieved (n); and, third (p^3), that the representation is false (f).

	p^1	p^2	p^3
er	v	n	f

How many possibilities can be found if the object of description contains more than one element? If there are two (er^1, er^2) elements of reality, then the number of possibilities (p) is nine.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
er^1	v	v	v	n	n	n	f	f	f
er^2	v	f	n	v	n	f	n	v	f

[4] See [11].

This means that the result of one representation must be equal to one of nine possible results, which, however, does not mean that the one completely truthful result is necessarily found among the nine representations. The reader should **recognize this table as a horizontal layout of the matrix of conjunction**. If there are three (er^1, er^2, er^3) elements of reality, then the number of possibilities is twenty–seven; if there are four elements, then the possibilities are eighty–one. Because of the possible increases in relation to the number of elements of reality, as described above, which has a power nature, the following formula is utilized instead of matrices:

Number of possible results of a process (in this case cognition) for reality with x elements N = 3x

Process of Cognition

The assumption is being made that the cognition of reality is realized by means of **identification** or **measurement**[5]. Identification refers to quality, whereas measurement has to do with the characteristic of quantity. The realization of identification and measurement discloses the existing relationship between the quality and quantity of a certain object. In the instance of identification, quantity is disregarded; in the case of quantity, identification is disregarded. The relationship between quality and quantity represents the structure of information. This can be described as follows:

measurement	me	(numerator)	quantity	ei^1
relation	re	(connector)		ei^2
identification	id	(denominator)	quality	ei^3

The value of information is a function of method (Me)[6], technique (Te), and goal (Go). If, within the scope of the tasks of identification or measurement, method (Me), technique (Te), and goal (Go) are in agreement regarding the object (Ob) of examination, adequate information (equal within assumed scope) is obtained. Such information is then accepted as true information:

[5] The most general categories utilized in the field of philosophy are substance, quality, and relation. See [7]. However, for a more detailed presentation of this concept better serving our purpose, the categories of quantity, quality, and relation must be considered.

[6] The actual level of development in a given field of knowledge, which is based upon a confirmed theory, is represented by a method. For this reason, the value of information is changing over time, which, in turn, means that truth depends not only upon correct logical procedure, but also upon progress in a given field of knowledge.

if er = ei then ei is true (v)

The principal issues within the scope of tasks of identification or measurement are a lack of, or an inconsistency of, method (Me), technique (Te), and goal (Go) in relation to the object (Ob) of examination. What would then transpire is either a lack of or an improper result. In other words, the result would be a lack of information or inadequate information. Inadequate representation would be considered false information:

if er = 0 then is null (n)

if er $\neq$ ei then if false (f)

When considering these components (Me, Te, Go) during the process of cognition, the following values are applied: (+), meaning proper; (o), meaning lack; (–), meaning improper.

These symbols are deemed suitable for describing the situation when the said components (Me, Te, Go) must act together. It seems more desirable to use the word **proper** than the word **true** when one is engaged in applying a method (Me), using a technique (Te), or pursuing a goal (Go). Thus, a slightly modified **matrix of conjunction** is utilized for the three arguments. The next table shows possible variants of cognition focused on an object (Ob) as related to method (Me), technique (Te), and goal (Go).

	Me	Te	Go	Result
1	+	+	+	+
2	+	+	–	–
3	+	+	o	o
4	+	–	+	–
5	+	o	+	o
6	+	o	o	o
7	+	–	o	o
8	+	o	–	o
9	+	–	–	–
10	o	+	+	o
11	o	+	–	o
12	o	+	o	o
13	o	o	+	o
14	o	o	o	o

	Me	Te	Go	Result
15	o	o	–	o
16	o	–	o	o
17	o	–	+	o
18	o	–	–	o
19	–	o	+	o
20	–	+	+	–
21	–	+	o	o
22	–	o	o	o
23	–	–	+	–
24	–	+	–	–
25	–	o	–	o
26	–	–	o	o
27	–	–	–	–

Possibilities 1–27 can be classified as follows: The first possibility represents a proper result. This can be interpreted as the **true** cognition of identification or measurement. A lack of result is represented by 19 possibilities, namely, 3, 5–8, 10–19, 21–22, and 25–26. An improper result is represented by 7 possibilities, i.e., 2, 4, 9, 20, 23–24, and 27. Even the lack of one factor causes a lack of result for the entire cognition. An improper result occurs when certain combinations of (+) and (–) arise for all factors. The following three possibilities indicate that the second (++–) method (Me) and technique (Te) were in order, but the goal (Go) was inconsistent (**teleological error**); the fourth (+–+) method (Me) and goal (Go) were in order, but the technique (Te) was inconsistent (**technical error**); the twentieth (–++) method (Me) was inconsistent, whereas the technique (Te) and goal (Go) were in order (**methodical error**). In the aforementioned three instances, cognition resulted in **false** information because one of the three necessary components was improper.

With regard to both identification and measurement, the obtained truth has relative character. In addition to the function of method (Me), technique (Te), and goal (Go), other components include time (Ti) and place (Pl). During the time of Ptolemy, the widely held view of the Solar Planetary System was different from the one following the time of Copernicus. The same can be said of the measurement of the distance between the moon and the earth. What was believed a hundred years ago is different than what is believed today.

Conclusion Regarding Fundamentals

The previous three sections of this paper disclosed **three different aspects** of understanding the concept of true (v), a lack of (n), and false (f) information. The first was the logical aspect, depicted in the form of matrices, which indicated in a most abstract manner what the result of our thinking would be for two arguments (p, q) if we considered three values (v, n, and f). The second was the probabilistic aspect, presented in the form of tables and formulas, which illustrated what could have occurred during the time when information originated, i.e., what the result could have been (v, n, f). This section also considered a different concept of quantity or amount of information by modeling how a number of cognizable elements translated into a number of possible results.

The third was the corresponding aspect. The name here being borrowed from the **correspondence theory of truth.**[7] This aspect consists of three

[7] See: [1] and [12]

categories – quality, quantity, relation – and, in a general way, depicts with participation of what components: method (Me), technique (Te), goal (Go), the value of information (v, n, f) originate.

Before proceeding with the presentation of the structure of information and the spaciousness of thought, the author requests that the reader stops for a moment and tries to imagine these three aspects of comprehending value (v, n, f) as a certain three-dimensional, geometrical figure assembled from three equilateral triangles, of different colored sides, inserted into a transparent sphere, connected by a common point and adjusted for 120 degrees. Such an illustration is proposed because it appears to represent the nature of analyzed values.

Part III

Structure of Information

The structure of information, in **horizontal projection**, on the zero (0) level of analysis can be presented in the following manner. There is a relationship between an element of information and reality, on the one hand, and an element of information and corresponding categories of information, on the other hand. This is a general matrix consisting of object (Ob), method (Me), technique (Te), goal (Go), subject (Su), time (Ti), and place (Pl). These categories would need to be utilized for any type of examination. Here, social connections are introduced to these categories because of their common character. This also means that they become the **vectors of the virtual space of information**. With the exception of covering the totality of representation, the structure of this space assumes an efficiency for the servicing of information. Here, individual thinking encounters its social foundation, allowing an objectivization of its result. The category of subject (Su), which personifies the usage of method (Me), technique (Te), and goal (Go), is necessary for processing. An equivalent to the subject could be a computer functioning as an autonomous causative system. The last two categories refer to and describe time (Ti) and place (Pl), **when** and **where**, particular information originated.

Ob(v1)	Me(v2)	Te(v3)	Go(v4)	Su(v5)	Ti(v6)	Pl(v7)
object	method	technique	goal	subject	time	place
what	how	what with	why	who	when	where
ei^1	ei^2	ei^3	ei^4	ei^5	ei^6	ei^7

According to our thesis, **these categories are necessary components of thinking**. And regardless which process of thought is being examined, it will always be constructed in a dynamic mode based upon these components.

Spaciousness of Thought

In the opening section of this paper, the **vertical structure** of information was described along with the concept of **levels of analysis**. This concept served to examine what is manifested in the shape of information. Now, by delving deeper into the virtual realm of thinking, we next focus upon the well-known concept of **hierarchical levels**. Thus, when adjacent levels are involved, the level above represents the location of something general (term, information), whereas the level below stands for the location of something particular (term, information). In addition to the relations between two levels during the thinking process, we also encounter the relation of something (term, information) with something else on the same level, between two sides that are named **coordinal** sides. Because something could represent inference, we therefore introduce the additional concept of consequential states, which applies to **diagonal** relations.

The analysis of truth-functional connectives for alternative, disjunction, conjunction, equivalence and implication has lead to the distinction of **possible**, **necessary**, and **sufficient** conditions within each function. Our present focus has to do with an examination of how **possible**, **necessary** and **sufficient** connections appear with regard to the **hierarchical levels**.

The logical frame of thinking appears in a spacious manner on hierarchical levels:

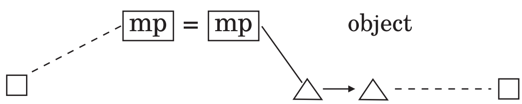
between two hierarchical levels,	–	vertical	$\uparrow$
between two coordinal sides,	–	horizontal	$\rightarrow$
between two consequential states,	–	diagonal	$\nearrow$

Connections such as potential (possible), essential (necessary), actual (existing), and reciprocal (sufficient) are realized within this framework.

Possible **vertical** connections will apply, for instance, in the case of language, to the term representing a class that has names of its members.; and, in the case of information, to the parameters of a class that has parameters of its members. Omitted here are descriptions of horizontal / possible and diagonal / possible for language, as well as for information.

Essential **horizontal** connections will apply, for instance, in the case of language, to the name of the characteristic of a class that has a term representing the class; in case of information, for the parameter of a member's feature that has parameters of its member. Again, we are omitting here the vertical / necessary and diagonal / necessary for language, as well as for information.

Sufficient **diagonal** connections, for instance, apply to a pair in which the first (essential) is represented by law (constant) and the second, by the given object of investigation:



Gold (□) has a melting point (mp) of 1064.43° C; object (△) melts at a temperature (mp) of 1064.43° C; therefore, object (△) is (made of) gold (□).

The above instances do not pretend to fully describe the logical frame of thinking; they merely aim to demonstrate its spaciousness. In conclusion, we can say that spaciousness of thought manifests itself virtually in the following dimensions: vertically, between the levels; horizontally, between the sides; and, diagonally, between the states.

Levels of Synthesis

In order to make a general analysis of the contents of thought in vertical projection, it is necessary to introduce another category of levels that we call **levels of synthesis**.

These levels ascend from the zero (0) level of elements of information to parts, numbers, and relations at the first ($\underline{1}$) level. And, from here to configuration and sequence, set on the second ($\underline{2}$) level. Finally, from the second level ($\underline{2}$), one moves to the structure of the model on the third ($\underline{3}$) level.

	structure of model	level $\underline{3}$	↔ HYPOTHESIS
	config./sequence/set	level $\underline{2}$	
	parts/numbers/relations	level $\underline{1}$	
REALITY ↔	elements of information	level 0	↑ [up]

Please note that the numbers of the **levels of analysis** are delineated by **overlining**, while the numbers of the **levels of synthesis** are delineated by **underlining**.

Planes of Realization

Leaving aside the presentation of elements of information in vertical and horizontal projection, we next focus on the planes of realization. The thought that goes through the mind – whatever it may be – has a particular meaning and is therefore connected to the proper word. It also has specific parameters that allow connections among the various elements of information. Similarly, the mind must take the value of the involved factors into consideration. Nevertheless, if any measurement is involved, computation is assumed.

- The **lingual** plane contains term, relation, and definition, and it serves to communicate results.
- The **informational** plane contains object, method, and goal, and it serves to achieve transformation.
- The **logical** plane contains alternative/disjunction, conjunction/equivalence, and implication/negation, and it serves to supervise evaluation.
- The **functional** plane contains addition/subtraction, multiplication/division, and equation/inequation, and it serves to control computation.

The above relationships can be presented in the form of a table:

Lingual	Informational	Logical	Functional
term	object (Ob)	alternative/disjunction	$+/-$
relation	method (Me)	conjunction/equivalence	$\times/\div$
definition	goal (Go)	implication/negation	$=/\neq$

Paradigm of Thinking

The sequence of possible thought activities is determined by the order of its component parts. To initiate thought activated by technique (Te), the starting point would be a single component. It could be the object (Ob), the method (Me) or the goal (Go). We assume that the reader remembers the function of these components which is involved in the **process of cognition**. Thus, technique (Te) is marked in the center, and object (Ob), method (Me), and goal (Go) on the circumference of a circle that represents a certain unity (whole). The following situation arises: Originating from any of the components, the directional options would be left or right. This movement discloses the **surprising simplicity of the paradigm of thinking**. At each beginning, there is a choice of either left or right. When one is dealing with two component parts, rather than merely one, a person has only one direction, i.e. does not have a choice. Any action is determined by this fact.

This paradigm of thinking can be synchronized with all three types of reasoning. An initial point might be the goal (Go). Then, if there is a certain problem to be resolved, the follow-up would consist of the path of **reductive** reasoning. If the initial point is the object (Ob) and if there are certain data to be processed, then the follow-up would consist of the path of **inductive** reasoning.

Suppose the primary point were method (Me), then, by applying a certain rule, the follow-up would be the path of **deductive** reasoning. The many forms in which these three kinds of reasoning are being utilized by the millions of human minds all over the world are irrelevant, for it appears that no additional types of reasoning could be discovered/invented. On the other hand, this paradigm represents a solid, evolving base that serves to circumvent initial problems by creating an autonomous causative system that can exceed the ability of the human mind.

This treatise has departed from the popular dichotomous division of reasoning into **induction – deduction**, or the rare division of reasoning into **deduction – reduction** (Czezowski[8]). Thus, the assumption is maintained that ***correspondence theories of truth*** establish the foundation for inductive reasoning, that ***coherence theories of truth*** establish the foundation for deductive reasoning, and that ***pragmatic theories of truth*** establish the foundation for reductive reasoning. In the first case, synthetic truth is dealt with by formulating a generalization/law; in the second case, analytic truth is dealt with by employing a law; and, in the third case, pragmatic truth will be dealt with by constructing a hypothesis.

In more comprehensive language, we could then say the following: If series of observations have led to a single conclusion, then the tool utilized was

[8] See [2].

inductive reasoning. If series of conclusions have been obtained on the basis of one law, then the necessary tool was deductive reasoning. On the other hand we utilize reductive[9] reasoning when we construct series of hypothesis regarding resolution of a certain task. Or in the opposite case, when constructing a single hypothesis to determine causality of series of events.

Conclusion

The dominance of computer science over the interpretation of many crucial problems related to the working of the **brain**, in addition to discoveries of important physiological characteristics of gray matter, has, in the author's opinion, created a perspective for understanding the thought process that is both limited and too general. The author believes that the philosophical sciences must provide a healthy counterbalance by focusing attention on specific rational activities of the **mind**, i.e., on thinking and reasoning. Alfred Whitehead wrote, "Philosophy is not one among the sciences with its own little scheme of abstractions which it works away at perfecting and improving. It is the survey of sciences, with the special objects of their harmony, and of their completion."[10]

The limited scope of computer science may be attributed to its perception of objects of investigation via the category of quantity, thus implying that any resolution should be available via computation. At this point, it is instructive to quote Ray Kurzweil, who is a prominent figure in the field of artificial intelligence: "Our human intelligence is based on the **computational process** that we are learning to understand. We will ultimately multiply our intellectual powers by applying and extending the method of human intelligence using the vastly greater capacity of **nonbiological computation**."[11] It is obvious that, to a certain extent, computers have the ability, via their **computational process**, to imitate or replace the process of thinking. However, it must be maintained that the proper description of the nature of these processes requires a more diversified approach.

[9] In American terminology, the term **abductive** is used; see [10]. .. (2) For C. S. Peirce (g.v.8), abduction is one of three basic forms of inference, along with induction and deduction. Abduction is the means whereby hypotheses are generated, moving from a particular case to a possible explanation of the case." We are convinced that inductive and deductive reasoning originated as a result of resolving a problem, that is, by employing reductive reasoning. See [5].

[10] See [13].

[11] See [9].

Given the current advanced level of computer and information technology, in order to better understand the issues involved in the modeling of **human intelligence**, we should replace the word **computation** with the term **transformation**, since, among other things, this term denotes **dynamism** and **spaciousness**. It is the author's opinion that issues presented in this paper could assist in the development of a new and fruitful alliance between philosophy and information technology in the field of thought modeling.

Bibliography

[1] Angeles Peter Adam, *Dictionary of Philosophy*, Barnes & Noble Books, New York 1981, p. 298.

[2] Czeżowski Tadeusz, *Główne Zasady Nauk Filozoficznych*, Zakład Narodowy Imienia Ossolińskich – Wydawnictwo, Wrocław 1959, ch. III, part 1, *O rozumowaniach*, pp. 136–141.

[3] Czeżowski Tadeusz, *Główne Zasady Nauk Filozoficznych*, Zakład Narodowy Imienia Ossolińskich – Wydawnictwo, Wrocław 1959, ch. I, pp. 68–69.

[4] Czeżowski Tadeusz, *Logika. Podręcznik dla studiujących nauki filozoficzne*, PZWS, Warszawa 1949, ch. 3, pp. 193–204.

[5] Goldstein Martin, *How We Know*, Plenum press, New York and London, New York 1978, ch. 2, p. 19.

[6] Haack Susan, *Philosophy of Logics*, Cambridge University Press, 1978, ch. 11, pp. 205–220.

[7] Harre Rom, *The Philosophies of Science*, Oxford University Press, London 1972, ch. 4, pp. 100–101.

[8] Johnson William Ernest, *Logic*, Dover Publications, Inc. 1964, part 1, ch. 3, p. 32–33.

[9] Kurzweil Ray, *The Singularity Is Near*, Viking, New York, 2005, pp. 652, Chapter: *The Limits of Computation*, p. 128.

[10] Reese William Lewis, *Dictionary of Philosophy and Religion*, Eastern and Western Thought. New Jersey: Humanities Press. Sussex: Harvester Press. 1980. p. 644. "ABDUCTION. From the Latin ab ("away") and ducere ("to lead").

[11] Shannon Claude Elwood, *A mathematical theory of communication*, Bell System Technical Journal, July and October, 1948, vol. 27, pp. 379–423 and 623–656,

[12] *The Correspondence Theory of Truth*, Stanford Encyclopedia of Philosophy (Internet).

[13] Whitehead Alfred, *North, Science and the Modern World*, The Press, New York 1953, Chapter V, p. 87.

Stanislaw Zadrozny
stanzadrozny@yahoo.com

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17 (30) 2009

Mariusz Giero
University of Białystok

APPLICATION OF BITEMPORAL DATABASES CONTAINING MEDICAL DATA[12]

Most databases store time-varying information. The research area of temporal databases aims to provide expressive and efficient ways to model, store, and query temporal data. The paper concentrates on the application of temporal logic in the research. It shows how the language of first-order temporal logic can be used to query temporal data.

1. Introduction

A temporal database [Etz, Ste, Tan1] is defined as the one maintaining object histories, i. e., past (before modification), present, and possibly future data (e.g. database is used for forecasts). Maintaining temporal data within traditional relational databases [Cod, Dat] is not straightforward.

Example 1

In a traditional relational database, deletion and modification operation causes physical removing of data and overwriting data, respectively. It is not possible to trace the history of records. Let us consider a database with one relation TREATMENT (id, medicine, when) where the course of patient's treatment is stored. Attribute *id* is for the patient identification number, attribute *medicine* is for the name of the medicine the patient is treated with and attribute *when* is for the date of treatment:

[12] The research reported in this paper is a part of the project entitled Temporal representation of knowledge and its implementation in the computer systems of medical conducts supported by the Polish Ministry of Science and Higher Education, grant no. 3 T11F 011 30.

ISBN 978–83–7431–231–8 ISSN 0860-150X

ID	MEDICINE	WHEN
1	A	10 X – 15 X
2	B	13 X – 16 X
3	A	13 X – 20 X

After deleting the first record and modifying the third one, we obtain:

ID	MEDICINE	WHEN
2	B	13 X – 16 X
3	C	13 X – 15 X

Data about patient 1 and treatment of patient 3 with A is lost.

There are numerous application domains that require keeping previous states: Medical Systems (e.g. patients records), Computer Applications (e.g. history of files back ups), Archive Management Systems (e.g. sporting events, publications and journals), Reservation Systems (e.g. when was a flight booked), and many others. Snodgrass [Sno] lists 16 various areas with time-varying data. In fact, it is hard to find areas that do not contain time-varying data.

One of the ways ([JenSno, Jen, Tan2]) to extend a traditional relational database to be a temporal one is by adding to each relation two attributes: *valid time* and *transaction time*. Valid time attribute (VT) is for recording time when a record is valid in the world represented by the database. Transaction time attribute (TT) is for recording time when a record is stored (from insertion to possible deletion) in the database. Applying the solution, the database from example 1 can be represented in the following way:

Table 1. Example of temporal database

ID	MEDICINE	VT	TT
1	A	10 X – 15 X	12 X – now
2	B	13 X – 16 X	10 X – now
3	A	13 X – 20 X	5 X – now

The attribute *when* has been replaced by the attribute VT and gives information about when the patient was treated. The attribute TT gives information about when the record was inserted and whether it was deleted or not. The values for the attributes VT, TT are intervals. The end date of the intervals can be value *now*. In case of VT it indicates that the end date of treatment has not been decided yet and means current date. In case of TT it indicates that the record was not deleted.

One can read the following information from table 1:

- The attribute VT reads: patient 1 was treated with A from 10 X to 15 X, patient 2 with B from 13 X to 16 X and patient 3 with A from 13 X to 20 X.
- The attribute TT reads: 3 records were inserted in the database, respectivly: 12 X, 10 X and 5 X and these records are still stored.

In a temporal database the basic operations: *insert*, *modification*, *deletion* are defined in a different way than in a traditional relational one.

The *deletion* operation is not a physical removing of a record but a logical one. It means that the end date of TT is changed from *now* to the current date. For example, after deletion of the first record (14 X) the database contains:

Table 2. Temporal database after deletion operation

ID	MEDICINE	VT	TT
1	**A**	**10 X – 15 X**	**12 X – 14 X**
2	B	13 X – 16 X	10 X – now
3	A	13 X – 20 X	5 X – now

The *modification* results in changing the end date of TT from *now* to the current date (this indicates that the record was deleted) and inserting the new record with new values of the attributes. For example, after modification of the third record (that took place on 10 X), the database contains:

Table 3. Temporal database after modification operation

ID	MEDICINE	VT	TT	
1	A	10 X – 15 X	12 X – 13 X	
2	B	13 X – 16 X	10 X – now	
3	**A**	**13 X – 20 X**	**5 X – 10 X**	the old version of the third record
3	**C**	**13 X – 15 X**	**10 X – now**	the new version of the third record

The *insertion* operation results in adding a new record. The start date of TT is the execution date of the operation. The end date of TT is *now*. For example:

Table 4. Temporal database after insertion operation

ID	MEDICINE	VT	TT	
1	A	10 X – 15 X	12 X – 13 X	
2	B	13 X – 16 X	10 X – now	
3	A	13 X – 20 X	5 X – 9 X	
3	C	13 X – 15 X	10 X – now	
2	**A**	**14 X – now**	**14 X – now**	the record inserted 14-th

Such a temporal database prevents from losing information. One can read both present and past states of data with respect to valid time and transactions time. For example:

State of data for the day 14 X of *TT*
in other words: records stored in the database on 14 X:

ID	MEDICINE	VT	**TT**
2	B	13 X – 16 X	**10 X – now**
3	C	13 X – 15 X	**10 X – now**
2	A	14 X – now	**14 X – now**

State of data for the day 13 X of *VT*
in other words: records (deleted and stored) about treatment taken on 13 X:

ID	MEDICINE	TT	**VT**
2	B	10 X – now	**13 X – 16 X**
3	A	5 X – 9 X	**13 X – 20 X**
3	C	10 X – now	**13 X – 15 X**

State of data for the day 14 X of *VT* and 10 X of *TT*
in other words: records stored (not deleted) in the database on 10 X
about treatment taken on 14 X:

ID	MEDICINE	**VT**	**TT**
2	B	**13 X – 16 X**	**10 X – now**
3	C	**13 X – 15 X**	**10 X – now**

Temporal databases can record only valid time (called valid–time databases), only transaction time (called transaction–time databases) or both valid time and transaction time (called bitemporal databases). The paper examines bitemporal databases. It is convenient to represent bitemporal databases graphically:

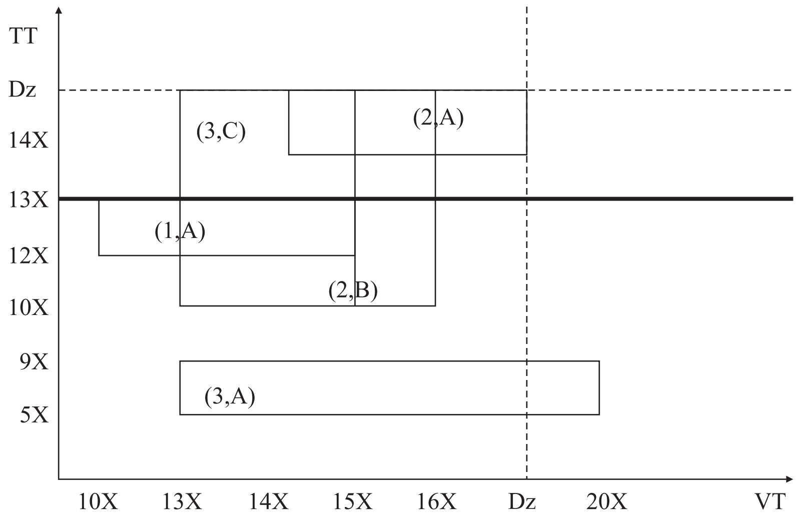


Figure 1. Graphical representation of the bitemporal database from Tab. 4

2. Time

In this paper, we assume that the flow of time $(T, <)$ is a linear, discrete and ordered structure with no end points. T is a set of time points and $<$ is a binary order relation defined on T that satisfies the following conditions:

- transitivity $\forall x, y, z(x < y \wedge y < z \Rightarrow x < z)$
- irreflexivity $\forall x \neg(x < x)$

3. Bitemporal database

There are various approaches to the definition of bitemporal database ([Tan1]). Bitemporal database can be defined as a structure $DB = (T, <, D, R_1, R_2, \ldots, R_n)$, where:

- $(T, <)$ is a discrete structure of time
- D is a non empty set (database domain),

- $R_1, R_2, \ldots, R_n$ are relations on $D \cup T$ such that

$$R_i \subseteq \underbrace{D \times \ldots \times D}_{k \text{ times}} \times 2^T \times 2^T$$

  where k is a natural number associated with each relation that indicates the number of non-temporal attributes, two last arguments of the relations indicate valid time and transaction time, respectively.

The state of relation R_i with k non-temporal arguments at the moment (t_v, t_t) is ${}_{tv,tt}R_i = \{(d_1, \ldots, d_k) :$ exist $\tau_v, \tau_t \subseteq T$ such that $(d_1, \ldots, d_k, \tau_v, \tau_t) \in R_i$ and $t_v \in \tau_v, t_t \in \tau_t\}$

Example 2

${}_{14\mathrm{X},10\mathrm{X}}$TR (where TR is the relation TREATMENT from the database presented in the Table 4):

ID	MEDICINE
2	B
3	C

4. The language of first order temporal logic

The first order temporal query language is an extension of the first order classical logic language of temporal connectives [Gab, ChoTom]. After some modifications, it is suited for formulating queries in bitemporal database. The alphabet, syntax and semantics are defined as following.

4.1. Alphabet

Alphabet has the following categories of basic symbols:

- domain variables: $x_1, x_2, \ldots$;
- domain constants: $c_1, c_2, \ldots$;
- time variables: $t_1, t_2, \ldots$;
- time constants: $e_1, e_2, \ldots$;
- predicate symbols: $P_1, P_2, \ldots, P_k$;
- equality symbol: $=$;
- classical connectives: $\neg, \wedge$;
- existential quantifier: $\exists$;
- temporal connectives: $\mathtt{U}, \mathtt{S}, \underline{\mathtt{U}}, \underline{\mathtt{S}}$;

- punctuation symbols:), (.
- additional predicate symbols: *date*, $\underline{date}$;

4.2. Syntax

A *term* is either a domain term or time term. A *domain (time) term* is either a domain (time) constant or a domain (time) variable. The atomic formulas of the language are of the form:

- $P_i(a_1, a_2, \ldots, a_n)$, where P_i is n-ary predicate symbol, $a_1, a_2, \ldots, a_n$ are domain terms
- $t = u$, where t, u are both either domain terms or time terms
- $date(t), \underline{date}(t)$, where t is time term

Formulas are finite strings of basic symbols defined in the following recursive manner:

(1) any atomic formula is a formula
(2) if φ, ψ are formulas, so also are $\neg\varphi$, $\varphi \wedge \psi$, $\exists x\varphi$, $\varphi\mathtt{U}\psi$, $\varphi\underline{\mathtt{U}}\psi$, $\varphi\mathtt{S}\psi$, $\varphi\underline{\mathtt{S}}\psi$, where x is any domain variable.
(3) there are not any other formulas than finite strings satisfying conditions (1) and (2)

Other operators are introduced as abbreviations:

- $\varphi \vee \psi \quad \equiv \quad \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \to \psi \quad \equiv \quad \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \quad \equiv \quad (\varphi \to \psi) \wedge (\psi \to \varphi)$
- $\forall x\varphi \quad \equiv \quad \neg\exists x\neg\varphi$
- $\mathtt{F}\varphi \quad \equiv \quad \top\mathtt{U}\varphi$
- $\mathtt{P}\varphi \quad \equiv \quad \top\mathtt{S}\varphi$
- $\mathtt{G}\varphi \quad \equiv \quad \neg\mathtt{F}\neg\varphi$
- $\mathtt{H}\varphi \quad \equiv \quad \neg\mathtt{P}\neg\varphi$
- $\mathtt{X}\varphi \quad \equiv \quad \bot\mathtt{U}\varphi$
- $\mathtt{X}^-\varphi \quad \equiv \quad \bot\mathtt{S}\varphi$,

where $\top$ denotes the propositional constant verum, $\bot$ denotes the propositional constant falsum.

Underline operators, i.e. $\underline{\mathtt{F}}, \underline{\mathtt{P}}, \underline{\mathtt{G}}, \underline{\mathtt{H}}, \underline{\mathtt{X}}, \underline{\mathtt{X}}^-$ are introduced in analogous way. For examle:

- $\underline{\mathtt{G}}\varphi \quad \equiv \quad \neg\underline{\mathtt{F}}\neg\varphi$
- $\underline{\mathtt{X}}\varphi \quad \equiv \quad \bot\underline{\mathtt{U}}\varphi$

4.3. Semantics

Let $DB = (T, <, D, R_1, R_2, \ldots, R_n)$ be a bitemporal database, Θ be an interpretation and θ be a valuation. The interpretation Θ associates every predicate symbol to a relation, every domain constant to an element of D, every time constant to an element of T. The valuation θ associates every domain term to an elelement of D, every time term to an element of T. If c is a constant term, we make $\theta(c) = c$.

We define a formula A to be true in DB at point $(t_v, t_t) \in T \times T$ under valuation θ, writing $DB, \theta, t_v, t_t \models \varphi$, by induction on the structure of formula:

(1) $DB, \theta, t_v, t_t \models P_i(x_1, \ldots, x_k)$
iff $(\theta(x_1), \ldots, \theta(x_k)) \in {}_{tv,tt}\Theta(P_i)$,

(2) $DB, \theta, t_v, t_t \models x_i = x_j$
iff $\theta(x_i) = \theta(x_j)$

(3) $DB, \theta, t_v, t_t \models \neg\varphi$
iff it is not the case that $DB, \theta, t_v, t_t \models \varphi$

(4) $DB, \theta, t_v, t_t \models \varphi \wedge \psi$
iff $DB, \theta, t_v, t_t \models \varphi$ and $DB, \theta, t_v, t_t \models \psi$

(5) $DB, \theta, t_v, t_t \models \exists x_i \varphi$
iff there exists an $a \in D$ such that $DB, \theta^*, t_v, t_t \models \varphi$,
where θ^* is a valuation which agrees with the valuation θ on the values of all variables except, possibly, on the values of x_i

(6) $DB, \theta, t_v, t_t \models \varphi \mathtt{U} \psi$
iff there exists a $t_1 \in T$ with $t_v < t_1$ and
$DB, \theta, t_1, t_t \models \psi$ and for every $t_2 \in T$,
whenever $t_v < t_2 < t_1$ then $DB, \theta, t_2, t_t \models \varphi$

(7) $DB, \theta, t_v, t_t \models \varphi \mathtt{S} \psi$
iff there exists a $t_1 \in T$ with $t_1 < t_v$ and
$DB, \theta, t_1, t_t \models \psi$ and for every $t_2 \in T$
whenever $t_1 < t_2 < t_v$ then $DB, \theta, t_2, t_t \models \varphi$

(8) $DB, \theta, t_v, t_t \models \varphi \underline{\mathtt{U}} \psi$
iff there exists a $t_1 \in T$ with $t_t < t_1$ and
$DB, \theta, t_v, t_1 \models \psi$ and for every $t_2 \in T$
whenever $t_t < t_2 < t_1$ then $DB, \theta, t_v, t_2 \models \varphi$

(9) $DB, \theta, t_v, t_t \models \varphi \underline{\mathtt{S}} \psi$
iff there exists a $t_1 \in T$ with $t_1 < t_t$ and
$DB, \theta, t_v, t_1 \models \psi$ and for every $t_2 \in T$
whenever $t_1 < t_2 < t_t$ then $DB, \theta, t_v, t_2 \models \varphi$

(10) $DB, \theta, t_v, t_t \models \mathit{date}(t)$
iff $\theta(t) = t_v$

(11) $DB, \theta, t_v, t_t \models \underline{\mathit{date}}(t)$
iff $\theta(t) = t_t$

The not underlined temporal connectives refer to valid time while the underlined ones refer to transaction time.

Semantics of the others temporal connectives $\mathtt{F}, \mathtt{P}, \mathtt{G}, \mathtt{H}, \mathtt{X}, \mathtt{X}^-$ and their underlined counterparts are the following:

(1) $DB, \theta, t_v, t_t \models \mathtt{F}\varphi$
iff there exists a $t_1 \in T$ with $t_v < t_1$ and $DB, \theta, t_1, t_t \models \varphi$

(2) $DB, \theta, t_v, t_t \models \underline{\mathtt{F}}\varphi$
iff there exists a $t_1 \in T$ with $t_t < t_1$ and $DB, \theta, t_v, t_1 \models \varphi$

(3) $DB, \theta, t_v, t_t \models \mathtt{P}\varphi$
iff there exists $t_1 \in T$ with $t_1 < t_v$ and $DB, \theta, t_1, t_t \models \varphi$

(4) $DB, \theta, t_v, t_t \models \underline{\mathtt{P}}\varphi$
iff there exists $t_1 \in T$ with $t_1 < t_t$ and $DB, \theta, t_v, t_1 \models \varphi$

(5) $DB, \theta, t_v, t_t \models \mathtt{G}\varphi$
iff for every $t_1 \in T$ whenever $t_v < t_1$ then $DB, \theta, t_1, t_t \models \varphi$

(6) $DB, \theta, t_v, t_t \models \underline{\mathtt{G}}\varphi$
iff for every $t_1 \in T$ whenever $t_t < t_1$ then $DB, \theta, t_v, t_1 \models \varphi$

(7) $DB, \theta, t_v, t_t \models \mathtt{H}\varphi$
iff for every $t_1 \in T$ whenever $t_1 < t_v$ then $DB, \theta, t_1, t_t \models \varphi$

(8) $DB, \theta, t_v, t_t \models \underline{\mathtt{H}}\varphi$
iff for every $t_1 \in T$ whenever $t_1 < t_t$ then $DB, \theta, t_v, t_1 \models \varphi$

(9) $DB, \theta, t_v, t_t \models \mathtt{X}\varphi$
iff $DB, \theta, t_v + 1, t_t \models \varphi$

(10) $DB, \theta, t_v, t_t \models \underline{\mathtt{X}}\varphi$
iff $DB, \theta, t_v, t_t + 1 \models \varphi$

(11) $DB, \theta, t_v, t_t \models \mathtt{X}^-\varphi$
iff $DB, \theta, t_v - 1, t_t \models \varphi$

(12) $DB, \theta, t_v, t_t \models \underline{\mathtt{X}}^-\varphi$
iff $DB, \theta, t_v, t_t - 1 \models \varphi$

Graphical representations of defined connectives presents Fig. 2.

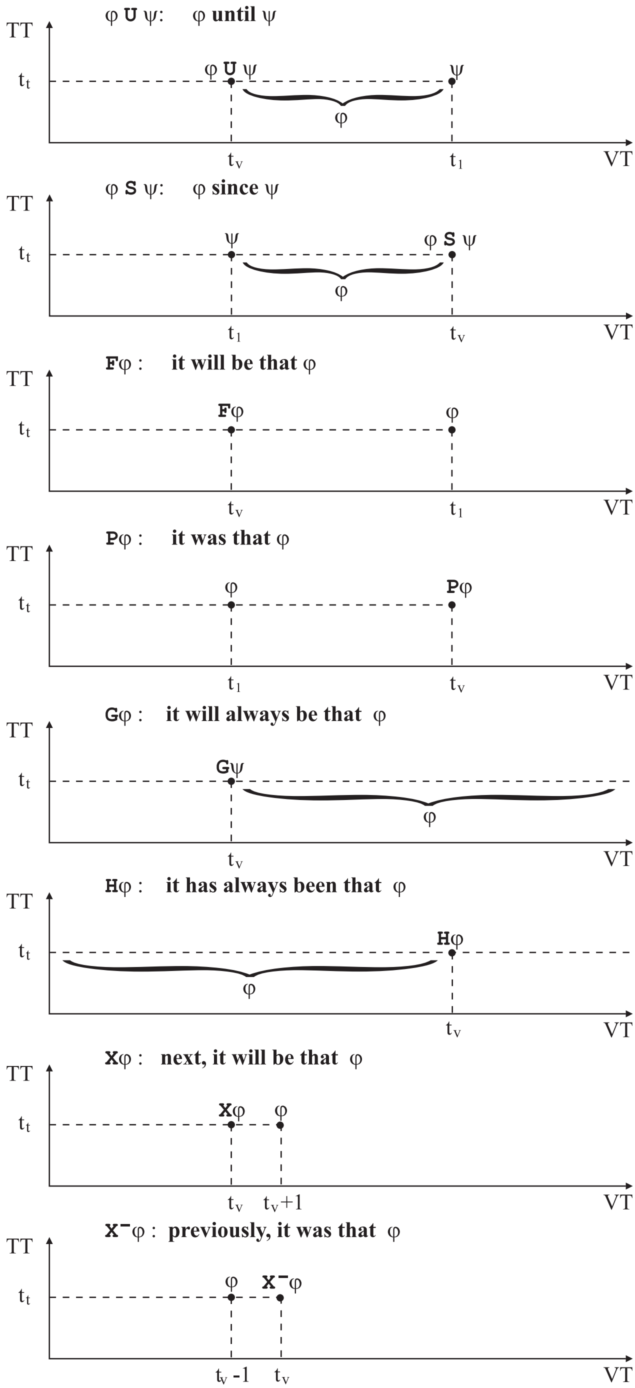


Figure 2. Graphical representation of temporal connectives referred to valid time

Graphical representation of temporal connectives with underlining is analogical (Fig. 3):

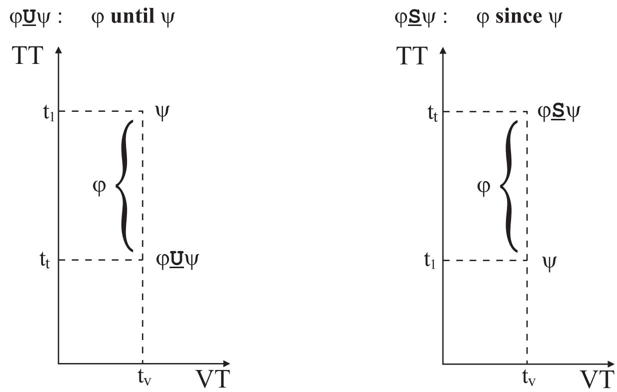


Figure 3. Graphical representation of connectives $\underline{\mathtt{U}}$ and $\underline{\mathtt{S}}$

Example 3

Let us interpret T_1 as the relation TREATMENT (Tab. 4).

- DB,θ,14X,10X $\models T_1$(2,B),
  because (2,B)$\in$ $_{14X,10X}$TREATMENT
- DB,θ,16X,12X $\models T_1$(2,B) $\mathtt{S}$ T_1(1,A),
  because there exists a t_1<16X with DB,θ,t_1,12X $\models T_1$(1,A) and for every $t_2 \in$T whenever t_1<t_2<16X then DB,θ,t_2,12X $\models T_1$(2,B) (e.g. t_1 can be associated to 13X: (1,A)$\in_{13X,12X}$TREATMENT and (2,B)$\in_{t2,10X}$TREATMENT, where 13X<t_2<16X)
- DB,θ,14X,11X $\models T_1$(1,A) $\underline{\mathtt{U}}$ T_1(2,B),
  because there exists a t_1>11X with DB,θ,14X,$t_1 \models T_1$(2,B) and for every $t_2 \in$T whenever 11X<t_2<t_1 then DB,θ,14X,t_2 $\models T_1$(1,A) (e.g. t1 can be associated to 13 X: (2,B)$\in_{14X,13X}$TREATMENT and (1,A)$\in_{14X,t2}$TREATMENT, where 11X<t_2<13X)

5. Querying a bitemporal database

The language defined in the previous section enables to query a bitemporal database in a convenient way. One can query a bitemporal database with respect to valid time (to retrieve data about what happened in the reality modelled by a database during given time), transaction time (to retrieve data stored in a database during given time) and both valid and transaction time (to retrieve data about what happened in the reality modelled by a database during given time according to records stored in a database during given time). In the first case, one uses temporal operators without underlining, in the second one temporal operators with underlining, and in the third case both operators.

A formula φ is a bitemporal database DB query if it contains at least one free variable and all the variables are domain ones. The answer of the query is defined as follows:

$$\varphi(DB) = \{(\theta(x_1), \ldots, \theta(x_n)) : DB, \theta, t_v, t_t \models \varphi\},$$

where $x_1, \ldots, x_n$ are the only free variables of the formula xx.

5.1. The examples of queries

The queries refer to the bitemporal database containing two relations: PATIENTS(id, name, VT, TT) and TREATMENT(id, medicine, VT, TT). The patients of a hospital and their course of treatment are recorded in this database. Two first attributes in both relations are non-temporal ones, the third attribute is for recording valid time, and the last one is for recording transaction time. The predicate symbol P_1 is interpreted as the relation PATIENTS and the predicate symbol L_1 is interpreted as the relation TREATMENT. We assume that the following constraint is defined in the database: the period of treatment of a patient is contained in (or is equal to) the period of their staying in the hospital.

Query 1

What medicines have been given every day till today (including today) to Kowalski (id. 1) during his last stay in hospital. The treatment could start on any day during his stay in hospital. The query refers to the current state of the database.

$$
\begin{aligned}
\varphi = (((&\mathtt{L_1(1,x)\ S\ P_1(1,Kowalski)})\\
&\quad \wedge \mathtt{X^-\ P_1(1,Kowalski)} \wedge \mathtt{X^-X^-P_1(1,Kowalski)}\\
&)\\
&\vee\\
&(\mathtt{X^-P_1(1,Kowalski)} \wedge \mathtt{X^-X^-}\neg\mathtt{P_1(1,Kowalski)} \wedge \mathtt{X^-L_1(1,x)}\\
&)\\
) &\wedge \mathtt{date(now+1)} \wedge \underline{\mathtt{date}}\mathtt{(now)}
\end{aligned}
$$

The predicate symbols `date` i <u>`date`</u> determine with respect to what kind of time one queries a database. According to the definition, one searches those valuations of a variable x in order to meet requirements::

$$
\begin{array}{l}
\texttt{DB},\Theta,\texttt{now+1},\texttt{now} \models \\
\quad ((\texttt{L}_1\texttt{(1,x)}\ \texttt{S}\ \texttt{P}_1\texttt{(1,Kowalski)}) \\
\quad\ \wedge\ \texttt{X}^-\ \texttt{P}_1\texttt{(1,Kowalski)} \wedge \texttt{X}^-\texttt{X}^-\texttt{P}_1\texttt{(1,Kowalski)} \\
\quad) \\
\quad \vee \\
\quad (\texttt{X}^-\texttt{P}_1\texttt{(1,Kowalski)} \wedge \texttt{X}^-\texttt{X}^-\neg\texttt{P}_1\texttt{(1,Kowalski)} \wedge \texttt{X}^-\texttt{L}_1\texttt{(1,x)})
\end{array}
$$

The valid time is `now+1` to include the present day to the period of treatment. In case the valid time is `now`, the treatment finished „yesterday" would satisfy formula φ (it is a consequence of the assumption that time is irreflexive). The following part of the query:

$$\texttt{X}^-\texttt{P}_1\texttt{(1,Kowalski)} \wedge \texttt{X}^-\texttt{X}^-\texttt{P}_1\texttt{(1,Kowalski)}$$

is a consequence of time being irreflexive. If this part was deleted, every treatment of patient admitted to hospital today would satisfy formula φ. The last part of the query is to ensure that among answers of the query is the following case: a patient is admitted to hospital today and his/her treatment starts that very day.

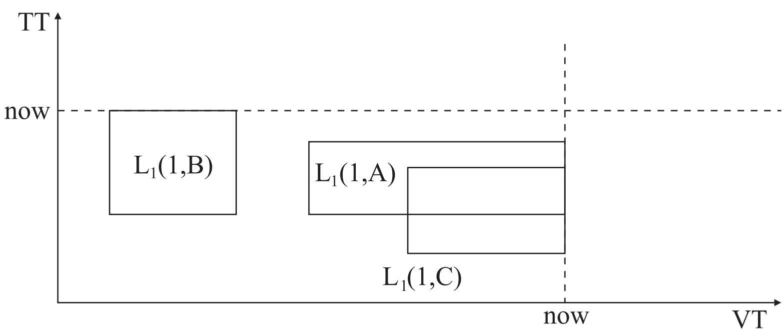


Figure 4. Contents of database at the moment of execution of query 1

The result of the query is medicine A (Fig. 4). The treatment with medicine B is finished and the record about treatment with medicine C does not belong to the current state of database.

Query 2

What medicines have been given to Kowalski? Search only records that have been inserted to the database since the record about admitting Kowalski to hospital was inserted.

$$
\begin{aligned}
\varphi = (((&\mathtt{L}_1\mathtt{(1,x)}\ \underline{\mathtt{S}}\ \mathtt{P}_1\mathtt{(1,Kowalski)} \wedge \underline{\mathtt{X}}^{-}\mathtt{P}_1\mathtt{(1,Kowalski)}\\
&\wedge \underline{\mathtt{X}}^{-}\underline{\mathtt{X}}^{-}\mathtt{P}_1\mathtt{(1,Kowalski)}\\
&)\\
&\wedge\\
&(\underline{\mathtt{X}}^{-}\mathtt{P}_1\mathtt{(1,Kowalski)} \wedge \underline{\mathtt{X}}^{-}\underline{\mathtt{X}}^{-}\neg\mathtt{P}_1\mathtt{(1,Kowalski)} \wedge \underline{\mathtt{X}}^{-}\mathtt{L}_1\mathtt{(1,x)}\\
&)\\
&) \wedge \underline{\mathtt{date}}\mathtt{(now+1)}
\end{aligned}
$$

We refer to the transaction time, hence, the operators with underlining, $\underline{\mathtt{S}}$ and $\underline{\mathtt{X}}^{-}$, are used. We search for records about treatment but valid time of the treatment is not of our interest, hence, the predicate symbol `date` (without underlining) is not used. The usage of the operator $\underline{\mathtt{X}}^{-}$ is to take into account the case that a patient is admitted to hospital today (analogical to query 1).

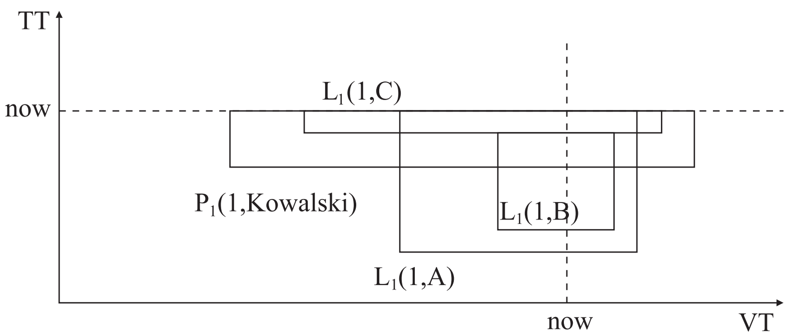


Figure 5. Contents of database at the moment of execution of query 2

The results of the query are medicines A and C (Fig. 5). The record about treatment with B does not belong to the current state of database.

Query 3

What are the identifiers of patients who are no longer treated with A according to the current state of the database.

$$
\begin{aligned}
\varphi = &\neg\mathtt{L}_1\mathtt{(x,A)} \wedge \mathtt{P}\ \mathtt{L}_1\mathtt{(x,A)} \wedge \neg\ \mathtt{F}\ \mathtt{L}_1\mathtt{(x,A)}\\
&\wedge \mathtt{date(now)} \wedge \underline{\mathtt{date}}\mathtt{(now)}
\end{aligned}
$$

There might be records about future treatments in the database. The subformula $\neg$`F L`$_1$`(x,A)` is to exclude those records.

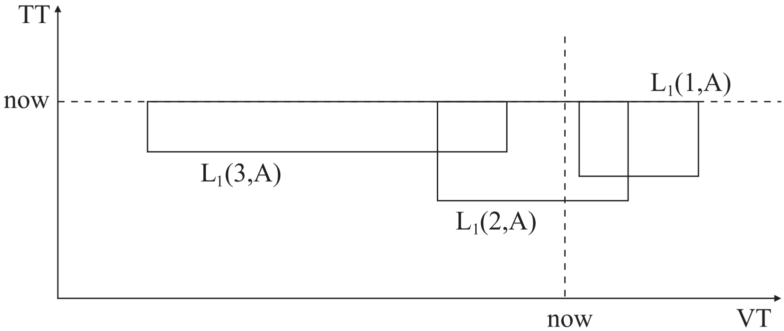


Figure 6. Contents of database at the moment of execution of query 3

The result of the query is identifier 3 (Fig. 6).

Query 4

What are the medicines that were given or are given or will be given to Kowalski according to the current state of database.

$$
\begin{aligned}
\varphi = \exists x\ (&\mathtt{P}(\mathtt{L}_1(\mathtt{x,y}) \wedge \mathtt{P}_1(\mathtt{x,Kowalski}))\\
&\vee\\
&\mathtt{L}_1(\mathtt{x,y}) \wedge \mathtt{P}_1(\mathtt{x,Kowalski}) \vee\\
&\mathtt{F}\ (\mathtt{L}_1(\mathtt{x,y}) \wedge \mathtt{P}_1(\mathtt{x,Kowalski}))\\
)&\\
\wedge\ &\mathtt{date(now)} \wedge \underline{\mathtt{date}}\mathtt{(now)}
\end{aligned}
$$

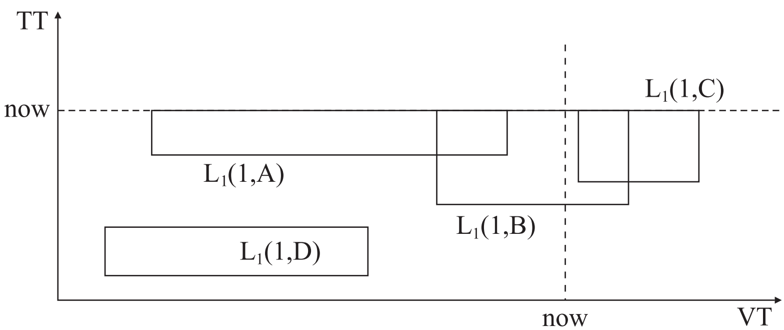


Figure 7. Contents of database at the moment of execution of query 4

The results of the query are medicines A, B, C (Fig. 7). The record with medicine D does not belong to the current state of the database, hence does not satisfy the query.

Query 5

Who will be treated with A during his/her whole stay in hospital in the future according to the current state of database?

$$
\begin{aligned}
\varphi = \exists \mathtt{x}\ \mathtt{F}(&\mathtt{X}^{-}\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{L}_1(\mathtt{x,A}) \\
&\wedge \\
&(\mathtt{L}_1(\mathtt{x,A})\ \mathtt{U}\ (\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{X}^{-}\mathtt{L}_1(\mathtt{x,A})) \\
&) \\
) \\
\wedge\ &\mathtt{date(now)} \wedge \underline{\mathtt{date}}\mathtt{(now)}
\end{aligned}
$$

The subformula $\mathtt{X}^{-}\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{L}_1(\mathtt{x,A})$ defines the day when a patient is admitted to hospital. The subformula $\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{X}^{-}\mathtt{L}_1(\mathtt{x,A})$ defines the day when a patient is discharged from hospital. Patients that satisfy the query are treated with A these both days but the connective $\mathtt{U}$ does not ensure it (under assumption that time is irreflexive). Hence, additional subformulas $\mathtt{X}^{-}\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{L}_1(\mathtt{x,A})$ and $\neg \mathtt{P}_1(\mathtt{x,y}) \wedge \mathtt{X}^{-}\mathtt{L}_1(\mathtt{x,A})$ are put in the query.

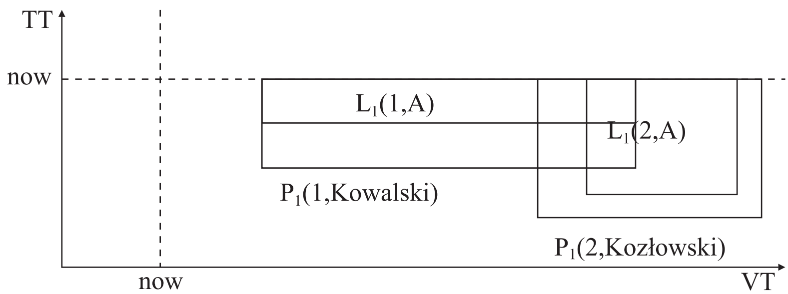


Figure 8. Contents of database at the moment of execution of query 5

The result of the query is the name *Kowalski* (Fig. 8).

Query 6

What are the identifiers of patients whose records about treatment with A have been deleted (logically) from database. We assume that the time of treatment is irrelevant.

$$\varphi = \underline{\mathtt{P}}(\mathtt{L}_1(\mathtt{x,A}) \wedge \mathtt{X}\neg \mathtt{L}_1(\mathtt{x,A})) \wedge \underline{\mathtt{date}}\mathtt{(now)}$$

The results of the query are identifier 1 (first rectangle from the left) and 2 (Fig. 9).

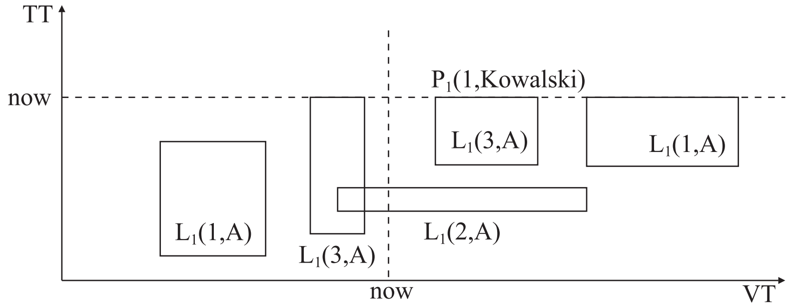


Figure 9. Contents of database at the moment of execution of query 6

Query 7

What records from table PATIENTS were inserted on the day of d_1 ($<$ *now*) and eventually deleted (logically). The time of staying in hospital is irrelevant.

$$
\begin{aligned}
\varphi = {} & \mathtt{P_1(x,y)} \wedge \underline{\mathtt{X}}^{-}\neg\mathtt{P_1(x,y)} \\
& \wedge \\
& \underline{\mathtt{F}}\mathtt{(}\neg\mathtt{(P_1(x,y)} \\
& \quad \wedge \\
& \quad \mathtt{(}\underline{\mathtt{F}}\ \underline{\mathtt{date}}\mathtt{(now)} \wedge \underline{\mathtt{date}}\mathtt{(now)} \\
& \quad \mathtt{)} \\
& \ \mathtt{)} \\
& \wedge\ \underline{\mathtt{date}}\mathtt{(d_1)}
\end{aligned}
$$

The subformulas $\mathtt{P_1(x,y)} \wedge \underline{\mathtt{X}}^{-}\neg\mathtt{P_1(x,y)}$ and $\underline{\mathtt{date}}\mathtt{(d_1)}$ define the day (d_1) of insertion of a record about staying of a patient in hospital. The subformula $\underline{\mathtt{F}}\mathtt{(}\neg\mathtt{P_1(x,y)} \wedge \mathtt{(}\underline{\mathtt{F}}\ \underline{\mathtt{date}}\mathtt{(now)} \vee \underline{\mathtt{date}}\mathtt{(now)))}$ defines the day of deletion of the record. The subformula $\underline{\mathtt{F}}\ \underline{\mathtt{date}}\mathtt{(now)} \vee \underline{\mathtt{date}}\mathtt{(now)}$ ensures that the day is between d_1 and *now* (including *now*). If the query did not contain this subformula every record would satisfy the query. This is because transaction time can not be later than *now* (transaction time is always the real time of the execution of an operation).

The result of the query is record (3, Nowak) (Fig. 10). Record (2, Kozłowski) was inserted before d1 and record (1, Kowalski) has not been deleted till *now*.

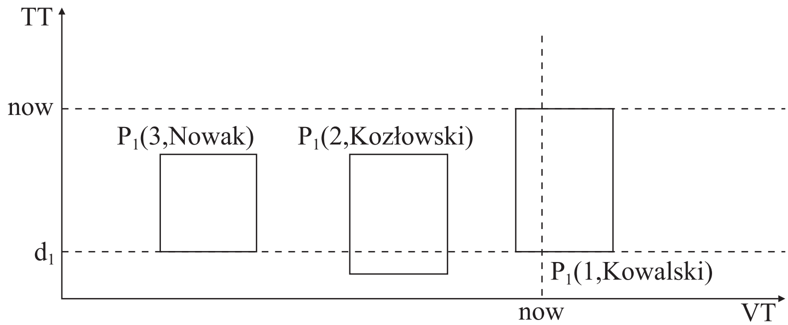


Figure 10. Contents of database at the moment of execution of query 7 and 8

Query 8

What records from table PATIENTS about staying in hospital inserted in the day of d_1 are in the current state of database (have not been deleted). The time of staying in hospital is irrelevant.

$$
\begin{aligned}
\varphi = \; & (\mathtt{P_1(x,y)} \\
& \quad \underline{\mathtt{S}} \\
& \quad (\mathtt{P_1(x,y)} \wedge \underline{\mathtt{X}}^{-}\neg \mathtt{P_1(x,y)} \wedge \underline{\mathtt{date}}\mathtt{(d_1)} \\
& \quad) \\
&) \wedge \mathtt{P_1(x,y)} \wedge \underline{\mathtt{date}}\mathtt{(now)}
\end{aligned}
$$

The subformula $\mathtt{P_1(x,y)} \wedge \underline{\mathtt{X}}^{-}\neg \mathtt{P_1(x,y)} \wedge \underline{\mathtt{date}}\mathtt{(d_1)}$ defines the day (d_1) of insertion of a record about staying in hospital. The subformula $\mathtt{P_1(x,y)}$ in the last part of the query ensures the record has not been deleted. If the query did not contain the subformula, records deleted the present day (henceforth not in the current state of database) would satisfy the query. The use of connective $\underline{\mathtt{S}}$ is for the case that a record was deleted and inserted again between $\mathtt{d_1}$ and `now`. Such a record should not satisfy the query.

The result of the query is record (1, Kowalski) (Fig. 10). Record (3, Nowak) was deleted. Record (2, Kozłowski) was inserted before d_1 and deleted as well.

Summary

The paper presented a formal definition of a bitemporal database, i.e., a database recording valid time and transaction time of data. It concentrated on the definition of a query language for this kind of database. The language is a slight modification of the language of the first order temporal logic. It enables to form queries in a convenient and intuitive way. One does not need to use variables to refer to time, hence the transcript of a query is shorter and less complicated. The paper presented a few examples of queries referring to the database containing medical data.

References

[ChoTom] Chomicki J., Toman D.: *Temporal Logic in Information Systems.* In Logics for Databases and Information Systems, Kluwer Academic Publishers, 1998, s. 31–70.

[Cod] Codd E. F.: *A Relational Model of Data for Large Shared Data banks.* In Communications of the ACM, 1970, 13 (6), s. 377–387.

[Dat] Date C. J.: *Introduction to Database system.* Addison-Wesley, 2003.

[Etz] Etzion, Jajodia, Sripada: *Temporal Databases: Research and practice.* Springer, 1998.

[Gab] Gabbay D. M., Reynolds M. A., Finger M.: *Temporal Logic, Mathematical Foundations and Computational Aspects.* Vol2, Clarendon Press, Oxford, 2000.

[Jen] Jensen, C. S.: *Temporal database Management.* PhD Thesis, Department of Computer Science, Aalborg University, 2000, [http://www.cs.aau.dk/∼csj/Thesis/].

[JenSno] Jensen C. S., Snodgrass R. T.: *Semantics of Time-Varying Information.* In Information Systems, 21(4), 1996, s. 311–352.

[Sno] Snodgrass R., editor.: *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

[Ste] Steiner A.: *A Generalisation Approach to Temporal Data Models and Their Implementations.* PhD Thesis, Departement Informatik, ETH Zurich, Switzerland, 1997.

[Tan1] Tansel A. U. i in.: *Temporal Databases: Theory, Design, and Implementation.* Benjamin-Cummings Publishing Co., Inc., 1993.

[Tan2] Tansel A., Tin E.: *Expressive Power of Temporal Relational Query Languages and Temporal Completeness.* In Temporal Databases: Research and Practice, LNCS 1399, Berlin Heidelberg, Springer-Verlag, 1998, s. 129–149.

Mariusz Giero
Chair of Logic, Informatics and Philisophy of Science
University of Białystok
ul. Sosonowa 64, 15–887 Białystok, Poland
giero@uwb.edu.pl

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 17 (30) 2009

Mariusz Giero
Institute of Computer Science, University of Bialystok

Robert Milewski
Institute of Computer Science, University of Bialystok

STORING AND RETRIEVING INFORMATION ON THE TREATMENT OF INFERTILITY WITH THE USE OF THE BITEMPORAL DATABASE AND TEMPORAL LOGIC

Abstract: In the paper, we present basic information on bitemporal databases and temporal logic. We show how the language of temporal logic can be used as a query language of a database. We use data of the treatment of infertility to present practical aspects of our research.

1. Introduction

The bitemporal database is a database which apart from "proper" data stores valid time and transaction time of data [7, 9, 11, 13]. Valid time is the time when an event represented by data takes place. Transaction time is the time when data is held in the database (the period from the insertion of data to possible deletion of data). Storing both kinds of time information enables more accurate description of events. We can retrieve not only when an event occured but also when the information about the event is inserted (deleted) into the database. Valid time enables to query the history of events and transaction time in the history of the database states.

The bitemporal database needs, among others, a redefinition of standard operations: modification and deletion. Modification does not cause overwriting of a record. It sets, with an additional attribute, that a given record is no longer in the database and inserts a new record with updated values. Deletion does not remove a record physically but logically. It sets, with an additional attribute, that a given record is no longer in the database [10].

The paper presents the application of the bitemporal database in the management of information connected with the course of treatment using

ISBN 978–83–7431–231–8 ISSN 0860-150X

the IVF method ICSI/ET (see next chapter). In this particular case, valid time is the time when medicine is given to a patient. As far as the transaction time is concerned, there is little modification. It is the time when the decision of giving medicine is taken (and holds). The moment of taking the decision is determined not through consultation with doctors but by the insertion of the decision into database. We assume that there are no delays. It is very important that the transaction (decision) time is recorded automatically by the database system (the moment when data is inserted). The user can not establish that time personally. The user can not insert the decision with a backdate. Such functionality is aimed at ensuring the integrity of data and preventing errors and fraud. Chapter 3 provides basic information on how the bitemporal database functions.

2. The treatment of infertility using the IVF ICSI/ET procedure

The problem of infertility concerns about 10–15% couples at reproductive age [1]. The number of couples requiring treatment due to problems with getting pregnant has been constantly increasing during the last decades. Social and economic conditions cause postponing maternity by women until subsequent years of life. A constant trend in the rise of the age of women giving birth to their first child has been observed [2]. The postponing of reproductive decisions is going hand in hand with the decrease of natural reproductive potential of women [3, 4]. It also causes delayed diagnosis of possible problems with reproduction, and next, it leads to the decrease of the effectiveness of infertility treatment [5]. The most effective method of infertility treatment is the application of the in vitro fertilization (IVF) procedure with intracytoplasmic sperm injection (ICSI) and embryo transfer (ET). The average effectiveness of the infertility treatment with IVF methods oscillates at about 30% value, and in the group of women over 40 is significantly lower, and oscillates between 8% and 18% [6].

The process of infertility treatment using the IVF ICSI/ET procedure proceeds according to the following stages:
- gathering of information concerning personal data of the woman and the man, who reported to the clinic,
- gathering information about previous treatment,
- general examination:
  - medical interview of the woman,
  - medical interview of the man,

    - laboratory tests,
    - ultrasonography imaging,
- treatment during subsequent days and observation of the developing ovarian follicles,
- embryology
    - ovarian follicles aspiration,
    - sperm preparation,
    - ART procedure,
- embryo transfer
    - evaluation of pronuclei,
    - evaluation of embryos,
    - evaluation of blastocysts,
    - transfer of embryos,
- registration of treatment results
    - pregnancy,
    - childbirth,
    - information about children.

In the paper we have only concentrated on one of the stages presented above, i.e. the treatment and ovarian hyperstimulation. At this stage, serial registration of the considerable quantity of information about the stimulation process is required. Initially, the protocol of stimulation is established, and next, all information concerning medicine to be served in the following days is noted. Also the response of the woman's organism is recorded. Various information is collected, e.g., the level of estradiol, the thickness of endometrium, the quantity and the size of developing ovarian follicles in both ovaries.

3. Basic operations of the bitemporal database

Let us examine a database consisting of one relation *Treatment* with 6 six attributes: *id*, *sort*, *name*, *dose*, VT, TT where the course of treatment is recorded. Values of the listed attributes are as follows: the identifier of the patient, the group medicine belongs to, the dose of medicine, the time when medicine is given (valid time), the time when the decision of giving medicine is undertaken (transaction time). The values of VT, TT are intervals. The starting point of the interval is always a specific date and the end point of the interval is either a specific date or infinity ∞. The attributes *id*, *sort*, *name*, *dose* are defined by the user while creating table (relation). Attributes VT, TT are added to each created relation by the database system.

Insertion of data

On the 10-th of March, the user inserts the following information: GonalF (FSH group) with a dose of 150 per day is to be given to patient 1 starting on the 11th of March (this decision is taken on the 10th). The following record is inserted to the database:

Id	*Sort*	*Name*	*Dose*	*VT*	*TT*
1	FSH	GonalF	150	$[11.03 - \infty)$	$[10.03 - \infty)$

If the user does not give the end date of VT the database system sets the value to ∞. This means that medicine is to be given until the decision is changed. The start date of VT can not be earlier than the current date. The user is not allowed to perform such action. The value of TT is set by the database system. The start date is equal to the date of the insertion (i.e. current date). The end date is equal to ∞. This means that the taken decision is in force. When the user inserts a record the value of the end date of TT is always set to ∞.

Modification of data

There is a difference between modification of attribute VT and other attributes (apart from TT attribute which is not allowed to be set by the user).

Modification of VT attribute

On 15th the user modifies the end date of VT in the inserted record. He sets the value of the end date of TT to 20.03 (in other words, an event is recorded that on the 15th a new decision has been taken: giving medicine is to be finished on the 20th). This operation results in the modification of the record inserted previously (this record is to inform that the decision is no longer in force) and in inserting a new record with information on the new decision:

Id	*Sort*	*Name*	*Dose*	*VT*	*TT*
1	FSH	GonalF	150	$[11.03 - \infty)$	$[10.03 -$ **15.03**$)$
1	**FSH**	**GonalF**	**150**	$\mathbf{[15.03 - 21.03)}$	$\mathbf{[15.03} - \infty\mathbf{)}$

New values are in bold. The change of the decision is recorded by setting the end date of TT to a specific date. The date is set by the system and is equal to the current date (the date of the execution of modification). Next, the system inserts a new record. The values of non temporal attributes (i.e. *id*, *sort*, *name*, *dose*) are the same as in the previous record.

The value of VT is the time that remains for giving the medicine and is computed by the system. As intervals representing VT and TT are right-open, the end date of VT is 21.03 which means that 20.03 is the last day when the medicine is given. Analogically, the value of VT in the first record is 15.03. This means that 14.03 is the last day when the decision is in force.

The value of TT is set as in insertion operation. The start date is the current date, the end date is ∞.

Modification of non temporal attributes

On the 17th, the user changes the dosage: from 19.03, the patient is to be given 75 of GonalF (in other words, on the 17th the decision of the change of the dosage is taken). Modification is conducted on the second record in the above database (the modification does not refer to the first record, besides, the first record is no longer allowed to be modified).

The modification results in the change of the end date of TT of the record (as in previous modification) and inserting 2 new records: the first one informs about the dosage between the 17th and the 19th (the period between the moment when the last decision is no longer in force and the moment a new decision i.e. new dosage starts to be in force), the second one informs about the new dosage:

Id	*Sort*	*Name*	*Dose*	*VT*	*TT*
1	FSH	GonalF	150	[11.03 – ∞)	[10.03 – 15.03)
1	FSH	GonalF	150	[15.03 – 21.03)	[15.03 – **17.03**)
1	**FSH**	**GonalF**	**150**	**[17.03 – 19.03)**	**[17.03 – ∞)**
1	**FSH**	**GonalF**	**75**	**[19.03 – 21.03)**	**[17.03 – ∞)**

New values are supplied in bold. The change of the last decision (the second record) is recorded in an analogous way as in the previous modification. The third record informs on how many days the medicine is to be given with the old dosage. The attribute values of the record are computed by the system (the record represents a new decision, therefore the end date of TT is ∞). The fourth record informs on when the new dosage starts (the record also represents the new decision and the end date of TT is ∞). The end date of VT remains not changed (the user wanted only to change the start date of VT).

Deletion of data

A record is not physically removed from database but logically. It results in changing the end date of TT from ∞ to a specific date (date of executing of deletion).

In order to retrieve the course of treatment one needs to take intersection of VT and TT. With the above database we have:

Id	*Sort*	*Name*	*Dose*	$VT \cap TT$
1	FSH	GonalF	150	[11.03 – 15.03)
1	FSH	GonalF	150	[15.03 – 17.03)
1	FSH	GonalF	150	[17.03 – 19.03)
1	FSH	GonalF	75	[19.03 – 21.03)

The course of treatment is as follows: GonalF was given to patient 1 with dosage 150 from the 11th to the 18th (included) (the three records), then from the 19th to the 20th (included) with dosage 75 (the fourth record). The treatment was finished on the 20th.

4. Querying data

The alphabet and syntax of the query language

Queries can be formed with the language of first-order temporal logic [8, 9, 12]. The language, compared to classical logic, is extended with temporal operators which enable to form queries in a convenient and intuitive way. The alphabet has the following categories of basic symbols:

- domain variables: $x_1, x_2, \ldots$;
- domain constants: $c_1, c_2, \ldots$;
- time variables: $t_1, t_2, \ldots$;
- time constants: $e_1, e_2, \ldots$;
- predicate symbols: $P_1, P_2, \ldots, P_k$;
- equality symbol: $=$;
- classical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$;
- quantifiers: $\exists, \forall$;
- temporal connectives: $\mathtt{F}, \mathtt{P}, \mathtt{G}, \mathtt{H}, \mathtt{X}, \mathtt{X}^{-}\mathtt{U}, \mathtt{S}$;

- punctuation symbols:), (.
- additional predicate symbol: vt;

A *term* is either a domain term or time term. A *domain (time) term* is either a domain (time) constant or a domain (time) variable. The atomic formulas of the language are of the form:

- $P_i(a_1, a_2, \ldots, a_n)$, where P_i is n-ary predicate symbol, $a_1, a_2, \ldots, a_n$ are domain terms
- $t = u$, where t, u are both either domain terms or time terms
- $vt(t)$, where t is time term

Formulas are finite strings of basic symbols defined in the following recursive manner:

(1) any atomic formula is a formula
(2) if φ, ψ are formulas, so also are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\varphi \leftrightarrow \psi$, $\exists x\varphi$, $\forall x\varphi$, $\mathtt{F}\varphi$, $\mathtt{P}\varphi$, $\mathtt{G}\varphi$, $\mathtt{H}\varphi$, $\mathtt{X}\varphi$, $X^{-}\varphi$, $\varphi\,\mathtt{U}\,\psi$, $\varphi\,\mathtt{S}\,\psi$ where x is any domain variable.
(3) there are not any other formulas than finite strings satisfying conditions (1) and (2)

Semantics of the query language

The model of the presented language is a bitemporal database defined as a structure $DB = (T, <, D, R_1, R_2, \ldots, R_n)$, where T is a set of dates, $<$ a relation ordering dates, D is a non empty set (database domain) and $R_1, R_2, \ldots, R_n$ are relations on $D \cup T$ such that

$$R_i \subseteq \{(d_1, \ldots, d_m, \tau_v, \tau_t) : d_1, \ldots, d_m \in D; \tau_v, \tau_t \text{ are integrals of the form } [t, u) \text{ or } [t, \infty), t, u \in T\}.$$

The valuation v associates every domain term to an element of D, every time term to an element of T. If c is a constant term, we make $v(c) = c$.

The interpretation I associates every predicate symbol P_i to a relation R_i of the database. We define formula A to be true in DB at time point $t \in T$ under valuation θ, writing $DB, \theta, t \models \varphi$, by induction on the structure of formula:

$DB, v, t \models P(a_1, \ldots, a_m)$ **iff** there exist τ_v, τ_t such that $(v(a_1), \ldots, v(a_m), \tau_v, \tau_t) \in I(P)$ and $t \in \tau_v \cap \tau_t$,

$DB, v, t \models x_i = x_j$ **iff** $\theta(x_i) = \theta(x_j)$

$DB, v, t \models \neg\varphi$ **iff** it is not the case that $DB, v, t \models \varphi$

$DB, v, t \models \varphi \wedge \psi$ **iff** $DB, v, t \models \varphi$ and $DB, v, t \models \psi$

$DB, v, t \models \varphi \vee \psi$ **iff** $DB, v, t \models \varphi$ or $DB, v, t \models \psi$

$DB, v, t \models \varphi \rightarrow \psi$ **iff** $DB, v, t \models \neg\varphi$ or $DB, v, t \models \psi$

$DB, v, t \models \varphi \leftrightarrow \psi$ **iff** $DB, v, t \models \varphi \rightarrow \psi$ and $DB, v, t \models \psi \rightarrow \varphi$

$DB, v, t \models \exists x_i \varphi$ **iff** there exists an $a \in D$ such that $DB, \theta^*, t \models \varphi$,

$DB, v, t \models \forall x_i \varphi$ **iff** for every $a \in D$ holds $DB, \theta^*, t \models \varphi$, where θ^* is a valuation which agrees with the valuation v on the values of all variables except, possibly, on the values of x_i

$DB, v, t \models \mathtt{P}\varphi$ **iff** there exists $t_1 \in T$ with $t_1 < t$ and $DB, v, t_1 \models \varphi$

$DB, v, t \models \mathtt{F}\varphi$ **iff** there exists $t_1 \in T$ with $t_1 > t$ and $DB, v, t_1 \models \varphi$

$DB, v, t \models \mathtt{H}\varphi$ **iff** for every $t_1 \in T$ whenever $t_1 < t$ then $DB, v, t_1 \models \varphi$

$DB, v, t \models \mathtt{G}\varphi$ **iff** for every $t_1 \in T$ whenever $t_1 > t$ then $DB, v, t_1 \models \varphi$

$DB, v, t \models \mathtt{X}\varphi$ **iff** $DB, v, t+1 \models \varphi$

$DB, v, t \models \mathtt{X}^{-}\varphi$ **iff** $DB, v, t-1 \models \varphi$

$DB, v, t \models \varphi\, \mathtt{S}\, \psi$ **iff** there exists $t_1 \in T$ with $t_1 < t$ and $DB, v, t_1 \models \psi$ and for every $t_2 \in T$ whenever $t_1 < t_2 < t$ then $DB, v, t_2 \models \varphi$

$DB, v, t \models \varphi\, \mathtt{U}\, \psi$ **iff** there exists $t_1 \in T$ with $t_1 > t$ and $DB, v, t_1 \models \psi$ and for every $t_2 \in T$ whenever $t_1 > t_2 > t$ then $DB, v, t_2 \models \varphi$

$DB, v, t \models vt(t)$ **iff** $vt(t) = t$

The temporal operators $\mathtt{F}, \mathtt{P}, \mathtt{G}, \mathtt{H}, \mathtt{X}, \mathtt{X}^{-}, \mathtt{U}, \mathtt{S}$ define the following expressions: "it will be that", "it was that", "it will always be that", "it has always been that", "next, it will be that", "previously, it was that", "... holds until ... holds", "... holds since ... holds", respectively. The predicate symbol vt is to set what time one poses the query at.

The query is every formula of the presented language with at least one free variable. The result of the query is defined as follows:

$$Q(DB, Q) = \{(v(x_1), \ldots, v(x_n))\colon DB, v, t \models \varphi(x_1, \ldots, x_n)\},$$

where DB is a database, φ is a query and $x_1, \ldots, x_n$ are the only free variables of φ.

Examples of queries

Let us consider a database consisting one relation *Treatment* with 6 six attributes: *id*, *sort*, *name*, *dose*, *VT*, *TT* where the course of treatment is recorded. Values of the listed attributes are as the following: the identifier of the patient, the group the medicine belongs to, the dose of medicine, the time when medicine is given (valid time), the time when the decision of giving medicine is held (transaction time).

id	*sort*	*name*	*dose*	*VT*	*TT*
1	aGnRH	Dipherelina	3,75	[24.02 – 25.02)	[20.02 – ∞)
1	FSH	Gonal F	150	[11.03 – ∞)	[10.03 – 15.03)
1	FSH	Gonal F	150	[15.03 – 21.03)	[15.03 – 17.03)
1	FSH	Gonal F	150	[17.03 – 19.03)	[17.03 – ∞)
1	FSH	Gonal F	75	[19.03 – ∞)	[17.03 – 22.03)
1	FSH	Gonal F	75	[22.03 – 23.03)	[22.03 – ∞)
1	HCG	Pregnyl	10000	[23.03 – 24.03)	[22.03 – ∞)
2	FSH	Gonal F	150	[16.03 – 22.03)	[15.03 – ∞)

One can form the following queries (the predicate symbol L is interpreted as the relation *Treatment*):

1. Who is given GonalF from the group of FSH with the dosage of 150 at the present time (we assume that today is the 11th of March)?

   $\varphi = L(x_1, \text{FSH}, \text{Gonal F}, 150) \wedge vt(11.03)$

   $Q(DB, \varphi) = \{1\}$

2. Who was treated with GonalF before the 18th of March. What was the dosage then?

   $\varphi = \mathtt{P}L(x_1, \text{FSH}, \text{Gonal F}, x_2) \wedge vt(18.03)$

   $Q(DB, Q) = \{(1, 150), (2, 150)\}$

3. Who took GonalF and after that Pregnyl? We refer the query to the 31st of March

   $$\varphi = \exists x_2, x_3\ \mathtt{P}(L(x_1, \text{FSH}, \text{Gonal F}, x_2) \wedge \mathtt{F}(L(x_1, \text{HCG}, \text{Pregnyl}, x_3) \wedge \mathtt{F}vt(31.03))) \wedge vt(31.03)$$

   $Q(DB, Q) = \{1\}$

4. Who was treated at least 5 days? We refer the query to the 23th of March.

   $$\varphi = \exists x_2, x_3, x_4 (L(x_1, x_2, x_3, x_4)\mathtt{S}(L(x_1, x_2, x_3, x_4) \wedge vt(17.03))) \wedge vt(23.03)$$

   $Q(DB, \varphi) = \{1\}$

5. Future work

Larger amount of data should be used to examine the efficiency of the bitemporal database. Complicated queries should be formed to examine the utility of the presented query language. The implementation of the bitemporal database is also taken into consideration.

6. References

[1] Adashi E.Y., Rock J.A., Rosenwaks Z. Human infertility: an introduction. Reproductive endocrinology, surgary and technology. Lippincott-Raven Publishers, Philadelphia 1996.

[2] te Velde E.R., Pearson P.L. The variability of female reproductive ageing. Hum. Reprod. 2002; 8: 141–154.

[3] Evers J.L. Female subfertility. Lancet 2002; 360: 151–159.

[4] Menken J. et al. Age and infertility. Science 1986; 233: 1389–1394.

[5] Broekmans F.J. et al. Antral follicle counts are related to age at natural fertility loss and age at menopause. Menopause 2004; 11: 607–614.

[6] Milewski R., Milewska A.J., Domitrz J., Wołczyński S. Zapłodnienie pozaustrojowe IVF ICSI/ET u kobiet po 40. roku życia. Przegląd Menopauzalny 2008; 2: 85–90.

[7] Etzion, Jajodia, Sripada: Temporal Databases: Research and practice. Springer, 1998.

[8] Gabbay D. M., Reynolds M. A., Finger M.: Temporal Logic, Mathematical Foundations and Computational Aspects. Vol1, Clarendon Press, Oxford, 1994.

[9] Gabbay D. M., Reynolds M. A., Finger M.: Temporal Logic, Mathematical Foundations and Computational Aspects. Vol2, Clarendon Press, Oxford, 2000.

[10] Giero M.: Application of bitemporal databases containing medical data. Studies in Logic, Grammar and Rhetoric, vol. 17(30), 2009, pp.

[11] Jensen, C. S.: Temporal database Management. PhD Thesis, Department of Computer Science, Aalborg University, 2000, [http://www.cs.aau.dk/∼csj/Thesis/].

[12] Trzęsicki K.: Logika temporalna. Wybrane zagadnienia. Wydawnictwo Uniwersytetu w Białymstoku, Białystok, 2008.

[13] Tansel A. U. i in.: Temporal Databases: Theory, Design, and Implementation. Benjamin-Cummings Publishing Co., Inc., 1993.

Mariusz Giero
Chair of Logic, Informatics and Philisophy of Science
University of Białystok
ul. Sosonowa 64
15–887 Białystok, Poland
giero@uwb.edu.pl

Robert Milewski
Medical University of Bialystok
Department of Statistics and Medical Informatics
Mickiewicza 2c
15–089 Białystok
milewski@umwb.edu.pl

Milewski Robert
Medical University of Bialystok

Jamiołkowski Jacek
Medical University of Bialystok

Milewska Anna Justyna
Medical University of Bialystok

Domitrz Jan
Medical University of Bialystok

Wołczyński Sławomir
Medical University of Bialystok

THE SYSTEM OF ELECTRONIC REGISTRATION OF INFORMATION ABOUT PATIENTS TREATED FOR INFERTILITY WITH THE IVF ICSI/ET METHOD

Abstract: Infertility treatment with IVF (in vitro fertilization) methods require collection, storage and analysis of large quantity of data. Existing hospital systems are not prepared to gather such detailed and specialized information. This situation stimulated the formation of a group dealing with the infertility treatment as well as software programming. The group also decided to create a suitable system for gathering this type of information.
After a long period of preparations, consultations and tests, the system has been initiated and implemented in the Clinic of Reproduction and Gynecological Endocrinology in Bialystok. The transparent structure and the form of implementation of the system allow it to be used by untrained personnel involved in work with other hospital systems. Data relating to few hundred couples treated for infertility using the IVF ICSI/ET method has been accumulated up to the date of writing of this paper.

Introduction

The character of treatment of infertility requires gathering, storage as well as continuous analysis of large quantity of specialized data, and also the assurance of the possibility of quick access. The hospital information systems are familiar to medical computer specialists [1], however the existing hospital systems are not adapted to accumulate specialized data. Each general system, designed to be applied in the unit of medical care accumulates only the most general data, mainly related to the personal information,

ISBN 978–83–7431–231–8 ISSN 0860-150X

diseases, executed treatment, diagnostic procedures, therapy, medical education as well as the hospital administration [2]. In numerous cases they cover demand of the medical units, resulting from administrative needs and the requirements imposed by the National Health Fund [3]. But it is necessary to find other solutions to gather much more specialized information.

Because of the still changing medical procedures [4], it is not likely to find on the medical software market an application, which will be prepared to gather our specific data. Moreover, one should not expect products which will conduct desirable statistical analyses of the gathered data. The best and probably the only satisfactory solution seems to be the creation of a local system designed by both programmers, as well as personnel who will use this system. Therefore, a team was formed, and its joint effort led to the creation of an application fulfilling the awaiting tasks.

The lack of a specialistic system for registering information concerning patients described above was noted at the Clinic of Reproduction and Gynecological Endocrinology of Medical University of Bialystok. One of the tasks of the clinic is to initiate and conduct the process of treatment of couples suffering from infertility problems. In the process, many complicated procedures are performed, among which a considerable part are procedures related to the process of in vitro fertilization (IVF) joint with intracytoplasmic sperm injection (ICSI), and then with the transfer of the created in this way embryo (ET) into the uterine cavity [4]. So far, the magnitude of information connected with every treated couple was recorded mainly in paper form, which had a restrictive influence on the possibility of the application and the analysis of accumulated data. This situation led to the formation of a group who decided to create a system able to collect data of patients going through the IVF ICSI/ET procedures.

The course of the treatment process of infertility using the IVF ICSI/ET procedure

The work of the team began with a detailed discussion and considering possible decisions relating to the range of accumulated information, as well as the form it will be stored. The pattern of process of treatment which consists of several main stages of accumulating information was created (Fig. 1). In the first order, personal data of the couple which applies for treatment is gathered. Then, information relating to previous treatment, as well as the results of various examinations are collected, including the medical interview of the woman and the man, laboratory tests, and also ultrasonography image

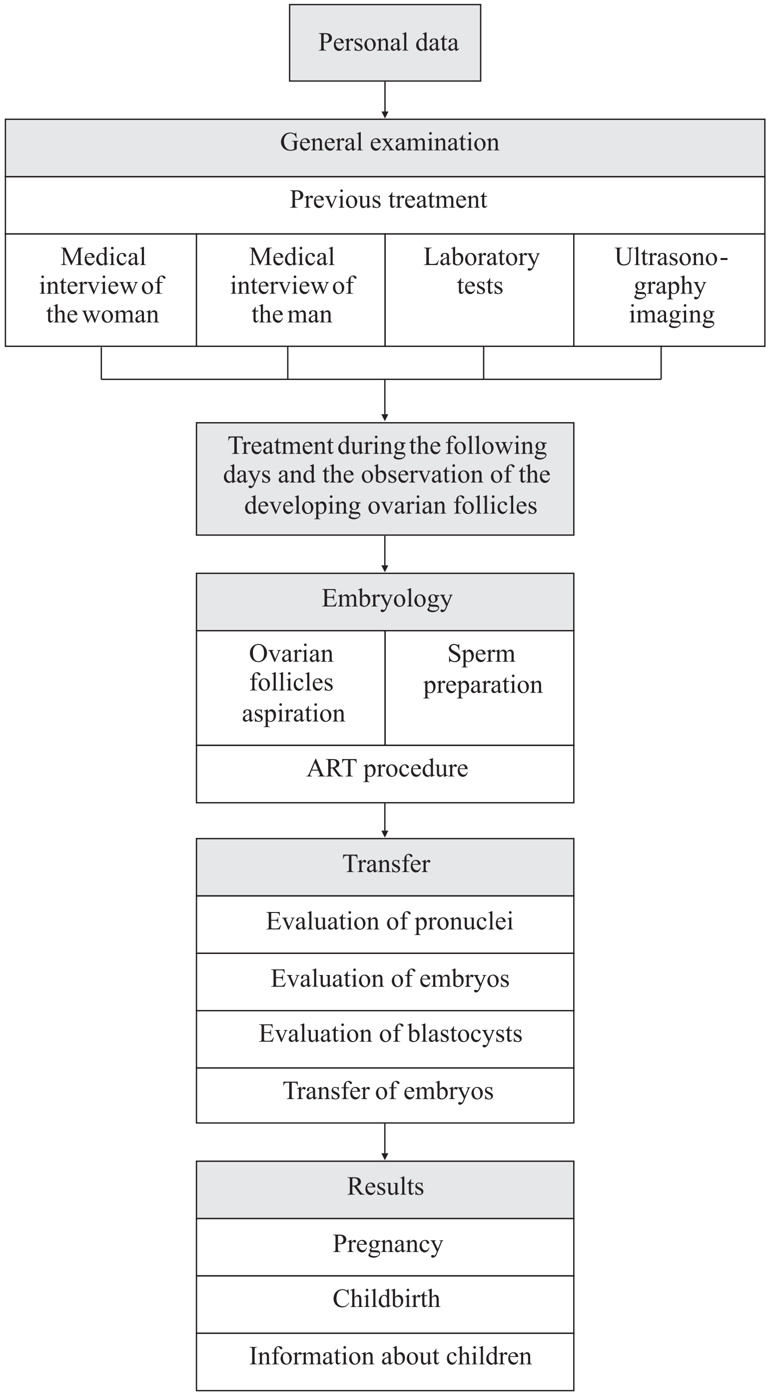


Figure 1. The pattern of the treatment process using the IVF ICSI/ET procedure

and its description. After the end of this stage, the couple is qualified for further treatment, the protocol of stimulation is chosen and the period of treatment begins. Information concerning drugs administered and ovulation stimulation in following days of the treatment process are gathered.

The level of estradiol, thickness of the endometrium, as well as the quantity and the size of the developing ovarian follicles in both ovaries are recorded in the later stage. The following part concerns embryology: the information relating to the process of ovarian follicles aspiration, preparation of sperm and the ART procedure are collected. In the next stage, the information about the developing embryo is gathered until the moment of its transfer. The last part concerns the results of the whole process, where the relating data of the potential pregnancy, childbirth, as well as basic information about children born as a result of the treatment are recorded.

The application handling the infertility treatment process

The application was created in the software development environment Delphi 2007 [5, 6], while the database was implemented using Microsoft Access [7] from the Microsoft Office 2003 software. The database consists of 47 tables, which keep all essential information recorded during the process of treatment. Every table contains several fields of different data type. The key tables are *Cycles*, containing basic information about each cycle of treatment, and also *Couples*, where personal data of treated couples is recorded. The tables are connected with relations of *one to many* type [7], because one couple can have many cycles of treatment registered. The database handling the application contains the information about the couples and the cycles on the main panel, visible in two grids after starting the program (Fig. 2). The system was designed to be used in Poland (there is no English version yet), therefore all print screens contain phrases in the Polish language.

The upper part of the grid allows to add a new couple and to remove or modify the existing one. The window, which appears after selecting option "add a new couple" is shown on (Fig. 3).

The process of addition of a new couple cannot be finished until suitable data is typed in the required fields. These fields concern the information which is indispensable for administrative or contact reasons (name, surname, PESEL number, address, area code and telephone). The information about the added couple is displayed in the upper part of the grid, which also unlocks the possibility of defining following cycles of treatment in the bottom part of the grid.

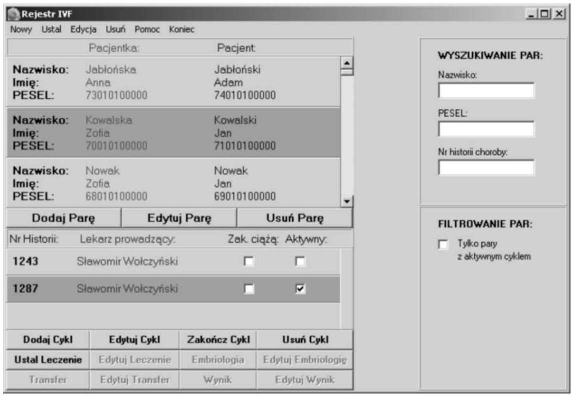


Figure 2. The grids of patients and the treatment cycles

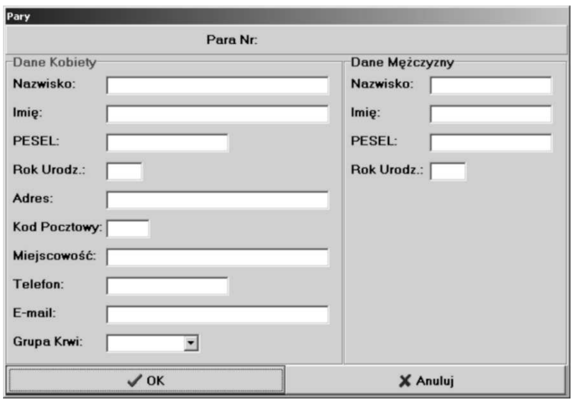


Figure 3. The window of a new couple addition

There are components for searching patients on the right side of the main application window (Fig. 2). Searching can be done according to three criteria: surname, PESEL number or the number of the case. The possibility of turning on filtering also exists, so only couples which are in the progress of an active cycle are displayed. Filtering according to the active cycle hides the whole history, leaving only presently treated couples.

Not all buttons are active in the cycle part because the process of treatment requires going through subsequent, linearly ordered stages. According to (Fig. 1), first it is necessary to create a new cycle, then plan the treatment, enter information relating to embryology and transfer, and finally enter the results of the treatment.

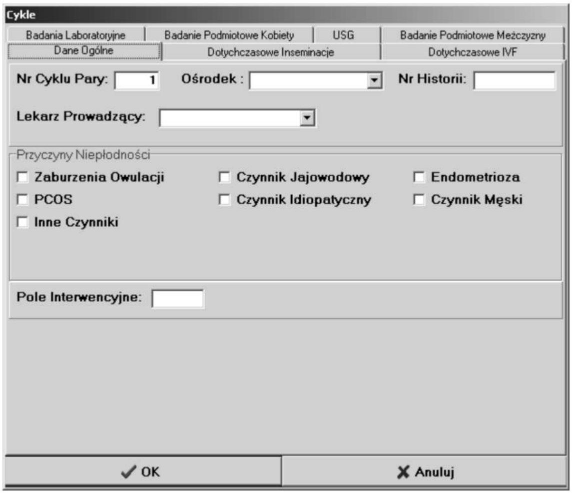


Figure 4. The window of a new cycle addition

The addition of a new cycle to the database is executed in the window (Fig. 4), consisting of 7 tabs:

- General information – identification information relating to the current cycle, physician's data and affirmed infertility causes,
- Previous inseminations – quantity and results of conducted inseminations using husband's (AIH) or donor's (AID) sperm, including division into natural and stimulated cycles,
- Previous IVF – quantity and results of previously conducted cycles of in vitro fertilization, including division into classical IVF cycles, IVF cycles with intracytoplasmic sperm injection (ICSI) and frozen embryos transfer,
- Laboratory tests – dates and results of more than ten laboratory tests, required from the woman before starting the treatment, as well as results of sperm examination,
- Medical interview of the woman – interview includes basic gynecological information, description of ovulation, the course of previous attempts to get pregnant and existing diseases, performed operations and gynecological interventions,
- Ultrasonography – date, description and ultrasonography image,
- Medical interview of the man – interview relating to the man's reproductive history, existing diseases, as well as results of medical examination of testicles, epididymides and vasa deferentia (for patients with severe oligoasthenoteratospermia or azoospermia).

The following tabs allow to enter appropriate data. The ultrasonography tab deserves a special attention (Fig. 5). It allows to read image recorded during ultrasonography examination of the patient from the disc file or from clipboard. This image, similarly to other data is placed in the database. The application handles different graphical image types: *.jpg, *.jpeg, *.bmp, *.ico, *.emf, *.wmf.

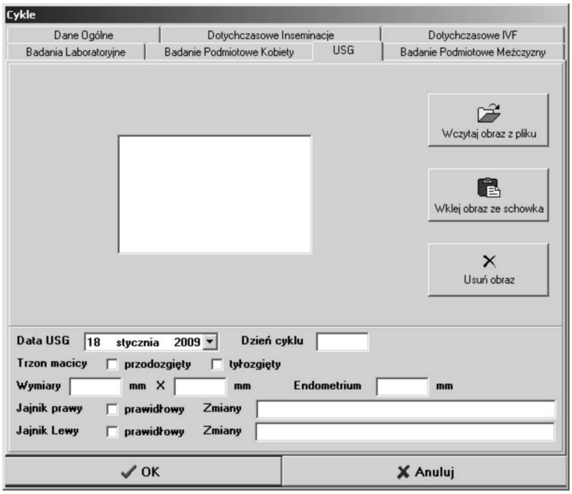


Figure 5. The "ultrasonography" tab of the window of a new cycle addition

After gathering preliminary information, it is required to choose the treatment protocol, which determines the order of proceedings in the first stage of the treatment. There are 11 predefined types of protocols in the database (Tab. 1), which we can divide into 4 main categories:
- short (protocol 1),
- long (protocols 2–5),
- with antagonist (protocols 6–9),
- other (protocols 10–11).

Individual protocols differ from one another as to the kind, dose and the time of administering the analogs of the GnRH, and the time of administering the hormones FSH, HMG and LH.

The treatment protocol establishing window is shown on (Fig. 6). Protocols can be chosen from the drop-down list. The short profile of the chosen pattern of treatment is displayed in the right side of the window. The date of treatment beginning is also set here, as well as the date of last menstruation. It is also possible to add information about potential contraception applied in the period previous to the current treatment.

Table 1

The kinds and the profile of treatment protocols

ID	Name	aGnRH from day	aGnRH till day	FSH	HMG	LH	from day	till day
1	short 0,05 (1)	1	8	yes	yes	yes	3	8
2	long 0,05 (2)	1	20	yes	yes	yes	15	20
3	long 0,05 (3)	1	14					
4	long 3,75 (4)	1	1					
5	long 3,75 (5)	1	1	yes	yes	yes	15	20
6	with antagonist 0,25 (6)	6	6	yes	yes	yes	1	5
7	with antagonist 3,0 (7)	6	6	yes	yes	yes	1	5
8	with antagonist 0,25 (8)			yes	yes	yes	1	5
9	with antagonist 3,0 (9)			yes	yes	yes	1	5
10	lupron stop 0,1 (10)	1	8					
11	free (11)							

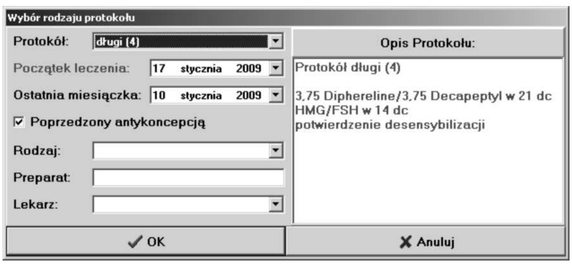


Figure 6. The treatment protocol establishing window

After confirming the chosen protocol, the window of drugs choice and doses of the first phase of the treatment is displayed. The appearance of the window depends on the selected pattern of the protocol (Fig. 7). The choice of drugs and their doses is required to pass on to the next stages. Treatment settled in this way is recorded in the database with division into the following days of treatment and is displayed on a treatment grid (Fig. 8).

It is possible to add new treatment days as well as to edit the existing ones. In the edition window (Fig. 9) it is possible to establish all accessible parameters relating to drugs and their doses. In the further stage also the thickness of the endometrium is defined, as well as the level of estradiol. The sizes of ovarian follicles in both ovaries obtained from ultrasonography imaging are also entered.

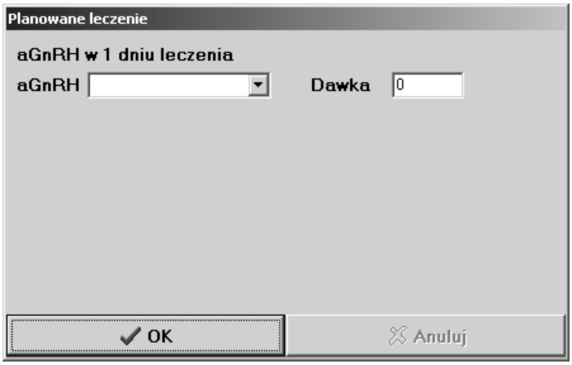


Figure 7. The drugs and doses of the first phase treatment window

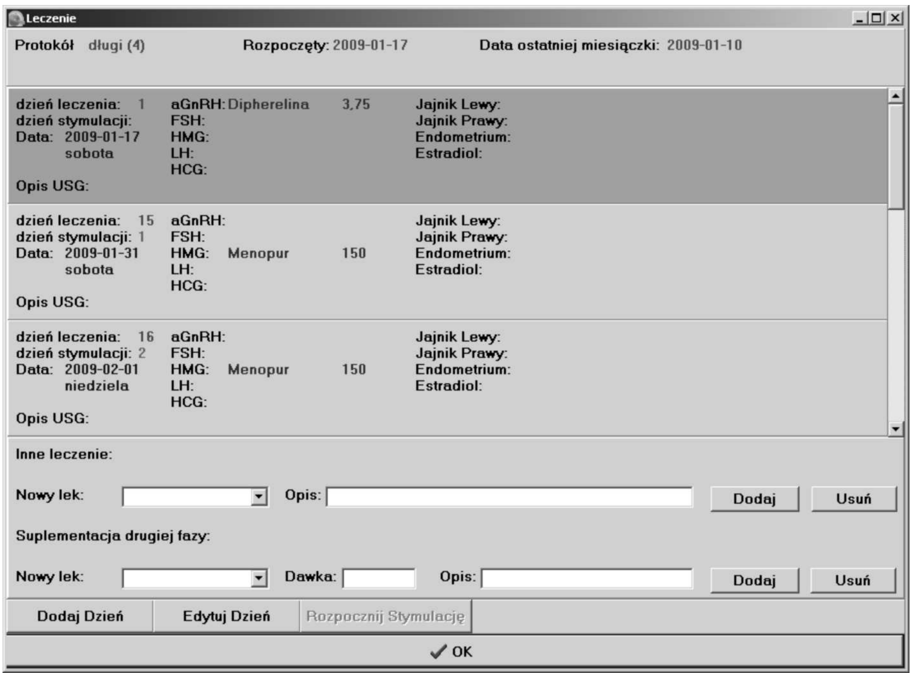


Figure 8. The following days of the treatment grid

The stimulation is initiated in the next step. In the application this procedure is started using "Begin stimulation" button in the following days of the treatment window (Fig. 8). It displays the window of the treatment planning with the possibility of choosing the drugs and their doses for 6 subsequent days of the stimulation (Fig. 10).

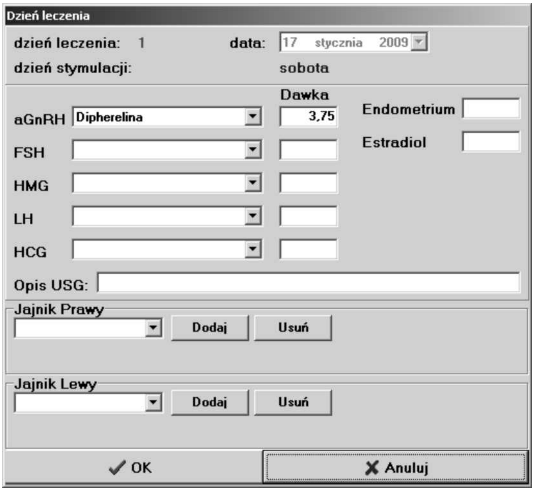


Figure 9. The individual day of treatment editing window

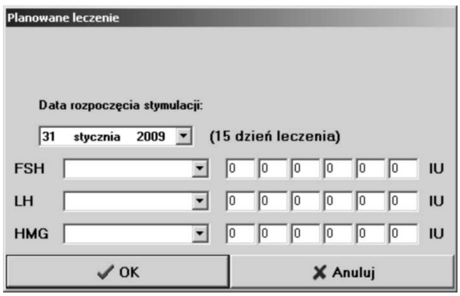


Figure 10. The ovulation stimulation planning window

After confirming the treatment the following days are added automatically on the grid of the stimulation window (Fig. 8). Depending on woman's organism response, the stage of stimulation of ovulation can be continued. As it was previously mentioned, the information concerning the developing ovarian follicles is also gathered here. At the moment, when the doctor affirms suitable time to finish the stimulation (the sufficient quantity of the suitably large ovarian follicles, the level of estradiol and the thickness of endometrium) a suitable HCG drug is administered to stimulate the ovulation peak and to make further treatment possible. The day of administering the HCG drug is the last day on the grid in the treatment window (Fig. 8). Application of HCG is required, to enable the opening of a new window. In cycles, without HCG administered yet, pressing the button *Embryology* will display a message about the necessity of HCG application first.

After completing the above described condition, the possibility of beginning the next stage appears. It results in the opening the embryology window (Fig. 11).

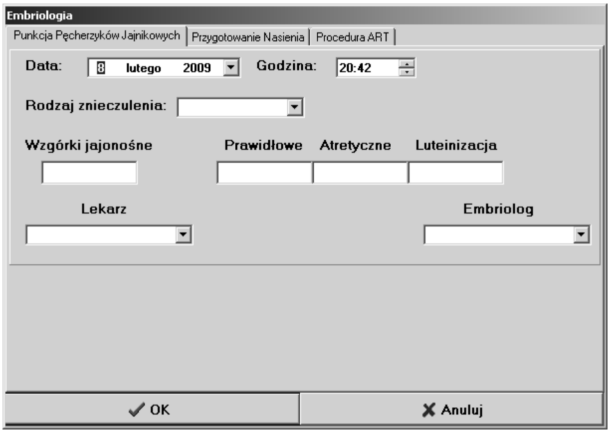


Figure 11. The embryology window

The window consists of three tabs, related to three stages of proceeding. These are:

– The aspiration of ovarian follicles – the time of aspiration, kind of anesthetics, quantity and the category of acquired cumulus oophorus, as well as the doctor's and embryologist's surname,
– The preparation of semen – the source of sperms, data related to the quality and the volume of semen, preparation of semen as well as the embryologist's surname,
– The ART procedure – data related to ART procedure, including division into classic case and ICSI, number of acquired oocytes as well as embryologist's surname.

The completion of the embryology window tabs makes possible the advancement to the next stage, which is related to transferring the fertilized ovum into uterine cavity.

Clicking the "Transfer" button results in opening the evaluation and transfer of embryos window (Fig. 12).

This window contains four tabs, related to three stages of proceeding. These are:

– The evaluation of pronuclei – the time of evaluation, the quantity of pronuclei in individual classes as well as the surname of the person, who performs the evaluation,

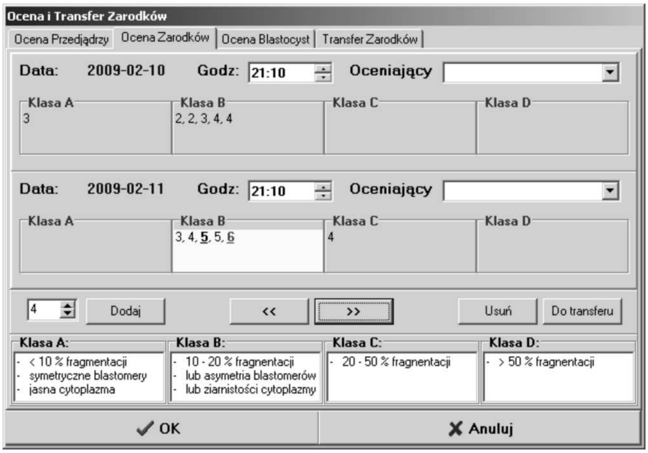


Figure 12. The embryos evaluation and transfer window

- The evaluation of embryos – the quantity of received embryos in second and third day of growth, with division into four classes, based on percentage of fragmentation. It is possible to select embryos intended for transfer in this tab,
- The evaluation of blastocysts – the evaluation of the blastocysts in the fifth and sixth day of growth, with division into categories and the surname of the person, who performs the evaluation. This tab also makes possible to define quantity of blastocyst intended to transfer,
- The transfer of embryos – the time of transfer into the uterine cavity, the qualification of the degree of transfer's difficulty and applied probe, the doctor's and the embryologist's surname, the quantity and class of transferred embryos and blastocysts, quantity of frozen embryos.

The completion of four tabs mentioned above allows to pass on to the last stage of the proceeding, in which the information related to the effectiveness of treatment is accumulated (Fig. 13).

Information related to the presence of pregnancy, the number of children, the week of childbirth, the method of childbirth, possible complications, as well as the state, weight and Apgar score of born children are recorded.

The status of the finished cycle of treatment can be changed to inactive by pressing the button "Finish cycle" in the main application window (Fig. 2). It allows to filter out couples with active cycle in progress.

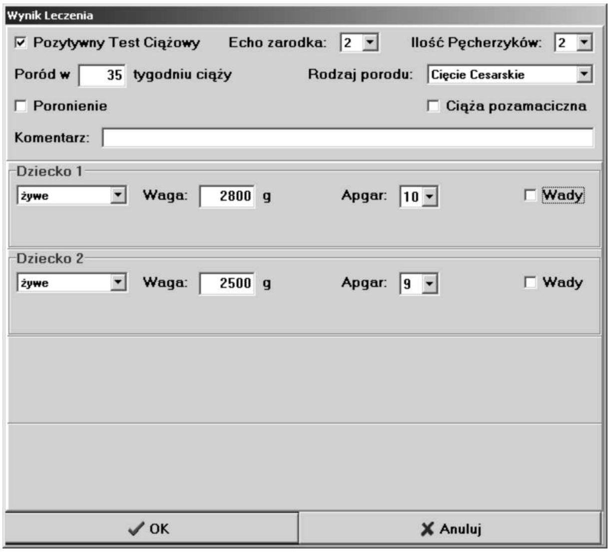


Figure 13. The results window

The final remarks

The created application is the result of co-operation of personnel who perform infertility treatment, as well as personnel who deal with programming. It is also the result of many years of co-operation and detailed consultations, permitting the whole group to acquaint with the subject matter – both with medical aspects and also with the possibilities of the programming environment in which the application was to be implemented.

The created system, as any large application is subject to continuous modification. In the first phase of the system's implementation many problems, both technical, as well as resulting from mistakes in the design stage, were found and improved. After achieving the stage of stability and undergoing tests performed by personnel, who didn't take a part in the design process, the system have been applied at the Clinic of Reproduction and Gynecological Endocrinology of Medical University of Bialystok. Data of few hundred of couples treated for infertility with the IVF ICSI/ET method has been accumulated up to the date of writing of this paper.

Significant improvement of accuracy and the reliability of entered data is one of first noticed effects of applying the system. It mainly is a consequence of the required filling in of some key fields in the database, which answered the recurrent questions of the lack in data. In comparison to gathering information using paper questionnaires such results would be impossible to accomplish.

The future plans

The created system has been implemented and is functioning properly but this is only the completion of the first stage of planned project. The second crucially important task will be the creation of the statistical module. This module should allow for quick calculation of descriptive statistics with the option of graphic presentation using charts, histograms, tables, and so on. It is intended that this application will enable to define and compare statistic parameters of a group of patients, as well as to calculate statistical significance of the differences. This functionality would simplify significantly the analysis of accumulated data and would also be of great help for the users of the system when taking a decision (for example during the choice of the protocol of the treatment).

The next step of this application's development is implementing it to work in a network, to allow multi-access to data. At present, the application is used on one selected computer. After adaptation to work in the network, many users from different locations could simultaneously add, edit and analyze the patients' data. This stage will enable a stable multi-access to data but also guarantee the safety of gathered information. Access by the Internet causes a whole range of threats to safety, and the confidentiality of recorded information requires putting special pressure on the security aspect.

After the completion of the third stage, it possible that the application will create a large and universal system which could become a standard in accumulation of information by entities performing infertility treatment using IVF methods.

References

[1] Piętka E. Zintegrowany system informacyjny w pracy szpitala. Wydawnictwo Naukowe PWN, Warszawa 2004.

[2] Kącki E., Kulikowski J.L., Nowakowski A., Waniewski E. Systemy komputerowe i teleinformatyczne w służbie zdrowia. Biocybernetyka i Inżynieria Biomedyczna, Akademicka Oficyna Wydawnicza EXIT, Warszawa 2003.

[3] Trąbka W. Szpitalne systemy informatyczne. Uniwersyteckie Wydawnictwo Medyczne Vesalius, Kraków 1999.
[4] Radwan J. Niepłodność i rozród wspomagany. Wydawnictwo Termedia, Poznań 2005.
[5] Boduch A. Delphi 2005. Kompendium programisty. Wydawnictwo Helion 2005.
[6] Wybrańczyk M. Delphi 7 i bazy danych. Wydawnictwo Helion 2003.
[7] Feddema H. Microsoft Access. Podręcznik administratora. Wydawnictwo Helion 2006.

Milewski Robert
Medical University of Bialystok
Department of Statistics and Medical Informatics
Mickiewicza 2c, 15–089 Białystok
milewski@umwb.edu.pl

Jamiołkowski Jacek
Medical University of Bialystok
Department of Public Health
I Armii Wojska Polskiego 2/2, 15–103 Bialystok
jacekj@umwb.edu.pl

Milewska Anna Justyna
Medical University of Bialystok
Department of Statistics and Medical Informatics
Mickiewicza 2c, 15–089 Białystok
milewska@umwb.edu.pl

Domitrz Jan
Medical University of Bialystok
Department of Reproduction and Gynecological Endocrinology
M. Sklodowskiej-Curie 24a, 15–276 Bialystok
jan@domitrz.pl

Wołczyński Sławomir
Medical University of Bialystok
Department of Reproduction and Gynecological Endocrinology
M. Sklodowskiej-Curie 24a, 15–276 Bialystok
wolczynski@sitech.pl