VERIFICA DI PROGRAMMI CONCORRENTI VPC 19-20 (Qualitative and quantitative verification of systems)

Prof.ssa Susanna Donatelli
Università di Torino
www.di.unito.it
susi@di.unito.it

Obiettivi

Obiettivo del corso e' di fornire gli strumenti, **teorici e pratici**, necessari alla **verifica dei sistemi**, ed in particolare del software. Per raggiungere tale obiettivo studieremo alcuni **paradigmi di base per la specifica** di processi distribuiti, focalizzando l'attenzione sulle capacità modellistiche e sugli strumenti di verifica di proprietà di buon comportamento.

Alla fine del corso lo studente sarà in grado di specificare sistemi concorrenti usando **linguaggi formali** e di utilizzare **strumenti software** per la verifica di proprieta' del sistema tramite verifica di proprietà del modello. Oltre alle classiche **proprietà** dei sistemi distribuiti quali assenza di deadlock, fairness e liveness, lo studente sarà in grado di definire e verificare proprietà in logica temporale quali ad esempio: "se il processo P manda un messaggio, allora non invierà il prossimo messaggio sino a che non riceve un acknowledgment", oppure "se il processo P manda un messaggio, riceverà un acknowledge entro 5 unità di tempo"..

Il corso **non ha come obiettivo** di fornire conoscenza su strumenti di verifica di linguaggi di programmazione, perché' non si focalizza sulla verifica dell' implementazione degli algoritmi, ma sulla verifica delle soluzioni algoritmiche legate alla concorrenza.
Nel ciclo di sviluppo del software le competenze acquisite in questo corso sono utili al fine della progettazione e verifica di soluzioni concorrenti/distribuite

Modalità di verifica dell' apprendimento

Combinazione di:

Piccoli esercizi e più consistenti progetti durante il corso, con precise date di consegna (pencil and paper and use of tools)

Gli esercizi devono essere accompagnati da una relazione (written report)

Verifica finale – discussione degli esercizi con uso dei tool e verifica della parte di teoria (dates can be agreed upon with the teacher, contact the teacher at least 10 days before)

Ci sono due modalita' per la valutazione

Orientata agli strumenti: sperimentazioni complete con gli strumenti di verifica. Discussione dei laboratori e una prova orale sugli argomenti principali

Orientata alla teoria: esercizi più semplici e più veloci e classico esame di teoria su tutti gli argomenti del corso

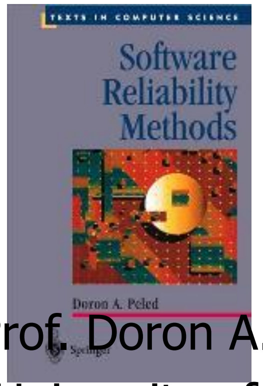
Argomenti per l’insegnamento da 6cfu

Non sono argomenti di esame dell’insegnamento da 6 cfu (e quindi non si devono portare i corrispondenti laboratori):

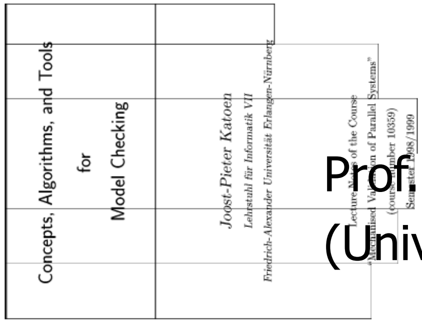
1. reti colorate e relativi esercizi

2. timed automata

Reference material books:



Prof. Doron A. Peled
(University of Warwick, UK)

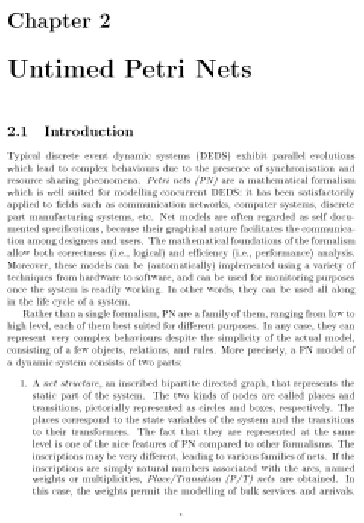

Concepts, Algorithms, and Tools for Model Checking

Joost-Pieter Katoen
Lehrstuhl für Informatik VII
Friedrich-Alexander Universität Erlangen-Nürnberg

Lecture Notes of the Course "Mechanised Validation of Parallel Systems"
(course number 10359)
Semester 1998/1999

Prof. Jost-Pieter Katoen
(University of Aachen, D)

Notes of the EU-sponsored Jaca MATCH school


Chapter 2

Untimed Petri Nets

2.1 Introduction

Typical discrete event dynamic systems (DEDS) exhibit parallel evolutions which lead to complex behaviours due to the presence of synchronisation and resource sharing phenomena. *Petri nets (PN)* are a mathematical formalism which is well suited for modelling concurrent DEDS: it has been satisfactorily applied to fields such as communication networks, computer systems, discrete part manufacturing systems, etc. Net models are often regarded as self documented specifications, because their graphical nature facilitates the communication among designers and users. The mathematical foundations of the formalism allow both correctness (i.e., logical) and efficiency (i.e., performance) analysis. Moreover, these models can be (automatically) implemented using a variety of techniques from hardware to software, and can be used for monitoring purposes once the system is readily working. In other words, they can be used all along in the life cycle of a system.

Rather than a single formalism, PN are a family of them, ranging from low to high level, each of them best suited for different purposes. In any case, they can represent very complex behaviours despite the simplicity of the actual model, consisting of a few objects, relations, and rules. More precisely, a PN model of a dynamic system consists of two parts:

1. A *net structure*, an inscribed bipartite directed graph, that represents the static part of the system. The two kinds of nodes are called places and transitions, pictorially represented as circles and boxes, respectively. The places correspond to the state variables of the system and the transitions to their transformers. The fact that they are represented at the same level is one of the nice features of PN compared to other formalisms. The inscriptions may be very different, leading to various families of nets. If the inscriptions are simply natural numbers associated with the arcs, named weights or multiplicities, *Place/Transition (P/T) nets* are obtained. In this case, the weights permit the modelling of bulk services and arrivals.

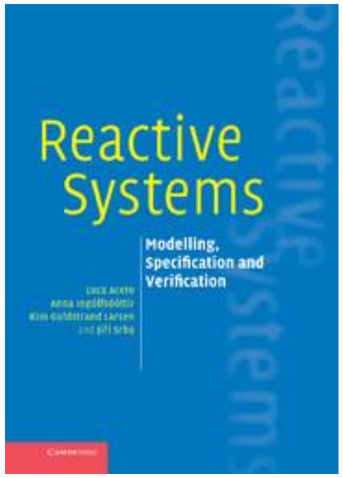
Reactive Systems
Modelling, Specification and Verification

Luca Aceto, Reykjavik University

Anna Ingólfsdóttir, Reykjavik University

Kim Guldstrand Larsen, Aalborg University, Denmark

Jiri Srba, Aalborg University, Denmark



Acknowledgements

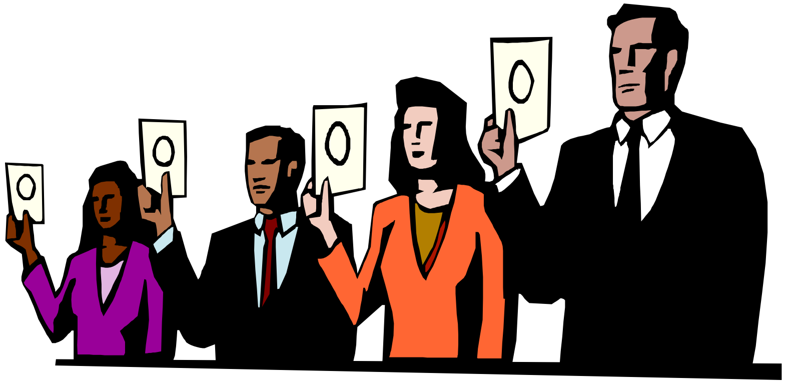
Transparencies adapted from the course notes and trasparencies of

- Prof. Doron A. Peled, University of Warwick (UK) and Bar Ilan University (Israel) http://www.dcs.warwick.ac.uk/~doron/srm.html
- Prof. Jost-Pieter Katoen, University of Aachen (Germany)
- Prof. Manuel Silva, Unievrsity of Zaragoza (Spain)
- Prof.ssa Giuliana Franceschinis, Università del Piemonte orientale (Italy)
- Dr. Jeremy Sproston, Università di Torino (Italy)
- Anders Moeller – BRICS University of Aurhus (DK)

Goal: system and software reliability

Use system and software engineering methodologies to develop the code.

<u>Use formal methods during code development</u>



Why system reliability?

System mulfunctioning can cause

- Loss of life (safety critical applications)
- Loss of money (mission critical applications)
- Loss of data/security/privacy

Why system reliability?

………and not validation and verification?

[Boehm’79]

- Verification: "Are we building the product right”?
- Validation: "Are we building the right product”?

Why should we care?

- NIST (National Institute of Standards and Technology) report
  - software bugs cost $60 billion annually
- High profile incidents of systems failure
  - Therac-25 radiation overdoses, 1985-87
  - Pentium FDIV bug, 1994
  - Northeast blackout, 2003
  - Air traffic control, LA airport, 2004
  - Chicago airport baggage delivery system
  - Survey of airline problems in http://www.wired.com/autopia/2013/04/airline-software-screw-ups/
  - American Airlines cancellation of 400 flights http://www.youtube.com/watch?v=W2o3d6QRL2k

The problem has been solved?

- Give a look at the abstract of the invited talk of Jeannete Wing from Microsoft at ACM SIGAda's Annual International Conference

Formal methods research has made tremendous progress since the 1980s when a proof using a theorem prover was worthy of a Ph.D. thesis and a bug in a VLSI textbook was found using a model checker. Now, with advances in theorem proving, model checking, satisfiability modulo theories (SMT) solvers, and program analysis, the engines of formal methods are more sophisticated and are applicable and scalable: to a wide range of domains, from biology to mathematics; to a wide range of systems, from asynchronous systems to spreadsheets; and for a wide range of properties, from security to program termination. In this talk, will present a few Microsoft Research stories of advances in formal methods and their application to Microsoft products and services. Formal methods use, however, is not routine—yet—in industrial practice. So, I will close with outstanding challenges and new directions for research in formal methods.

Target properties

Some properties can be easier than other to verify…..

Properties of Interest

Where does (semi-)**automated** analysis and verification have a chance?

- resource management (memory, files, allocation, locking)
- temporal properties (event ordering, concurrency, deadlocks, safety/liveness)
- datatype invariants (shapes, memory errors)
- security (integrity, confidentiality)
- numerical computations
- ...

Typical bugs? See e.g. bugzilla.mozilla.org or bugzilla.kernel.org

Verification vs. other programming techniques

Analysis

Debugging

Program optimization

Programming at design level

Verification vs. Analysis

- Verification: *checking* invariants
- Analysis: *detecting* invariants

  - this is just one possible definition of the words
  - in practical tools, there is a large overlap
  - analysis is not necessarily “harder” than verification

Relation to Debugging

- **Debugging**: you know there is a bug,
  but not exactly where it is
- **Verification**: you hope there are no bugs,
  but you want to be sure

- Analysis tools can be used to enhance program understanding (e.g. "program slicing")
- Verification tools can often provide counterexamples

Relation to Testing

"*Program testing can be used to show the presence of bugs, but never to show their absence.*" [Dijkstra, 1972]

Relation to Programming Language Design

- Programming at a higher level of abstraction can reduce the possibility of errors

- Examples:
  - type systems (note: types are invariants!)
  - abstract data-types
  - object oriented encapsulation and inheritance
  - domain-specific languages

What are formal methods?

Techniques for analyzing systems, based on some mathematics.

This does not mean that the user must be a mathematician.

Some of the work is done in an informal way, due to complexity.

Examples for FM

Deductive verification:

Using some logical formalism, ***prove*** using a theorm prover that the software satisfies its specification.

Model checking:

Use some software to automatically ***check*** that the software satisfies its specification.

Testing:

Check executions of the software according to some coverage scheme.

Formal methods' adoption:

- Boss: Mark, I want that the new internet marketing software will be flawless. OK?
- Mark: Hmmm. Well, ..., Aham, Oh! Ah??? Where do I start?
- Bob: I have just the solution for you. It would solve everything.

Why is software hard?

- The human element
  - Getting a consistent and complete set of requirements is difficult
  - Requirements often change
  - Human beings use software in ways never imagined by the designers

Why is software hard?

- The mathematical element
  - Huge set of behaviors
  - Nondeterminism
    - External due to inputs
    - Internal due to concurrency
  - Even if the requirements are unchanging, complete and formally specified, it is unfeasible to check all the behaviors

Badmouth

- Formal methods can only be used by mathematicians.
- The verification process is itself prone to errors, so why bother?
- Using formal methods will slow down the project.

Some answers…

Formal methods can only be used by mathematicians.

Wrong. They are based on some math but the user should not care.

The verification process is itself prone to errors, so why bother?

We opt to reduce the errors, not eliminate them.

Using formal methods will slow down the project.

Maybe it will speed it up, once errors are found earlier.

Some exaggerations

Automatic verification can only find errors.

Deductive verification can show that the software is completely safe.

Testing is the only industrial practical method.

"Program testing can be used to show the presence of bugs, but never to show their absence." [Dijkstra, 1972]

Some exaggeration

- Myth:
  - Think of the peace of mind you will have when the verifier finally says “`Verified`”, and you can relax in the mathematical certainty that no more errors exist
- Reality:
  - Use instead to find bugs (like more powerful type checkers)
  - We should change “Verified” to “Sorry, I can’t find more bugs”

Holy grail of algorithmic verification

Nei proof systems

- Soundness
  - si possono solo derivare affermazioni corrette
- Completeness
  - tutte le affermazioni corrette possono essere derivate

System Verification methods

- Light- weight
  - **unsound**
  - bug searching via simulation
  - simple properties, efficient and easy to use
- Medium- weight
  - **sound** but **incomplete**
  - analysis via fixed- point computation, type checking/ inference
- Heavy- weight
  - **sound** and **complete**
  - verification via theorem proving and user annotations
  - complex properties, resource demanding and difficult to use

Steps in the verification process

1. Check the kind of system to analyze.
2. Choose formalisms, methods and tools.
3. Express system properties.
4. Model the system.
5. Apply methods.
6. Obtain verification results.
7. Analyze results.
8. Identify errors.
9. Suggest correction.

Example of verification steps – java program verification

1. Check the kind of system to analyze.
2. Choose formalisms, methods and tools.
3. Express system properties. (square function of x does compute x*x)
4. Model the system. (add java modelling language specification)
5. Apply methods. (deductive verification using Why through Krakatoa)
6. Obtain verification results. (property is verified or not)
7. Analyze results.
8. Identify errors.
9. Suggest correction.

Example of verification steps – Mutual exclusion of Dekker

1. Check the kind of system to analyze.
2. Choose formalisms, methods and tools. (Petri nets and the GreatSPN tool)
3. Express system properties. (It is never the case that two processes can access the common resource at the same time)
4. Model the system. (build a Petri net specification of Dekker mutual exclusion)
5. Apply methods. (state space exploration – RG generation)
6. Obtain verification results. (property is verified or not, if not example of execution that violates the property)
7. Analyze results.
8. Identify errors.
9. Suggest correction.

Some concerns

- Which technique?
- Which tool?
- Which experts?
- What limitations?
- What methodology?
- At which points?
- How expensive?
- How many people?

- Needed expertise.
- Kind of training.
- Size limitations.
- Exhaustiveness.
- Reliability.
- Expressiveness
- Efficiency
- Support
- Degree of automation

Our course

Concentrate on distributed systems (as inherently protocols are)

Learn several formalisms to model system and properties (automata, process algebras, Petri Nets, temporal logic, timed automata).

Learn advantages and limitations, in order to choose the right methods and tools.

Learn how to combine existing formalisms and existing “solution” methods to pose and answer the right questions.

Emphasis

Analysis objective:

- Qualitative properties (e.g. absence of deadlock, liveness, fairness, "for each message sent an ack is received")
- Quantitative properties (e.g. "reach a deadlock/absorbing state no later than time 10", "for each msg sent an ack is received in within 5 ms", "for each msg sent an ack is received in within 5 ms, with 99% probability")

The process:

Selecting the tools, modeling, verification, locating errors.

Use of tools:

Hands on at least GreatSPN, nuSMV, Uppaal

Valutazione

Combinazione di:

- Piccoli esercizi e più consistenti progetti durante il corso, con precise date di consegna (pencil and paper and use of tools)
- Gli esercizi devono essere accompagnati da una relazione (written report)
- Verifica finale – discussione degli esercizi con uso dei tool e verifica della parte di teoria (dates can be agreed upon with the teacher, contact the teacher at least 10 days before)

Valutazione

Due modalita’ per la valutazione

- Orientata agli strumenti: sperimentazioni complete con gli strumenti di verifica e partecipazione alle attivita’ del model checking context a cui partecipa il nostro gruppo di ricerca. Solo la discussione dei laboratori e una prova orale sugli argomenti principali

- Orientata alla teoria: esercizi piu’ semplici e classico esame di teoria

Example topics

Project:

Check the web page for the exercises given in previous academic years

Oral exam:

1. Give the definition of a Petri net
2. Definition of deadlock in a Petri nets
3. Etc…

Prerequisiti – primo esercizio

- Imparare a fare e presentare definizioni di “ oggetti matematici”