ICAPS05

TU3

Tutorial on Directed Model Checking

Stefan Edelkamp
University of Dortmund, GERMANY

Tilman Mehler
University of Dortmund, GERMANY

Shahid Jabbar
University of Dortmund, GERMANY

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

CONFERENCE CO-CHAIRS:

Susanne Biundo
University of Ulm, GERMANY

Karen Myers
SRI International, USA

Kanna Rajan
NASA Ames Research Center, USA

Cover design: L.Castillo@decsai.ugr.es

Tutorial on Directed Model Checking

Stefan Edelkamp
University of Dortmund, GERMANY

Tilman Mehler
University of Dortmund, GERMANY

Shahid Jabbar
University of Dortmund, GERMANY

Tutorial on Directed Model Checking

Table of contents

Tutorial on Directed Model Checking

Preface

The tutorial is concerned with error-guided and counter-example driven exploration in model checking as an increasingly important technology used mainly in the area of software validation, but with a growing impact on the AI planning community.

Heuristic guided search techniques have first been proposed in the area of AI where they have been used quite successfully in solving complex planning and scheduling problems. On the other hand, Model Checking has evolved into one of the most successful verification techniques. Examples range from mainstream applications such as protocol validation, software model checking and embedded systems verification to exotic areas such as business workflow analysis, scheduler synthesis. Model checking technology has also been proven to be effective in automated test case generation.

The sheer size of the reachable state space of realistic models imposes tremendous challenges on the algorithmics of model checking technology. Complete exploration of the state space is often unfeasible and approximations are needed. Also, the error trails reported by depth-first search model checkers are often exceedingly lengthy – in many cases they consist of mutliple thousands of computation steps which greatly hampers error interpretation. In the meantime, state space exploration is a central aspect of AI planning, and the AI planning community has a long and impressive line of research in developing and improving search algorithms over very large state spaces under a broad range of assumptions.

We study directed search in explicit state model checking. evaluate the combination of heuristic search with bit-state hashing, and partial-order reduction, just to name a few. We discuss approaches to symbolic model checking by studying BDD versions of traditional AI exploration algorithms. One important focus is the exploration with abstraction databases. While some approaches integrate AI planning methods into a model checker, we will also study the possibility of using planners to perform model checking directly. Simple protocols have been successfully tested as benchmarks in international planning competitions.

The term "Directed Model Checking"was coined by an paper on the application of heuristic search in the symbolic model checker checker mucke. Earlier work mainly included best-first search-like search strategies for accelerating the search for concrete type of errors in the model cheker Murphi. Since then the interest on applying AI-based search techniques grew and was applied in different domains: Java and C++ programs, real-time systems, hardware circuits. In some cases ressearch produced or extended verification tools with heuristic search capabilities. As a case study we will introduce to the HSF-SPIN tool.

The goal of the tutorial is to bring together research interests in formal automatic analysis and in guided search. Intended participants are researchers that are involved in the development and analysis of systems with large state spaces. The tutorial will discuss in the algorithmic foundations as well as in the implementation of directed explicit and symbolic model checking. The main focus is on the analysis of concurrent software systems. Although seminar focus is to attract researchers, we invite people from industry to participate in the seminar, as industrial interest in directed model checking (either symbolic or explicit) is rising. Prior intimate knowledge of automata based explicit state or symbolic model checking is not required.

The presentation slides cover most of the tutorial content. The print divides into units of a few pages followed by a list of references. The sections are almost chronologically sorted, with the exception that directed search is placed to the end of this notes, as it may be optional for many attendees. The slides were produced mainly by Stefan Edelkamp, while my PhD students Tilman Mehler and Shahid Jabbar were responsible for additional material and for preparing example cases. Many thanks to Alberto Lluch-Lafuente, who helped assembling the slides. He also came up with many of the ideas that are presented in this tutorial. Further thanks go to Stefan Leue, who came up with the idea of having a tutorial on this topic.

We all personally hope that you will enjoy the tutorial and invite all attendees to join this research branch. As a further side remark there will be a Dagstuhl seminar with this topic organized by Willem Visser, Stefan Leue, Alberto Lluch-Lafuente, and Stefan Edelkamp in April 2006. We hope that we might see one or the other participant of this tutorial there.

Instructors

- *Stefan Edelkamp, University of Dortmund, GERMANY*
- *Tilman Mehler, University of Dortmund, GERMANY*
- *Shahid Jabbar, University of Dortmund, GERMANY*

Directed Model Checking

– Motivation –

Stefan Edelkamp

1 Motivation

Critical Role of Software: medicine, aviation, finance, transportation, space technology and communication

Software Failure: financial and commercial disaster, human suffering and fatalities

Verifiable development: Software industry not yet ready for emerging standards

Research Aims:

- Develop intelligent, heuristically-*directed model checking* algorithms and implement these algorithms in formally-based tools
- Lay foundation of a technique that will enable software engineers to design and debug complex systems
- Verify that these systems behave predictably and correctly

Verification Engineering

Systems are harder to verify than in earlier days

Design groups spend 50-70% of the design time on verification

Verification Engineering

- studies techniques to verify that a system is correct
- rapidly expanding as society's expectation of system infallibility and reliance on computer systems and communication devices increases

pan-European survey (Davy 2001) of electronic companies:

- verification is the single biggest problem in electronics design today
- incomplete verification of design by far greatest single cause of severe bugs

IBM Research Center

"It is widely recognized that functional verification emerges as the bottleneck of the design development cycle

This is due to a combination of several correlated factors:

1. exponential increase in design complexity
2. tighter time-to-market requirements
3. higher quality expectations

In parallel, verification means are not evolving at a matching pace.

The cost of the late discovery of buts is enormous, justifying the fact that, for a typical microprocessor design project, up to half of the overall resources spent, are devoted to its verification. "

Model Checking

Traditional verification line *simulation* and *testing* inadequate

- hit-and-miss techniques, relying on the appropriate input being chosen that will reveal errors in the system
- the greater the complexity, the less effective the techniques.
- simulation takes a week to verify a 500,000 gate chip design, while formal verification can accomplish this task in 1/2 hour

Model checking is a cornerstone of verification engineering.

Amir Pneuli - 1996 ACM Turing Award Recipient

Lightweight vs. Heavy-Duty Tools

Model Checking (MC):

“Automatic method for proving that an application satisfies its specification, represented by a temporal-logic formula.”

MC is *lightweight* (Heitmeyer 1998)

Contrast: *heavy-duty* tool e.g. *proof assistants* require skilled use, which is a barrier to their adoption by industry

Major hindrance to the adoption of formal methods: lack of work-force skills

Research and Development of MC in Companies

Using or evaluating MC: IBM, Intel, Lucent, Telenokia, BMW, and Siemens

MC included Computer Aided Design (CAD) toolkits:

- *CVE* (Siemens)
- *Telelogic Tau* (Telelogic)
- *Design Insight* (Chrysalis Symbolic Design)
- *NP-Tools* (Prover Technology)
- *RuleBase* (IBM Research Lab)
- *FormalPro* (Mentor Graphics)

Affirma FormalCheck (originally Lucent, now Cadence Design Techniques) costs e.g. US$ 95,000

Most popular developed-in-academia model checkers in the public-domain: *SPIN* (Holzman), *SMV* (McMillan)

Working and Limits of Traditional MC

Working: evaluate the given property in every reachable state of the model, where each state is stored explicitly

Problem: state space grows exponentially

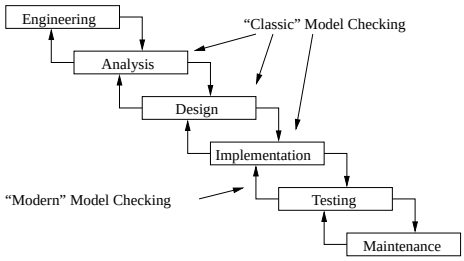
Limits: simple protocols, circuits and algorithms

On-the-Fly Model Checkers:

1. verify states as they are reached
2. still exhaustive
3. virtually all specifications contain errors at least initially
4. perform in general better during system development

Waterfall Model

The waterfall life cycle model:



Symbolic Model Checking

. . . can reduce the size of the state space very dramatically by verifying sets of states

. . . use Binary Decision Diagrams to represent the state space.

Example: SMV, can verify specifications with up to 10^{1300} reachable states (Clarke, Grumberg, Long 1992)

Directed Model Checking

Traditional explicit and symbolic model-checking research invloves an exhaustive search of the state space

Notable exceptions:

- BDD-version of the A* algorithm: replace BFS search with heuristic search (Edelkamp and Reffel 1998)
- Target enlargement heuristic used in chip design (Yang 1998)

These research lines form the starting point for this lecture

Model Checking as a Debugger

Dilemma:

- Applying (traditional) model checking too early is inefficient
  Typical runs are in the order of 15 min to overnight
- Applying (traditional) model checking too late is expensive

The later an error is discovered the more difficult it is to remove

The error may require a substantial part of the design to be redone

This research fills a void;

Serious and usable tool for software developers at the time they need it most: when designing a complex systems

Directed Model Checking

. . . novel and different in a number of important ways

- hones in on errors
- can be used early in the software live cycle
- can handle an incomplete description of the system because it searches in just a portion of the state space
- avoids the problem of *state explosion*

Adapt heuristics to model checking enables model checker to be used

Cross discipline: Formal methods and AI technologies are integrated

Likely to be direction of future research (Buccafurri, Eiter, Gottlob and Leone 1997)

References

[1] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing symbolic model checking by AI thechniques. Technical Report 9701, IFIG, 1997.

[2] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[3] P. Davy. What's wrong with design. Hyperactive Eldectronics Weekly Online, 2001.

[4] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *KI: Advances in Artificial Intelligence*, LNAI, pages 81–92. Springer, 1998.

[5] C. Heitmeyer. On the need for practical formal methods. In A. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS, pages 18–26. Springer, 1998.

[6] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[7] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[8] C. H. Yang. *Prioritized Model Checking*. PhD thesis, Stanford University, 1998.

Directed Model Checking

– Model Checking –

Stefan Edelkamp

1 Overview

- Design Phases
- Finite State Automata over Infinite Words
- Communicating Processes
- Labeled Transition Systems
- Composition of Processes
- Temporal Logics, in particular LTL

2 Design Phases

System designers and analysts have different expectations at different stages of the system development process (Cobleigh et al. 2001)

Exploratory Mode: Usually applicable to earlier stages of the process, where errors in the design are expected, one wishes to find errors as fast as possible.

Fault-finding Mode: One expects to obtain meaningful error trails in order to fix the design error, while one is willing to tolerate somewhat longer execution times.

Correctness Mode: In a third mode, one is interested in checking the correctness of the design.

Example: Plain Old Telephony System

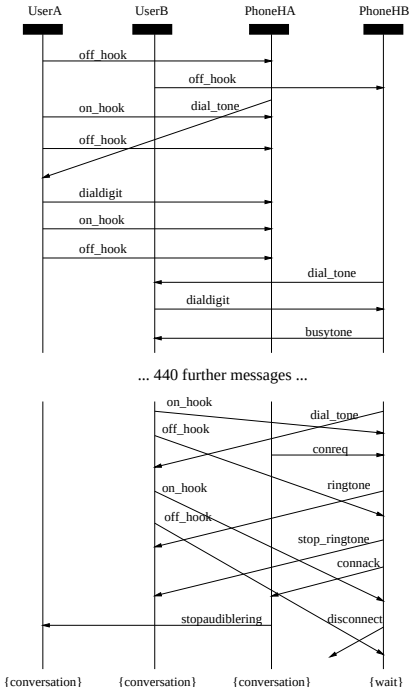
Model POTS (Kamel, Leue) : generated with the visual modeling tool VIP.

- “first cut” implementation of a simple two-party call processing
- two user processes `UserA` and `UserB` representing the environment behavior of the switch
- two phone handler processes `PhoneHA` and `PhoneHB` representing the software instances that control the internal operation of the switch according to signals (on-hook, off-hook, etc.) received from the environment
- full of faults of various kinds

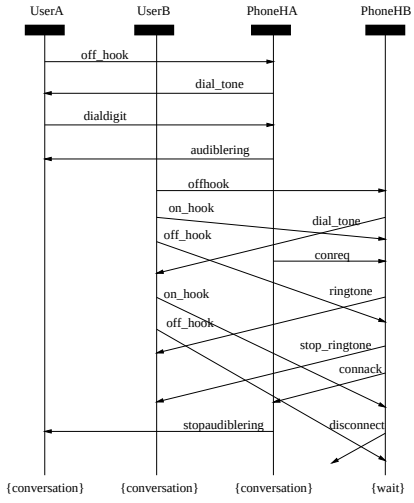
SPIN model checker: model is actually capable of connecting two telephones.

Error: invariant violation

Error Trail Produced by SPIN



Manually Generated Error Trail in POTS

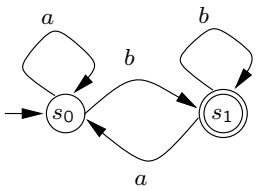


3 Finite Automata

Finite-state automaton: tuple $\langle S, \Sigma, T, S_0, F \rangle$ where:

- S is a finite set of states;
- Σ is a finite alphabet;
- $T \subseteq S \times \Sigma \times S$ is the transition relation;
- $S_0 \subseteq S$ is the set of initial states;
- $F \subseteq S$ is the set of final states.

A simple finite automaton represented graphically:



Runs

Run of an automaton over an finite word $v \in \Sigma^*$: an alternating sequence of states and letters $s_0 \xrightarrow{v_1} \ldots \xrightarrow{v_n} s_n$ such that

- the first state is an initial state, that is, $s_0 \in S_0$, and
- moving from the i-th state s_i to the $(i+1)$-st state s_{i+1} upon reading the i-th input letter v_i is consistent with the transition relation, that is, for all $0 \leq i \leq n$ we have $(s_i, v_{i+1}, s_{i+1}) \in T$.

Run ρ *over a word* v *accepting*: *ends* in a final state

$\mathcal{A}$ *accepts word* v: exists an accepting run of $\mathcal{A}$ over v.

Language of the automaton $\mathcal{L}(\mathcal{A})$*:* set of all accepted words by an automaton $\mathcal{A}$

4 Finite Automata over Infinite Words

Concurrent systems often do not terminate $\Rightarrow$ automata over infinite words

Büchi Automata same syntax as finite automata, but their semantic is different

F: set of *accepting* states instead of final states
inf (ρ): set of states that appear infinitely often in a run ρ.

Run ρ of a Büchi automaton $\mathcal{B}$ over an infinite word accepting: some accepting state appears infinitely often in ρ, that is, when *inf* $(\rho) \cap F \neq \emptyset$.

Language $\mathcal{L}(B)$ accepted by the Büchi automaton $\mathcal{B}$: set of infinite words, over which all runs of $\mathcal{B}$ are accepting.

Example interpreted as a Büchi automaton: set of all infinite words in which a letter b appears infinitely often.

5 Communicating Processes

A finite process: tuple $\langle S, E, T, S_0, F, V \rangle$ where:

- S is a finite set of states;
- E is a finite set of transition labels;
- $T \subseteq S \times E \times S$ is the transition relation;
- $S_0 \subseteq S$ is the set of initial states;
- $F \subseteq S$ is the set of final states;
- V is a finite set of variables.

Variables: current state of process, numbers, message channels (FIFO queues)

Guards, Propositions and Predicates

Guards: boolean predicates over the set of variables that determine the *executability* or *enabledness* of a transition in concrete system states.

Propositions and Predicates: Predicates over queues q: *full*(q), *empty*(q)

Other Predicates over Message Channels: *head*(q), *capacity*(q) and *length*(q)

AP: atomic boolean predicates over V; consists of all possible boolean constants, variables, relations between numerical expressions, and boolean queue predicates

Boolean predicates *BP*: If p is an atomic boolean proposition of *AP* then p is also in *BP*. If p and q are predicates of *BP* so are $\neg p, p \vee q, p \wedge q, p \rightarrow q$ and $p \leftrightarrow q$.

Every guard is a boolean predicate contained in *BP*

Executability

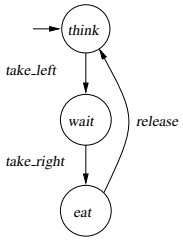
Asynchronous Receive ($\mathtt{q?x}$*)* extract message from channel q and assigns it to the variables in the tuple x; corresponding transition is not executable if the message channel is empty, i.e. *guard*$(\mathtt{q?x}) \equiv \neg$*empty*$(q)$.

Asynchronous Send ($\mathtt{q!m}$*)*: insert a message m in channel q; not executable if the channel is full, i.e. *guard*$(\mathtt{q!m}) \equiv \neg$*full*$(q)$.

Assignments ($v \leftarrow r$): change the value of a variable v, setting it to the new value r; always executable

Finite Process for a Dining Philosopher

:



- $S = \{\mathit{eat}, \mathit{wait}, \mathit{think}\}$
- $E = \{\mathit{take\_left}, \mathit{take\_right}, \mathit{release}\}$
- $T = \{(\mathit{think,take\_left,wait}), (\mathit{wait,take\_right,eat}), (\mathit{eat,release,think})\}$
- $S_0 = \{\mathit{think}\}$
- $F = \emptyset$

Labeled Transition System

. . . global state transition graph, unfolding of above systems

A *labeled finite transition system* is a tuple $\langle S, S_0, T, \mathit{AP}, L\rangle$ where:

- S is a finite set of states;
- S_0 is the set of initial states;
- T is a finite set of transitions, where each transition $t \in T$ is a partial function $t : S \to S$;
- AP is a finite set of atomic propositions;
- L is a labeling function $S \to 2^{\mathit{AP}}$.

t with $t(s) = s'$ in $T \iff (pc(s), t, pc(s'))$ in set of transitions of the process being unfolded

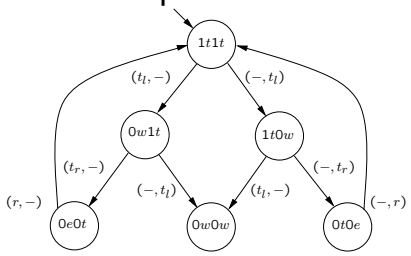
Asynchronous Composition of Processes

The asynchronous composition of n finite processes $\mathcal{P}_i = \langle S_i, E_i, T_i, S_0^i, F_i, V_i \rangle$, $1 \leq i \leq n$ is a finite process $\mathcal{P} = \langle S, E, T, S_0, F, V \rangle$ such that:

- $S = \prod_{i=1}^{n} S_i$; $E = \prod_{i=1}^{n} (E_i \cup {}'-')$;
- $S_0 = \prod_{i=1}^{n} S_0^i$;
- $F = \prod_{i=1}^{n} F_i$; $V = \bigcup_{i=1}^{n} V_i$;
- $T = \{((s_1, .., s_n), (e_1, .., e_n), (s'_1, .., s'_n)) \mid \exists j : 1 \leq j \leq n \wedge e_j \in E_j \wedge (s_j, e_j, s'_j) \in T_j \wedge \forall_{1 \leq i \leq n, i \neq j} : s_i = s'_i \wedge e_i = {}' - '\}$.

Asynchronous Composition of Philosophers

Convenient: n variables f_i with $1 \leq i \leq n$ for the forks $\Rightarrow$ state space of the asynchronous composition of two such processes:



t, w, e, t_r, and t_l are the resp. abbrev. for *think*, *wait*, *eat*, *take_left* and *take_right*.

The labeling of a state is the concatenation of the values of variables f_0, pc_0, f_1 and pc_1, where boolean values *true* and *false* are respectively represented by 1 and 0.

Synchronous Composition of Processes

The synchronous composition of n finite processes $\mathcal{P}_i = \langle S_i, E_i, T_i, S_0^i, F_i, V_i \rangle$, $1 \leq i \leq n$ is a finite process $\mathcal{P} = \langle S, E, T, S_0, F, V \rangle$ such that:

- $S = \prod_{i=1}^{n} S_i$; $E = \prod_{i=1}^{n} E_i$;
- $S_0 = \prod_{i=1}^{n} S_0^i$;
- $F = \prod_{i=1}^{n} F_i$; $V = \bigcup_{i=1}^{n} V_i$;
- $T = \{((s_1, .., s_n), (e_1, ..e_n), (s'_1, .., s'_n)) \mid \forall_{i=1..n} : e_i \in E_i \wedge (s_i, e_i, s'_i) \in T_i\}$.

6 Temporal Logics

Reasoning about programs: sufficient to concentrate on in/output requirements.

Concurrent systems usually don't terminate $\Rightarrow$ reasoning about ordering of events.

TL are a sort of modal logic specifically tailored to specify temporal requirements.

TL offer a way for specifying ordering of events without using time explicitly.

Temporal properties such as "an event p must must always hold" or "an event p is always followed by an event q" can be expressed with these logics.

Several temporal logic formalisms exist: μ-calculus, CTL*, CTL and LTL, ...
The more expressive a logic is the harder it is to check it.

Linear-time Temporal Logic

LTL formulae; describe properties of runs of a system, extend boolean logic with the following set of temporal operators

*Next Time (**X**f):* specifies that property f must hold in the *next* state of the run.

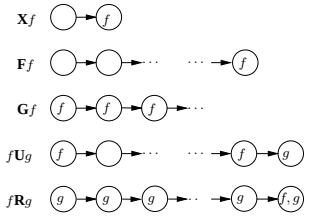
*Eventually (**F**f):* requires that property f must hold in some *future* state of the run.

*Always (**G**f):* used to express that property f must hold in *every* state of the run.

*Until (f**U**g):* requires that property g holds in *some* state of the run, and at *every preceding* state property f must hold.

*Release (f**R**g):* specifies that property f holds up to and includes the first state where property g holds, which is not required to occur.

Illustration



A possible run is depicted which satisfies the temporal formula.

$$r = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$$

$$r_i = s_i \xrightarrow{\alpha_i} s_{i+1} \xrightarrow{\alpha_{i+1}} \dots$$

Model Checking Problem: Given model $\mathcal{M}$ and LTL formula f check $\mathcal{M}, s \models f$ for every initial state s

LTL Semantics in Labeled Transition Systems

- $\mathcal{M}, s \models p \Leftrightarrow p \in L(s)$.
- $\mathcal{M}, s \models \neg f \Leftrightarrow \mathcal{M}, s \not\models f$.
- $\mathcal{M}, s \models f \vee g \Leftrightarrow \mathcal{M}, s \models f$ or $\mathcal{M}, s \models g$.
- $\mathcal{M}, s \models \mathbf{X}f \Leftrightarrow$ for every run r starting at s we have $\mathcal{M}, r_1 \models f$.
- $\mathcal{M}, s \models \mathbf{F}f \Leftrightarrow$ for every run r starting at s there exists an $i \geq 0$ such that $\mathcal{M}, r_i \models f$.
- $\mathcal{M}, s \models \mathbf{G}f \Leftrightarrow$ for every run r starting at s and for all $i \geq 0$ we have $\mathcal{M}, r_i \models f$.
- $\mathcal{M}, s \models f\mathbf{U}g \Leftrightarrow$ for every run r starting at s, there exists an $i \geq 0$ such that $\mathcal{M}, r_i \models g$ and for all $0 \leq j < i$ we have $\mathcal{M}, r_j \models f$.
- $\mathcal{M}, s \models f\mathbf{R}g \Leftrightarrow$ for every run r starting at s and for all $j \geq 0$, if for every $i < j$ we have $\mathcal{M}, r_i \not\models f$ implies $\mathcal{M}, r_j \models g$.

Typical Specification Properties

Safety: Informally, safety properties express that, under certain conditions, an event will never occur.

Invariant: LTL formula $\mathbf{G}p$ expressing that p must hold in every state of the system

Example: Invariant $\mathbf{G}(pc_i = eat \rightarrow \neg f_i \wedge \neg f_{i \oplus 1})$

What does it mean?

Does it hold?

Assertion: concrete case of invariance in which the the predicate is the conjunct of two predicates: $pc_i = s \wedge p$ for given i, s, and q

Liveness

Informally, liveness properties express that, under certain conditions, some event will ultimately occur.

Example: **GF**$(pc_i = eat)$ for the $i-th$ philosopher

What does it mean?

Does it hold?

Response: Most frequently used, **G**$p \rightarrow$ **F** q

Deadlocks: *Deadlock absence* is a temporal property that requires that there exists no state from which no progress is possible.

Deadlock absence can be expressed in LTL with the formula **GX***true*.

References

[1] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Conference on Logic, Methodology, and Philosophy of Science*, LNCS, pages 1–11. Stanfort University Press, 1962.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[3] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *International Conference on Software Engineering*, pages 37–46, 2001.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.

[5] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[6] M. Kamel and S. Leue. VIP: A visual editor and compiler for v-Promela. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 471–486. Springer, 2000.

Directed Model Checking

– Automata-Based MC –

Stefan Edelkamp

1 Overview

- Model Checking Procedure
- Intersection of Büchi Automata
- Checking Emptiness
- Nested Depth-First Search
- Snapshot and Correctness
- Improved Nested DFS

2 Automata-Based Model Checking

. . . common approach to the model checking problem (not LTL specific)

Transforming model and specification into Büchi automata:

- systems can be modeled by finite automata
- model automaton transformed into Büchi by considering all states as accepting
- (L)TL formulae can also be transformed into an equivalent Büchi automaton
- (contrary not always possible, since Büchi automata more expressive than LTL)
- combine automata to one that accepts the *bad* behaviors of the system
- checking correctness is reduced to checking emptiness of this automaton

Counterexamples; infinite accepting runs which are called accepting cycles.

Model Checking Procedure

Checking that a model represented by an automaton $\mathcal{M}$ satisfies its specification represented by an automaton $\mathcal{S}$:

verify if $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$

- “language accepted by the model is included in that of the specification”

- “set of behaviors of the model is accepted by the specification”

$$\overline{\mathcal{L}(\mathcal{S})} = \Sigma^{\omega} - \mathcal{L}(\mathcal{S}) \Rightarrow \mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S}) \iff \mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$$

- “model has no *bad* behavior”

Checking emptiness is in practice more efficient than checking inclusion.

Remarks

Büchi automata are closed under intersection and complementation $\Rightarrow$

- $\exists$ automaton that accepts $\overline{\mathcal{L}(\mathcal{S})}$

- $\exists$ automata that accepts $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})}$.

Theory:

- possible to complement Büchi automaton equivalent to LTL formula

but double-exponential in the size formulae (Safra 1998, Sistla 1983)

Practice: construct Büchi automaton for negation of LTL formula, avoiding complementation

Computing the Intersection

Automaton that recognizes language of the intersection of a

- Büchi automaton $\mathcal{M}$ representing the model
$\mathcal{M} = \langle \Sigma, S_{\mathcal{M}}, \Delta_{\mathcal{M}}, S_0^{\mathcal{M}}, F_{\mathcal{M}} \rangle$ with $S_{\mathcal{M}} = F_{\mathcal{M}}$

- Büchi automaton $\mathcal{N}$ representing the negation of the specification
$\mathcal{N} = \langle \Sigma, S_{\mathcal{N}}, \Delta_{\mathcal{N}}, S_0^{\mathcal{N}}, F_{\mathcal{N}} \rangle$

$\Rightarrow \mathcal{M} \cap \mathcal{N}$ of $\mathcal{M}$ and $\mathcal{N}$ is a Büchi automaton $\langle S, \Sigma, \Delta, S_0, F \rangle$, where:

- $S = S_{\mathcal{M}} \times S_{\mathcal{N}}$; $S_0 = S_0^{\mathcal{M}} \times S_0^{\mathcal{N}}$;
- $F = F_{\mathcal{M}} \times F_{\mathcal{N}} = S_{\mathcal{M}} \times F_{\mathcal{N}}$;
- $\Delta = \{((s_{\mathcal{M}}, s_{\mathcal{N}}), a, (s'_{\mathcal{M}}, s'_{\mathcal{N}})) \mid s_{\mathcal{M}}, s'_{\mathcal{M}} \in S_{\mathcal{M}} \wedge s_{\mathcal{N}}, s'_{\mathcal{N}} \in S_{\mathcal{N}} \wedge (s_{\mathcal{M}}, a, s'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \wedge (s_{\mathcal{N}}, a, s'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}\}$

On-the-Fly Model Checking

. . . efficient way to perform model checking

. . . compute the global state transition graph during the construction of the intersection.

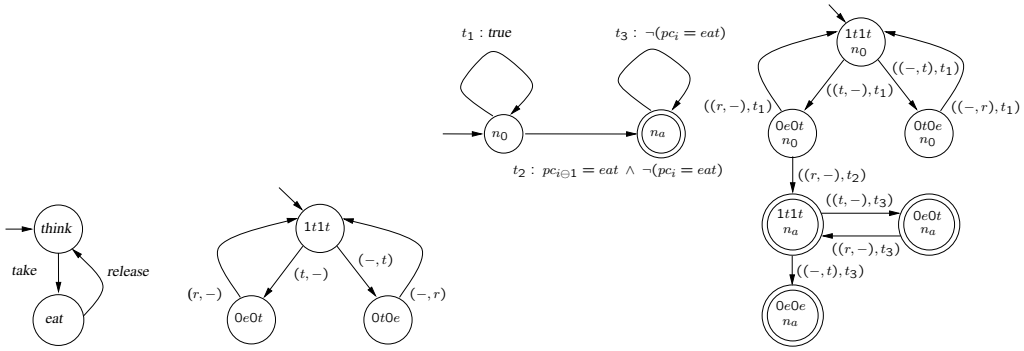
Advantage: only that part of the state space is constructed that is needed in order to check the desired property

Evident for error detection: not necessary to continue generating states

Example: Envy

LTL Property: **G**$((pc_{i\ominus 1} = eat) \rightarrow$ **F**$(pc_i = eat))$,

Model, Composition, Never claim and state space of intersection:



Infinite acc. run: $(1t1t, n_0) \xrightarrow{((t,-),t_1)} (0e0t, n_0) \xrightarrow{((r,-),t_2)} (1t1t, n_a) \xrightarrow{((t,-),t_3)} (0e0t, n_a) \xrightarrow{((r,-),t_3)} (1t1t, n_a) \Rightarrow$ language $\neq \emptyset$, property does not hold

Checking Emptiness

Checking emptiness: check that automaton accepts no word

Accepting runs: $\sim$ strongly connected components (SCC),
reachable from initial state and contain at least one accepting state

$\sim$ reachable cycle containing at least one accepting state

$\Rightarrow$ checking emptiness $\iff$ exists no such cycle

All SSCs of a graph: Tarjan's algorithm

Accepting cycles: more efficiently by *nested depth-first search* (Holzmann)

Nested Depth-First Search

- explores the state space in a depth-first manner
- stores visited states in *Visited* list
- marks states which are on the current search stack
- invokes *DFS2* for accepting states in postorder

procedure *DFS1*(s);
    Stack.push(s); *Visited.insert*(s);
    for each successor s' of s **do**
        if $s' \notin$ *Visited* **then** *DFS1*(s');
    if *accepting*(s) **then** *DFS2*(s);
    Stack.pop();

DFS2

- explores states already visited by DFS1 but not by any secondary search
- states visited by the second search are *flagged*
- if a state is found on the stack of first search, accepting cycle is found

procedure *DFS2*(s);
    flagged(s) $\leftarrow$ *true*;
    for each successor s' of s **do**
        if $s' \in$ *Stack* **then return solution**;
        if not *flagged*(s') **then** *DFS2*(s');

Typical implementation: 2 bits per state, one for marking, one for flagging

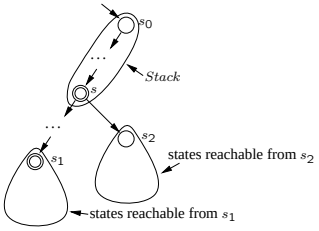
Complexity: linear in the size of the intersected state transition graph

Snapshot of the Nested DFS

All successors of s have been explored by DFS1 $\Rightarrow$ DFS2 started, since s is accepting (assume s not flagged by second search).

s_2 successor of s and all reachable states through s_2 non-accepting $\Rightarrow$ DFS2 will explore s_2

Accepting state s_1 reachable from $s \Rightarrow$ since s_2 was explored by DFS1 after $s \Rightarrow$ second search has been started from it and all reachable states from it have been flagged $\Rightarrow$ DFS2 started at s does not explore s_1



Correctness Nested DFS

Lemma q node not on any cycle $\Rightarrow$ DFS backtracks from q only after all nodes reachable from q are backtracked from.

Theorem: Nested DFS returns counterex. for emptiness of $B \iff L(B) = \emptyset$

Proof: Counterexample for state and acc. cycle $\Rightarrow L(B) \neq \emptyset$

Automata reports emptiness $\stackrel{!}{\Rightarrow} L(B) = \emptyset$

Flagged states all reached by DFS1; DFS2 starts at q and $\exists$ path to p on DFS1 stack $\stackrel{!}{\Rightarrow}$ to be completed to cycle

$\stackrel{!}{\Rightarrow} \exists$ unflagged path from q to state in DFS1 stack

Proof ctd.

Assume $\forall$ path to state on DFS1 stack $\exists$ flagged state

q: first such state, r first flagged state reached from q
q': acc. state that invokes DFS2, in which r is encountered

$\Rightarrow$ DFS2 starts from q' earlier than from q

q' reachable from q: $\exists$ cycle $(q', \ldots, r, \ldots, q, \ldots, q')$ not been found previously in contradiction to q first state from which DFS2 fails

q' not reachable from q: either q' appears on a cycle such that q' reachable from q via r (contradiction to chosen case) - or -

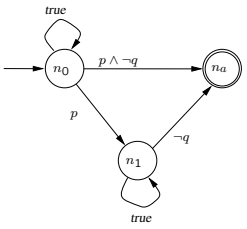
q' appears not on a cycle, s.t. backtrack from q earlier than backtrack from q' and DFS2 started from q prior to start from q (contradiction to ordering of DFS2)

Special Liveness Properties

Some liveness properties can be checked by a simple rechability algorithm (DFS).

$\Rightarrow$ nested DFS needed only when properties which never claim automaton contains at least one SCC

Interpretation depends on the translation of LTL into Büchi automata



Never claim automaton for LTL formula $\mathbf{G}(p \rightarrow \mathbf{G}q)$

System state violates property if there is a reachable state in the intersection such that the never claim automaton is in its local accepting end state n_a

3 Improved Nested DFS

Observation: Cycle in the state transition graph of the intersection of the system $\mathcal{M}$ and the never claim automaton $\mathcal{N}$ is accepting if and only if the corresponding cycle in $\mathcal{N}$ is accepting.

Projection: π maps global state to state in never claim

SCC *non-accepting*: none of its states is accepting, *full-accepting*; each cycle formed by states of the SCC is accepting, *partial-accepting*: otherwise.

Idea: Partitioning the never claim into SCCs, apply DFS2 only in case of partially accepting cycles

Analyze never claim beforehand.

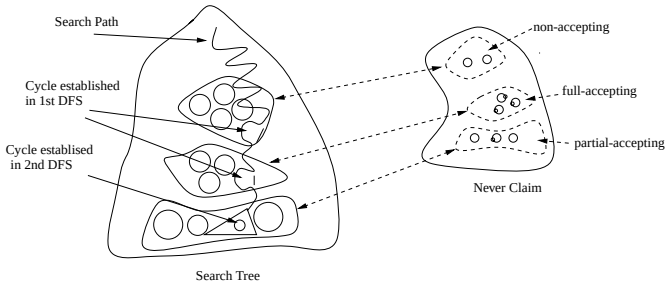
Pseudo-Code

Procedure *INDFS1*(s);
  Stack.push(s); *Visited.insert(s)*
  if $s \in Stack$ and *full-acepting*$(\pi(s))$ **then** **return solution**;
  for each successor s' of s **do**
    if $s' \notin$ *Visited* **then** *INDFS1*(s');
  if *accepting(s)* **and** *partial-accepting*$(\pi(s))$ **then** *INDFS2*(s);
  Stack.pop$()$;

procedure *INDFS2*(s);
  $flagged(s) \leftarrow true$;
  for each successor s' of s **do**
    if $s' \in$ *Stack* then **return solution**;
    if (not *flagged*(s)**)** **and** $\pi(s) = \pi(s')$ **then** *INDFS2*(s');

Example

Different Cases in Improved-Nested-DFS



References

[1] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Conference on Logic, Methodology, and Philosophy of Science*, LNCS, pages 1–11. Stanfort University Press, 1962.

[2] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Symposium Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[3] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *The SPIN Verification System*, pages 23–32, 1972.

[4] S. Safra. On the complexity of omega-automata. In *Annual Symposium on Foundations of Computer Science*, pages 319–237. IEEE Computer Society, 1998.

[5] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1983.

[6] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

Directed Model Checking

– Heuristics –

Stefan Edelkamp

1 Overview

- Error Detection
- Unknown Error States
  - Invariants
  - Deadlock
  - Formula-based Estimate
- Known Error States
  - Hamming Distance
  - FSM Distance
- Trail Improvement

2 (Safety) Error Detection

Errors: Boolean predicate *error*(s) of the set *BP* saying state s violates the property or not

System to be checked: asynchronous composition of n finite communicating process $\mathcal{P}_i$, with $i = 1, \ldots, n$.

$\mathcal{P}_i = \langle S_i, E_i, T_i, S_0^i, F_i, V_i \rangle$: component process

$\mathcal{S} = \langle S, S_0, T, AP, L \rangle$: unfolding of the asynchronous composition if the $\mathcal{P}_i$s

Two kinds of heuristics:

- no error state has been found yet
- exploit information of a given error state

3 Heuristics for Unknown Error States

Problem: define a function that estimates the distance from a given state to the nearest error state, without knowing in advance if there exist such a state at all

Solution: property itself can be utilized to define a boolean predicate that characterizes error states, and as basis for an estimate function

Special-proposed Heuristics: Deadlock detection

Formula-based Heuristic: exploits structure of property, given a state s and predicate f, returns how close state is to an error

Invariant and Assertion Errors

Invariants: properties express that a certain proposition p will always hold

$\Rightarrow$ states violating an invariant **G**p are those in which $\neg p$ holds

$\Rightarrow$ Negation of p is the formula for invariant error states.

Assertions: special case of invariants that require that a boolean predicate must always hold in a certain control state of a process of the system

$\Rightarrow$ Assertions are defined as the conjunction of a boolean predicate q and a boolean expression $pc_i = s$

$\Rightarrow$ $pc_i = s$ true only in global states where process $\mathcal{P}_i$ is in its local state $s \in S_i$

$\Rightarrow$ states violating assertion **G**$pc_i = s \wedge q$ satisfy $\neg(pc_i = s \wedge q)$.

Deadlocks

. . . global states in which no progress is possible $\Rightarrow$ each process is blocked

Problem: define a boolean predicate that characterizes a state as a deadlock state which could at the same time be used as an input for the estimation function

s: global state of S $\Rightarrow$

$$\textit{blocked\_locally}(i,u,s) \equiv \textit{pc}_i(s) = u \ \wedge \bigwedge_{(u,e,v)\in T_i} \neg \textit{guard}(e,s)$$

Let $B_i \subseteq S_i$ be the set of potentially blocking states within process $\mathcal{P}_i$.

$$\textit{blocked(i,s)} \equiv \bigvee_{u\in B_i} \textit{blocked\_locally}(i,u,s)$$

Deadlock: disjunction of *blocked(i,s)* for every process $\mathcal{P}_i$, with $i = 1,\ldots,n$

$$\textit{deadlock}(S) = \bigwedge_{i=1,\ldots,n} \textit{blocked}(i,S)$$

Potentially Blocking

- some transitions are always executable, e.g. assignments
- conditional statements/communication operations are not always executable

Local state *potentially blocking*: has only potentially non-executable transitions

Example (dining philosophers):
- guard of *take_left*/ *take_right* evaluate to true only if left/right fork is available
- transition *release* is always enabled since its guard is the predicate *true*

$\Rightarrow$ *think* and *wait* are potentially blocking states, since there is no outgoing transition from these states whose guard is always evaluated to *true*

$\Rightarrow$ predicate *blocked* for philosopher i:

$$\textit{blocked}(i,s) \equiv (\textit{pc}_i(s) = \textit{think} \wedge \neg f_i(s)) \vee (\textit{pc}_i(s) = \textit{wait} \wedge \neg f_{i\oplus 1}(s))$$

Formula-Based Estimate

f: boolean predicate of *BP* that characterizes error states.

$h_f(s)$: estimate of number of transitions necessary until a global state $s' \in S$ is reached where f holds, starting from $s \in S$

Function $h_f(s)$ recursively defined as follows:

- $h_{\mathit{true}}(s) = 0$, no transition is necessary since *true* holds in every state and, therefore, also in s.
- $h_{\mathit{false}}(s) = \infty$, evidently *false* can never become *true*.
- $h_v(s) = 0$ (if v is *true* in s)
- $h_v(s) = 1$ (if v is *false*) in s

- $h_{\mathit{pc}_i=s_i}(s) = D_i(\mathit{pc}_i(s), s_i)$, the minimum number of transitions required for process $\mathcal{P}_i$ to reach its local state s_i
- $h_{\neg f}(s) = \overline{h}_f(s)$, where $\overline{h}_f$ is the dual of h_f
- $h_{f \vee g}(s) = \min\{h_f(s), h_g(s)\}$, at least on of f and g must become true
- $h_{f \wedge g}(s) = h_f(s) + h_g(s)$, both f and g must become true
- $h_{x \otimes y}(s) = 1$ (if $x \otimes y$ already holds in state s),$\otimes$ relational operator
- $h_{x \otimes y}(s) = |x - y|$ (if $x \otimes y$ does not hold in s), assuming that the transitions of the system can only increment or decrement numerical variables
- $h_{\mathit{full}(q)}(s) = \mathit{capacity}(q) - \mathit{length}(q)$
- $h_{\mathit{empty}(q)}(s) = \mathit{length}(q)$

Matrix D_i pre-computed in $O(|S_i|^3)$ time with the APSP algorithm of Floyd/Warshall

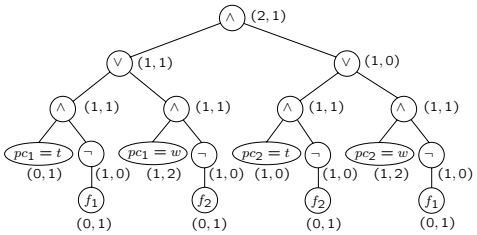
Dual

- $\overline{h}_{\mathit{true}}(s) = \infty$; $\overline{h}_{\mathit{false}}(s) = 0$
- $\overline{h}_v(s) = 1$, if v is *false* in s; $\overline{h}_v(s) = 0$, if v is *false* in s.
- $\overline{h}_{\mathit{pc}_i=s_i}(s) = 1$ (if $\mathit{pc}_i = s_i$); $\overline{h}_{\mathit{pc}_i=s_i}(s) = 0$ (if $\mathit{pc}_i \neq s_i$).
- $\overline{h}_{\neg f}(s) = h_f(s)$.
- $\overline{h}_{f \vee g}(s) = \overline{h}_f(s) + \overline{h}_g(s)$; $\overline{h}_{f \wedge g}(s) = \min\{\overline{h}_f(s), \overline{h}_g(s)\}$.
- $\overline{h}_{x \otimes y}(s) = 0$ (if $x \otimes y$ does not hold in state s).
- $\overline{h}_{x \otimes y}(s) = x - y$ (if $x \otimes y$ holds in state s).
- $\overline{h}_{\mathit{full}(q)}(s) = 0$ (if q is full in s); $\overline{h}_{\mathit{full}(q)}(s) = 1$ (if q is not full in s).
- $\overline{h}_{\mathit{empty}(q)}(s) = 1$ (if q is empty in s).
- $\overline{h}_{\mathit{empty}(q)}(s) = 0$ (if q is not empty in s).

Example

$$((\mathit{pc}_1(s) = \mathit{think} \wedge \neg f_1) \vee (\mathit{pc}_1(s) = \mathit{wait} \wedge \neg f_2)) \wedge$$

$$((\mathit{pc}_2(s) = \mathit{think} \wedge \neg f_2) \vee (\mathit{pc}_2(s) = \mathit{wait} \wedge \neg f_1))$$



The estimate is computed for the global states $s_1 = 1t1t$ and $s_2 = 0e0t$

$\Rightarrow h_f(s_1) = 2$, which is also the real distance.

$\Rightarrow h_f(s_2) = 1$, wrong: real distance 3

Alternative Heuristics for Deadlocks

Active process estimate $H_a(s)$: # number of non-blocked processes in s

$$H_a(s) = \sum_{i \in \{1,\ldots,n\} \wedge active(i,s)} 1,$$

where *active*(i, s) is a flag that determines, whether or not a process $\mathcal{P}_i$ can progress in global state s

$$\textit{active}(i, s) \equiv \bigvee_{(pc_i(s),e,v) \in T_i} \textit{guard}(e),$$

Complexity: (assuming bounded outdegree): linear wrt # processes of the system

Example: $h_a(0e1t)$ is 1, the shortest distance to a deadlock state is 3

Range: 0,. . . ,# processes

4 Heuristics for Known Error States

Two options: focus at exactly the state that was found, focuses on equivalent error states

Hamming Distance Heuristic s: global state as bit vector $s = (s_1, \ldots, s_k)$
s': error state we are searching for

Hamming distance $H_d(s, s')$: bit-flips necessary to transform s into s'

$$H_d(s, s') = \sum_{i=1}^{k} |s_i - s'_i|$$

Complexity: tlinear to the size of the binary enconding

FSM Distance Heuristic

. . . sum for each $\mathcal{P}_i$ of the distance between the local state of $\mathcal{P}_i$ in s and the local state of $\mathcal{P}_i$ in s':

$$H_m(s) = \sum_{i=1}^{n} D_i(pc_i(s), pc_i(s'))$$

Alternative:

- boolean predicate f as conjunction of predicates $pc_i(s) = pc_i(s')$ for each $i = 1, \ldots, n$

- formula-based estimate such that $\forall s \in S : h_f(s) = H_m(s)$

Theorem FSM Distance is monotone / consistent.

5 Trail Improvement

Exploratory mode: provide non-optimal counterexamples

Fault-finding mode: find shortest possible error trail.

*A**: finds optimal or near to optimal counterexamples.

Idea: Profit from the information of the error trail found in the exploratory mode to obtain shorter counterexample

Two approaches

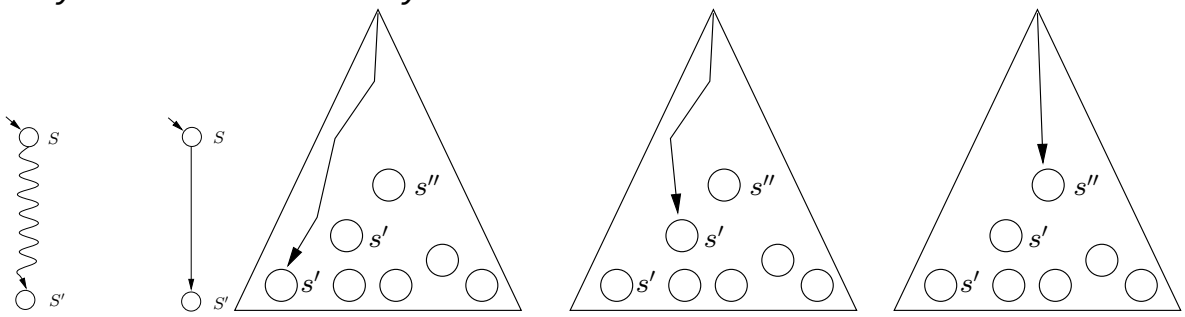
Safety Trails: the goal is the error state

Liveness Trails: Seed state of the accepting cycle.

Improving Safety Trails

Extracted terminal state from provided counterexample to define Hamming distance (exact error states) FSM distance (equivalent error states).

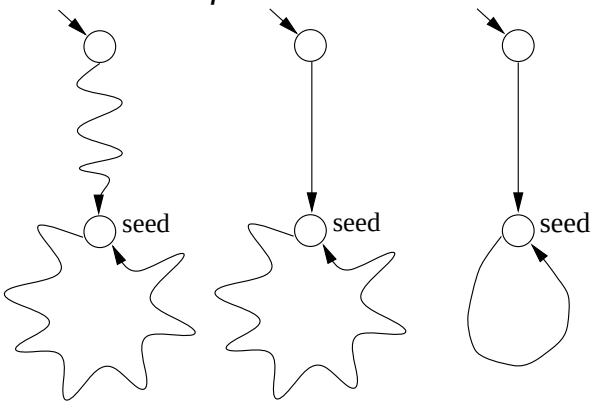
Safety error trail shortened by directed search:



DAC strategy: If shortening path to n-th state not possible (memory constraints) $\Rightarrow$ horten path from 1-st state to $\lfloor n/2 \rfloor$-st, and path from $\lfloor n/2 \rfloor + 1$-st to the n-th state

Improving Liveness Trails

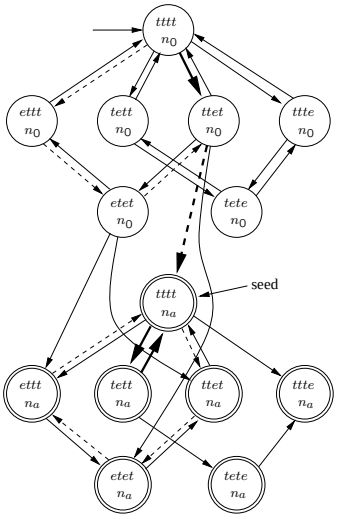
Liveness error trail shortened in two phases:



In both cases: necessary to search for exact same state, and not for equivalent one

Example

Intersection of the envy never claim with four deadlock-free philosophers:



Nested DFS: dashed path of length $4 + 4$ A* with FSM: thick path of length $2 + 2$

References

[1] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 5(2-3):247–267, 2004.

[2] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology (STTT)*, 6(4):277–301, 2004.

[3] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Workshop on Model Checking Software (SPIN)*, pages 57–79, 2001.

[4] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[5] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.

[6] A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, University of Freiburg, 2003.

Directed Model Checking

– HSF-SPIN –

Stefan Edelkamp

1 Overview of HSF-SPIN

Experimental model checker for empirical evaluation of different verification strategies

Implementation Language: C/C++

Available from: Alberto Lluch-Lafuente, Shahid Jabbar, Stefan Edelkamp

Name: reveals its origins; inherits code from the heuristic search framework HSF (Stefan Edelkamp) and model checker SPIN (Gerald Holzmann)

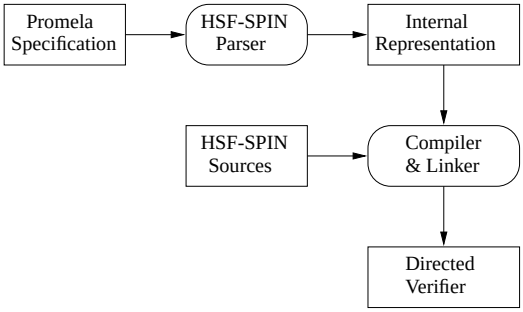
Modeling Language: HSF-SPIN and SPIN use the same language (*Promela*) for specifying models, and the same format for counterexamples

Architecture: separation of the search algorithms from the problem being solved; inherits implementation of several data structures and algorithms

2 Structure of HSF-SPIN

Two phase verification process:

- The *parser*, generates a specific verifier for a given Promela model.



  Promela specification of the model and the correctness requirement are parsed and translated into an internal representation.

- This internal representation is compiled and linked with additional source code resulting in the specific directed verifier

Modeling in HSF-SPIN

The modeling language accepted by HSF-SPIN is the same as in SPIN, namely *Promela*

Promela allows to define parameterizable classes of finite processes.

A special process called *init* is usually started in the first state and instantiates the processes of the system.

Communication in a Promela model is done via shared variables and message channels.

Further documentation of the Promela specification language can be found in SPIN's web site
`http://netlib.bell-labs.com/netlib/spin/whatispin.html`

Example

```
#define left forks[my_id]
#define right forks[(my_id+1)%N]
bool forks[N];

proctype philosopher(int my_id)
{
think:  do
        ::left; /* try to get left fork */
wait:     right; /* try to get right fork */
eat:      atomic{left=true; right=true} /* release forks */
        od
}
```

For the sake of brevity, we have avoided the definition of the init process, which just instantiates each of the N philosophers.

Features HSF-SPIN

For checking safety properties, one can apply

- depth-first search, breadth-first / Dijkstra shortest path search
- A* and greedy best-first search, IDA* and Hill Climbing
- Trail-directed search, interactive simulation

For checking liveness properties one can

- apply *nested depth-first search*
- shorten liveness error trails by applying a two-phase A*.

Specification in HSF-SPIN

Property specification is done as in SPIN, i.e. by

- giving the never claim corresponding to an LTL formula
- using assertion statements within the Promela source
- explicitly expressing that one wants to check deadlock absence.

All provided heuristics can be applied

Ample set partial order reduction including different cycle conditions

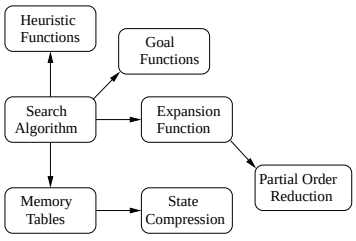
Error Trails written in SPIN's format to used by XSPIN for a graphical visualization, can also reads an error trail to simulate and shorten it

Bitstate hashing compression is supported for IDA*

The Verifier

- performs the verification process.
- takes as input a sequence of parameters that define the type of exploration to be done

Components:



Core of verifier is the search algorithm

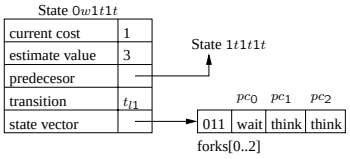
State Representation

The internal representation of a state consists on two parts:

1. information necessary for the search algorithms: the estimate value for the state, the cost of the current optimal path to the state, a link to the predecessor state and information about the transition that lead to the state.

2. representation of the state of the system and is usually called state vector

Example:



Protocol Class

Expansion function: takes the representation of a state as input and delivers a list containing each successor state.

If partial order is active, only a subset of the successors is returned

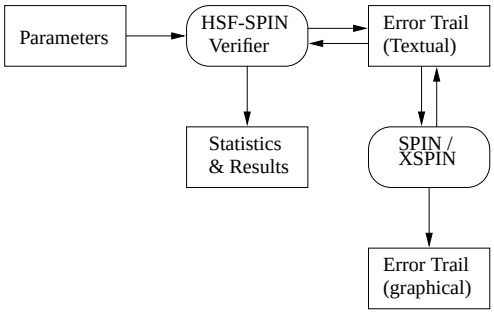
Heuristic function: returns a positive integer value for a given state.

Goal functions: boolean function that determines whether a state is an error state or not.

Several goal functions are implemented according to the kind of error: deadlock, assertion or safety violation of an LTL property.

There is no such function for liveness errors, since liveness errors refer to paths and not to states $\Rightarrow$ detected by the search algorithm itself

Running HSF-SPIN




```
Parameters:
        -Ax: x=Search Algorithm
        -Ex: x=Error to be checked
        -Hx: x=Heuristic Function
        -Wx: x=Weighting for h in A*
...
```


Statistics

```
Printing Statistics...
        State-vector 148 bytes, depth reached 42, errors: 1
              103 states, stored
                8 states, matched
              111 transitions (transitions performed)
               31 atomic steps
               21 states, expanded
        Range of heuristic was: [0..10]

        Memory Statistics: [...]
Writing Trail
Wrote models/deadlock.philosophers.prm.trail
        Length of trail is 42
```

References

[1] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 5(2-3):247–267, 2004.

[2] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Workshop on Model Checking Software (SPIN)*, pages 57–79, 2001.

[3] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[4] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.

[5] A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, University of Freiburg, 2003.

Directed Model Checking

– Symbolic Search –

Stefan Edelkamp

1 Overview

- Boolean encodings of sets of states
- Transitional Relation and Relational Product
- Binary decision diagrams
- Symbolic BFS
  - bidirectional search
  - forward set simplification
- Symbolic Dijkstra
- Symbolic A* (BDD-A*)

2 Characteristic Function

General assumption: Finite state description given in binary

The *characteristic function* $\Phi_S(x)$ for a set of states S is a boolean function mapping $\{0,1\}^n$ to $\{0,1\}$, s.t.

- $\Phi_S(a) = 1$, for all states $s = (a)_2$ in S and

- $\Phi_S(a) = 0$, otherwise.

Characteristic function for two or more states:

$$\Phi_{\{s_1\}\cup\{s_2\}}(x) = \Phi_{\{s_1\}}(x) \vee \Phi_{\{s_2\}}(x)$$

Transition Relation

The *transition relation* has twice as many variables than the encoding of the board:

$T(x, x') = 1 \iff x$ is the encoding of a given state and x' is the encoding of a successor position

Example Transition System: $\bigcirc_{(00)} \leftrightarrows \bigcirc_{(01)} \leftrightarrows \bigcirc_{(10)} \leftrightarrows \bigcirc_{(11)}$

Safety Error: $x_0 \wedge x_1$.

Transitions: $(00) \rightarrow (01)$, $(01) \rightarrow (00)$, $(01) \rightarrow (10)$, $(10) \rightarrow (01)$, $(10) \rightarrow (11)$, $(11) \rightarrow (10)$.

3 Relational Product

Image I of state set *From* wrt. transition relation T:

$$I(x) = \exists x' \; (T(x, x') \; \wedge \; \mathit{From}(x'))$$

Disjunctive partitioning:

$$T(x, x') = \bigvee_{O \in \mathcal{O}} T_O(x, x') \Rightarrow$$

$$I(x) = \bigvee_{O \in \mathcal{O}} (\exists x' \; T_O(x, x') \; \wedge \; \mathit{From}(x'))$$

($\exists$ and $\vee$ commute)

$\Rightarrow T(x, x')$ is not required to be built monolithically

4 Binary Decision Diagrams

Ordered BDD for the boolean function f wrt. variable ordering π;

- an acyclic graph with one source and two sinks labeled with 0 and 1
- all other (internal) nodes are labeled with a boolean variable x_i and have two outgoing edges labeled with 0 and 1.
- on all paths the variables respect π

Reduced Ordered BDD - general assumption - is a *BDD*, where

- nodes with identical successors are omitted and
- isomorphic sub-*BDD*s are unique

Remark: π has exponential impact on BDD size

5 Symbolic BFS

S_i: set of states reachable from initial state s in i steps, initialized by $S_0 = \{s\}$.

Task: determine ϕ_{S_i} given both $\phi_{S_{i-1}}$ and T

$$\phi_{S_i}(x) = \exists x' \, (\phi_{S_{i-1}}(x') \wedge T(x', x))$$

Read as: x *belongs to* S_i *if it has a predecessor* x' *in set* S_{i-1} *and there is a transition from* x' *to* x

On right hand side ϕ depends on x' compared to x on the left hand side

$\Rightarrow$ substitute x with x' in ϕ_{S_i} beforehand (textual replacement $[x/x']$)

Pseudo-Code

Procedure *Symbolic Breadth-First Search*
Input: Transition system with relation T,
Error formula G, start state s

Open $\leftarrow \phi_{\{s\}}$
do
Succ $\leftarrow \exists x' \ (Open(x') \wedge T(x', x))$
Open $\leftarrow$ *Succ*
while (*Open* $\wedge \ \phi_G \equiv 0$)

Bidirectional Search

Backward Search: starts with error and iterates until it encounters initial state

- take advantage that T is relation
- iterate according to formula $\phi_{B_i}(x') = \exists x \ (\phi_{B_{i-1}}(x) \wedge T(x', x))$

Bidirectional BFS: forward and backward search are carried out concurrently

- forward search frontier F_f with $F_0 = \{s\}$
- backward search frontier B_b with $B_0 = G$

Two search frontiers meet $\Rightarrow$ found optimal solution of length $f + b$

Pseudo-Code

Procedure *Bidirectional Breadth-First Search*
Input: Transition system with relation T,
Error formula G, start state s

fOpen $\leftarrow \phi_{\{s\}}$; *bOpen* $\leftarrow \phi_G$
do
  if (*forward*)
    Succ $\leftarrow \exists x' \ (fOpen(x') \wedge T(x', x))$
    fOpen $\leftarrow$ *Succ*
  else
    Succ $\leftarrow \exists x \ (bOpen(x) \wedge T(x', x))$
    bOpen $\leftarrow$ *Succ*
while (*fOpen* $\wedge$ *bOpen* $\equiv$ 0)

Duplicate Elimination

. . . introduction of list *Closed* containing all states ever expanded

- common in single state exploration
- avoids duplicates

Ordinary memory structure: *hash/transposition table*

Symbolic search $\Rightarrow$ *forward set simplification*

Advantage: terminate search in case of complete validation

Pseudo-Code

Procedure *Forward Set Simplification*
Input: Transition system with relation T,
Error formula G, start state s

Closed $\leftarrow$ *Open* $\leftarrow \phi_{\{s\}}$
do
Succ $\leftarrow \exists x' \ (Open(x') \wedge T(x', x))$
Open $\leftarrow$ *Succ* $\wedge \neg$ *Closed*
Closed $\leftarrow$ *Closed* $\vee$ *Succ*
while (*Open* $\wedge \phi_G \equiv 0$)

6 Symbolic Dijkstra

Weighted transition relation: $T(w, x', x) = 1 \iff$ step from x' to x has cost w

$w \in \{0, \ldots, C\} \Rightarrow f$-values restricted to finite domain

Stages:

- *Open* initialized with (representation of) start state, value 0
- extract *all* states with *Min* minimal f-value, determine remaining queue *Rest*
- if no error is encountered, substitute variables in *Min*
- determine successor set *Succ* of *Min*
- attach new f-values to this set
- *Open* for next iteration $=$ disjunct of *Succ* with the remaining queue *Rest*

Pseudo-Code

Procedure Symbolic Dijkstra

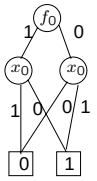
$Open(f, x) \leftarrow (f = 0) \wedge \phi_s(x)$
do
  $f_{\min} = \min\{f \mid f \wedge Open \neq \emptyset\}$
  $Min(x) \leftarrow \exists f\ (Open \wedge f = f_{\min})$
  $Rest(f, x) \leftarrow Open \wedge \neg\, Min$
  $Succ(f, x) \leftarrow \exists x', w\ (Min(x') \wedge$
    $T(w, x', x) \wedge add(f_{\min}, w, f))$
  $Open \leftarrow Rest \vee Succ$
while ($Open \wedge \phi_G \equiv 0$)

7 Symbolic A*

New value of successor s of s':
$f(s) = g(s) + h(s) = g(s') + w(s', s) + h(s) = f(s') + w(s', s) - h(s') + h(s)$

Estimator H: relation of tuples (*value*,*state*) $= 1 \iff h$(*state*)=*value*

Example: $h(00) = h(01) = 1, h(10) = h(11) = 0$



Arithmetics:
formula$(h', h, w, f', f) = \exists\, t_1, t_2\ add(t_1, h', f')\ \wedge\ add(t_1, w, t_2)\ \wedge\ add(h, t_2, f)$

Optimality and completeness: inherited from A*/IDA* (consistent estimate)

Pseudo-Code

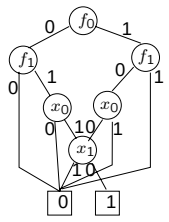
Procedure BDD-A*

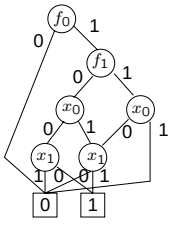
$Open(f,x) \leftarrow H(f,x) \wedge \phi_{S^0}(x)$
do
  $f_{\min} = \min\{f \mid f \wedge Open \neq \emptyset\}$
  $Min(x) \leftarrow \exists f\ (Open \wedge f = f_{\min})$
  $Rest(f,x) \leftarrow Open \wedge \neg\, Min$
  $Succ(f,x) \leftarrow \exists w, x'\ (Min(x') \wedge T(w,x',x) \wedge$
    $\exists h'\ (H(h',x') \wedge \exists h\ (H(h,x) \wedge$
    $formula(h',h,w,f_{\min},f))))$
  $Open \leftarrow Rest \vee Succ$
while $(Open \wedge \phi_G \equiv 0)$

Example

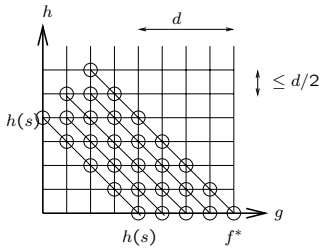
- minterm $\overline{x}_0\overline{x}_1$ with $f = h = 1$
- $\overline{x}_0 x_1$ $h = 1, f = 2$ an f value of two.
- minterm $x_0\overline{x}_1$ is associated with an $f = 2$ and $\overline{x}_0\overline{x}_1$ is assigned to $f = 3$
- extract $x_0\overline{x}_1$ with value 2, find successors 01 and 11.
- Combining the characteristic function x_1 with $h \Rightarrow$ split BDD of x_1, since $x_0 x_1$ relates to $h = 0$, whereas $\overline{x_0} x_1$ relates to $h = 1$.

$\Rightarrow$ Minimal solution length $= 3$





Complexity



Assume consistent und undirected system, $d = \delta(s, T) - h(s)$

- "below" $h(s)$ we have $\leq dh(s) + h(s)$ ◯s
- "roof" above $h(s)$ has $\leq 1 + 3 + \ldots + 2(d/2) - 1 = d^2/4$ ◯s

$\leq dh(s) + h(s) + d^2/4$ iterations

Matrix Computation (*gh-Search*)

Goal: avoid arithmetic computation with BDDs

Idea (Jensen et al.): 2D bucket-layout for BDDs

Advantages:

- state sets to be expanded next are generally smaller
- hope is that BDD representation is as well
- arithmetics to compute f-values no longer needed

Note: tight connection to bucket represenation in *External A**

Pseudo Code

Procedure gh**-BDD-A***

$Open[0, h(\mathcal{I})] \leftarrow \Phi_s$
$f_{\min} \leftarrow \min\{i + j |\ Open[i, j] \neq 0\}$
while $(f_{\min} \neq \infty$ **or** $\Phi_{\mathcal{G}} \wedge Open = 0)$
  $g_{\min} \leftarrow \min\{i \mid Open[i, f_{\min} - i] \neq 0\}$
  $h_{\max} \leftarrow f_{\min} - g_{\min}$
  $Reduce(Open[g_{\min}, h_{\max}])$
  $A(x') \leftarrow$
    $\bigvee_{O \in \mathcal{O}} (\exists x.\ T(x, x') \wedge Open[g_{\min}, h_{\max}](x)[x \setminus x'])$
  for each $i \in \{0, \ldots, \max\}$
    $A_i \leftarrow H[i] \wedge A$
    $Open[g_{\min} + 1, i] \leftarrow Open[g_{\min} + 1, i] \vee A_i$
  $f_{\min} \leftarrow \min\{i + j |\ Open[i, j] \neq 0\} \cup \{\infty\}$

References

[1] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transaction on Computing*, 35(8):677–691, 1986.

[2] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *European Conference on Planning (ECP)*, pages 130–142, 1997.

[3] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.

[4] E. A. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 83–98, 2002.

[5] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *National Conference on Artificial Intelligence (AAAI)*, pages 668–673, 2002.

[6] K. Qian and A. Nymeyer. Heuristic search algorithms based on symbolic data structures. In *Australian Conference on Artificial Intelligence (ACAI)*, pages 966–979, 2003.

[7] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 497–511, 2004.

[8] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods (FM)*, pages 195–211, 1999.

[9] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD based model checking. In *Formal methods in Computer-Aided Design (FMCAD)*, pages 255–289, 1998.

Directed Model Checking

– Symbolic Model Checking –

Stefan Edelkamp

1 Overview

- Kripke Structure
- CTL/CTL*
- Symbolic Model Checking CTL
- μ-calculus
- Symbolic Model Checking μ-Calculus
- Simulation

Kripke Structures

Kripke structure: $M = (S, L, AP, T)$, consists of

- a set of states S,
- a set of atomic propositions AP,
- a labeling function $L : S \rightarrow 2^{AP}$ and
- a transition relation $T \subseteq S \times S$.

S_0: *set* of initial states associated with M, $s \rightarrow s'$ abbreviates $(s, s') \in T$

Note: (labeled) transition systems and Kripke structures closely related

Model checking problem: deciding if model M satisfies specification ϕ, i.e. $M \models \phi$

Specialized model checking problem: $M, S_0 \models \phi$

2 CTL and CTL*

CTL: another temporal logic for specifying properties in model checking.

Basic CTL operators: **AX** and **EX**, **AF** and **EF**, **AG** and **EG**, **AU** and **EU**.

Example of CTL formula that has no equivalent LTL formula: $\mathbf{A}(\mathbf{FG}\phi)$

Example of LTL formula that has no equivalent CTL formula: $\mathbf{AG}(\mathbf{EF}\phi)$

Example of CTL* formula that is neither LTL or CTL: $\mathbf{A}(\mathbf{FG}\phi) \vee \mathbf{AG}(\mathbf{EF}\phi)$

Semantics CTL

ν and μ: (largest and smallest) fixpoint operators

$Sat(p) = \{s \mid p \in L(s)\}, \quad Sat(\phi \vee \psi) = Sat(\phi) \cup Sat(\psi)$

$Sat(\mathbf{EX}\phi) = \{s \mid \exists s'; s' \in Sat(\phi) \wedge s \rightarrow s'\}$
$Sat(\mathbf{AX}\phi) = \{s \mid \forall s'; s' \in Sat(\phi) \Rightarrow s \rightarrow s'\}$

$Sat(\mathbf{A}(\phi \mathbf{U} \psi)) = \mu v.\psi \vee (\phi \wedge \mathbf{AX}v),$
$Sat(\mathbf{E}(\phi \mathbf{U} \psi)) = \mu v.\psi \vee (\phi \wedge \mathbf{EX}v)$

$Sat(\mathbf{AF}\phi) = \mu v.\phi \vee \mathbf{AX}v$, $Sat(\mathbf{EF}\phi) = \mu v.\phi \vee \mathbf{EX}v$

$Sat(\mathbf{AG}\phi) = \nu v.\phi \wedge \mathbf{AX}v,$
$Sat(\mathbf{EG}\phi) = \nu v.\phi \wedge \mathbf{EX}v$

Example

Kripke system: $\rightarrow \bigcirc_0^p \rightarrow \bigcirc_1^p \rightleftarrows \bigcirc_2^q \leftarrow \bigcirc_3$

Check property: $\mathbf{E}(p\mathbf{U}q) = \mu v.p \vee (q \wedge \mathbf{EX}v)$

$f(v) = p \vee (q \wedge \mathbf{EX}v) \Rightarrow f^1(\emptyset) = \{2\}$, $f^2(\emptyset) = \{1,2\}$, $f^3(\emptyset) = \{1,2,0\}$

$S_0 = \{0\} \in f^*(\emptyset) \Rightarrow M, S_0 \models \mathbf{E}(p\mathbf{U}q)$

To compute $Sat(\phi)$ in general:

- evaluate $L(S)$ for proposition AP
- set operations for propositional operators,
- evaluate transitional relation for $\mathbf{X}$ operator
- compute fixpoints for $\mathbf{A}$, $\mathbf{E}$ and $\mathbf{U}$ operators

Symbolic Model Checking CTL

Use BDDs on bitvectors $s_1, \ldots, s_{AP}$, symbolic search for μ and ν

$Check(s_i) = s_i,\ s_i \in L(s) \qquad Check(\phi \vee \psi) = Check(\phi) \vee Check(\psi)$

$Check(\mathbf{EX}, \phi(s)) = \exists s' \phi(s)[s/s'] \wedge T(s,s'),$
$Check(\mathbf{AX}, \phi(s)) = \forall s' \phi(s)[s/s'] \Rightarrow T(s,s'),$

$Check(\mathbf{AU}(\phi(s), \psi(s))) = \mu v.\psi(s) \vee (\phi(s) \wedge Check(\mathbf{AX}, v(s))),$
$Check(\mathbf{EU}(\phi(s), \psi(s))) = \mu v.\psi(s) \vee (\phi(s) \wedge Check(\mathbf{EX}, v(s)))$

$Check(\mathbf{AF}(\phi(s)) = \mu v.\phi(s) \vee Check(\mathbf{AX}, v(s)),$
$Check(\mathbf{EF}(\phi(s)) = \mu v.\phi(s) \vee Check(\mathbf{EX}, v(s))$

$Check(\mathbf{AG}\phi) = \nu v.\phi(s) \wedge Check(\mathbf{AX}, v(s)),$
$Check(\mathbf{EG}\phi) = \nu v.\phi(s) \wedge Check(\mathbf{EX}, v(s))$

Complexities

Theorem (Carke, Grumberg, Long 1993) CTL model checking is polynomial in M and ϕ

Theorem (Sistla, Clarke 1985) LTL model checking is PSPACE complete

Most MC algorithms are polynomial in M and exponential in ϕ

Theorem (Sistla, Clarke 1985) CTL* model checking is PSPACE complete

Theorem $NP \neq P \Rightarrow$ For every Kripke structure M there exists a LTL formula ϕ, s.t. every CTL formula equivalent to $\mathbf{E}\phi$ has more than polynomial length

Theorem $NP \neq P \Rightarrow$ For every Kripke structure M there exists a LTL formula ϕ, s.t. every CTL formula equivalent to $\mathbf{A}\phi$ has more than polynomial length

3 μ-Calculus

V_μ variables of the μ-calculus, F_μ formulae

- $p \in AP \Rightarrow p \in F_\mu$
- $v \in V_\mu \Rightarrow v \in F_\mu$
- $\phi, \psi \in F_\mu \Rightarrow \neg\phi, \phi \vee \psi, \phi \wedge \psi \in F_\mu$
- $\phi \in F_\mu \Rightarrow [t]\phi, \langle t \rangle \phi \in F_\mu$
- $v \in V_\mu, \phi \in F_\mu \Rightarrow \mu v.f, \nu v.\phi \in F_\mu$

$[t]\phi$: "ϕ holds for all state reachable making a t-transition"
$\langle t \rangle \phi$: "possible to make transition t where ϕ holds"

ϕ (in μ,ν) *syntactiacally monoton*: occurences of v in ϕ have even number of negations

Semantics μ-Calculus

$Sat(p) = \{s \mid p \in L(s)\}, \quad Sat(\neg p) = S \setminus Sat(p),$

$Sat(v) =$ environment(v), with

environment : $V_\mu \rightarrow 2^S$ and *environment*$[v/v'](v) = v'$

$Sat(\phi \vee \psi) = Sat(\phi) \cup Sat(\psi), \quad Sat(\phi \wedge \psi) = Sat(\phi) \cup Sat(\psi)$

$Sat(\langle t \rangle \phi) = \{s \mid \exists s \rightarrow s' \wedge s' \in Sat(\phi)\}$
$Sat([t]\phi) = \{s \mid \forall s \rightarrow s' \Rightarrow s' \in Sat(\phi)\}$

$Sat(\mu v.\phi) = \bigcup_{i>0} \tau^i(\emptyset)$
$Sat(\nu v.\phi) = \bigcap_{i>0} \tau^i(S)$

with *predicate transformer* $\tau(v) = Sat(\phi)[v/v']$, $S \subseteq S' \Rightarrow \tau(S) \subseteq \tau(S')$

Symbolic Model Checking μ-Calculus

$Check(p) = s_p$, $Check(v) =$ *environment*(v), $Check(\neg\phi) = \neg Check(\phi)$

$Check(\phi \wedge \psi) = Check(\phi) \wedge Check(\psi)$, $Check(\phi \vee \psi) = Check(\phi) \vee Check(\psi)$

$Check(\mu v.\phi) = Fixpoint(\phi,$ *environment*$, 0)$
$Check(\nu v.\phi) = Fixpoint(\phi,$ *environment*$, 1)$

$Fixpoint(\phi, e, s)$ is following symbolic BFS traversal: $r \leftarrow s$
repeat $o \leftarrow r$; *environment*$[s/o]$; $r \leftarrow Check(\phi)$ *until* $(o = r)$ *return* r

Example: $\mu v.(q \wedge u) \vee \langle t \rangle v$

$S^0 = 0$; $S^{i+1} = (s_q \wedge$ *environment*$(u)) \vee (\exists s'.T(s, s') \wedge S^i)$

Translating CTL in μ-Calculus

Algorithm η performs the translation process

$\eta(p) = p$, $\eta(\neg\phi) = \neg\eta(\phi)$, $\eta(\phi \wedge \psi) = \eta(\phi) \wedge \eta(\psi)$

$\eta(\mathbf{EX}\phi) = \langle t \rangle \eta(\phi)$, $\eta(\mathbf{EG}\phi) = \nu v.\eta(\phi) \wedge \langle t \rangle v$

$\eta(\mathbf{E}(\phi \mathbf{U} \psi)) = \mu v.\eta(\psi) \vee (\eta(\phi) \wedge \langle t \rangle v$

Example: $\eta(\mathbf{EG}(\mathbf{E}[p \mathbf{U} q])) = \nu v.(\mu v'.q \vee (p \wedge \langle t \rangle v')) \wedge \langle t \rangle v)$

Theorem $M = (S, L, AP, T)$ Kripke structure, ϕ formula $\Rightarrow$
$M, S_0 \models \phi \iff S \subseteq Sat(\eta(\phi))$

Simulation

Given: Kripke structures $M = (S, L, AP, T)$ and $M' = (S', L', AP', T')$, with $AP' \subseteq AP$

Simulation relation $\sim \subseteq S \times S'$ between M and M' ($M' \preceq M$):

- for all $s \sim s'$ then $L(s) \cap AP' = L'(s')$
- for every state s_1 such that $s \rightarrow s_1$ there is a state s'_1
- $s' \rightarrow' s'_1$ and $s_1 \sim s'_1$

Theorem If $M' \preceq M$, then for every CTL* formula f, which atomic propositions are contained in AP', $M' \models f \Rightarrow M \models f$

References

[1] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *REX School Symposium, A Decade of Concurrency*, LNCS, pages 124–175. Springer, 1993.

[2] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[5] T. Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

[6] K. L. McMillan. Temporal logic and model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 36–54. Springer, 1998.

[7] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

Directed Model Checking

– Planning and Model Checking –

Stefan Edelkamp

1 Overview

- Common Ground
- Model Checking via Planning
- Model Checking Integrated Planning
- Planning via Model Checking
- Promela Planning

Common Ground

Model checker searches counterexample to falsify a given specification, planner searches for sequence of actions to satisfy a goal

Kripke Structure: finite set of states, set of initial states, transition relation, and state labeling function.

$\Rightarrow$ propositional planning problems can be modeled based on Kripke structures

Thorem Any propositional deterministic planning problem can be modeled as an LTL model checking problem.

Achieving any propositional goal g can be expressed in form of a counter-example to the temporal formula $f = \mathbf{A}(\mathbf{G}\ \neg g)$ in LTL.

The inverse is often available, in particular when model checking safety properties

2 Planning via Model Checking

- *Symbolic Planning:* use symbolic state information in form of BDDs to cope with the increasing number of states to be represented
- *Nondeterministic Planning*: A plan (state-action table) is *weak* if for each state a goal can be reached, *strong* if all terminal states are goals, and *strong cyclic* strong and for each state a terminal one is reachable
- *Conformant and Contingent Planning*: Belief-state planning has been dealt with e.g. the power of BDD representations
- *Planning with Temporary Extended Goals*: Extend the goal specification by general temporal logic formulae
- *Real-Time Planning:* PDDL2.1 has been converted to the timed automata language of the model checker UPPAAL
- *Planning with Control-Rules:* Hand-tailored planners attach supplementary information in form of temporal formulae to prune the state space

3 Model Checking Integrated Planning

Initial Work: planning with the μcke model checker

Minimized Encoding: Fact-space exploration, partition and merge predicates

BDD package: Buddy

MIPS at IPC-2: explicit heuristic search algorithms based on bit-vector state representation and relaxed planning heuristic as well as symbolic heuristic search

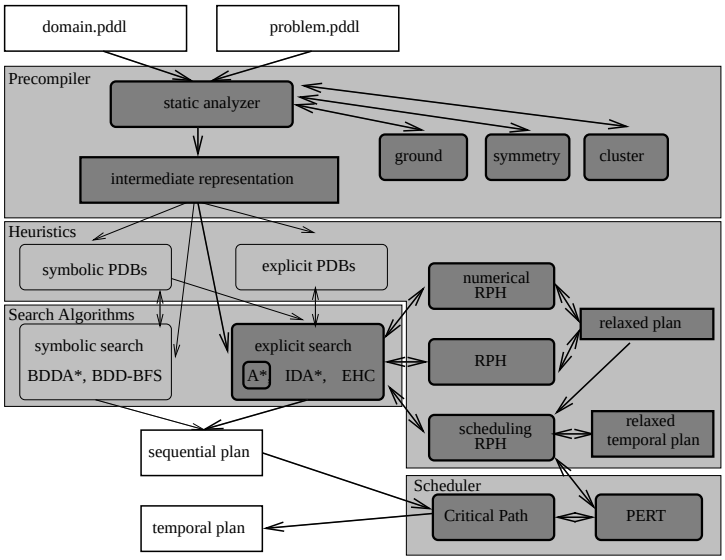
Between the Planning Competitions: explicit and symbolic pattern databases

MIPS at IPC-3: PDDL2.1 expressiveness, (plan at least in every domain)

After 2002: non-linear relaxed planning heuristics, posterior plan scheduling with time-windows, external (symbolic) search

Architecture of MIPS

4 parts: *pre-compilation*, *heuristics*, *search algorithms*, and post-compilation



4 Model Checking via Planning

1. *Bounded Model Checking*: applies the same techniques to Model Checking that are used in the *Planning as Satisfiability* approach
2. *Directed Model Checking*: mimics the success of heuristic search in action planning, e.g. in the domain of hardware validation

Here: Direct Conversion of Model Checking Problems in PDDL (IPC4 Benchmark)

Compiler: Automatically generating a PDDL model from Promela syntax

Restrictions: *Safety Properties*, especially *Deadlocks*

fixed number of processes: dynamic creation of processes in PDDL would require a language extension for dynamic object creation

Example

Automata representation for the model of the 10 Dining Philosophers problem: (Intermediate file produced by SPIN):

```
proctype philosopher
 state 1 -(tr 3)-> state 6 line 11 => forks[_pid]!fork
 state 6 -(tr 4)-> state 3 line 12 => forks[_pid]?fork
 state 3 -(tr 5)-> state 4 line 14 => forks[((_pid+1)%10)]?fork
 state 4 -(tr 3)-> state 5 line 16 => forks[_pid]!fork
 state 5 -(tr 6)-> state 6 line 16 => forks[((_pid+1)%10)]!fork
```

Process P: finite graph $(S(P), \mathit{trans}, \mathit{init}(P), \mathit{curr}(P), \delta(P))$

Channel Q: finite graph $(S(Q), \mathit{head}(Q), \mathit{tail}(Q), \delta(Q), \mathit{mess}(Q), \mathit{cont}(Q))$

Shared and local variables are modeled by PDDL fluents

Domain Encoding

Operators: `queue-read`, `queue-write`, `advance-queue-head`, `advance-empty-queue-tail`, `advance-non-empty-queue-tail`, `process-trans`

- `activate-trans`: activates a transition in a process of a given type if in the current state we have an option to perform the local transition
- `queue-read` and `queue-write` actions, initialize reading/writing of a message.
- `advance-queue-head`, `advance-empty-queue-tail`, `advance-non-empty-queue-tail`: queue update operators, set `settled` flag, which is a precondition of every queue access action
- `process-trans`: executes local transition resets the flags.

Blocking and Deadlocks

Blocked Transitions:

1. read message does not match the requested message
2. queue capacity is either too small or too large

All active transitions in a process block $\Rightarrow$ process itself will block

All processes are blocked $\Rightarrow$ deadlock in the system

Planning Goal: conjunction of atoms requiring that all processes are blocked

Elegant Model: Blocking implemented as a set of derived predicates (PDDL2.2)

References

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[2] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search symbolic model checking = efficient conformant planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 467–472, 2001.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer, 1999.

[5] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Conference on Design Automation (DAC)*, pages 29–34, 2000.

[6] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, pages 875–881, 1998.

[7] H. Dierks, G. Behrmann, and K. Larsen. Solving planning problems using real-time model checking. In *Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Model Checking*, pages 30–39, 2002.

[8] S. Edelkamp. Promela planning. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, pages 197–212. Springer, 2003.

[9] F. Giunchiglia and P. Traverso. Planning as model checking. In *European Conference on Planning (ECP)*, pages 1–19, 1999.

[10] H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1194–1201, 1996.

[11] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 479–486, 2001.

Directed Model Checking

– State Compaction and Incremental Hashing –

Stefan Edelkamp

1 Overview

- State Compression
- Bitstate Hashing
- Paritial A*
- Partial IDA*
- Rabin Karp String Matching
- Incremental Hashing (static, abstract, and dynamic)

2 State Compression

Problem: Given M bity, an uncompressed state representation that requires R bits allows to store M/R distinct states

$\Rightarrow$ find state representation that requires less than R bits of information.

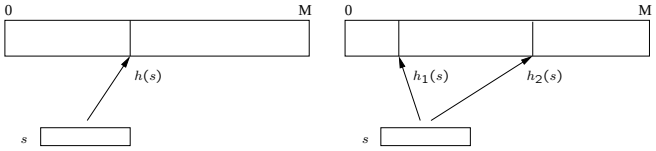
Total compression: assign distinct compressed representation to each state; either reversible or not; e.g., *collapse method*

Partial compression: allow two different states to have the same compressed representation.

- drastically reduce the amount of bits to represent states
- incomplete, since it is not possible to distinguish each state
- false prunings of some parts of the state space

Bitstate Hashing

Single and multi bitstate hashing:



Collisions detected during the exploration of the state space are not resolved, since it is not possible to determine the representation of a compressed state

$\Rightarrow$ not possible to decide whether the colliding state has been already visited or not

Sequential bitstate hashing (Supertrace): improve coverage of bitstate hashing by performing the exploration repeatedly with different, independent hash functions

3 Partial A*

If we apply state compaction for the entire visited list, we arrive at *Partial GSEA* and *Partial A**.

. . . organizes visited list *Closed* in form of an array without any collision strategy on board (We will write *Closed*$[i]$ to highlight the difference).

. . . should additionally be applied without reopening strategy, even if the estimate used is not admissible, since the resulting algorithm cannot guarantee optimal solutions anyway.

In that manner one can avoid node expansions introduced by the reopening issue.

For reducing Open-List, *state reconstruction scheme* is needed

Pseudo-Code

Procedure Partial A* (s, m)
    Closed $[0..m]$ $\leftarrow$ *false*
    Open $\leftarrow \{s\}$
    while (*Open* $\neq \emptyset$)
        $u \leftarrow$ **arg min**$_f$*Open*
        Closed [*Hash*(u)] $\leftarrow$ *true*
        if (*goal*(u)) **return solution**
        $\Gamma(u) \leftarrow$ *Expand*(u)
        for each $v = to(e)$ e **in** outgoing(u)
            if ($v \notin$ *Open*) **and** (*Closed* [*Hash*(v)] $=$ *false*)
                Open $\leftarrow$ *Open* $\cup \{v\}$

4 Partial IDA*

Applying bit-state compression for search algorithms such as BFS and A* is not as effective as it is in DFS or IDA*.

$\Rightarrow$ apparent aspirant for such state compaction techniques is IDA*.

Bit-state hashing can be hardly combined with transposition table updates propagating f- or h-value back to the root

$\Rightarrow$ simpler to initialize the hash table in each iteration.

In opposite to A*, IDA* tracks the solution path on the stack

$\Rightarrow$ omit the predecessor link and additional state information in the state description for the set of visited states.

Pseudo Code

Procedure Partial IDA*(s, m)
  Push(*Stack*, $\{s, h(s)\}$); $U' \leftarrow U \leftarrow h(s)$
  while $(U' \neq \infty)$
    $U \leftarrow U'$; $U' \leftarrow \infty$ *Closed* $[0..m] \leftarrow$ *false*
    while $(S \neq \emptyset)$
      $\{u, f(u)\} \leftarrow$ *Pop*(S)
      if (*goal*(u)) **return solution**
      for each $v = to(e)$, e **in** outgoing(u)
        if not (*Closed* [*Hash*(v)]) *Closed* [*Hash*(v)] $\leftarrow$ *true*
          if $(f(u) +$ *cost*$(u, v) - h(u) + h(v) > U)$
            if $(f(u) +$ *cost*$(u, v) - h(u) + h(v) < U')$
              $U' \leftarrow f(u) +$ *cost*$(u, v) - h(u) + h(v)$
            else *Push*$($*Stack*$, v, f(u) +$ *cost*$(u, v) - h(u) + h(v))$

5 Rabin Karp String Matching

Task: search pattern $P \in \Sigma^k$ in a text $S = (s_1, \ldots, s_n) \in \Sigma^n$.

First compute hash code $h(P)$

In i-th iteration, check, if $S_i = (s_i, \ldots, s_{i+k-1})$ equals P:

$h(s_i, \ldots, s_{i+k-1}) = h(P) \Rightarrow$ character-by-character comparison of S with P

No match is found $\Rightarrow$ compare pattern to $S_{i+1} = (s_{i+1}, \ldots, s_{i+k})$.

Hash value $\sum_{j=i}^{i+k} s_i |\Sigma|^j$ *mod* q of S_{i+1} computed in constant time as

$$h(S_{i+1}) = (h(S_i) - s_i \cdot |\Sigma|^{k-1}) \cdot |\Sigma| + s_{i+k} \, \textit{mod} \, q$$

6 Incremental Hashing

Assumption: state vector v present with v_i in $\Sigma = \{0, \ldots, l-1\}$ (generalizes to $v_i \in D_i$, $|D_i| < \infty$)

Practical Software Model Checking: state vector is a byte array.

For state vectors $v = (v_1, \ldots, v_k)$ and their successor state vectors $v' = (v'_1, \ldots, v'_k)$ we calculate

$$\begin{aligned} h(v') &= \sum_{j=1}^{k} v_j \cdot |\Sigma|^j - v_i \cdot |\Sigma|^i + v'_i \cdot |\Sigma|^i \, mod \, q \\ &= h(v) - v_i \cdot |\Sigma|^i + v'_i \cdot |\Sigma|^i \, mod \, q. \end{aligned}$$

General Result

. . . operators change the value of some component (e.g. atomic blocks)

Let $I(v, v') = \{i \mid v_i \neq v'_i\}$ the set of modified vector indices, then

$$\begin{aligned} h(v') &= \sum_{i=1}^{k} v_i \cdot |\Sigma|^i - \sum_{i \in I(v,v')} v_i \cdot |\Sigma|^i + \sum_{i \in I(v,v')} v'_i \cdot |\Sigma|^i \, mod \, q \\ &= h(v) - \sum_{i \in I(v,v')} v_i \cdot |\Sigma|^i + \sum_{i \in I(v,v')} v'_i \cdot |\Sigma|^i \, mod \, q. \end{aligned}$$

Theorem Computing the hash value of v' given the one for v is available in time

- $O(|I(v, v')|)$; using $O(k)$ extra space
- $O(1)$; using $O((k \cdot |\Sigma|)^{I_{\max}})$ extra space, where $I_{\max} = \max_{(v,v')} |I(v, v')|$.

Abstraction

State space abstractions are used to compute pattern databases

Let ϕ_i be a mapping $v = (v_1, \ldots, v_k)$ to $\phi_i(v) = (\phi_i(v_1), \ldots, \phi_i(v_k))$, $1 \leq i \leq l$.

Theorem Combined incremental state and abstraction state vector hashing of state vector v' with respect to its predecessor v is available in time

- $O\left(|I(\phi(v,v'))| + \sum_{i=1}^{l} |I(\phi_i(v), \phi_i(v'))|\right)$ using $O(kl)$ extra space, where $I(\phi_i(v), \phi_i(v'))$ denotes the set of affected indices in database i
- $O(l)$; using $O(l \cdot (k \cdot |\Sigma|)^{I_{\max}})$ extra space.

Dynamic Distributed Incremental Hashing

Distributed and dynamic incremental hashing of vector $v' = (v_1, \ldots, v_m)$ with respect to its predecessor $v = (v_1, \ldots, v_m)$ assuming a modification in component i (update, insertion or deletion) is available in time

- $O(|I(v_i, v_i')|)$ for update, $O(m + \log n_i)$ for insertion, and $O(m)$ for deletion using $O(m + \max_{i=1}^{m} n_i)$ extra space.
- $O(|I(v_i, v_i')| \log m)$ for update, $O(\log m + \log n_i)$ for insertion, and $O(\log m)$ for deletion using $O(m + \max_{i=1}^{m} n_i)$ extra space.

First approach uses shifts within bitvector

Second approach uses AVL trees

References

[1] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology (STTT)*, 6(4):277–301, 2004.

[2] S. Edelkamp and T. Mehler. Incremental hashing in state space search. In *Workshop "New Results in Planning, Scheduling and Design"*, 2004.

[3] S. Edelkamp and T. Mehler. Incremental hashing for pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2005. Poster, To Appear.

[4] G. J. Holzmann. State compression in SPIN. In *3rd Workshop on Software Model Checking*, 1997.

[5] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.

[6] W. Holzmann. Memory efficient storage in SPIN. In *2nd Workshop on Software Model Checking*, 1996.

[7] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. In *KI*, pages 229–243, 2001.

[8] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[9] T. Mehler and S. Edelkamp. Dynamic distributed incremental hashing and state reconstruction in program model checking. Draft.

[10] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer-Aided Verification*, LNCS, pages 59–70. Springer, 1993.

Directed Model Checking

– Abstraction –

Stefan Edelkamp

1 Overview

- Pattern Databases
- Abstraction
- Data/Predicate abstraction
- Abstraction Databases
- Abstraction in Practice

Pattern Databases

. . . automatic technique for designing estimator functions

. . . first applied to define effective heuristics for the 15-puzzle

Subsequently, pattern databases defined heuristics to solve random instances to Rubik's Cube for the first time

Recently, they have shown general applicability in action planning

Abstraction

- copes with large and infinite state spaces in model checking
- reduces verification efforts
- abstract system often smaller than the original one
- if abstract system satisfies correctness spec, so does the concrete one
- abstractions may introduce behaviors not present in the original system

Combination of abstraction with heuristic search to improve error detection

- abstract system is explored in order to create a database that stores the exact distances from abstract states to the set of abstract error states
- check, whether or not the abstract behavior is present in the original system, efficient exploration algorithms exploit the database as a guidance

Abstraction as Simulation

Problem: find abstractions that are correct wrt. correctness specification and, at the same time, provide significant reductions $\Rightarrow$ research focused on *simulations*

System $\widehat{M}$ simulates a system M: every *behavior* of M is also present in $\widehat{M}$

Abstract system simulates the original one $\Rightarrow$ if it satisfies the specification so does the original one

Opposite direction: not true!

Spurious error: bad behavior in the abstract system might not be present in the original one.

Abstract-and-refine loop: approximation refined for a one that is consistent with the counterexample established, and the verification process starts again.

Data/Predicate Abstraction

Data Abstraction: If variables v_i in state vector $(v_1, \ldots, v_k)$ have domain D and all abstract variables have domain $A \subseteq D \Rightarrow$ abstraction $\phi : D \rightarrow A$.

In many cases: abstracted system simulates the original one

———————

Predicate abstraction: abstract concrete states under a finite set of predicates

The predicates $p_1, \ldots, p_n$ define an abstraction function $\phi : S \rightarrow \{0,1\}^n$ with $\phi(s) = \widehat{s}$ if for all i we have $\widehat{s}_i = p_i(s)$

Abstract state reachable: abstraction of a concrete state that is reachable

As above: predicate abstractions are simulations

Example

```
x:=0; while x<n do x++; end do
```

Abstraction on variable x: $\phi(0) = \texttt{ZERO}$, $\phi(n) = N$, and $\phi(i) = \texttt{MIDDLE} \Rightarrow$

```
x:=ZERO
while x<N
 if(x=0) then x=MIDDLE
 else x=nondeterministically MIDDLE or N
```

Original system has finite path $(s_0, \ldots s_n)$, where s_i is state in which $x = i$

Abstraction ϕ: $\phi(s_i) = \phi(s_{i+1})$ for $1 \leq i \leq (n-2) \Rightarrow \phi$ is a simulation that induces abstract system with infinitely many paths $(\phi(s_0), \phi(s_1)^j, \phi(s_n))$

$(\phi(s_0), \phi(s_1)^{(n-2)}, \phi(s_n))$: has corresponding concrete path
$(\phi(s_0), \phi(s_1), \phi(s_n))$: spurious counterexample

Abstraction a Kripke Structures

$\phi : M = (S, L, AP, T) \rightarrow \widehat{M} = (\widehat{S}, \widehat{L}, \widehat{AP}, \widehat{T})$ with

- $\widehat{S} = \phi(S) = \{\phi(s) \mid s \in S\}$,
- $\widehat{T} = \{(\phi(s), \phi(s_1)) \mid T(s, s_1)\}$

General: apply a surjection of the state space graph (S, T) into $(\widehat{S}, \widehat{T})$

ϕ *homomorphism*: for all $s \rightarrow s_1$ we have $\phi(s) \widehat{\rightarrow} \phi(s_1)$
$\Rightarrow t$ is reachable from state s implies $\phi(t)$ is reachable from $\phi(s)$

Note: abstraction ϕ can be extended to modify the label set

Theorem: For each two concrete states s and t:
shortest path from $\phi(s)$ to $\phi(t) \leq$ shortest path from s to t

Abstraction Database

Abstraction database: according to an abstraction ϕ table with entries

$$(m, \delta_H(m, \phi(t))$$

for each m in abstract space

Construction: Database computed in a backward traversal of abstract space:

Directed graph with invertible operators: Backward BFS or Backward Dijkstra's single-source shortest path search algorithm if the graphs are weighted

Directed graph with non-invertible operators: in forward traversal of abstract space construct graph inverse on-the-fly, collect horizon nodes and uses only inverse graph links

2 Abstractions in Practice

α-SPIN: extends SPIN by allowing the Promela model to be abstracted.

Processing: model is read with an XML parser and combined with an abstraction function, also coming in XML

Result: XML file representing abstract model, translated back into Promela.

$\Rightarrow$ approach produces no source conflict with the model checker

Storage/Retrieval: Maintain hash table with abstract state information on disk

Addressing: dummy assignments of the form `variable := variable;` in the original Promela source. The abstraction mechanism results in an assignment `variable := map(variable);`.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[2] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *Static Analysis Symposium*, Lecture Notes in Computer Science, pages 51–53. Springer, 1995.

[5] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[6] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science. Springer, 2001. 13-24.

[7] S. Edelkamp and A. Lluch-Lafuente. Abstraction in directed model checking. In *ICAPS-Workshop on Connecting Planning Theory with Practice*, 2004.

[8] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[9] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[10] P. Merino, M. del Mar Gallardo, J. Martinez, and E. Pimentel. aspin: Extending spin with abstraction. In *Model Checking Software (SPIN)*, Lecture Notes in Computer Science. Springer, 2002.

[11] R. Milner. An algebraic definition of simulation between programs. In *Joint Conference of Artificial Intelligence (IJCAI)*, pages 481–489, 1995.

[12] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 497–511, 2004.

[13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 72–83. Springer, 1997.

Directed Model Checking

– Partial Order Reduction –

Stefan Edelkamp

1 Overview

- Motivation
- Independence and Invisibility
- Stuttering Equivalence
- Ample Set
- General State Expanding Search
- Hierarchy of Cycle Conditions
- (Recovering) Solution Quality

2 Motivation

Idea: exploit the commutativity of asynchronous systems to reduce the size of the state space

Resulting state space: equivalent to the original one wrt. specification

Two main families:

- net unfoldings
- diamond properties (here)

Several approaches:

- “stubborn” sets, “persistent” sets, and
- “ample” sets (here)

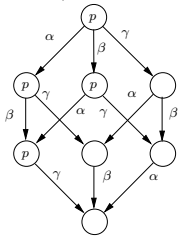
Independence and Invisibility

$\alpha, \beta \in T$ *independent*: for each state $s \in S$ in which α, β are defined we have

1. $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$ (enableness preserving)
2. $\alpha(\beta(s)) = \beta(\alpha(s))$ (communtative)

α *invisible* wrt. a set of propositions P: $\forall s, s'$, $s' = \alpha(s)$: $L(s) \cap P = L(s') \cap P$.

Example: Transitions α, β and γ are pairwise independent; α and β are invisible

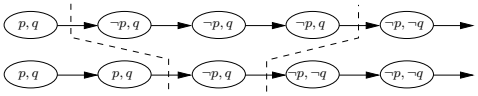


with respect to the set of propositions $P = \{p\}$, while γ is not

Stuttering Equivalence

Two executions are *stuttering equivalent* wrt. P: atomic propositions of the i-th block of both executions have the same intersection with P, for each $i > 0$

Example: stuttering equivalent paths wrt. LTL property in which only p and q occur.



Two transition systems *stuttering equivalent*: same set of initial states and for each execution in one systems $\exists$ stuttering equivalent execution in the other one

LTL$_{-X}$ cannot distinguish between stuttering equivalent transition systems

$\mathcal{M}$ and $\mathcal{N}$ are two stuttering equivalent transition systems $\Rightarrow$ $\mathcal{M}$ satisfies a given LTL$_{-X}$ specification $\Longleftrightarrow$ $\mathcal{N}$ does

Ample Set

C0: $ample(s)$ is empty exactly when $enabled(s)$ is empty.

C1: Along every path in full state space starting at s, a transition dependent on one in $ample(s)$ does not occur without a transition in $ample(s)$ occurring first.

C2: If a state s is not fully expanded, then each transition α in the ample set of s must be invisible with regard to P.

C3: If for each state of a cycle in the reduced state space, a transition α is enabled, then α must be in the ample set of some of the states of the cycle.

C0 and **C2**: independent to search algorithm applied

Checking Condition **C1** at least as hard as checking reachability for the full state space $\Rightarrow$ over-approximation

Dynamically Checking the Cycle Condition

Condition C3$_{cycle}$: Every cycle in the reduced state space contains at least one state that is fully expanded.

$\Rightarrow$ checking **C3** can be reduced to detecting cycles during the search

Avoiding ample sets containing backward edges except when state is fully expanded ensures **C3** when using DFS/IDA*

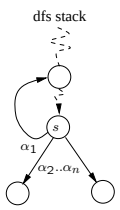
$\Rightarrow$ **C3**$_{stack}$:

C3$_{stack}$: If a state s is not fully expanded, then no transition in $ample(s)$ leads to a state on the search stack.

Safety Cycle Condition

C3^{-}: If for each state of a cycle in the reduced state space, α is enabled, then α must be in the ample set of some successor of a state of the cycle.

Condition C3$^{-}_{stack}$: If a state s is not fully expanded, then at least one transition in $ample(s)$ does not lead to a state on the search stack.

Example: 

Contrary to **C3**$_{stack}$, **C3**$^{-}_{stack}$ accepts $\{\alpha_1, \alpha_2\}$ as valid ample set

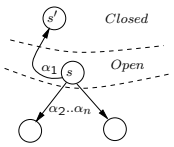
$\Rightarrow$ **C3**$^{-}_{stack}$ not sufficient to guarantee **C3** (necessary for checking liveness)

3 Checking Cycle Condition with GSEA

Common: assumes cycle to exist whenever an already visited state is found.

$\Rightarrow$ weaker reductions, as it is known that state spaces of concurrent systems usually have a high density of duplicate states.

C3$_{duplicate}$: If a state s is not fully expanded, then no transition in *ample*(s) leads to an already visited state.



$\{\alpha_1\}$ and $\{\alpha_1, \alpha_2\}$ are examples of non valid ample sets. On the other hand, the set $\{\alpha_2\}$ is not refuted.

Safety Cycle Condition for a GSEA

. . . alternative condition in order to enforce the cycle condition **C3**$^-$

. . . sufficient to guarantee a correct reduction when checking safety properties.

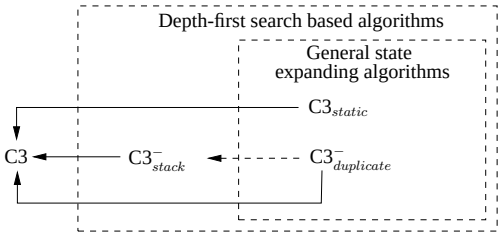
. . . based on the same idea as **C3**$_{duplicate}$

Condition C3$^-_{duplicate}$: If a state s is not fully expanded, then at least one transition in *ample*(s) does not lead to an already visited state.

Example: $\{\alpha_1, \alpha_2\}$ is rejected as ample set by condition **C3**$_{duplicate}$, but not by **C3**$^-_{duplicate}$.

4 Hierarchy of Cycle Conditions

C3 Conditions: All presented cycle conditions for checking safety properties.



Arrows indicate which condition enforces which other.

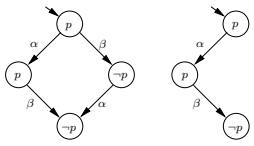
5 Solution Quality

Observation: Shortest path to error in reduced space often longer than shortest path to error in full space

Intuition: Concept of stuttering equivalence does not make assumptions about length of equivalent blocks.

Example: α and β independent, α invisible wrt. p; invariant **G**p to be checked

$\Rightarrow$ reduced state space stuttering equivalent to full one, shortest path length of 2 vs. 1 for original space



Recovering Solution Quality

Idea: process error trail after the verification; ignore those transitions that are independent from the one that directly lead to the error state

Approach: extracting irrelevant transitions from the counterexamples until there is no such transition

Observation: after extracting an irrelevant transition, more new transitions may become irrelevant, e.g. if they were dependent on the recently extracted transition.

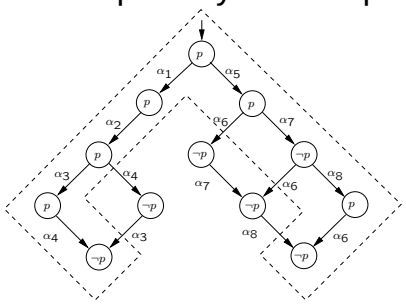
$\Rightarrow$ an efficient algorithm must extract transitions beginning from the last one.

6 Pseudo-Code

Procedure *Filter* (r)
  $r' \leftarrow r$; $j \leftarrow n-1$;
  while $1 \leq j < n$ **do**
    irrelevant $\leftarrow$ *true*;
    if *visible*$(r'[j])$ **then** *irrelevant* $\leftarrow$ *false*;
    else
      for i **in** $j+1$ .. n **do**
        if *dependent*$(r'[i], r'[j])$ **or** *can_enable*$(r'[j], r'[i])$ **then**
          irrelevant $\leftarrow$ *false*; **break**;
    if *irrelevant* **then** $r' \leftarrow$ *extract*(r', j); $n \leftarrow n-1$;
    else $j \leftarrow j-1$;
  return r';

Admissability

Observation: resulting counterexample may not be optimal.



Error: p doesn't hold; *Independent:* (α_3, α_4), (α_6, α_7), (α_6, α_8); *Visible:* α_6, α_4

Path formed by transitions α_1, α_2, α_3 and α_4 can be established as shortest path in the reduced state space denoted by the dashed region.

Established error path $\alpha_1\alpha_2\alpha_4$; Optimal error path in full space: $\alpha_5\alpha_6$.

References

[1] R. Alur, R. Brayton, T. Henzinger, S. Qaderer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *International Conference on Computer-Aided Verification*.

[2] S. Bornot, R. Morin, P. Niebert, and S. Zennou. Black box unfolding with local first search. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 241–257, 2002.

[3] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 241–257, 1996.

[4] P. Godefroid. Using partial orders to improve automatic verification methods. In *International Conference on Computer-Aided Verification*.

[5] D. A. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in Systems Design*, 8:39–64, 1996.

[6] A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165, 1991.

Directed Model Checking

– Directed Search –

Stefan Edelkamp

1 Overview

- Search in Directed Graphs
- General State Expanding Search
- Depth-Bounded Search and Re-opening
- Heuristics
- Dijkstra / A*
- IDA*
- Other Heuristic Search Algorithms

2 Search in Directed Graphs

Implicit Graphs: graph $G = \langle V, E \rangle$ constructed starting from the initial vertex and generating new vertices in each step

Strongly connected component (SCC): maximal set of vertices, such that each vertex in the set is reachable from each other vertex of the set.

Computing the SCCs: Algorithm of Tarjan

Cost of $p = \langle e_1..e_n \rangle$: $cost(p) = \sum_{i=1..n} cost(e_i)$

Termination: search algorithm does not run forever.

Completeness: search algorithm terminates exactly when a solution exists.

Admissibility/Optimality: always returns optimal solutions.

General State Expanding Search Algorithm

General state expanding search algorithm (GSEA): divides the vertices of a graph into three sets: the set *Open*, *Closed* and the rest

- iteratively extracts vertices from *Open* and *moves* them into *Closed*.

- if extracted vertex is a goal $\Rightarrow$ algorithm terminates

- vertices extracted from *Open* are expanded

- if successor of expanded vertex neither in *Open* nor in *Closed* $\Rightarrow$ add to *Open*

BFS implements *Open* as a FIFO queue and DFS as a stack.

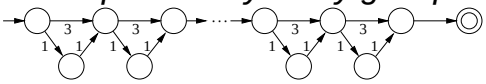
Reopening: visit vertices and their successors more than once by inserting already visited and newly generated vertices into *Open*

Pseudo-Code

Procedure *GeneralStateExpandingAlgorithm*(s)
  Closed $\leftarrow \emptyset$;
  Open $\leftarrow \emptyset$;
  Open.insert(s);
  while not *Open.empty*$()$ **do**
    $u \leftarrow$ *Open.extract*$()$;
    Closed.insert(u);
    if *goal*(u) **then return solution**;
    for each $e \in$ *outgoing*(u) **do**
      $v \leftarrow$ *to*(e);
      if $v \notin$ *Closed* **and** $v \notin$ *Open* **then**
        Open.insert(v);

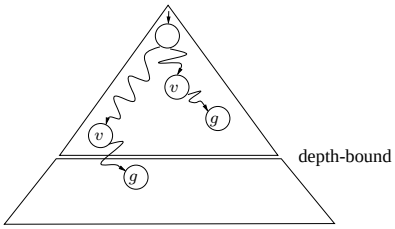
Bounding the Depth

Directed weighted graph with exponentially many goal paths:



Depth-Bounded Search: Bound maximal depth reachable by DFS

Anomaly:



Guarantee completeness and admissibility: reopen vertices, when reached through a shorter path than the current shortest one.

Pseudo-Code

Procedure *AdmissibleDepthFirstSearch*(s);
  if *goal*(s) **then**
    goal $\leftarrow$ s; *depthbound* $\leftarrow$ s.*depth*;
  end if;
  Closed.insert(s)
  for each $e \in$ *outgoing*(s) **do**
    $s' \leftarrow$ *to*(e);
    if ($s \notin$ *Closed* **or** s'.*depth* $>$ *depth* $+ 1$)
    and s'.*depth* $<$ *depthbound* **then**
      s'.*depth* $=$ *depth* $+ 1$;
      AdmissibleDepthFirstSearch(s');

Initially: s.depth for each s and *depthbound* are set to ∞, and *depth* is set to 0

Dijktra's Single Source Shortest Path

Change to original: new algorithm terminates if a goal vertex is found.

Specializes GSEA algorithm by implementing the *Open* list as a priority queue.

- Priority of a vertex: cost of the current optimal cost path to that vertex

- Value is denoted by attribute f in the algorithm.

- Every time a vertex is reached through a path with lower cost the priority of the vertex is updated

Pseudo-Code

Procedure *Dijktra*(s)
  Closed $\leftarrow \emptyset$; *Open* $\leftarrow \emptyset$; $s.f \leftarrow 0$;
  Open.insert(s);
  while not *Open.empty*$()$ **do**
    $u \leftarrow$ *Open.extractmin*$()$; *Closed.insert*(u);
    if *goal*(u) **then return solution**;
    for each $e \in$ *outgoing*(u) **do**
      $v \leftarrow$ *to*(e); $f' \leftarrow u.f +$ *cost*(e);
      if $v \in Open$ **then**
        if $(f' < v.f)$ **then** $v.f \leftarrow f'$;
      else $v.f \leftarrow f'$; *Open.insert*(v);

3 Heuristics

Heuristic search: exploits information of the underlying problem being solved in order to improve the search process.

Information: usually represented by means of functions that rank the desirability of exploring a vertex

A heuristic estimate function h is

- *admissible*: $h(u) \leq h^*(u)$ for every vertex u of the graph.

- *monotone*: $h(u) \leq$ *cost*$(e) + h(v)$, for every pair of vertices u, v of the graph where e is the edge going from u to v with minimum cost.

Theorem: Monotone heuristic is admissible

4 A* Search Algorithm

. . . similar to BFS and Dijktra: instead of expanding the vertices according to their depth or cost, rank the vertices by using the value assigned by a heuristic function.

. . . identify vertices having higher probability to lead to goal vertex

If successor vertex v is in

- *Open* with new cost $<$ old one $\Rightarrow$ update current optimal cost

- *Closed* with new cost $<$ old one $\Rightarrow$ delete from *Closed* and re-insert into *Open*

Reopening: required for guaranteeing admissibility with non-monotone heuristics

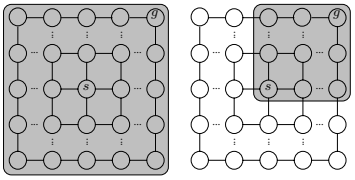
Theorem: A* optimal on finite graphs. Optimality on infinite graphs is also guaranteed if the cost of every infinite path is unbounded (Pearl)

Pseudo-Code

Procedure *A**(s)
  Closed $\leftarrow \emptyset$; *Open* $\leftarrow \emptyset$; $s.f \leftarrow h(s)$; $s.g \leftarrow 0$; *Open.insert*(s);
  while not *Open.empty*$()$ **do**
    $u \leftarrow$ *Open.extractmin*$()$; *Closed.insert*(u);
    if *goal*(u) **then return solution**;
    for each $e \in$ *outgoing*(u) **do**
      $v \leftarrow$ *to*(e); $v.g \leftarrow u.g + cost(e)$; $f' \leftarrow v.g + h(v)$;
      if $v \in$ *Open* **then**
        if $(f' < v.f)$ **then** $v.f \leftarrow f'$;
      else if $v \in$ *Closed* **then**
        if $(f' < v.f)$ **then**
          $v.f \leftarrow f'$; *Closed.delete*(v); *Open.insert*(v);
      else $v.f \leftarrow f'$; *Open.insert*(v);

Example

Search in a grid with Dijkstra's algorithm (left) and A* (right):



Estimator: Euclidean distance

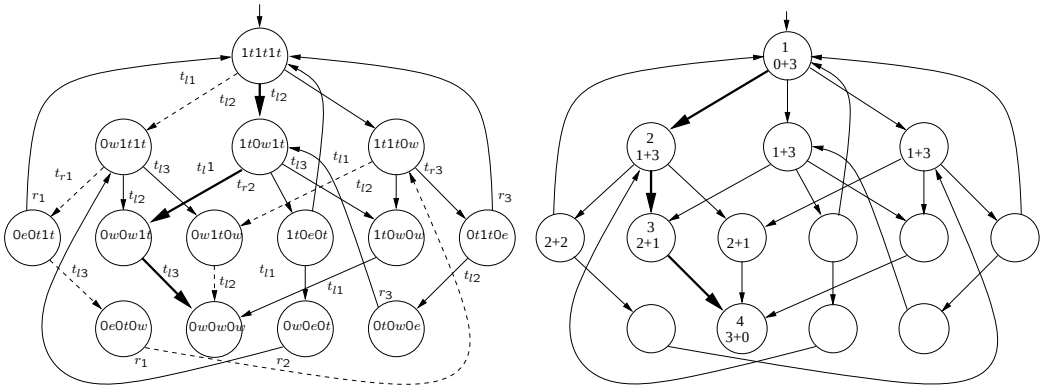
Goal vertex: upper corner of the grid

Initial vertex: center of the grid.

Reduction: number of visited vertices $\approx 1/4$ with respect to Dijkstra's algorithm.

Application to Dining Philosophers

State space of the asynchronous composition of three philosophers (left)



Search with A* in this state space (right) – order on top of g+h

Error: Deadlock; *Heuristic*: number of active processes

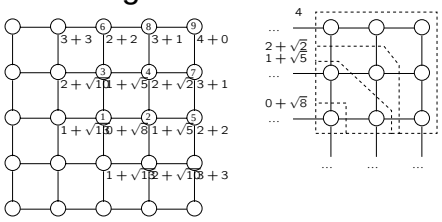
5 Iterative Deepening A*

Depth-first search iterative deepening (DFID): performs several depth-first search traversals with increasing depth bound

$\Rightarrow$ linear space with respect to the depth, while optimal solutions are guaranteed.

Iterative deepening A*: successive iterations correspond to increased cost bounds

$\Rightarrow$ space complexity remains linear to depth reached, admissibility is preserved, less vertex expansions wrt DFID if good heuristic function are used.



Pseudo-Code

Procedure *IDA**()
    threshold $\leftarrow h(s)$;
    loop *new_threshold* $\leftarrow \infty$; *depth* $\leftarrow 0$;
        search$(s,$ *threshold*$)$; *threshold* $\leftarrow$ *new_threshold*;

Procedure *search*$(u,$ *threshold*$)$
    if *goal*(u) **then return solution**;
    depth $\leftarrow$ *depth* $+ 1$;
    for each $e \in$ *outgoing*(u) **do**
        $v \leftarrow$ *to*(e); $f \leftarrow$ *depth* $+ h(v)$;
        if $f \leq$ *threshold* **then** *search*$(v,$ *threshold*$)$;
        else if $f <$ *new_threshold* **then** *new_threshold* $\leftarrow f$;
    depth $\leftarrow$ *depth* $- 1$;

6 Other Directed Search Strategies

Hill Climbing: DFS variant where the order in which successors of a vertex are explored is determined by a heuristic function; not admissible, local optima.

*WA**: modification of A* with $f(u) = (1 - w) * u.g + w * h(u)$, with $0 \leq w \leq 1$; trade-off between solution quality and acceleration of the search.

Greedy Best-First: A* without reopening in which the heuristic function used is just an evaluation function; not admissible but able to find goal vertices very fast.

Beam Search: similar to best-first, but uses a bounded $Open$ list, incomplete and applied for searching in large graphs with high density of goals

Frontier Search: The frontier search reduce memory requirements of A* by avoiding storage of closed vertices, effective in graphs where $|Open| >> |Closed|$

References

[1] R. Bisiani. Beam search. In *Encyclopedia of Artificial Intelligence*, pages 1467–1568. Wiley Interscience Publication, 1992.

[2] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[3] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.

[4] J. Pearl. *Heuristics*. Addison-Wesley, 1985.

[5] A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[6] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, (1):146–160, 1972.

[7] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Science Department, University of Wisconsin, 1970.