Algorithmic Methods for Formal Verification

by Yaniv Sa'ar

Supervisors: Prof. Amir Pnueli, Prof. Lenore D. Zuck, and Prof. David Harel

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science

July 10, 2011 / Ph.D. Defense

Introduction

Thesis

A little software engineering in verification techniques goes a long way.

Introduction

- Currently formal verification tools are black boxes
  - Hard to incorporate new model checking algorithms, unless one is familiar with tool (low-level C code)
- An exception is TLV (Temporal Logic Verifier)
  - TLV slightly changed the usual design perspective
  - Special purpose scripting language to create and manipulate BDDs on-the-fly
  - Tightly coupled with BDD package (CMU SMV)

Introduction

- Currently formal verification tools are black boxes
  - Hard to incorporate new model checking algorithms, unless one is familiar with tool (low-level C code)
- An exception is TLV (Temporal Logic Verifier)
  - TLV slightly changed the usual design perspective
  - Special purpose scripting language to create and manipulate BDDs on-the-fly
  - Tightly coupled with BDD package (CMU SMV)

Introduction

- Currently formal verification tools are black boxes
  - Hard to incorporate new model checking algorithms, unless one is familiar with tool (low-level C code)
- An exception is TLV (Temporal Logic Verifier)
  - TLV slightly changed the usual design perspective
  - Special purpose scripting language to create and manipulate BDDs on-the-fly
  - Tightly coupled with BDD package (CMU SMV)

There is a need for easier-to-use developing environment!

Introduction

There is a need for easier-to-use developing environment!

JTLV *Java Temporal Logic enVironment*:

- Computer-aided verification framework
- Allows for easy development of formal algorithms, in a high-level programming environment (e.g., verification, analysis, synthesis, abstraction)
- On top of state-of-the-art IDE; Eclipse
- Without having to sacrifice performance

Outline

Synthesis

- Automatic synthesis of programs and (digital) designs from logical specifications is one of the most ambitious and challenging problems in computer science
- A solution would lift programming from the current mostly imperative level, to a declarative, logical style
- This is of major importance when concurrency is involved

Synthesis of Reactive(1) Designs

Synthesis process for general LTL has been identified as hopelessly intractable, yet:

- We identify an important and expressive fragment of LTL (GR(1)) which we can synthesize in quadratic time

AspectLTL: An Aspect Language for LTL Specifications

- Declarative language for specification and implementation of crosscutting concerns, based on LTL
- Supported by a JTLV-based prototype tool, that (literally) lifts declarative programming into practice
  - Generation of LTL aspect composition and synthesis is sound and complete
- Synthesis of GR(1) is an integral part of the compilation process

Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem

Compositional Methods

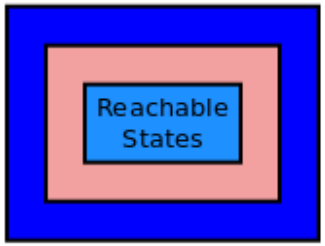
- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem

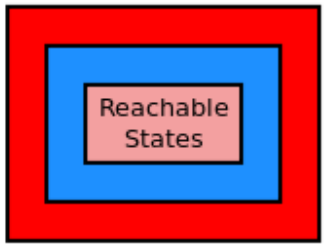
Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem

Compositional Methods

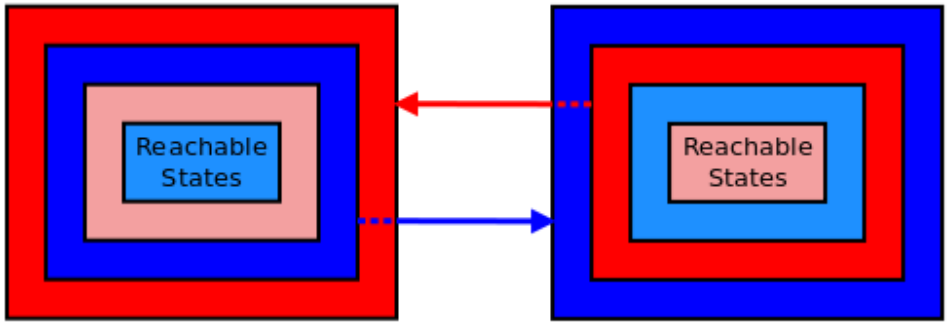
- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem





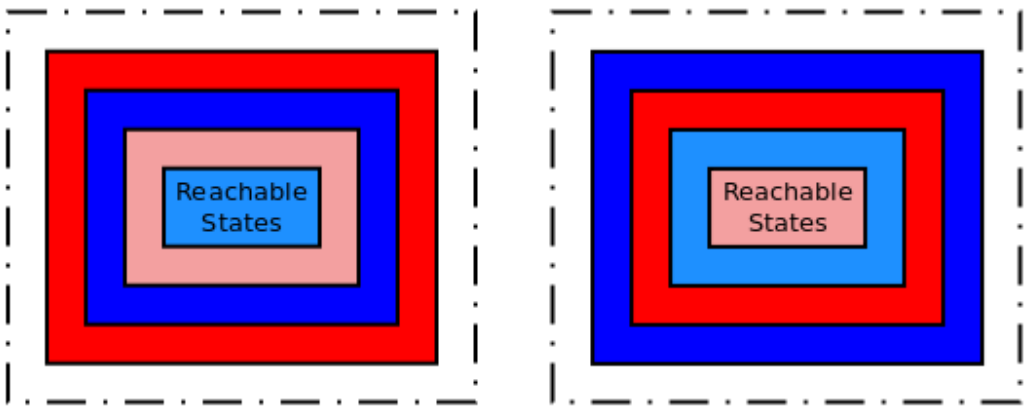
Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem



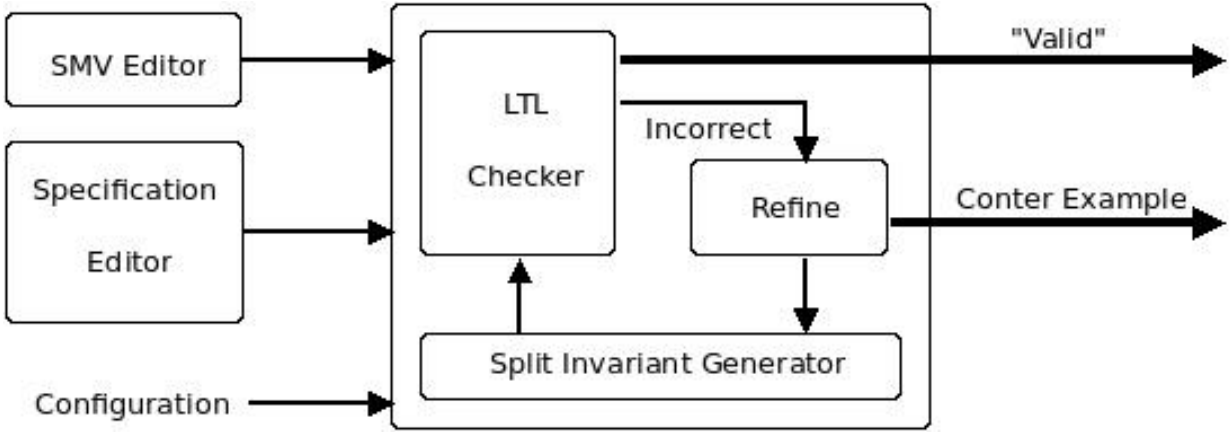
Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem



SPLIT: A Compositional LTL Verifier

- SPLIT is the first tool to implement a fully automated compositional method for arbitrary LTL properties



Parallelizing A Symbolic Compositional Model-Checking Algorithm

- Many attempts were made to parallelize model checking under general settings
- For compositional reasoning, both interference-freedom and locality suggest distributing each local computation.
  - How to distribute the computations?
  - How to handle the BDD structure(s)?
- Our solution:
  - BDD manager for each local computation
  - BDD communication between threads by copy
  - Number of threads slightly larger than number of cores
  - Good thread topology (usually star)

A Dash of Fairness for Compositional Reasoning

- Fairness is necessary for liveness
- Incorporating (global) fairness compositionally is difficult
- Previous work handles justice
- We developed a new algorithm to verify general LTL formulae for systems with compassion

And...

Two-Way Traceability and Conflict Debugging for AspectLTL

JTLV enables traceability and debugging support for AspectLTL programs.

Verification of Multi-Linked Heaps

JTLV enables the automatic reasoning on programs that perform destructive updating on heaps.

All You Need is Compassion

There are cases where mechanical solution is not good enough.

Introduction

Thesis

A little software engineering in verification techniques goes a long way.

Outline

JTLV: A Framework for Developing Verification Algorithms

Verification tools are based on two common elements.

- Computational Model – represents the system implementation
  - Representation for the various programming languages
  - Assigns a semantics to each reactive system
- Specification Language – represents the expectations from the implementation
  - Essentially temporal logic
  - The semantics of the computational structures serves as models for the formula

JTLV: A Framework for Developing Verification Algorithms

- Currently formal verification tools are black boxes
  - Hard to incorporate new model checking algorithms, unless one is familiar with tool (low-level C code)
- An exception is TLV (Temporal Logic Verifier)
  - TLV slightly changed the usual design perspective
  - Special purpose scripting language to create and manipulate BDDs on-the-fly
  - Tightly coupled with BDD package (CMU SMV)

JTLV: A Framework for Developing Verification Algorithms

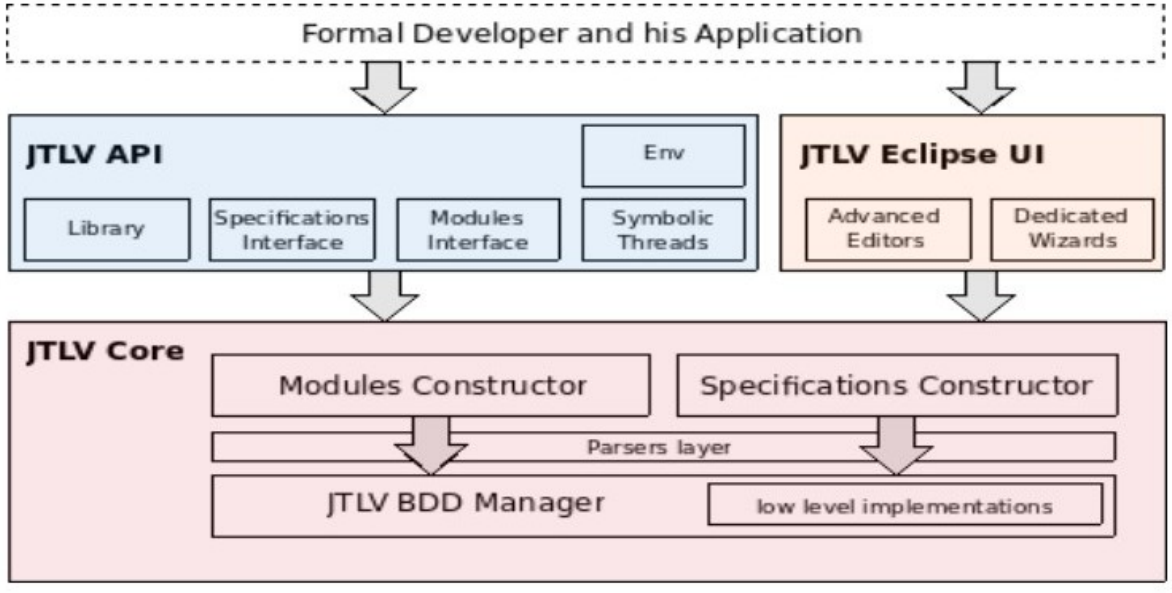
There is a need for easier-to-use developing environment!

JTLV *Java Temporal Logic enVironment*:

- Computer-aided verification framework
- Allows for easy development of formal algorithms, in a high-level programming environment (e.g., verification, analysis, synthesis, abstraction)
- On top of state-of-the-art IDE; Eclipse
- Without having to sacrifice performance

JTLV: A Framework for Developing Verification Algorithms

JTLV is composed of three major components:



JTLV: A Framework for Developing Verification Algorithms

API:

- Env – Basic functionalities, e.g. load modules and specifications, save and load BDD to and from files, etc.
- OpenVDL – Open Verification Developer Library, contains implementations of known formal algorithms
- Module – A pure Java interface to the SMV code
- Specification – A pure Java interface to the specifications
- Symbolic Threads – Each instance is coupled with its own BDD manager, allowing it to execute freely without synchronization with other threads

JTLV: A Framework for Developing Verification Algorithms

So,. . . what can we do with it?

- Advanced software engineering techniques:
  - Object oriented design, and patterns
  - Increasing level of (programming) abstraction
  - Symbolic multi-threaded algorithms
- Java/Eclipse ecosystem with respect to JTLV:
  - Debugger, software verifiers, documentation methodology (JavaDoc), scripting environment (BeanShell), code refactoring, unit testing, etc.

JTLV: A Framework for Developing Verification Algorithms

So,. . . what can we do with it?

- Advanced software engineering techniques:
  - Object oriented design, and patterns
  - Increasing level of (programming) abstraction
  - Symbolic multi-threaded algorithms
- Java/Eclipse ecosystem with respect to JTLV:
  - Debugger, software verifiers, documentation methodology (JavaDoc), scripting environment (BeanShell), code refactoring, unit testing, etc.

JTLV: A Framework for Developing Verification Algorithms

So,. . . what can we do with it?

- Advanced software engineering techniques:
  - Object oriented design, and patterns
  - Increasing level of (programming) abstraction
  - Symbolic multi-threaded algorithms
- Java/Eclipse ecosystem with respect to JTLV:
  - Debugger, software verifiers, documentation methodology (JavaDoc), scripting environment (BeanShell), code refactoring, unit testing, etc.

Conclusion

JTLV makes symbolic techniques easier.
(demonstrated in rest of talk)

Synthesis

- Automatic synthesis of programs and (digital) designs from logical specifications is one of the most ambitious and challenging problems in computer science
- A solution would lift programming from the current mostly imperative level, to a declarative, logical style
- This is of major importance when concurrency is involved

Synthesis of Reactive(1) Designs

Synthesis process for general LTL has been identified as hopelessly intractable ([Pnueli, Rosner 89]), yet:

Synthesis of Reactive(1) Designs

Synthesis process for general LTL has been identified as hopelessly intractable ([Pnueli, Rosner 89]), yet:

- [Asarin, Maler, Pnueli, Sifakis 98]: (cubic) polynomial solutions to games (and synthesis) where the acceptance condition is: $\Box p$, $\Diamond q$, $\Box \Diamond p$, or $\Diamond \Box q$
- [Alur, La Torre 04]: efficient synthesis for Boolean combinations of formulae of the form $\Box p$

Synthesis of Reactive(1) Designs

Synthesis process for general LTL has been identified as hopelessly intractable ([Pnueli, Rosner 89]), yet:

- We present a quadratic game-based synthesis algorithm for General Reactivity of rank 1 (GR(1)) specifications:

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)$$

where each p_i and q_i is a Boolean combination of atomic propositions

Synthesis of Reactive(1) Designs

GR(1) fragment

$$(\Box \Diamond p_1 \wedge \cdots \wedge \Box \Diamond p_m) \rightarrow (\Box \Diamond q_1 \wedge \cdots \wedge \Box \Diamond q_n)$$

- Specifications are usually exponentially more succinct then their implementations
- Past LTL formulae can be included in both assumptions and guarantees

Synthesis of Reactive(1) Designs

GR(1) fragment

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)$$

- GR(1) seems to place an undue burden on the user or have a too restrictive expressive power
  - Assume/Guarantee specifications are common practice in industry
  - Expressiveness of GR(1) is well established
  - Solution to GR(1) implies a solution to GR(k) ($\bigwedge_k$ GR(1)) that is exponential in k. GR(k) is at the top of LTL hierarchy

Synthesis of Reactive(1) Designs

GR(1) fragment

$$\underbrace{(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m)}_{\text{Assume}} \rightarrow \underbrace{(\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)}_{\text{Guarantee}}$$

- GR(1) seems to place an undue burden on the user or have a too restrictive expressive power
  - Assume/Guarantee specifications are common practice in industry
  - Expressiveness of GR(1) is well established
  - Solution to GR(1) implies a solution to GR(k) ($\bigwedge_k$ GR(1)) that is exponential in k. GR(k) is at the top of LTL hierarchy

Synthesis of Reactive(1) Designs

GR(1) fragment

$$\underbrace{(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m)}_{\text{Assume}} \rightarrow \underbrace{(\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)}_{\text{Guarantee}}$$

- GR(1) seems to place an undue burden on the user or have a too restrictive expressive power
  - Assume/Guarantee specifications are common practice in industry
  - Expressiveness of GR(1) is well established
  - Solution to GR(1) implies a solution to GR(k) ($\bigwedge_k$ GR(1)) that is exponential in k. GR(k) is at the top of LTL hierarchy

Synthesis of Reactive(1) Designs

GR(1) fragment

$$\underbrace{(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m)}_{\text{Assume}} \rightarrow \underbrace{(\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)}_{\text{Guarantee}}$$

- GR(1) seems to place an undue burden on the user or have a too restrictive expressive power
  - Assume/Guarantee specifications are common practice in industry
  - Expressiveness of GR(1) is well established
  - Solution to GR(1) implies a solution to GR(k) ($\bigwedge_k$ GR(1)) that is exponential in k. GR(k) is at the top of LTL hierarchy

Synthesis of Reactive(1) Designs

- The synthesis heavily uses symbolic analysis of state space
- The synthesis, with all its stages, was implemented in JTLV

Synthesis of Reactive(1) Designs

- The synthesis heavily uses symbolic analysis of state space
- The synthesis, with all its stages, was implemented in JTLV

Conclusion

The high complexity established for LTL synthesis does not necessarily identify it as intractable. JTLV allows for efficient mechanization of the synthesis.

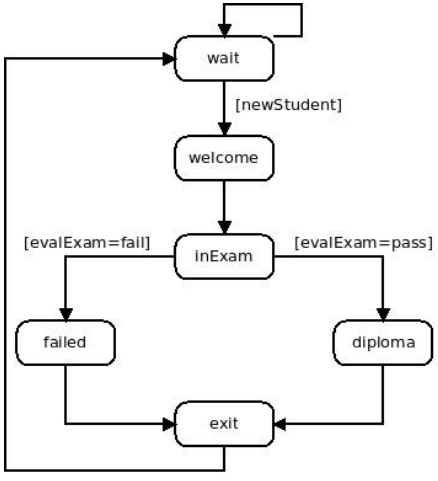
AspectLTL: An Aspect Language for LTL Specifications

- Declarative language for specification and implementation of crosscutting concerns, based on LTL
- Supported by a JTLV-based prototype tool, that (literally) lifts declarative programming into practice
  - Generation of LTL aspect composition and synthesis is sound and complete
- Synthesis of GR(1) is an integral part of the compilation process

AspectLTL: An Aspect Language for LTL Specifications

Exam service

```
MODULE ExamService
  VARENV -- envrinoment variables
    evalExam : {pass, fail};
    newStudent : boolean;
  VAR    -- system variables
    state : {wait, welcome, inExam, diploma,
             failed, exit};
  INIT
    state=wait;
  TRANS
    ((state=wait) -> ( (next(state)=wait) |
       (newStudent & next(state)=welcome) )) &
    ((state=welcome) -> (next(state)=inExam)) &
    ((state=inExam)
      -> ( (next(state)=diploma &
            next(evalExam)=pass    ) |
           (next(state)=failed  &
            next(evalExam)=fail)   )    ) &
    ((state=diploma) -> (next(state)=exit)) &
    ((state=failed) -> (next(state)=exit)) &
    ((state=exit) -> (next(state)=wait));
```



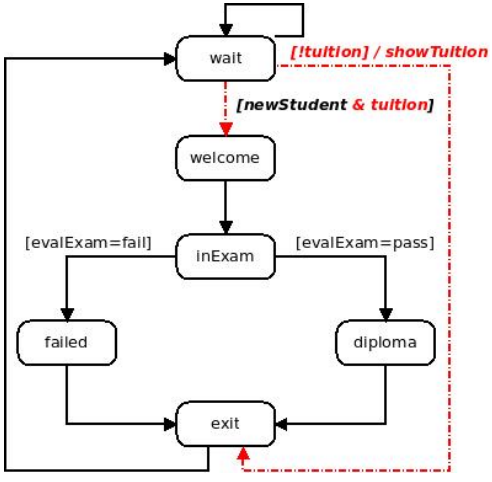
AspectLTL: An Aspect Language for LTL Specifications

Exam service – with a tuition concern

```
MODULE ExamService
  VARENV -- envrinoment variables
    evalExam : {pass, fail};
    newStudent : boolean;
  VAR    -- system variables
    state : {wait, welcome, inExam, diploma,
             failed, exit};
  INIT
    state=wait;
  TRANS
    ((state=wait) -> ( (next(state)=wait) |
       (newStudent & next(state)=welcome) )) &
    ((state=welcome) -> (next(state)=inExam)) &
    ((state=inExam)
      -> ( (next(state)=diploma &
            next(evalExam)=pass   ) |
           (next(state)=failed  &
            next(evalExam)=fail)  )   ) &
    ((state=diploma) -> (next(state)=exit)) &
    ((state=failed) -> (next(state)=exit)) &
    ((state=exit) -> (next(state)=wait));
```

+

```
ASPECT Tuition
  VARENV -- envrinoment variables
    new tuition : boolean;
  VAR    -- system variables
    ext state : {wait, welcome, exit};
    new showTuition : boolean;
  TRANS
    -- adding a transition from wait directly
    -- to exit if the tuition was not paid.
    ( state=wait & next(state)=exit &
      !tuition & next(showTuition) );
  LTLSPEC
    -- there is no transition from wait to
    -- welcome if the tuition was not paid.
    [] ( !(state=wait & !tuition &
           next(state)=welcome) );
```



AspectLTL: An Aspect Language for LTL Specifications

- AspectLTL supports the language features considered as distinguishing characteristics of aspect languages:
  - Obliviousness – the base system makes no assumptions about LTL aspects
  - Quantification – an aspects assumes little about the base system (allowing modularity of concerns)

AspectLTL: An Aspect Language for LTL Specifications

- As opposed to prior works, ours addresses the correct composition at the semantics level
- We use GR(1) to define a novel declarative style programming language
- The language is supported by a JTLV-based prototype

AspectLTL: An Aspect Language for LTL Specifications

- As opposed to prior works, ours addresses the correct composition at the semantics level
- We use GR(1) to define a novel declarative style programming language
- The language is supported by a JTLV-based prototype

Conclusion

AspectLTL lifts programming from the current, mostly imperative style, to a declarative, logical style. JTLV enables the development of an efficient compiler for the language.

Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem
- [Namjoshi 05] Construct a vector, split invariant, of local (i.e., per-process) inductive invariants,

$$\theta = \theta_1, \theta_2, \ldots, \theta_n$$

  - interference-free – a step by one process does not violate the invariant of another
  - the conjunction of local invariants ($\bigwedge_i \theta_i$), is always globally inductive invariant
  - Locality is enforced syntactically

Compositional Methods

- Model checking suffers from state explosion
- In the case of asynchronous concurrent systems, local reasoning can often ameliorate the problem
- [Namjoshi 05] Construct a vector, split invariant, of local (i.e., per-process) inductive invariants,

$$\theta = \theta_1, \theta_2, \ldots, \theta_n$$

  - interference-free – a step by one process does not violate the invariant of another
  - the conjunction of local invariants ($\bigwedge_i \theta_i$), is always globally inductive invariant
  - Locality is enforced syntactically

Compositional Methods

Split invariant computation:

- Reachability analysis: PSPACE(N)-complete. Computation of split invariant: PTIME(N) and incomplete
- [Cohen, Namjoshi 07] automatically refine using *auxiliary variables* yielding completeness

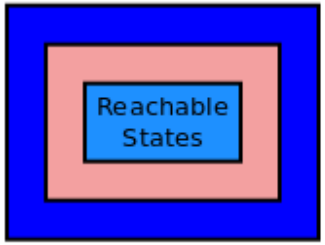
Compositional Methods

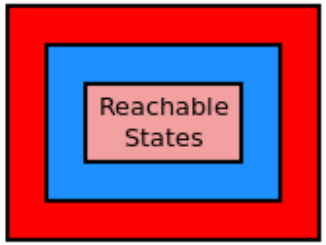
Split invariant computation:

- Reachability analysis: PSPACE(N)-complete. Computation of split invariant: PTIME(N) and incomplete
- [Cohen, Namjoshi 07] automatically refine using *auxiliary variables* yielding completeness

Compositional Methods

Split invariant computation:

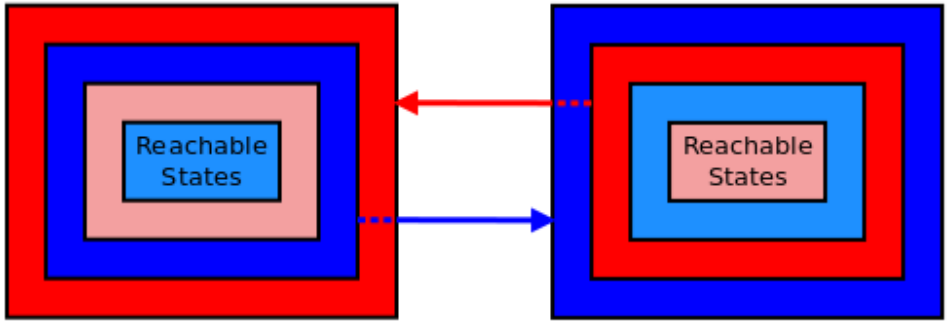




- Reachability analysis: PSPACE(N)-complete. Computation of split invariant: PTIME(N) and incomplete
- [Cohen, Namjoshi 07] automatically refine using *auxiliary variables* yielding completeness

Compositional Methods

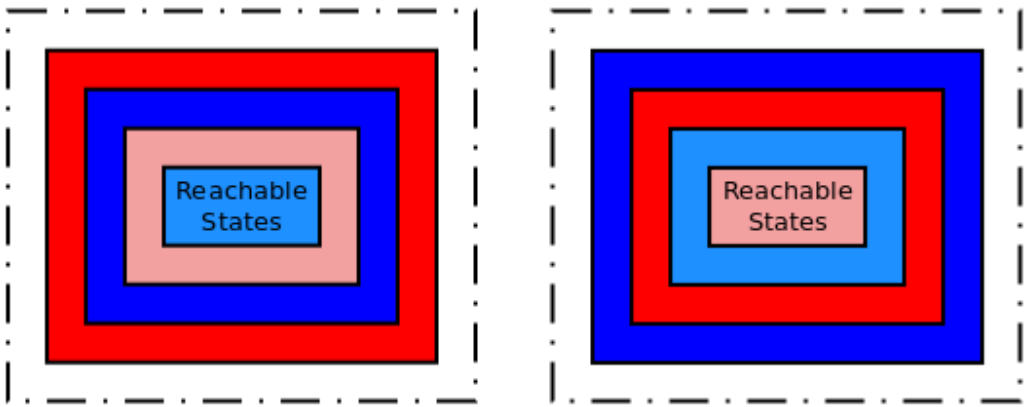
Split invariant computation:



- Reachability analysis: PSPACE(N)-complete. Computation of split invariant: PTIME(N) and incomplete
- [Cohen, Namjoshi 07] automatically refine using *auxiliary variables* yielding completeness

Compositional Methods

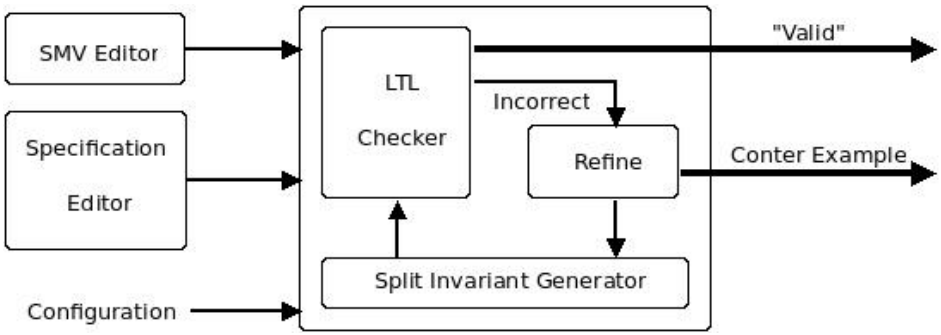
Split invariant computation:



- Reachability analysis: PSPACE(N)-complete. Computation of split invariant: PTIME(N) and incomplete
- [Cohen, Namjoshi 07] automatically refine using *auxiliary variables* yielding completeness

SPLIT: A Compositional LTL Verifier

- SPLIT is an overarching system implementing:
  1. Simultaneous least fixpoint [Namjoshi 05]
  2. [Cohen, Namjoshi 07] safety refinement (completeness)
  3. [Cohen, Namjoshi 08] algorithm for progress under justice
  4. [CNS10a] algorithm for arbitrary LTL properties under both justice and compassion



SPLIT: A Compositional LTL Verifier

- Experimental results support claim that local reasoning allows verifying *significantly* larger systems
- SPLIT is the first tool to implement a fully automated compositional method for arbitrary LTL properties

Conclusion

JTLV enabled the development of a complete off-the-shelf standalone application for fully compositional reasoning.

Parallelizing A Symbolic Compositional Model-Checking Algorithm

- Many attempts were made to parallelize model checking under general settings
- For compositional reasoning, both interference-freedom and locality suggest distributing each local computation.
  - How to distribute the computations?
  - How to handle the BDD structure(s)?
- Our solution:
  - BDD manager for each local computation
  - BDD communication between threads by copy
  - Number of threads slightly larger than number of cores
  - Good thread topology (usually star)

Parallelizing A Symbolic Compositional Model-Checking Algorithm

	sequential	4 cores			8 cores		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
MUX-SEM							
512	27	8.3	3.25	0.81	4.8	5.6	0.70
1024	117	34.8	3.3	0.82	19.2	6.1	0.76
1536	360	112	3.2	0.80	65	5.5	0.69
2048	561	165	3.4	0.85	92	6.1	0.76
SZYMANSKI							
5	3.1	1.6	1.93	0.48	1.1	2.81	0.35
6	20.5	6.5	3.15	0.78	4.4	4.65	0.78
7	130	41	3.17	0.79	23.7	5.48	0.78
8	564	163	3.46	0.86	93	6.06	0.76
9	2896	739	3.91	0.97	492	5.88	0.73
GERMAN							
9	489	126	3.88	0.97	76	6.40	0.80
10	1076	268	4.00	1.00	164	6.56	0.82
11	2867	691	4.14	1.03	385	7.44	0.93
12	over BDD limit	1819	-	-	1013	-	-
PETERSON'S							
4	0.7	0.9	0.77	0.19	0.9	0.77	0.17
5	8.5	4	2.1	0.50	3.2	2.6	0.52
6	183	66	2.77	0.70	46	3.98	0.66

Parallelizing A Symbolic Compositional Model-Checking Algorithm

- Multi-core processing is a promising direction for next generation model checking
- Compositional algorithms can exploit the multi-core
- JTLV is an ideal environment to develop multi-processing applications

Conclusion

Model checking on multi-core architectures is a promising direction. JTLV enables the development non-trivial multi-threaded applications.

A Dash of Fairness for Compositional Reasoning

- Fairness is necessary for liveness
- Incorporating (global) fairness compositionally is difficult
- Previous work handles justice
- We developed a new algorithm to verify general LTL formulae for systems with compassion

A Dash of Fairness for Compositional Reasoning

1. Compute the split invariant vector $\langle \theta_1, \theta_2, \ldots, \theta_n \rangle$
2. For each process P_i build an abstraction Q_i containing transitions T_i, and summaries of others transitions
   - Summary S_j for P_j is defined as $(\exists L_j, L'_j : \theta_j \wedge T_j)$
3. For each process P_i abstract others fairness conditions
   - A compassion $\langle p_j, q_j \rangle$ of P_j is transformed to its corresponding abstraction
     $\langle \underbrace{\forall L_j : \theta_j \rightarrow p_j}_{\text{increasing states}}, \underbrace{\exists L_j : \theta_j \wedge q_j}_{\text{reducing states}} \rangle$
4. Verify the property over every abstract process Q_i
5. If the check succeeds, HALT with success
6. If all summary transitions in cex are concrete, HALT with failure. Otherwise, refine and RETURN to step 1

A Dash of Fairness for Compositional Reasoning

- New algorithm demonstrates how to handle compassion in compositional framework
- Experimental results are encouraging
- JTLV ideal for experiments

Conclusion

Compositional reasoning can also handle compassion. JTLV makes reasoning simple.

And...

Two-Way Traceability and Conflict Debugging for AspectLTL

JTLV enables traceability and debugging support for AspectLTL programs.

Verification of Multi-Linked Heaps

JTLV enables the automatic reasoning on programs that perform destructive updating on heaps.

All You Need is Compassion

There are cases where mechanical solution is not good enough.

Conclusion

There is a need for easier-to-use developing environment!

[PSZ10b]

JTLV makes symbolic techniques easier.

Conclusion

There is a need for easier-to-use developing environment!

[PSZ10b]

JTLV makes symbolic techniques easier.

[PPS06, BPPS11]

The high complexity established for LTL synthesis does not necessarily identify it as intractable. JTLV allows for efficient mechanization of the synthesis.

[MS11]

AspectLTL lifts programming from the current, mostly imperative style, to a declarative, logical style. JTLV enables the development of an efficient compiler for the language.

Conclusion

There is a need for easier-to-use developing environment!

[PSZ10b]

JTLV makes symbolic techniques easier.

[CNS+10c]

Model checking on multi-core architectures is a promising direction. JTLV enables the development non-trivial multi-threaded applications.

[CNS10a]

Compositional reasoning can also handle compassion. JTLV makes reasoning simple.

[CNS10b]

JTLV enabled the development of a complete off-the-shelf standalone application for fully compositional reasoning.

Conclusion

There is a need for easier-to-use developing environment!

[PSZ10b]

JTLV makes symbolic techniques easier.

[MS12]

JTLV enables traceability and debugging support for AspectLTL programs.

[BPSZ11]

JTLV enables the automatic reasoning on programs that perform destructive updating on heaps.

[PS08]

There are cases where mechanical solution is not good enough.

Conclusion

There is a need for easier-to-use developing environment!

[PSZ10b]

JTLV makes symbolic techniques easier.

Other Users

We should also note that JTLV already has an avid user community, including researchers from Imperial College London, New York University, Bell Labs Alcatel-Lucent, Weizmann Institute, Microsoft Research Cambridge, RWTH-Aachen, California Institute of Technology, GRASP Laboratory University of Pennsylvania, and University of California Los Angeles.

Algorithmic Methods for Formal Verification

by Yaniv Sa'ar

Supervisors: Prof. Amir Pnueli, Prof. Lenore D. Zuck, and Prof. David Harel

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science

July 10, 2011 / Ph.D. Defense

Partial Bibliography

- Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. *JTLV: A Framework for Developing Verification Algorithms.* In *Proc. 22nd Int. Conf. on Computer Aided Verification*, 2010.
- Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. *Synthesis of Reactive(1) Designs.* In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2006.
- Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. *Synthesis of Reactive(1) Designs.* *JCSS, Special Issue in Honor of Amir Pnueli*.
- Shahar Maoz and Yaniv Sa'ar. *AspectLTL: An Aspect Language for LTL Specifications.* In *Proc. 10th Int. Conf. on Aspect-Oriented Software Development*, 2011..
- Ariel Cohen, Kedar S. Namjoshi, Yaniv Sa'ar, Lenore D. Zuck, and Katya I. Kisyova. *Parallelizing a Symbolic Compositional Model-Checking Algorithm.* In *Proc. 6th Int. Haifa Verification Conf.*, 2010.
- Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. *A Dash of Fairness for Compositional Reasoning.* In *Proc. 22nd Int. Conf. on Computer Aided Verification*, 2010.
- Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. *SPLIT: A Compositional LTL Verifier.* In *Proc. 22nd Int. Conf. on Computer Aided Verification*, 2010.
- Ittai Balaban, Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. *Verification of Multi-Linked Heaps.* *JCSS, Special Issue in Honor of Amir Pnueli*.
- Shahar Maoz and Yaniv Sa'ar. *Two-way Traceability and Conflict Debugging for AspectLTL Programs.* In *Proc. 11th Int. Conf. on Aspect-Oriented Software Development*, 2012. Submitted.
- Amir Pnueli and Yaniv Sa'ar. *All You Need is Compassion.* In *Proc. 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2008.