Yet Another Layered Approach to Validation Tools

FMT “Tool Architecture”

Observation at FMT in the Spring of 2000 (!):

- Abandonware: several (prototype) tools were left by PhD or MsC students for the group to maintain.
- Several PhD students were working on a state exploration tool for their own specific needs.
- Two new projects (Côte de Resyste and HaaST) in which the design and implementation of validation tools took prominent places.

- Working group to define a “common tool architecture” that could be (re)used by all tools.

Implementation language: C++.

Compiler Layer

Application Layer

Q: Why is the model checker SPIN so successful?
A: Because SPIN has not been developed by a PhD student.

Disclaimer: This work is less concerned with specific research goals. We just want to reuse our software tools.

A: Because SPIN has not been developed by a PhD student.
An Object-Oriented, Layered Approach to Model Checking Tools

These are the features that you want in your own model checker (at least)!

Remember: this all happened before the ‘Extreme Programming’ movement.

However, that period (i.e. the HaaST project) led to MoDeST & MoToR.

The effectiveness of these advanced and powerful options account for the popularity of SPIN.

Most (if not all) of these features are controlled using C compiler options.

But hard to maintain (spaghetti-code) and impossible to reuse.

The resulting verifier is fast and tuned to the maximum.

Project died a silent death.

Building all components required a lot of work.

Design was a compromise; no one really felt responsible to 'sell the design'.

It seemed (again) easier to build the isolated functionality from scratch.

Software engineers involved were ‘religious’ about their own code conventions and standards; they did not feel like changing their habits.

Remember: this all happened before the ‘Extreme Programming’ movement.

These features are controlled using C compiler options.

# ifndef FULLSTACK
# define MA
#endif

# ifdef DMA
# define NOREDUCE
# difficult
# ifndef NOREDUCE
# ifdef DMA
# define MA
# endif

# if NCORE>1 && defined(FULL_TRAIL)
// ... other pan files
// all functions

# ifdef BITSTATE
// ... other pan files
// all functions

• data types
• state vector definition
• tables

• automata for all processes (i.e. transitions)
• forward moves (C code for transitions)
• backward moves (C code for transitions)

The r

http://www.spinroot.com

Remember: this all happened before the ‘Extreme Programming’ movement.

However, that period (i.e. the HaaST project) led to MoDeST & MoToR.

The effectiveness of these advanced and powerful options account for the popularity of SPIN.

Most (if not all) of these features are controlled using C compiler options.

But hard to maintain (spaghetti-code) and impossible to reuse.

The resulting verifier is fast and tuned to the maximum.

Project died a silent death.

Building all components required a lot of work.

Design was a compromise; no one really felt responsible to 'sell the design'.

It seemed (again) easier to build the isolated functionality from scratch.

Software engineers involved were ‘religious’ about their own code conventions and standards; they did not feel like changing their habits.

Remember: this all happened before the ‘Extreme Programming’ movement.

However, that period (i.e. the HaaST project) led to MoDeST & MoToR.

The effectiveness of these advanced and powerful options account for the popularity of SPIN.

Most (if not all) of these features are controlled using C compiler options.

But hard to maintain (spaghetti-code) and impossible to reuse.

The resulting verifier is fast and tuned to the maximum.

Project died a silent death.

Building all components required a lot of work.

Design was a compromise; no one really felt responsible to 'sell the design'.

It seemed (again) easier to build the isolated functionality from scratch.

Software engineers involved were ‘religious’ about their own code conventions and standards; they did not feel like changing their habits.

Remember: this all happened before the ‘Extreme Programming’ movement.

However, that period (i.e. the HaaST project) led to MoDeST & MoToR.

The effectiveness of these advanced and powerful options account for the popularity of SPIN.

Most (if not all) of these features are controlled using C compiler options.

But hard to maintain (spaghetti-code) and impossible to reuse.

The resulting verifier is fast and tuned to the maximum.
MCKit

Model-Checking Kit

- 1-safe Petri-nets
- Automata + channels
- + shared memory

Glue between existing modelling languages and existing checkers.

Internal structure from higher layers is lost, which makes it (very) hard to re-use the internal structure (e.g. when writing counter-examples, or store states in a state store).

ClP (deadlock)
DSSZ (CTL, LTL)
LoLa (CTL, deadlock)
Mcsmodels (unfolding)
PEP (LTL, deadlock)
PRoD (CTL, LTL, deadlock)
SPIN (LTL, reachability)
SMV (CTL, deadlock)

http://www.fmi.uni-stuttgart.de/szs/tools/mckit/

Last website update: Sep 2004

IF Toolkit

http://www-verimag.imag.fr/~async/IF/

IF specifications

domain specific

IF specifications

IF Toolkit 10

Last website update: Jan 2005

http://www-verimag.imag.fr/~async/IF/

LTS

IF specifications

NIPS

http://www.cs.utwente.nl/~michaelw/nips/

Virtual Machine approach to state space generation: high-level modeling languages are translated to NIPS bytecode.

Promela
Assembly

compiler
compiler

NIPS bytecode

The NIPS VM can be used to generate the state space of a NIPS bytecode program.

Due to the black box nature of the NIPS VM, the reusability of the algorithms is low.

Motivation: model checkers are specialised.

- Reusing functionality requires model transformations.
- Most tools use their own formalism.
- Typically built from scratch.

Approach

- Implement functionality generically.
- Reusable functionality for different models.
- Focus on explicit-state model checking.

OO Framework

[Kattenbelt et al. 2007]
**Conceptual Architecture**

The model interface should provide sufficient functionality for any algorithm that we wish to implement.

"Generic layers can be defined for explicit state model checking* symbolic model checking* bounded model checking*...

We abstract from actual types by means of type parameterisation (e.g. generics, templates).

---

**Example: DFS**

Template arguments:

- `template <typename S, typename L, typename T>`
- `class BasicDepthFirstSearch`  

Private:

- `ExplicitStateModelInterface & s;`  
- `GoalCondition& t;`  
- `GenericLayer& g;`  

Public:

- `void dfs(S* & s, L* & l, T* & t)`

- `if (goalCondition(s))`  
- `if (goal(s))`

- `Statespace.insert(s);`  
- `while (t != 0)`

- `if (Statespace.find(t) == Statespace.end())`  
- `if (goalCondition(s))`  

- `s = goal(s);`  
- `t = goal(t);`

* The `GoalCondition` specifies what we are looking for in the DFS.

---

**Concrete Architecture**

Template:

- `template <typename S, typename L, typename T>`
- `class ExplicitStateModelInterface`

Public:

- `virtual S* getInitialState(T* tr) = 0;`
- `virtual T* getFirstTransition(S* s) = 0;`
- `virtual T* getNextTransition(T* tr) = 0;`
- `virtual T* getLabel(T* tr) = 0;`
- `virtual T* getTransition(S* s) = 0;`
- `virtual T* forgetTransition(T* tr) = 0;`

Generic layer:

- `Simulator Algorithms`
- `Testing Algorithms`
- `Verification Algorithms`

---

**More generic interfaces**

To make the search algorithms even more generic, an interface `SearchStrategy` is defined.

- The `Condition` specifies what we are looking for (e.g. deadlock, acceptance cycle, etc.).
- The `Action` specifies what to do when the condition is satisfied (e.g. always store the state, report this state, etc.).

---

*Note that all interfaces and algorithms are still generic; they are only defined in terms of `S`, `T`, and `L`.*
An Object-Oriented, Layered Approach to Model Checking Tools

Abstract...

Graph-based intermediate representation (cf. BOGOR).

- Reason: the dynamic nature of software: dynamic objects, dynamic process creation.
- A state is represented by a state graph:
  - data values and process instances are nodes
  - variables induce edges
- Heap (and thread) symmetry for ‘free’.
- Talk by Arend Rensink will give more information on using graphs as intermediate representation.

Prom+

- Prototype implementation in C++
- Prom+ language: small subset of Promela + dynamic objects
- Built on top of generic and abstract layer
- Results on (too) small benchmark suite:
  - memory: comparable with SPIN
  - time: three orders of magnitude slower than SPIN

SpinJ

- SpinJ (= Spin in Java) - re-implementation of SPIN in Java.
  - Goal: a well-designed, reusable model checking framework that can be used for experiments and reused for other input formalisms.
  - Question: Can we build a fast model checker in Java?

Features of initial version of SpinJ:

- full Promela (+ never claims)
- DFS (+ nested), BFS
- bisimulation, state compression
- partial order reduction
- state compression
**SUMO project (1)**

- Part of FMT Master Course “System Validation”
- Application and implementation of explicit state model checkers.
- SUMO project: develop a state space explorer for the modelling language SUMO, a subset of Promela. The explorer should check for safety properties.
- Input: system description in SUMO
- Output
  - no errors
  - error: deadlock / assertion violation + trace leading to error state
- Implementation language: Java.

**SUMO project (2)**

- The grade for the SUMO project depends on the relative ‘speed’ of the explorer.

<table>
<thead>
<tr>
<th>grade</th>
<th>status of implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>does not compile</td>
</tr>
<tr>
<td>≤ 5</td>
<td>does not work correctly on all test files</td>
</tr>
<tr>
<td>≥ 6</td>
<td>works correctly on all test files</td>
</tr>
<tr>
<td>6</td>
<td>30% slowest implementations</td>
</tr>
<tr>
<td>7</td>
<td>40% average implementations</td>
</tr>
<tr>
<td>8</td>
<td>30% fastest implementations</td>
</tr>
<tr>
<td>9</td>
<td>the fastest implementation</td>
</tr>
</tbody>
</table>

**Bonus points**

- 0.5 shortest counterexample
- 0.5 state compression
- 0.5 brave hashing + hash compaction
- 1.0 other Promela features (max +1.0)
- 1.5 partial order reduction
- 2.0 LTL model checking
Perhaps surprisingly, the implementation effort for the three approaches is more-or-less the same.

Conclusions

- **Prom**: original approach is pure, elegant and extendible. (but much too slow)
- **Spin**: Java implementation is fast enough. (without sacrificing elegance and reusability)

**Future work**
- **Prom**: analysis of bottlenecks of implementation.
- **Prom**: other generic layers (symbolic, bounded model checking).
- **Prom** and **Spin**: extend functionality within framework.
- ... plus make sure that the framework and tools do not get lost!