Compositional Software Model Checking

Dan R. Ghica
Queen’s University, Kingston, Canada

April 18, 2002
Outline of talk

• software model checking: methods, successes, challenges

• a model-checking-friendly semantics of software

• a model-checking-friendly specification logic
What is model checking?

- a formal verification technique

**model checking**: verifying if a system is a model for (i.e. satisfies) a logical formula (property);

**software model checking (SMC)**: applying model checking techniques to software.

- is an alternative to *automated proving* or *proof checking*
Why model checking

<table>
<thead>
<tr>
<th>Model checking</th>
<th>Proof checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>semantics-driven</td>
<td>syntax-driven</td>
</tr>
<tr>
<td>less user involvement</td>
<td>more user involvement</td>
</tr>
<tr>
<td>easier to automate</td>
<td>harder to automate</td>
</tr>
</tbody>
</table>
Who uses model checking?

Hardware verification:

- becoming common industrial practice

Software verification:

- Microsoft: SLAM
- AT&T: SPIN
- KSU (Honeywell, NASA): Bandera
SMC: the idea

Adaptation of hardware verification techniques:

- model a program as an automata-like system
- verify properties expressed in a temporal logic:
  - safety: a certain event never occurs
  - liveness: a certain event always occurs
SMC: successes

Especially suitable for small, flat programs:

- device drivers (Microsoft certified driver initiative)
- network protocols
- embedded software
SMC: main challenges†

SMC is great...
...but it could be better:

- local verification of program fragments
- source-level specification
- unity of programming and specification semantics
- compositional model checking

The semantic gap of SMC

A good semantic model of software:

- sound (all bugs reported) and complete (no false bugs)
- denotational (model program fragments)
- also a model of a Hoare-like programming logic

Current models of software used in SMC are none of the above.
Traditional semantic models of software

- operational:
  formalized interpreters, heavily syntactic, not denotational

- denotational:
  highly mathematical, unsuitable for automatic verification
An alternative semantics: game theoretic models

1910s: foundations of games theory: Zermelo

1950s: games in economics: van Neumann, Nash

1960s: games in logic: Lorenzen

1970s: games in mathematics: Conway

1990s: games in programming semantics:

Abramsky, Jagadeesan, Hyland, Ong
Hyland-Ong games for computation

Computation seen as a dialogue between a term (P–player) and its environment (O–opponent).

P wins if it can give an answer to O’s opening question.

P plays according to a strategy.

The strategy is the interpretation of the term.
Moves and their interpretation

<table>
<thead>
<tr>
<th></th>
<th>$O$</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$</td>
<td>request for output</td>
<td>request for input</td>
</tr>
<tr>
<td>$a$</td>
<td>input</td>
<td>output</td>
</tr>
</tbody>
</table>
A simple example

\[ f: \mathbb{N} \rightarrow \mathbb{N} \vdash f(1) + f(2) : \mathbb{N} \]

---

**Interpretation**

<table>
<thead>
<tr>
<th>(N → N) ⊢ N</th>
<th>O</th>
<th>q</th>
<th>Initial request for output</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>q</td>
<td></td>
<td>RI: value of f</td>
</tr>
<tr>
<td>O</td>
<td>q</td>
<td></td>
<td>RO: what is the argument of f?</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td></td>
<td>The system outputs 1</td>
</tr>
<tr>
<td>O</td>
<td>f(1)</td>
<td></td>
<td>The environment provides f(1)</td>
</tr>
<tr>
<td>P</td>
<td>q</td>
<td></td>
<td>RI: value of f</td>
</tr>
<tr>
<td>O</td>
<td>q</td>
<td></td>
<td>RO: what is the argument of f?</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td></td>
<td>The system outputs 2</td>
</tr>
<tr>
<td>O</td>
<td>f(2)</td>
<td></td>
<td>The environment provides f(2)</td>
</tr>
<tr>
<td>P</td>
<td>f(1) + f(2)</td>
<td></td>
<td>The system gives the final output.</td>
</tr>
</tbody>
</table>
A more complicated example

\[
f : \mathbb{N} \to \mathbb{N} \vdash f(f(1)) : \mathbb{N}
\]
A frustrating situation

Strategies: sets of justified sequences of moves with various combinatorial constraints (alternation, bracketing, visibility, innocence).

The stateful view is conceptually familiar and allows particular equivalences to be validated quite easily; the behavioural view is conceptually more sophisticated but allows proofs of general properties such as full abstraction.†

Regular language semantics

Basic idea: in a restricted language fragment (no higher order functions, tail recursion only) justification pointers are redundant and strategies can be modeled as sets of complete plays—which are regular languages!†

Consequence: a truly elementary denotational semantics!

The programming language

Data sets:
  booleans, bounded integers

Ground types:
  variables, expressions, commands;

Imperative features:
  assignment, branching, iteration, local variables;

Functional features:
  first-order functions (uniformly on all types), call-by-name;
Other languages and languages features

• call-by-value, arrays


• second-order functions

The regular-language model

Semantic valuation:

\[
\left[ \Gamma \vdash M : \theta \right] : \text{Term} \times \text{Environment} \rightarrow \text{RegularLanguage}
\]

Language constants:

\[
\left[ \text{skip} : \text{com} \right] u = \text{run} \cdot \text{done} \quad \left[ n : \text{int} \right] u = q \cdot n \quad \left[ \text{loop} : \text{com} \right] u = \emptyset
\]

Identifiers:

\[
\left[ c : \text{com} \vdash c : \text{com} \right] u = \begin{cases} 
\text{run} \cdot \text{run}^c \cdot \text{done}^c \cdot \text{done} & \text{if } c \notin \text{dom}(u) \\
\text{u}(c) & \text{if } c \in \text{dom}(u)
\end{cases}
\]
Some helpful notation

\[
[\Gamma \vdash E:\text{int}] u = \sum_n q \cdot ([\Gamma \vdash E:\text{int}]_n u) \cdot n
\]

\[
[\Gamma \vdash C:\text{com}] u = \text{run} \cdot ([\Gamma \vdash C:\text{com}] u) \cdot \text{done}
\]

Constants:

\[
(\text{skip}) u = \epsilon \quad (n)_n u = \epsilon \quad (n)_{n'}\neq n = \emptyset
\]

Identifiers:

\[
([e:\text{int} \vdash e:\text{int}]_n u = \begin{cases} q^e \cdot n^e & \text{if } e \notin \text{dom}(u) \\ u(x) & \text{if } e \in \text{dom}(u) \end{cases}
\]
Operators

Arithmetic:

\[(E \star F)_v u = \sum_{v_1 \star v_2 = v} (E)_{v_1} u \cdot (F)_{v_2} u\]

Imperative:

\[(C; C') u = (C') u \cdot (C') u\]

\[(\text{if } B \text{ then } C \text{ else } C':\text{com}) u = (B:\text{bool})_{tt} u \cdot (C:\text{com}) u + (B:\text{bool})_{ff} u \cdot (C':\text{com}) u\]

\[(\text{while } B \text{ do } C) u = (B)_{tt} u \cdot (C) u \ast (B)_{ff} u\]
A simple example

\[\Gamma \vdash \text{while true do } C \equiv \text{loop}\]

\[
(\text{while true do } C') u = (\text{true})_{tt} u \cdot (C') u)^* \cdot (\text{true})_{ff} u
\]

\[
= (\epsilon \cdot (C') u)^* \cdot \emptyset
\]

\[
= \emptyset = (\text{loop}) u
\]

Because: \((\text{true})_{tt} u = \epsilon, (\text{true})_{ff} u = \emptyset\).

Usual proof: fixed-point induction\(\dagger\).

Variables

\[ [V:\text{var}] u = \sum_n \text{read} \cdot (V)^r_n u \cdot n + \text{write}(n) \cdot (V)^w_n u \cdot \text{ok} \]

Assignment:
\[ (V := E) u = \sum_n (E)_n u \cdot (V)^w_n u \]

Dereferencing (reading):
\[ (!V : \text{int})_n u = (V : \text{var})^r_n u \]

\textbf{Obs:} no causal relation between read and write actions:
\[
\text{let } x \text{ be } y := !y + 1; y \text{ in } \cdots
\]
Local (block) variables

Local variables are the only ones guaranteed to be well behaved:

\[ \cdots \text{write}^x(n) \cdot \text{ok}^x \cdots \text{read}^x \cdot n^x \cdots \text{read}^x \cdot n^x \cdots \]

Interpretation of block variables:

\[ (\Gamma \vdash \text{int } x; C) \ u = \left( (\Gamma, x:\text{var} \vdash C) \ u \cap \tilde{\gamma}^x \right) \upharpoonright A^x \]

Where \( \gamma^x = \left( \sum_n \text{write}^x(n) \cdot \text{ok}^x \cdot (\text{read}^x \cdot n^x)^* \right)^* \)

- \( \tilde{\gamma} \) is broadening, \( A^x \) the alphabet of moves not tagged by \( x \),

- \( \upharpoonright \) is restriction.
An example

\[ \text{c:com} \vdash \text{int } x; \text{c} \equiv c \]

First proved using possible-worlds-style functor categories\(^\dagger\).

\[
\begin{align*}
(c: \text{com} \vdash \text{int } x; c) u &= (\{c: \text{com}, x: \text{var} \vdash c\} u \cap \overline{\gamma}^x) \upharpoonright A^x \\
&= (\text{run}^c \cdot \text{done}^c \cap \overline{\gamma}^x) \upharpoonright A^x \\
&= \text{run}^c \cdot \text{done}^c \\
&= (c: \text{com} \vdash c) u
\end{align*}
\]

Functions

- non-local functions

\[ [f : \text{comm} \rightarrow \text{comm}, c : \text{comm} \vdash f(c) : \text{comm}] \]
\[ = \text{run} \cdot \text{run}_f \cdot (\text{run}_f,^1 \cdot \text{run}_c \cdot \text{done}_c \cdot \text{done}_f,^1)^\ast \cdot \text{done}_f \cdot \text{done} \]

- abstraction, application
The semantic gap is bridged.

- algorithmic regular-language based model
- sound and complete (fully abstract)
- decidable (for most properties)
How can we apply this to model checking?†

- a programming logic based on the same semantic model

Then:

- local, source-level verification (and compositional reasoning)

- unity of assertion and programming languages

 Specification language

- assertions = boolean expressions + quantifiers
  ...can have computational effects!
  ...do not use an environment (no declarations in assertions)

- specifications = properties of assertions
  ...can only be true or false, no computational effects
A problem: variables in assertions

We have *bad variables*:

\[ x := a; \neg x = a \]

does not necessarily evaluate to \( tt \).

But we also have *bad expressions*:

let inc be \( x := \neg x + 1; \neg x \) in \( \cdots \)

then \( inc = inc, inc + inc = 2 \times inc \), etc, may not evaluate to \( tt \).

- Static reasoning becomes difficult.

- How to formulate meaningful assertions?
Yet side-effects in expressions are useful

- common low-level programming idiom

- signal special situations by setting flags

...and unavoidable

- overflow, division by zero, non-termination

We do not want to ban side-effects, just to control them.
A solution: stability

Impose *global constraints* on the behaviour of identifiers with regular language intersection:

\[
(\Gamma \vdash \nabla x: \theta. B: \text{assert}) = \left( (\Gamma, x: \theta \vdash B: \text{assert}) \cap \widetilde{\gamma}_\theta \right) \upharpoonright A^x
\]

The following assertions always evaluate to *tt*, i.e.:

\[
(\nabla x: \text{int}. x = x)_{ff} = \emptyset
\]
\[
(\nabla x: \text{int}. x + x = 2 \times x)_{ff} = \emptyset
\]
\[
(\nabla x: \text{var}. \nabla y: \text{int}. x := y; !x = y)_{ff} = \emptyset
\]
Semantics of stability

Inspired by the game-semantic model of local variables:

- \( \gamma_{\text{bool}}^e = \sum_{b \in \{tt,ff\}} (q^e \cdot b^e)^* \)

- \( \gamma_{\text{varbool}}^x = \sum_{b \in \{tt,ff\}} (\text{read}^x \cdot b^x)^* \cdot \left(\sum_{b \in \{tt,ff\}} \text{write}^x(b) \cdot \text{ok}^x \cdot (\text{read}^x \cdot b^x)^*\right)^* \)
Interpretation of specifications

- an environment-like parameter to track global constraints;
- a state-like parameter to track initial values ("states");

$$[\nabla x: \theta. S: \text{spec}] (g, w) \iff [S: \text{spec}] (g | x \mapsto \gamma^x_{\theta}, w | x \mapsto I^x_{\theta})$$

Atomic specification: $$[A: \text{spec}] (g, w) \iff$$

safety: $$G \cap W \cdot [A: \text{assert}] \cap A \cdot ff = \emptyset$$

liveness: $$G \cap W \cdot [A: \text{assert}] \cap A \cdot tt \neq \emptyset$$

for all $$W \in w(x_1) \cdots w(x_n)$$, where $$G = \bigcap_x g(x)$$. 
Connectives

Implication: $[S \Rightarrow S':\text{spec}] (g, w) \iff \forall w'. \forall x. w'(x) \subseteq w(x), [S:\text{spec}] (g, w')$ implies $[S':\text{spec}] (g, w')$

Conjunction: $[S \land S':\text{spec}] (g, w) \iff [S:\text{spec}] (g, w)$ and $[S':\text{spec}] (g, w)$
Semantics of specs (cont’d)

Validity:

\[ [S]_{\text{spec}} (\emptyset, \emptyset) \]

Hoare triple:

\[ \forall \bullet A\{P\}A' ::= \forall \bullet (A \Rightarrow P; A') \]

where \( \forall \) is \( \nabla x_0: \theta_0 \ldots \nabla x_n: \theta_n \).

With this interpretation specs have good logical properties.
Discharging stability quantifiers

\[ \nabla x : \sigma . \exists \cdot S \quad \exists \cdot \nabla_\sigma (P) \]

\[ \exists \cdot S[P/x] \]

and \( A \not\# P \) for all assertions \( A \) occurring in \( S \) containing \( x \).

- in this spec language, non-interference is a simple \textit{syntactic} side-condition;

- \( \nabla_\sigma (P) \) asserts that \( P \) is stable as a phrase.
Semantics of stable phrases

\([\nabla_{\text{int}}(E)](g) \iff \]

\[G \cap (W \cdot \lbrack E\rbrack_{v} \cdot \mathcal{P}_E(g) \cdot \lbrack E\rbrack_{v'}) = \emptyset, \quad v \neq v'\]

\[G \cap (W \cdot \lbrack E\rbrack) \neq \emptyset,\]

for all \(W \in A\). where

\[\mathcal{P}_E(g) = (A \setminus \{\text{write}(v)^x, \text{ok}^x \mid x \in \text{FV}(E), g(x) = \gamma_{\text{var}}^x\})^*\].

\textbf{Obs:} Stability of a phrase may be proven either inferentially or by model-checking.
Examples of stability

Stable:

\[ \nabla x : \text{int} \cdot x + 1 \]

\[ \nabla x : \text{var} \cdot x := !x + 1; 7 \]

Not stable:

\[ x + 3 \]

\[ \nabla x : \text{var} \cdot x := !x + 1; !x \]
Example: some valid axioms and inference rules

Frame:

\[ \Upsilon \bullet C; A \Rightarrow A^C \# A \]

Composition of effects:

\[ \Upsilon \bullet C; A \land (A \Rightarrow C'; A') \Rightarrow C; C'; A' \]

Procedural invariant:

\[ \begin{align*}
\Upsilon \bullet A \Rightarrow P_i; A & \quad \Upsilon \bullet \nabla_{\text{assert}}(A) \\
\Upsilon \bullet A \Rightarrow f(P_1, P_2, \ldots, P_n); A
\end{align*} \]

Reynolds-like inference rules for the language\(^\dagger\).

... except procedure specifications

For procedure specs, stability is too strong a constraint.

We use

- qualified stability quantifiers: $\nabla x/x_1, x_2 \ldots$

- effects quantifiers for non-local procedures: $A\{x\}A'$
Conclusion and future work

Major accomplishments:

- algorithmic model: model checking
- decidability: model checking
- soundness: compositional reasoning
Theoretical research work needed

- proof theory for generalized quantifiers
- complexity results
- extending the programming language
- from generalized to relativized quantifiers

\[ \nabla x. S \text{ vs. } \nabla(x) \Rightarrow S \]
But first objective

A VERIFICATION TOOL