Growing Solver-Aided Languages with ROSETTE

Emina Torlak & Rastislav Bodik

U.C. Berkeley

ExCAPE · June 10, 2013
solver-aided domain-specific language

Solver-aided DSL (SDSL)

Noun

1. A high-level language in which partially implemented programs can be executed, verified, debugged and synthesized with the aid of a constraint solver.
programming ...

specification

P(x) {
  ...
}

programming ...

```
P(x) {
  ...
}
```

formula, input/output pairs, traces, another program, ...
assume pre(x)
assert post(P(x))

P(x) {
  ...
}
programming with a solver-aided language

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```
solver-aided programming: code checking

Is there a valid input x for which P(x) violates the spec?

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

∃ x . pre(x) ∧ ¬post(P(x))

SAT/SMT solver

CBMC [Oxford], Dafny [MSR], Jahob [EPFL], Miniatur / MemSAT [IBM], etc.
solver-aided programming: code checking

Is there a valid input \( x \) for which \( P(x) \) violates the spec?

\[
\begin{align*}
\text{assume } & \text{pre}(x) \\
& P(x) \{ \\
& \quad \ldots \\
& \} \\
\text{assert } & \text{post}(P(x))
\end{align*}
\]

\[\exists x . \text{pre}(x) \land \neg \text{post}(P(x))\]

SAT/SMT solver

counterexample

model

\( x = 42 \)
solver-aided programming: localizing faults

Given $x$ and $x'$, what subset of $P$ is responsible for $P(x) \neq x'$?

```plaintext
assume pre(x)
P(x) {
    v = x + 2
    ...
}
assert post(P(x))
```

$pre(x) \land post(x') \land x' = P(x)$

SAT/SMT solver
Given $x$ and $x'$, what subset of $P$ is responsible for $P(x) \neq x'$?

```plaintext
assume pre(x)
P(x) {
    v = x + 2
    ...
}
assert post(P(x))
```

repair candidates

```
pre(x) \land post(x') \land x' = P(x)
```

SAT/SMT solver

MAXSAT/ MIN CORE
solver-aided programming: angelic execution

Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

$$\exists v. \text{pre}(x) \land \text{post}(P(x, v))$$

Kaplan [EPFL], PBnJ [UCLA], Skalch [Berkeley], Squander [MIT], etc.
Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

$$\exists v . \text{pre}(x) \land \text{post}(P(x, v))$$

$$v = 0, \ldots$$

trace

model

SAT/SMT solver

Kaplan [EPFL], PBnJ [UCLA], Skalch [Berkeley], Squander [MIT], etc.
solver-aided programming: synthesis

Replace ?? with expression e so that \( P_e(x) \) satisfies the spec on all valid inputs.

```plaintext
assume \( \text{pre}(x) \)
\( \text{P}(x) \) {
  \( v = ?? \)
  ...
}
assert \( \text{post}(\text{P}(x)) \)
```

\[ \exists e . \forall x . \text{pre}(x) \Rightarrow \text{post}(P_e(x)) \]

SAT/SMT solver
Replace ?? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

\[
\begin{align*}
\text{assume } & \quad \text{pre}(x) \\
\text{P(x)} & \quad \{ \\
\quad & \quad v = x - 2 \\
\quad & \quad \ldots \} \\
\text{assert } & \quad \text{post}(P(x))
\end{align*}
\]

\[
\exists e . \ \forall x . \ \text{pre}(x) \Rightarrow \text{post}(P_e(x))
\]
but building solver-aided languages is hard ...

Each new SDSL created by careful custom compilation to constraints, requiring years of training and experience.
a solver-aided framework for building SDSLs

Implement a library or an interpreter for your SDSL, and get a synthesizer, verifier, debugger and angelic oracle for programs in that SDSL.
a tiny solver-aided extension of racket ...

\[
\text{top-level-form} = \begin{cases} \text{general-top-level-form} \\ \text{(#expression expr)} \\ \text{(module id name-id} \\ \text{(#plain-module-begin} \\ \text{module-level-form ...}) \\ \text{(begin top-level-form ...)} \\ \text{(begin-for-syntax top-level-form ...)} \end{cases}
\]

\[
\text{module-level-form} = \begin{cases} \text{general-top-level-form} \\ \text{(#provide raw-provide-spec ...)} \\ \text{(begin-for-syntax module-level-form ...)} \end{cases}
\]

\[
\text{general-top-level-form} = \begin{cases} \text{expr} \\ \text{(define-values (id ...) expr)} \\ \text{(define-syntaxes (id ...) expr)} \\ \text{(#require raw-require-spec ...)} \end{cases}
\]

\[
\text{expr} = \begin{cases} \text{id} \\ \text{(#plain-lambda formals expr ...+)} \\ \text{(case-lambda (formals expr ...+)} \\ \text{(if expr expr expr)} \\ \text{(begin expr ...+)} \\ \text{(begin0 expr expr ...)} \\ \text{(let-values (((id ...) expr] ...)} \\ \text{expr ...+)} \\ \text{(letrec-values (((id ...) expr] ...)} \\ \text{expr ...+)} \\ \text{(set! id expr)} \\ \text{(quote datum)} \\ \text{(quote-syntax datum)} \\ \text{(with-continuation-mark expr expr expr)} \\ \text{(#plain-app expr ...+)} \\ \text{(#top . id)} \\ \text{(#variable-reference id)} \\ \text{(#variable-reference (#top . id))} \\ \text{(#variable-reference)} \end{cases}
\]

\[
\text{formals} = \begin{cases} \text{id ...} \\ \text{id ...+ . id} \end{cases}
\]

\[
\text{(define-symbolic id expr)}
\]

\[
\text{(assert expr)}
\]

\[
\text{(solve expr)}
\]

\[
\text{(verify expr)}
\]

\[
\text{(debug [expr] expr)}
\]

\[
\text{(synthesize}
\begin{cases}
\text{#:forall expr} \\
\text{#:guarantee expr}
\end{cases}
\]

\[
\text{Racket}
\]

\[
\text{ROSETTE}
\]
... with a symbolic evaluator and compiler

- debug
- solve
- verify
- synthesize

SDSL + program

ROSSETTE racket

transform, evaluate & compile to constraints

KODKOD

SAT solver
... with a symbolic evaluator and compiler

- **debug**
- **solve**
- **verify**
- **synthesize**

SDSL + program → Rosette racket → KODKOD

Transform, evaluate & compile to constraints

Easy to connect to another solver or backend, such as SMT or SynthLib.
... with a symbolic evaluator and compiler

- SDSL + program
- map solution to program level
- transform, evaluate & compile to constraints
- Easy to connect to another solver or backend, such as SMT or SynthLib.

RÚSETTE
racket

KODKOD
SAT solver

debug
solve
verify
synthesize
... with a symbolic evaluator and compiler

Easy to connect to another solver or backend, such as SMT or SynthLib.
web, spatial programming, superoptimization
websynth: web scraping by demonstration

- Web scraping usually done with fragile, hand-written scripts using regex.
- Websynth synthesizes a web scraper from a URL and a few examples of desired data, using a declarative SDLS (XPaths).
- Implemented by two undergraduate students in a few weeks.
- 4 seconds in total to construct a 1100+ node DOM, synthesize the XPath, and return the data.
partitioning code & data for a low power chip

Instructions/Second vs Power

GreenArrays GA144 Processor

~100x

Figure by Per Ljung
partitioning code & data for a low power chip

Instructions/Second vs Power

GreenArrays GA144 Processor

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core
partitioning code & data for a low power chip

**Instructions/Second vs Power**

![Comparison chart showing Instructions/Second vs Power for various processors. The chart includes data points for different processors, with GreenArrays GA144 Processor highlighted, showing a significant improvement (≈100x).](chart.png)

**GreenArrays GA144 Processor**

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

---

Manual function partitioning:
break functions up into a pipeline with a few operations per core.

![Manual function partitioning illustration.](partitioning.png)

**Figure by Per Ljung**

**GreenArrays GA144 Processor**

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

---

**Instructions/Second vs Power**

![Comparison chart showing Instructions/Second vs Power for various processors. The chart includes data points for different processors, with GreenArrays GA144 Processor highlighted, showing a significant improvement (≈100x).](chart.png)

**GreenArrays GA144 Processor**

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

---

Manual function partitioning:
break functions up into a pipeline with a few operations per core.

![Manual function partitioning illustration.](partitioning.png)

**Figure by Per Ljung**
partitioning code & data for a low power chip

High-Level Program

Partitioner

Per-core High-Level Programs

Code Generator

Per-core Optimized Machine Code

New Programming Model

New Approach

Using Synthesis

Mangpo Phothilimthana and Nishant Totla (first year graduate students)
partitioning code & data for a low power chip

High-Level Program

Per-core High-Level Programs

Partitioner

Per-core Optimized Machine Code

Code Generator

New Programming Model

New Approach

Using Synthesis

Mangpo Phothilimthana and Nishant Totla (first year graduate students)
partitioning code & data for a low power chip

one iteration of MD5 hash

optimal code partitioning by the synthesizer

256-byte mem per core
initial data placement specified

512-byte mem per core
different initial data placement

512-byte mem per core
same initial data placement
component-based synthesis of loop free code

(define-fragment (fast-max x y)
  #:ensures
  (lambda (x y result)
    (= result (if (> x y) x y)))

#:library
  (bvlib [{bvneg bvge bvand} 1]
     [{bvxor} 2]))

▶ A clever algorithm for synthesis of bitvector programs by Gulwani et al, PLDI’11.
▶ Produces a loop-free program from a multi-set of instructions and a functional spec (superoptimization).
▶ Encoding cannot be reproduced with tools using hardwired, general-purpose synthesis algorithms.
▶ 600 lines of Rosette, and performance matches the published algorithm.
thank you!

code checking
angelic execution
debugging
synthesis