Automating System Configuration

Nestan Tsiskaridze

07/01/2021

Joint work with:

**AHA Faculty**: Clark Barrett, Mark Horowitz

**Lake Team**: Qiaoyi Liu, Kavya Sreedhar, Maxwell Strange

**Pono Team**: Ahmed Irfan, Florian Lonsing, Makai Mann

**Boolector Team**: Aina Niemetz, Mathias Preiner
Configurable Systems

Configurable systems are parameterized to increase their flexibility or functionality.

Such systems can be used in various contexts or applications.

Goal: Choose the appropriate parameter values for the context or application in which the system will be used.
Configurable Systems

Why configurable systems?

• Make designs more **widely applicable** and **re-usable**.

• Systems can be configured at **design time**, **setup time**, or **during normal operation**.

• Most hardware and software systems require **some degree of configuration**.

• Often used when **integrating decoupled parts** of a system, including integrating software and hardware.

• **Agile design approaches** require **rapid integration** of changing parts of a system while continuously maintaining correct end-to-end functionality.
Configurable Systems

Why automation?

- Today’s systems increase in **scale** and **complexity**.

- **Manual** configuration is **error-prone** and may even be **impossible**, depending on how frequently the systems need to be reconfigured.

- **Automation is needed**.

- Especially useful for an agile design process.
Formal System Model

Symbolic Transition System is built by translating another representation:

E.g. a program, a mathematical model, a hardware description, etc.
Symbolic Transition System

- **States** are made up of state variables $\nu \in V$;

- A **state** is an assignment to all variables;

- A **Symbolic Transition System** is a tuple $S := \langle V, I, T \rangle$
  - $V$ – a finite set of state variables;
  - $I(V)$ – a formula denoting the initial states of $S$;
  - $T(V, V')$ – a formula denoting a transition relation, $V'$ denotes next state variables.
Symbolic Transition System

- An **execution of length** $k$ is a sequence of states $\pi := s_0 \, s_1 \ldots s_{k-1}$ such that:
  - the first state $s_0$ respects $I$;
  - every adjacent pair $(s_{i-1}, s_i), \ 0 \leq i < k$, respects $T$.

- An **unrolling of length** $k$ of a symbolic transition system is a formula that captures an execution of length $k$ by:
  - introducing fresh timed variables $V_{@i}$;
  - creating copies of the transition relation $T(V_{@i}, V_{@(i + 1)})$.

\[
unroll(S, k) := I(V_{@0}) \land \bigwedge_{i=0}^{k-1} T(V_{@i}, V_{@(i + 1)})
\]
Formal System Model

\[ S \coloneqq \langle V, I, T \rangle \]

\[ V_{in} \cup V_{out} \cup V_{conf} \subseteq V. \] Pairwise intersections of these sets may either be empty or non-empty.

\[ V_{conf} \neq \emptyset \]

\[ V_{in} \text{ do not appear in } I(V) \text{ and } V'_{in} \text{ do not appear in } T. \]
Formal System Model

- \( P \) — an **application-supplied input-output property**, or an **input-output specification**, a formula capturing an input-output relationship for \( k \) transitions:

\[
P (V_{in}@0, ..., V_{in}@k, V_{out}@0, ..., V_{out}@k).
\]

**Example**

For some constant values of inputs \( c_{in}^i \) and constant values of outputs \( c_{out}^i \):

\[
P := \bigwedge_{i=0}^{k-1} (V_{in}@i = c_{in}^i \land V_{out}@i = c_{out}^i),
\]

- Typically, we want a **configuration constancy constraint** to hold:

\[
conf (V_{conf}, k) := \bigwedge_{i=0}^{k-1} (V_{conf} @(i + 1) = V_{conf}@i)
\]
A **Configuration Problem** is a tuple: \( CP := \langle S, k, V_{in}, V_{out}, V_{conf}, P \rangle \)

A **configuration** \( C \) is an assignment to the variables in \( V_{conf} \).

**Configuration finding problem:**

Given a \( CP \), find a configuration \( C \) for \( S \) such that \( S \) satisfies \( P \) with configuration \( C \).

We reduce to **satisfiability checking** of the **configuration formula**:

\[
\varphi(CP) := \text{unroll}(S, k) \land \text{conf}(V_{conf}, k) \land P(V_{in}\,\!@0, \ldots, V_{in}\,\!@k, V_{out}\,\!@0, \ldots, V_{out}\,\!@k).
\]
Configuration Problem Example

(simple ALU)

Let $S := \langle x: \text{int}, a: \text{int}, c: \text{Bool}, \ x = 0, \ x' = \text{ite}(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, V_{out} = \{x\}, V_{con_f} = \{c\}$.

Two ways to configure $S$:

- **always adds** the current input to the current state variable,
- **always subtracts** the current input from the current state variable.
Configuration Problem Example

(simple ALU)

Let $S := \langle x: \text{int}, a: \text{int}, c: \text{Bool}, x = 0, \ x' = \text{ite}(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, V_{out} = \{x\}, V_{conf} = \{c\}$.

Consider an input-output relation for $k = 2$:

$P_1(a@0, a@1, a@2, x@0, x@1, x@2): = a@0 = 1 \land a@1 = 1 \land a@2 = 1 \land x@0 = 0 \land x@1 = 1 \land x@2 = 2$

Check satisfiability of:

$\text{unroll}(S, 2) \land \text{conf}(c@0, c@1, c@2) \land P_1(a@0, a@1, a@2, x@0, x@1, x@2)$.

$x@0 = 0 \land$
$x@1 = \text{ite}(c@0, x@0 + a@0, x@0 - a@0) \land$
$x@2 = \text{ite}(c@1, x@1 + a@1, x@1 - a@1) \land$
$c@1 = c@0 \land c@2 = c@1 \land$
$a@0 = 1 \land a@1 = 1 \land a@2 = 1 \land x@0 = 0 \land x@1 = 1 \land x@2 = 2$

Satisfiable when $c@0 = \text{True}$!
Configuration Problem Example

(simple ALU)

Let $S := \langle x: \text{int}, a: \text{int}, c: \text{Bool}, \ x = 0, \ x' = \text{ite}(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, \ V_{out} = \{x\}, \ V_{conf} = \{c\}$.

Consider an input-output relation for $k = 2$:

$P_2(a@0, a@1, a@2, x@0, x@1, x@2) := a@0 = 1 \land a@1 = 1 \land a@2 = 1 \land x@0 = 0 \land x@1 = 1 \land x@2 = 0$

Check satisfiability of:

$\text{unroll}(S, 2) \land \text{conf}(c@0, c@1, c@2) \land P_2(a@0, a@1, a@2, x@0, x@1, x@2)$.

$\begin{align*}
  x@0 &= 0 \land \\
  x@1 &= \text{ite}(c@0, x@0 + a@0, x@0 - a@0) \land \\
  x@2 &= \text{ite}(c@1, x@1 + a@1, x@1 - a@1) \land \\
  c@1 &= c@0 \land c@2 = c@1 \land \\
  a@0 &= 1 \land a@1 = 1 \land a@2 = 1 \land x@0 = 0 \land x@1 = 1 \land x@2 = 0
\end{align*}$

Unsatisfiable!
Configuration Solving Framework (Basic) Scheme

Input: $CP$

Configuration Solving Framework
Construct Formula

Output: $\varphi$

SMT Solver

Yes

Output: A Correct Configuration $C$

No

Output: Not configurable
Coarse-Grained Reconfigurable Architecture (CGRA)

Input:
- @0
- @1
- @2
- data_in
- valid_in

CGRA Memory Tile
- MEM
- Configuration Registers

Output:
- @0
- @1
- @2
- data_out
- valid_out
Btor2 format allows specifying word-level model checking problems.

**Pono**: our Performant, Adaptable, and Extensible SMT-based Model Checker.

We use **Boolector** as our underlying SMT solver.
<table>
<thead>
<tr>
<th>Design</th>
<th>Size (Σ bw)</th>
<th>FF</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>34 980</td>
<td>34 998</td>
<td>164 696</td>
</tr>
<tr>
<td>agg</td>
<td>1 354</td>
<td>1 372</td>
<td>19 676</td>
</tr>
<tr>
<td>tb</td>
<td>1 108</td>
<td>1 126</td>
<td>18 538</td>
</tr>
<tr>
<td>sram</td>
<td>33 694</td>
<td>33 712</td>
<td>150 750</td>
</tr>
</tbody>
</table>

**Identity Stream**

2x Intel Xeon E5-2620 v4 2.10GHz 8-core 128GB.

**Timeout:** 20 min

**Memout:** 100 GB.
Scalability Bottlenecks

Two main bottlenecks:

• Size of the transition system $S$;

• Length of the input sequence $k$ — number of transitions in $P$, size of the formula $unroll(S, k)$. 
Modular Decomposition

Let $CP := \langle S, k, V_{in}, V_{out}, V_{conf}, P \rangle$ with $S := \langle V, I, T \rangle$.

$(CP_1, CP_2)$ is a decomposition of $CP$ into two configuration problems:

$CP_i := \langle S_i, k, V_{in}^i, V_{out}^i, V_{conf}^i, P_i \rangle$ with $S_i := \langle V_i, I_i, T_i \rangle$ for $i = 1, 2$

if:

(i) $T_1 (V_1, V'_1) \land T_2 (V_2, V'_2) \Rightarrow T (V, V')$;

(ii) $I_1 (V_1) \land I_2 (V_2) \Rightarrow I (V)$;

(iii) $P_1 \land P_2 \Rightarrow P$. 
Modular Decomposition

Given a candidate decomposition \((CP_1, CP_2)\) of a configuration problem \(CP\), procedure \textbf{SOLVEMODULAR} attempts to solve \(CP\) by solving \(CP_1\) and \(CP_2\).

\begin{algorithm}
\caption{Modular configuration finding.}
\textbf{Procedure} \textbf{SOLVEMODULAR} \\
\textbf{Input:} \((CP_1, CP_2)\) a decomposition of \(CP\). \\
\textbf{Output:} a pair \((r, C)\) where if \(r = \text{sat}\), then \(C\) is a configuration of \(S\)
1: \(\phi_1 := \text{MAKECP}(CP_1)\) \\
2: \((r, I_1) := \text{SOLVE}(\phi_1)\), \\
3: if \(r = \text{sat}\) then \\
4: \(\phi_2 := \text{MAKECP}(CP_2) \land \text{GETABDUCT}(\phi_1, I_1)\) \\
5: \((r, I) := \text{SOLVE}(\phi_2)\) \\
6: end if \\
7: \text{return} \((r, I^{\text{conf}})\)
\end{algorithm}

\textbf{MAKECP} constructs the configuration formula for a \(CP\).

\textbf{SOLVE} invokes a solver to check the satisfiability of the configuration formula.

\textbf{GETABDUCT} returns an \textit{abduct} formula \(\psi\) of \(\phi_1\): all interpretations that satisfy \(\psi\) also satisfy \(\phi_1\).

\textbf{Goal}: use information in \(I_1\) to find a simple formula \(\psi\).
We search for a set of sub-terms in \(\phi_1\) such that, if we constrain them to be equal to their values in \(I_1\), this ensures that \(\phi_1\) is satisfied.
Modular Decomposition

**Theorem.** *(Soundness)*

If \((CP_1, CP_2)\) is a decomposition of a configuration problem \(CP\), and \textsc{SolveModular} \((CP_1, CP_2)\) returns a pair \((\text{sat}, C)\), then \(C\) is a correct configuration of \(CP\).

---

**Algorithm 1** Modular configuration finding.

```plaintext
Procedure \textsc{SolveModular}
Input: \((CP_1, CP_2)\) a decomposition of \(CP\).
Output: a pair \((r, C)\) where if \(r = \text{sat}\), then \(C\) is a configuration of \(S\)

1: \(\phi_1 := \text{MAKECP}(CP_1)\)
2: \((r, I_1) := \text{SOLVE}(\phi_1)\),
3: \text{if } r = \text{sat} \text{ then}
4: \(\phi_2 := \text{MAKECP}(CP_2) \land \text{GETABDUCT}(\phi_1, I_1)\)
5: \((r, I) := \text{SOLVE}(\phi_2)\)
6: \text{end if}
7: \text{return } (r, {}^V_{\text{conf}})
```

**Note:** If \textsc{SolveModular} returns \(r = \text{unsat}\), this does not (in general) imply that \(CP\) is unconfigurable.

It may be that the particular decomposition fails, or the particular solution found for \(CP_1\) is at fault.
Modular Decomposition in Practice

In practice, the algorithm works well when the decomposition separates a module into two largely independent parts.

A modular decomposition of system $S$ into systems $S_1$ and $S_2$.

$V_{in}$ and $V^1_{conf}$ are the output and the configuration variables of $S_1$.

$V^2_{in}$ and $V^2_{conf}$ are the input and the configuration variables of $S_2$.

$V_{conf} \subseteq V^1_{conf} \cup V^2_{conf}$. 
Memory Tile Architecture
<table>
<thead>
<tr>
<th>Design</th>
<th>Size (Σ bw)</th>
<th>FF</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>34 980</td>
<td>34 998</td>
<td>164 696</td>
</tr>
<tr>
<td>agg</td>
<td>1 354</td>
<td>1 372</td>
<td>19 676</td>
</tr>
<tr>
<td>tb</td>
<td>1 108</td>
<td>1 126</td>
<td>18 538</td>
</tr>
<tr>
<td>sram</td>
<td>33 694</td>
<td>33 712</td>
<td>150 750</td>
</tr>
</tbody>
</table>

Identity Stream

2x Intel Xeon E5-2620 v4
2.10GHz 8-core 128GB.

Timeout: 20 min
Memout: 100 GB.
Configuration Optimization Problem

- Human-readable configurations are particularly important for users;

- Humans tend to generate the simplest solutions for simplest application-specific scenarios;

- Producing simple solutions is often analogous to minimizing some metric when finding solutions. E. g.: readable solutions might minimize the value of some objective term representing the simplicity of a desired scenario;

- Can we produce configurations that are best according to some metric?
Configuration Optimization Problem

An optimization problem $\mathcal{OP}$ is a tuple $\langle t, A, \preceq, \phi, \mathcal{O} \rangle$ where:

- $t$ is an objective term to optimize of sort $\sigma$;
- $A$ is a set and $\preceq$ is a total order over $A$.
- $\phi$ is a formula to satisfy; and
- $\mathcal{O} \in \{min, max\}$ is the optimization objective.

$\mathcal{I}$ is a solution to $\mathcal{OP}$ if $\sigma^\mathcal{I} = A$, $\mathcal{I} \models \phi$, and for any $\mathcal{I}'$, such that $\sigma^{\mathcal{I}'} = A$ and $\mathcal{I}' \models \phi$:

$$
(O = min \rightarrow t^\mathcal{I} \preceq t^{\mathcal{I}'}) \land (O = max \rightarrow t^{\mathcal{I}'} \preceq t^\mathcal{I}).
$$
Configuration Optimization Problem

A multi-objective optimization problem $\mathcal{MOP}$ is a finite sequence of optimization problems $\{\mathcal{OP}_1, \ldots, \mathcal{OP}_n\}$ over the same formula $\phi$, where

- $\mathcal{OP}_i := \langle t_i, A_i, \preceq_i, \phi, \mathcal{O}_i \rangle$ and
- $t_i$ is of sort $\sigma_i$ for $i \in [1, n]$.

$\mathcal{I}$ is a solution to $\mathcal{MOP}$ if $\sigma_i^\mathcal{I} = A_i$, $\mathcal{I} \models \phi$, and for any $\mathcal{I}'$, such that $\sigma_i^{\mathcal{I}'} = A_i$ and $\mathcal{I}' \models \phi$, either:

(i) $t_i^\mathcal{I} = t_i^{\mathcal{I}'}$ for all $i \in [1, n]$; or
(ii) for some $j \in [1, n]$, $t_i^\mathcal{I} = t_i^{\mathcal{I}'}$ for all $i \in [1, j)$, and

\[
(O_j = \min \rightarrow t_j^\mathcal{I} \preceq_j t_j^{\mathcal{I}'} \land \neg (O_j = \max \rightarrow t_j^{\mathcal{I}'} <_j t_j^\mathcal{I})),
\]

where $\prec$ is the strict total order associated with $\preceq$. 
Optimization-Assisted Configuration Solver

$\phi'$ is a conjunction of the configuration formula and the optional verification properties.
Optimize Memory Tile Configurations

The addressors and accessors use affine sequence generators to generate sequences of values for reading and writing.

4 types of configuration variables in streaming memory controllers:

They manage loop-nests and address calculations.

- Dimensionalities
  control the structure

- Ranges

- Strides
  control particular address

- Starting Addresses

Goal: find an addressing pattern for the simplest looping.
Minimize the *dim* variables – fewer nested loops and fewer loop counters, simpler solutions.

Prioritize minimizing *dim* variables controlling writes over those controlling reads – lower write complexity leads to lower read complexity.

A multi-objective optimization problem:

\[
\mathcal{OPT}_1 := \{\mathcal{OPT}_1, \mathcal{OPT}_w^1, \ldots, \mathcal{OPT}_w^{d_w}, \mathcal{OPT}_r^1, \ldots, \mathcal{OPT}_r^{d_r}\}:
\]

\[
\mathcal{OPT}_1 := \langle \sum_i \text{dim}_i, A_{BV}, \preceq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d],
\]

\[
\mathcal{OPT}_w^i := \langle \text{dim}_w^i, A_{BV}, \preceq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d_w]
\]

\[
\mathcal{OPT}_r^i := \langle \text{dim}_r^i, A_{BV}, \preceq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d_r]
\]

*A_{BV} – is the domain of bit-vectors;*

*\preceq_{BV} – the usual total order on bit-vector values;*

*d – the number of affine sequence generators in the module;*

*\text{dim}_i \text{ for } i \in [1; d] – all of the *dim* variables in the module;*

*\text{dim}_w^i \text{ for } i \in [1; d_w] – write *dim* variables;*

*\text{dim}_r^i \text{ for } i \in [1; d_r] – read *dim* variables, \(d_w + d_r = d;\)*

*\phi – the configuration formula.*
Minimize the products of the *range* configuration variables in each loop-nest structure – eliminate unnecessary reads and writes to the memory.

An optimization problem:

$$\mathcal{OP}_2 := \langle \sum_{i=0}^{d-1} \prod_{j=0}^{\text{dim}_i-1} \text{ranges}_i[j], A_{BV}, \preceq_{BV}, \phi, \text{min} \rangle$$

$$\sum_{i=0}^{d-1} \prod_{j=0}^{\text{dim}_i-1} \text{ranges}_i[j]$$ – the aggregate number of reads or writes that occur to a particular memory.
Optimization Objective 3

Minimize stride variables – avoid generating configurations using unnecessarily large addresses.

An optimization problem:

$$\mathcal{OP}_3 := \langle \sum_i \text{strides}_i, A_{BV} \preceq_{BV} \phi, \text{min} \rangle$$
Optimization Objective 4

Minimize offset configuration variables in addressor modules – prevents unnecessary offsets, improves the readability.

Note: values of offset variables in the accessors are fixed by the application.

An optimization problem:

$$\mathcal{OP}_4 := \langle \Sigma_i \text{offset}_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle$$
A multi-objective optimization problem for finding human-readable configurations:

\[ \text{MOP} \mathcal{H} := \{ \text{MOP}_1, \text{MOP}_2, \text{MOP}_3, \text{MOP}_4 \} \]

with:

- \[ \text{MOP}_1 := \{ \text{OP}_1, \text{OP}_w^1, \ldots, \text{OP}_w^{d_w}, \text{OP}_r^1, \ldots, \text{OP}_r^{d_r} \} : \]
  \[ \text{OP}_1 := \langle \sum_i \dim_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d], \]
  \[ \text{OP}_w^i := \langle \dim_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d_w] \]
  \[ \text{OP}_r^i := \langle \dim_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \text{ for } i \in [1, d_r] \]

- \[ \text{OP}_2 := \langle \sum_{i=0}^{d-1} \prod_{j=0}^{\dim_i-1} \text{ranges}_i[j], A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \]

- \[ \text{OP}_3 := \langle \sum_i \text{strides}_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \]

- \[ \text{OP}_4 := \langle \sum_i \text{offset}_i, A_{BV}, \leq_{BV}, \phi, \text{min} \rangle \]
Stencil Applications

• **Identity** – simply streams the input back out in the same order. Useful as a baseline test, can also be used to implement a fixed delay on a stream.

• **3x3 Convolution** – multiplies a 3x3 sliding image window by a 3x3 kernel of constant values. Used in a variety of image processing applications.

• **Cascade** — implements a pipeline with two convolution kernels executed in sequence. Requires configuration of two memory tiles (conv and hw).

• **Harris** – a corner detection algorithm that can be used to infer image features. Requires configuration of five different memory tiles (cim, lxx, lxy, lyy, and pad).
<table>
<thead>
<tr>
<th>Design</th>
<th>Size (Σ bw)</th>
<th>FF</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>34 980</td>
<td>34 998</td>
<td>164 696</td>
</tr>
<tr>
<td>agg</td>
<td>1 354</td>
<td>1 372</td>
<td>19 676</td>
</tr>
<tr>
<td>tb</td>
<td>1 108.</td>
<td>1 126</td>
<td>18 538</td>
</tr>
<tr>
<td>sram</td>
<td>33 694</td>
<td>33 712</td>
<td>150 750</td>
</tr>
</tbody>
</table>

2x Intel Xeon E5-2620 v4 2.10GHz 8-core 128GB.

Timeout: 4000 sec
Memout: 100 GB.
Summary

• Our new approach provides a general framework for automatically configuring systems representable as transition systems.

• Key contributions include the ability to leverage modularity and the use of optimization.

• Optimal configurations are more human-understandable.

• It works well for our CGRA memory tile.

• Both modularity and optimization can improve scalability:

• These results suggest that modular configuration with optimization may be the best strategy in practice.