Automating System Configuration

Nestan Tsiskaridze ID, Maxwell Strange ID, Makai Mann ID, Kavya Sreedhar ID, Qiaoyi Liu ID,
Mark Horowitz ID, Clark Barrett ID

Stanford University, Stanford, CA 94305, USA

E-mail: {nestan, mstrange, makaim, skavya, joeyliu}@stanford.edu, horowitz@ee.stanford.edu, barrett@cs.stanford.edu

Abstract—The increasing complexity of modern configurable systems makes it critical to improve the level of automation in the process of system configuration. Such automation can also improve the agility of the development cycle, allowing for rapid and automated integration of decoupled workflows. In this paper, we present a new framework for automated configuration of systems representable as state machines. The framework leverages model checking and satisfiability modulo theories (SMT) and can be applied to any application domain representable using SMT formulas. Our approach can also be applied modularly, improving its scalability. Furthermore, we show how optimization can be used to produce configurations that are best according to some metric and also more likely to be understandable to humans. We showcase this framework and its flexibility by using it to configure a CGRA memory tile for various image processing applications.

I. Introduction

In systems engineering, the *system configuration* problem arises when systems are parameterized to increase their flexibility or functionality. It refers to the problem of choosing the appropriate parameter values for the context or application in which the system will be used. Most hardware and software systems, including hardware IPs, operating systems, networks, servers, and data centers, require some degree of configuration. The need for configuration also often arises when integrating decoupled parts of a system, including integrating software and hardware.

The difficulty of the system configuration problem has been gradually growing as systems increase in scale and complexity. In particular, in an effort to make designs more widely applicable and re-usable, there has been an increasing use of hardware that is configurable, not only at design time or setup time, but even during normal operation. Manual configuration of such systems is error-prone and may even be impossible, depending on how frequently the systems need to be reconfigured.

Automation of the configuration problem can also be beneficial during the system design process. In particular, it obviates the need for new hand-coded configuration files every time some configurable component changes. Increased automation of such steps supports a move towards more agile design processes. Agile approaches typically require the ability to rapidly and (largely) automatically integrate changing parts of a system while continuously maintaining correct end-to-end functionality. Having design blocks that are flexibly configurable aids this effort, as does the ability to automate the configuration.

A potential disadvantage of automated configuration is that it could lead to an increase in the opacity of the overall system. Hand-written configurations can be documented and explained to allow for easier understandability and maintainability. Thus, an additional goal when automating configuration should be to produce results that are comprehensible to humans and that can be easily reviewed and maintained.

In this paper, we present a general framework for automated system configuration. It provides a flexible approach for solving the configuration problem for systems composed of software, hardware, or both. The systems are modeled using transition systems, where transition formulas can use the full expressive power of SMT-LIB [1], the language used by satisfiability modulo theories (SMT) [2] solvers. The framework provides a systematic approach to facilitate fully automated or automation-guided system configuration. It is well-suited for both stand-alone designs and for designs with multiple configurable parts. For the latter, it is especially useful during system integration and rapid development.

The main contributions of this paper are:

- We introduce a "programming by example" approach for formalizing common input-output specifications. In an exact formulation of the configuration problem, the input-output specification would need to universally quantify over the input variables. Our approach avoids the need for quantifiers.
- We propose a new modular approach for configuration finding in a general SMT setting that makes use of abduction.
- We show how to leverage optimization to obtain human-readable configurations.
- We present a case study—automated configuration of a memory tile in the context of an agile hardware design project targeting image processing applications.

The remainder of the paper is organized as follows. Section II presents background and notation. Section III formalizes the configuration solving problem and introduces our framework, including some extensions and limitations. In Section IV, we show how optimization techniques can be integrated into the approach, both for the purpose of improving performance as well as for improving human readability, and we discuss a few additional extensions of the framework. In Section V we present a case study, giving the details of a specific system design and showing how our framework can be applied. Experimental results for this case study are

then reported in Section . We survey the related work in Section and conclude in Section .

II. BACKGROUND

We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, formula, and interpretation. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, i.e., the *models* of $\mathcal{T}$. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* $\mathcal{T}$ if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. We define $\models_{\mathcal{T}}$ over Σ-formulas: if φ and ψ are Σ-formulas, then $\varphi \models_{\mathcal{T}} \psi$ if all interpretations which satisfy φ also satisfy ψ. In this case, we also call φ an *abduct* of ψ under $\mathcal{T}$. For generality, we assume an arbitrary but fixed background theory $\mathcal{T}$ (which could be a combination of theories) with signature Σ and an infinite set $\mathcal{X}$ of variables. We will assume that all terms and formulas are Σ-terms and Σ-formulas whose free variables are in $\mathcal{X}$, that entailment is entailment modulo $\mathcal{T}$, and that interpretations are $\mathcal{T}$-interpretations that assign every variable in $\mathcal{X}$.

Given an interpretation $\mathcal{I}$, a variable assignment s over a set of variables V is a mapping that assigns each variable $v \in V$ of sort σ to an element of $\sigma^{\mathcal{I}}$, denoted v^s. The assignment over V *induced* by an interpretation $\mathcal{I}$ (i.e., the assignment that maps each variable in V to its interpretation in $\mathcal{I}$) is denoted $\mathcal{I}^V$. The assignment s restricted to the domain $U \subseteq V$ is denoted by s^U. We write $\mathcal{I}[s]$ for the interpretation that is equivalent to $\mathcal{I}$ except that each variable $v \in V$ is mapped to v^s. We write $f \circ g$ for functional composition, i.e., $f \circ g(x) = f(g(x))$.

Satisfiability Modulo Theories (SMT). Satisfiability Modulo Theories [2] is an extension of the Boolean satisfiability (SAT) problem to satisfiability in first-order theories. SMT solvers combine the Boolean reasoning of a SAT solver with specialized theory solvers to check satisfiability of many-sorted first-order logic formulas. Some examples of commonly supported theories are: fixed-width bit-vectors, uninterpreted functions, linear arithmetic, and arrays. In our case study, we utilize fixed-width bit-vectors for modeling a hardware design.

Symbolic Transition Systems.

A symbolic transition system (STS) $\mathcal{S}$ is a tuple $\mathcal{S} := \langle V, I, T \rangle$, where V is a finite set of state variables (possibly of different sorts), $I(V)$ is a formula denoting the initial states of the system, and $T(V, V')$ is a formula expressing a transition relation, with V' defined as follows. Let $prime$ be a bijection that maps each variable $v \in V$ to a new variable (not in V) v' of the same sort. V' is the codomain of $prime$.

A state s of $\mathcal{S}$ is a variable assignment over V. A sequence of states is called a *path*. An *execution* of $\mathcal{S}$ of length k is a pair $\langle \mathcal{I}, \pi \rangle$, where $\mathcal{I}$ is an interpretation and $\pi := s_0, s_1, \dots, s_{k-1}$ is a *path* such that $\mathcal{I}[s_0] \models I(V)$ and $\mathcal{I}[s_i][s_{i+1} \circ prime^{-1}] \models T(V, V')$ for all $0 \leq i < k - 1$.

Unrolling and Bounded Model Checking.

An *unrolling* of length k of a symbolic transition system is a formula that captures an execution of length k by creating copies of the transition relation. This is accomplished by introducing fresh copies of every state variable for each state in the execution path. We use $V@i$ to denote the set of variables obtained by replacing each variable $v \in V$ with a new variable called $v@i$ of the same sort. We refer to these as *timed* variables. Given an STS $\mathcal{S}$, let $unroll(\mathcal{S}, k) = I(V@0) \wedge \bigwedge_{0 \leq i < k} T(V@i, V@(i+1))$.

Bounded model checking (BMC) [3] is an unrolling-based symbolic model checking approach. Let $P(V)$ be a formula representing a desired property of a symbolic transition system. BMC creates an unrolled transition system and adds an additional constraint that the property is violated at time k. The BMC formula at bound k is thus: $unroll(\mathcal{S}, k) \wedge \neg P(V@k)$. A typical approach for BMC starts with $k = 0$ and incrementally increases it if no counterexample is found at the current bound. A satisfiable BMC formula can easily be converted into an execution that violates the property.

Optimization. An *optimization problem* $\mathcal{OP}$ is a tuple $\langle t, A, \preccurlyeq, \phi, \mathcal{O} \rangle$ where:
- t is an *objective term* to optimize of sort σ;
- A is a set and $\preccurlyeq$ is a total order over A.
- ϕ 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 $\mathcal{I}' \models \phi$:

$$(\mathcal{O} = min \rightarrow t^{\mathcal{I}} \preccurlyeq t^{\mathcal{I}'}) \wedge (\mathcal{O} = max \rightarrow t^{\mathcal{I}'} \preccurlyeq t^{\mathcal{I}}).$$

A *multi-objective optimization problem* $\mathcal{MOP}$ is a finite sequence of optimization problems $\{\mathcal{OP}_1, \dots, \mathcal{OP}_n\}$ over the same formula ϕ, where $\mathcal{OP}_i := \langle t_i, A_i, \preccurlyeq_i, \phi, \mathcal{O}_i \rangle$ and t_i is of sort σ_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

$$(\mathcal{O}_j = min \rightarrow t_j^{\mathcal{I}} \prec_j t_j^{\mathcal{I}'}) \wedge (\mathcal{O}_j = max \rightarrow t_j^{\mathcal{I}'} \prec_j t_j^{\mathcal{I}}),$$

where $\prec$ is the strict total order associated with $\preccurlyeq$.

III. CONFIGURATION SOLVING FRAMEWORK

In this section, we formalize the configuration problem and introduce our automated framework for solving it. We also describe how to improve scalability using a modular approach.

A. Problem Formalization

Suppose we have a configurable system that we want to use in a particular application context. We assume the application context can precisely define an input/output relationship that it expects the system to adhere to. The *configuration finding problem* is then: given a system S and an application-supplied input-output relationship P for S, find a configuration $\mathcal{C}$ for S such that S satisfies P with configuration $\mathcal{C}$. In this paper, we assume that P specifies behavior for only a finite number of steps. The rationale is that for many configurable systems, a segment of a desired execution is sufficient to partially (or fully) determine what the configuration should be. This is the case for the systems we target and for the case study we
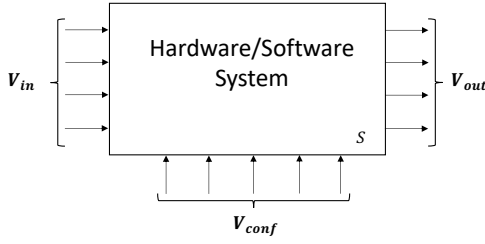
Fig. 1: Formal system model.

Fig. 2: Configuration solving framework (basic) scheme. $\mathcal{CP}$ is a configuration problem. ϕ is a configuration formula.

describe later. More general specifications are an important direction for future work.

Formally, a configuration problem $\mathcal{CP}$ is a tuple $\langle \mathcal{S}, k, V_{\text{in}}, V_{\text{out}}, V_{\text{conf}}, P \rangle$ where:

- $\mathcal{S} := \langle V, I, T \rangle$ is a symbolic transition system representing a configurable system S, as in Figure 1;
- k is the number of transitions over which the input-output specification will be defined;
- $V_{\text{in}}, V_{\text{out}}, V_{\text{conf}}$ are three distinguished subsets of the state variables V of $\mathcal{S}$; V_{in} contains *input* variables (input variables do not appear in $I(V)$, and their primed versions do not appear in T); V_{out} contains *output* variables; and $V_{\text{conf}} \neq \emptyset$ contains the *configuration* variables; pairwise intersections of these sets may either be empty or non-empty, and V may contain variables that are not in any of these sets; and
- P is *an input-output property*, or *an input-output specification*, a formula capturing an input-output relationship for k transitions: $P(V_{\text{in}}@0, \ldots, V_{\text{in}}@(k-1), V_{\text{out}}@0, \ldots, V_{\text{out}}@k)$; in this paper, we use a "programming by example" property, specifying a set of exact values on input and output variables at each transition: $\bigwedge_{0 \leq i < k} V_{\text{in}}@i = c_{in}^i \wedge \bigwedge_{0 \leq i \leq k} V_{\text{out}}@i = c_{out}^i$. This approach works well on our case study (i.e. the configuration found for the given example generalizes to other inputs), and it avoids the need for universal quantification on the input variables. Handling other kinds of properties is an important direction for future work.

A *configuration* $\mathcal{C}$ is defined as an assignment to the variables in V_{conf}.

In this paper, we assume the configuration variables V_{conf} remain unchanged once configured (a reasonable assumption for many systems, including the one in the case study we present in Section V). We enforce this by explicitly adding an additional *configuration constancy constraint*: $conf(V_{\text{conf}}, k) = \bigwedge_{0 \leq i < k} V_{\text{conf}}@(i+1) = V_{\text{conf}}@i$. The configuration finding problem then reduces to checking the satisfiability of the *configuration formula*:

$$\phi(\mathcal{CP}) = unroll(\mathcal{S}, k) \wedge conf(V_{\text{conf}}, k) \wedge$$
$$P(V_{\text{in}}@0, \ldots, V_{\text{in}}@(k-1), V_{\text{out}}@0, \ldots, V_{\text{out}}@k) \quad (1)$$

A configuration $\mathcal{C}$ is *correct* for $\mathcal{CP}$ if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \phi$ and $\mathcal{C} = \mathcal{I}^{V_{\text{conf}}}$.
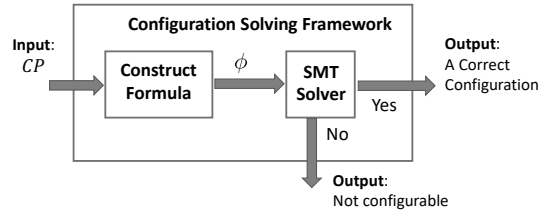
Example 1. *(simple ALU)*

Let $\mathcal{S} := \langle \{x : int, a : int, cfg : Bool\}, x = 0, x' = ite(cfg, x+a, x-a) \rangle$ be a transition system in a configuration finding problem, where $V_{\text{in}} = \{a\}$, $V_{\text{out}} = \{x\}$, $V_{\text{conf}} = \{cfg\}$, and ite is the if-then-else operator. There are two ways to configure $\mathcal{S}$: as a system that always adds the current input to the current state, or as a system that always subtracts the current input from the current state. Let us consider two instances of an input-output relation for $k = 2$:

1) *$P_1(a@0, a@1, x@0, x@1, x@2) = a@0 = 1 \wedge a@1 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 2$. We are interested in whether there exists a value of cfg which satisfies both the configuration constancy constraint (i.e., remains unchanged) and P_1. To determine this, we check the satisfiability of $unroll(\mathcal{S}, 2) \wedge conf(cfg@0, cfg@1, cfg@2) \wedge P_1(a@0, a@1, x@0, x@1, x@2)$, which expands to:*

$$x@0 = 0 \wedge$$
$$x@1 = ite(cfg@0, x@0 + a@0, x@0 - a@0) \wedge$$
$$x@2 = ite(cfg@1, x@1 + a@1, x@1 - a@1) \wedge$$
$$cfg@1 = cfg@0 \wedge cfg@2 = cfg@1 \wedge$$
$$a@0 = 1 \wedge a@1 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 2$$

The formula is satisfiable when $cfg@0 = True$.

2) *$P_2(a@0, a@1, x@0, x@1, x@2) = a@0 = 1 \wedge a@1 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 0$. For this case, the formula to be checked is:*

$$x@0 = 0 \wedge$$
$$x@1 = ite(cfg@0, x@0 + a@0, x@0 - a@0) \wedge$$
$$x@2 = ite(cfg@1, x@1 + a@1, x@1 - a@1) \wedge$$
$$cfg@1 = cfg@0 \wedge cfg@2 = cfg@1 \wedge$$
$$a@0 = 1 \wedge a@1 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 0$$

This formula is unsatisfiable, and thus there is no value of cfg that satisfies the desired property.

The framework for the basic scheme just outlined is shown in Figure 2. The input to the framework is a configuration problem. The framework constructs formula (1) and calls a solver to determine whether it is satisfiable. The output is either "not configurable" or the configuration $\mathcal{C}$.

There are two main sources of complexity that limit the scalability of the approach. The first is the complexity of the

Algorithm 1 Modular configuration finding.

Procedure SOLVEMODULAR
Input: $(\mathcal{CP}_1, \mathcal{CP}_2)$ a decomposition of $\mathcal{CP}$.
Output: a pair $(r, \mathcal{C})$ where if $r = sat$, then $\mathcal{C}$ is a configuration of $\mathcal{S}$
1: $\phi_1 := \text{MAKECP}(\mathcal{CP}_1)$
2: $(r, \mathcal{I}_1) := \text{SOLVE}(\phi_1)$,
3: **if** $r = sat$ **then**
4:      $\phi_2 := \text{MAKECP}(\mathcal{CP}_2) \wedge \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$
5:      $(r, \mathcal{I}) := \text{SOLVE}(\phi_2)$
6: **end if**
7: **return** $(r, \mathcal{I}^{V_{\text{conf}}})$

---

design itself, and the second is the bound k required by P. To address design complexity, we propose designing for modular configuration, discussed in more detail in Section III-B below. Designing systems that can be configured using only small values of k is an interesting research challenge that we plan to investigate in future work.

Another way to improve scalability is by using design knowledge to strengthen the formula ϕ. For example, if a configuration variable must be within a specific range, then this can be added as a constraint. Any constraint expressible in the language supported by the backend SMT solver can be supported.

B. Modular Configuration

A natural remedy for design complexity is modular decomposition. Here, we explain a systematic approach for modular configuration, including conditions under which a full configuration can be recovered.

Given $\mathcal{CP} = \langle \mathcal{S}, k, V_{\text{in}}, V_{\text{out}}, V_{\text{conf}}, P \rangle$ with $\mathcal{S} = \langle V, I, T \rangle$, we say $(\mathcal{CP}_1, \mathcal{CP}_2)$ is a *decomposition* of $\mathcal{CP}$ (where $\mathcal{CP}_i := \langle \mathcal{S}_i, k, V_{\text{in}}^i, V_{\text{out}}^i, V_{\text{conf}}^i, P_i \rangle$ and $\mathcal{S}_i := \langle V_i, I_i, T_i \rangle$ for $i = 1, 2$) if: (i) $T_1(V_1, V_1') \wedge T_2(V_2, V_2') \implies T(V, V')$; (ii) $I_1(V_1) \wedge I_2(V_2) \implies I(V)$; (iii) $P_1 \wedge P_2 \implies P$; and (iv) $V_{\text{conf}} \subseteq V_{\text{conf}}^1 \cup V_{\text{conf}}^2$.

We now describe a procedure SOLVEMODULAR, presented in Algorithm 1, which, given a decomposition $(\mathcal{CP}_1, \mathcal{CP}_2)$ of a configuration problem $\mathcal{CP}$, attempts to solve $\mathcal{CP}$ by solving $\mathcal{CP}_1$ and $\mathcal{CP}_2$. The call to MAKECP on line 1 constructs the configuration formula for $\mathcal{CP}_1$. The call to SOLVE on line 2 invokes a solver to check the satisfiability of the configuration formula. If the formula is satisfiable, SOLVE returns a pair $(sat, \mathcal{I})$ where $\mathcal{I}$ is a satisfying interpretation found by the solver. If the formula is unsatisfiable, SOLVE returns a pair $(unsat, \mathcal{I})$ where $\mathcal{I}$ is an arbitrary interpretation. Line 4 creates the configuration formula for $\mathcal{CP}_2$. The formula is additionally constrained to ensure that the solution for $\mathcal{CP}_2$ still satisfies ϕ_1. The call to GETABDUCT returns a formula ψ such that $\psi \models_{\mathcal{T}} \phi_1$. The goal is to use the information in $\mathcal{I}_1$ to generate a simple formula for ψ. The approach we take is to find a set of sub-terms in ϕ_1 such that, if we constrain them to be equal to their values in $\mathcal{I}_1$, this ensures that ϕ_1 is satisfied. In the worst case, we could constrain ϕ_1 itself to be equal to $\top$, which would effectively require solving all of ϕ_1 again at the same time as solving ϕ_2. However, in practice, we can do much better. For example, it is often sufficient to let
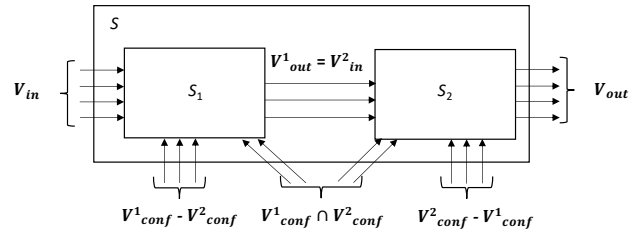

Fig. 3: Modular decomposition of system $\mathcal{S}$ into systems $\mathcal{S}_1$ and $\mathcal{S}_2$. V_{out}^1 and V_{conf}^1 are the output and the configuration variables of $\mathcal{S}_1$. V_{in}^2 and V_{conf}^2 are the input and the configuration variables of $\mathcal{S}_2$. $V_{conf} \subseteq V_{conf}^1 \cup V_{conf}^2$.

ψ be the formula that assigns the free variables in ϕ_1 to their model values from $\mathcal{I}_1$.[1] If the second call to SOLVE succeeds, the result is a correct configuration for $\mathcal{CP}$.

Theorem III.1. *(Soundness)*
If $(\mathcal{CP}_1, \mathcal{CP}_2)$ is a decomposition of a configuration problem $\mathcal{CP}$, and SOLVEMODULAR$(\mathcal{CP}_1, \mathcal{CP}_2)$ returns a a pair $(sat, \mathcal{C})$, then $\mathcal{C}$ is a correct configuration of $\mathcal{CP}$.

Proof. Let SOLVEMODULAR return $(sat, \mathcal{I}^{V_{\text{conf}}})$. We prove that $\mathcal{I}^{V_{\text{conf}}}$ is a correct configuration of $\mathcal{CP}$. First, we notice that SOLVEMODULAR returns $r = sat$ iff both calls to SOLVE(ϕ_1) and SOLVE(ϕ_2) return $r = sat$. Let $(sat, \mathcal{I}_1)$ and $(sat, \mathcal{I})$ be the results of SOLVE(ϕ_1) and SOLVE(ϕ_2), respectively. Let $\psi = \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$. From line 5, $\mathcal{I} \models \phi_2$. Thus, $\mathcal{I} \models \text{MAKECP}(\mathcal{CP}_2)$ and $\mathcal{I} \models \psi$. Since $\psi \models_{\mathcal{T}} \phi_1$, we also have $\mathcal{I} \models \phi_1$. Consequently, $\mathcal{I}$ satisfies: I_1, $T_1(V_1@i, V_1@(i+1))$ for $i \in [0, k-1]$, $conf(V_{\text{conf}}^1, k)$, and P_1. Furthermore, $\mathcal{I}$ satisfies: I_2, $T_2(V_2@i, V_2@(i+1))$ for $i \in [0, k-1]$, $conf(V_{\text{conf}}^2, k)$, and P_2. By the definition of decomposition, then, $\mathcal{I}$ satisfies $I(V)$, $T(V@i, V@(i+1))$ for $i \in [0, k-1]$, and P. Finally, from $\mathcal{I} \models conf(V_{\text{conf}}^1, k)$, $\mathcal{I} \models conf(V_{\text{conf}}^2, k)$, and condition (iv) of the definition of decomposition ($V_{\text{conf}} \subseteq V_{\text{conf}}^1 \cup V_{\text{conf}}^2$), it follows that $\mathcal{I} \models conf(V_{\text{conf}}, k)$. Thus, $\mathcal{I}$ satisfies the configuration formula of $\mathcal{CP}$. Therefore, $\mathcal{C} := \mathcal{I}^{V_{\text{conf}}}$ is a correct configuration of $\mathcal{CP}$. $\qquad\square$

If SOLVEMODULAR returns $r = unsat$, this does not (in general) imply that $\mathcal{CP}$ is unconfigurable. Rather, it may be that the particular decomposition fails, or even that the particular solution found for $\mathcal{CP}_1$ is at fault (and another solution would have succeeded).

However, in practice, we have found that the algorithm works well when the decomposition separates a module into two largely independent parts. An example is shown in Figure 3. Here, the two submodules share only a subset of the configuration variables as well as an interface where outputs of the first module flow into inputs of the second module.

---

[1] See the appendix of an extended version of this paper for details on when and why this works [4]. Investigating other possible implementations for GETABDUCT is an interesting direction for future work.
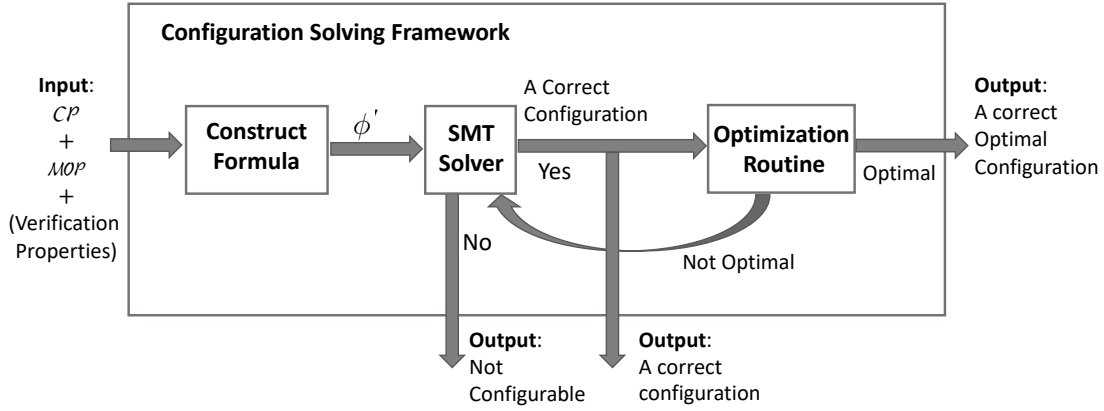
Fig. 4: Optimization-assisted configuration framework. The input is a configuration problem with optional optimization and verification objectives. The framework can return: (i) a non-optimal but correct configuration, or (ii) an optimal and correct configuration, or (iii) unsat. ϕ' is a conjunction of the configuration formula ϕ and the optional verification properties.

IV. OPTIMIZATION-ASSISTED CONFIGURATION

A solver can return an unnatural or non-intuitive configuration, complicating the ability of users to understand or maintain the configuration.

We observe that users tend to prefer the simplest configurations, where the notion of simplest corresponds to minimizing some metric when finding solutions. To this end, we show how to extend our framework with optimization goals.

Figure 4 depicts our configuration framework extended with support for multi-objective optimization. There are various ways to combine optimization with configuration solving; we depict one approach using iteration. One instance of this approach works as follows: first a solution is found and the value of the objective term is calculated; then the search space is systematically explored by iteratively constraining the value to be better than the current best value; when no better value can be found, the optimal value has been discovered. There are many different kinds of optimizations that fit this general framework. We present several useful examples in the context of the case study in Section V.

Further extensions. Figure 4 also includes an extension to support combining configuration-finding with verification. In this scheme, any invariants that the system should obey are conjoined to the configuration formula. This ensures that any configuration found satisfies the invariant up to bound k. To check that an invariant holds for all reachable states requires a separate run of an unbounded model checker.

Finding the configuration itself using unbounded model checking is an interesting direction for future work. A significant challenge is that this requires writing the input-output property as a single state formula, which may be much harder than writing it as a bounded set of input, output pairs (in much the same way that loop invariants are difficult to come up with in software). If the input-output property can be written as a state formula P, it may be possible to utilize invariant synthesis techniques by seeking to synthesize an invariant of the form: $\bigwedge_i (V_{\mathsf{conf}}^i = C^i) \implies P$, where the left-hand side of the implication contains all configuration variables $V_{\mathsf{conf}}^i \in V_{\mathsf{conf}}$, and each C^i is a constant value to be synthesized.

V. CASE STUDY

We present a case study with a course-grained reconfigurable architecture (CGRA) design developed in the Agile Hardware Center at Stanford University [5]. Reconfigurable architectures are appealing because they offer the high performance of hardware with software-like flexibility. CGRAs in particular use sophisticated reconfigurable elements with the aim of narrowing the performance gap with custom ASICs [6].

However, configuring a CGRA is challenging, typically requiring manual effort by an experienced engineer who fully understands the application and the design. To the best of our knowledge, ours is the first framework that finds correct CGRA configurations fully automatically.

In this paper, we focus on configuring a *memory tile* of the CGRA for image processing applications. In these applications data is streamed into the memory tile and must be reordered in various ways before being streamed out. Only the timing and order of the data are changed; the data itself remains the same. Below, we first describe the memory tile design, then present some specific applications, and then explain how we automate configuration of the design for these applications.

A. CGRA Memory Tile Design

The memory tile is a non-trivial design (34998 FF and 164696 gates). Figure 5 shows its architecture . It contains three types of units: *memories*, *addressors*, and *accessors*. Addressors and accessors are reconfigurable units. The accessors control *when* to write or read. The addressors control *where* to write or read. There are three memory modules: an *aggregator* module (AGG), a *static random-access memory* module (SRAM), and a *transpose buffer* module (TB). Each module has an *input accessor* and an *input addressor* associated with it for writes, and an *output accessor* and an *output addressor* for reads. The modules are chained: outputs of AGG are intputs
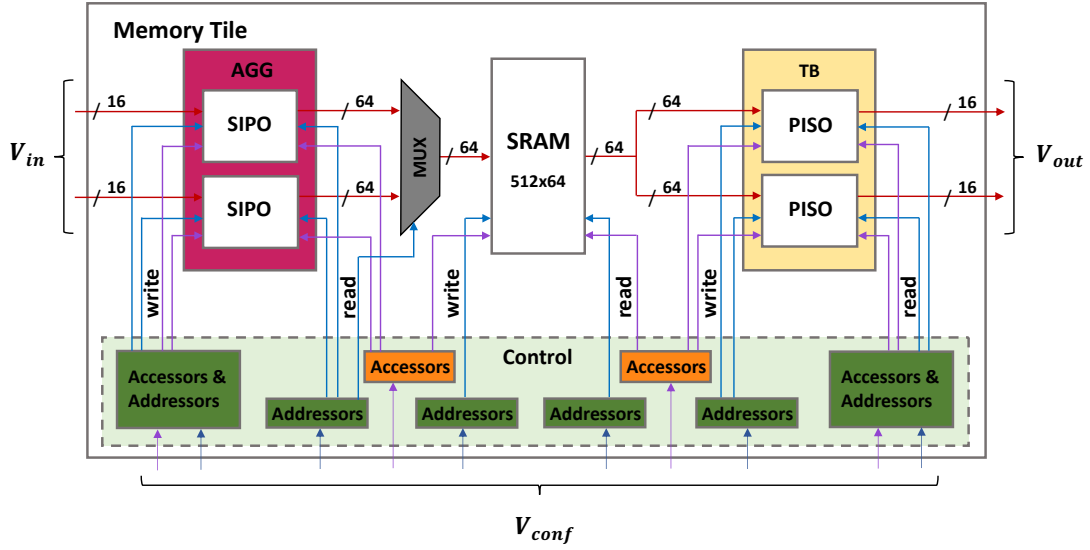
Fig. 5: Memory tile architecture. All accessors and addressors are included in the *control* box. Red arrows represent data flow. Blue and purple arrows represent addressor and accessor control signals, respectively. Green boxes are local to a single module. Orange boxes are shared between modules. V_{conf} consists of all accessor and addressor configuration variables.

Procedure AFFINESEQUENCE
Input: dim: a value indicating the number of nested loops,
    $ranges[dim]$: an array of loop bounds, one for each loop,
    $strides[dim]$: an array of strides, one for each loop,
    $offset$: the offset for the address computation
Output: $vals[\Pi_i ranges[i]]$: a set of output addresses
 1: var $c[dim]$;   ▷ Index variables for each loop
 2: var $i := 0$;
 3: **for** $c[dim-1]$ **in** $[0, ranges[dim-1])$ **do**
 4:    ...
 5:    **for** $c[0]$ **in** $[0, ranges[0])$ **do**
 6:       $vals[i] := \Pi_{j=0}^{dim-1} c[j] * strides[j] + offset$;
 7:       $i := i + 1$;
 8:    **end for**
 9: **end for**

Fig. 6: Affine sequence generator using nested loops.

to SRAM, and outputs of SRAM are inputs to TB. Accessors are *shared* between each pair of connected memory modules. Shared accessors act as *schedule generators* for each memory connection. They specify when the data should be transferred and set any required delays between when the data is produced and consumed. Addressors are unique for each module.

The addressors and accessors in the memory tile make use of affine sequence generators to generate sequences of values for reading and writing. Figure 6 shows pseudocode for an affine sequence generator. It takes as input a number dim of loops, an array $ranges$ with bounds for each loop, an array $strides$ with strides for each loop, and $offset$ which is a base value. It then computes a sequence of outputs, $vals$, by running dim nested loops, and computing the sum of the offset and the product of each stride with its loop index in the innermost loop. Each of the inputs to the procedure corresponds to a configuration register in the hardware.

While each addressor and accessor contains an affine se-

quence generator, they differ in how they interpret $vals$. For an addressor, $vals$ contains raw addresses sent to a memory (for either reading or writing). For an accessor, $vals$ contains clock cycle counts that are compared to a running cycle counter to determine when to read or write. Note that an (accessor, addressor) pair should have the same values for their dim and $ranges$ variables to ensure that they produce the same number of values. There are 4 accessors (including 2 shared with SRAM) and 4 addressors for AGG (1 for each memory port). TB has 4 accessors (including 2 shared with SRAM) and 4 addressors (1 for each memory port). SRAM has 2 addressors, and shares 2 accessors with AGG and 2 acessors with TB.

The memory tile processes 16-bit words. However, it uses a 512x64-bit SRAM which stores four 16-bit words at each address. The rationale for this design is to emulate a multi-ported SRAM while minimizing the energy consumption per memory access [7]. To match the data width at the SRAM interface, AGG and TB implement width converters. AGG implements a *serial-in to parallel-out* (SIPO) converter—serial data is loaded, one 16-bit word at a time, and these are packed into 64-bit outputs. TB implements a *parallel-in to serial-out* (PISO) converter—parallel data is loaded into the PISO as a 64-bit word and is shifted out of the PISO serially, one 16-bit word at a time. The memory tile uses a 2-input and 2-output port architecture to support more throughput. Thus, AGG and TB contain two SIPOs and two PISOs, respectively.

B. Stencil Applications

We consider a common class of image-processing techniques called *stencils*. Stencil computations usually consist of a multi-stage pipeline, where each stage is a dense linear algebra computation in a local region. So-called *push memories* are

inserted between computation units, whose job is to orchestrate the order and the timing of the data explicitly [8]. We explore configuring memory tiles as push memories for four stencil applications:

- *Identity.* The identity stencil simply streams the input back out in the same order. It is useful as a baseline test and also can be used to implement a fixed delay on a stream.
- *3x3 Convolution.* This stencil is used in a variety of image processing applications [9] (e.g., to blur images). It multiplies a 3x3 sliding image window by a 3x3 kernel of constant values.
- *Cascade.* This application implements a pipeline with two convolution kernels executed in sequence. The Cascade application requires configuration of two memory tiles, denoted by *conv* and *hw*.
- *Harris.* Harris is a corner detection algorithm that can be used to infer image features [10]. It extracts the gradients of an image in different orientations and combines this information using multiple convolutions. This is the most complex of our applications, requiring the configuration of five different memory tiles, which we denote as *cim*, *lxx*, *lxy*, *lyy*, and *pad*.

C. Automating the Memory Tile Configuration

We decompose the memory tile into three sub-modules (for scalability), following the approach shown in Figure 3. The first sub-module includes AGG, its input/output accessor/addressor modules, and the MUX (1372 FF, 19676 gates). The second sub-module includes SRAM, both AGG read accessors, and both TB write accessors (33712 FF, 150750 gates). The third sub-module includes TB and its input/output accessor/addressor modules (1126 FF, 18538 gates). Shared accessors contain the shared configuration variables, whose values are propagated to the next module during modular configuration.

In order to configure each module in the memory tile, we look at the transition system defined by its memory and its accessors and addressors. We then use the "programming by example" approach described above. We specify the input-output property P as a sequence of distinct input values (e.g., 1,2,3,...), paired with the corresponding application-specific desired output sequence based on those values. We then solve for the configuration variables as described in Section III-A above.

As mentioned in Section IV, it is important to generate configurations that can easily be read and understood. Working together with the designers, we devised a set of optimization objectives that greatly improve the readability of memory tile configurations. We explain these next. We apply the framework of Figure 4 to configure and optimize each module separately.

Objective 1: we first minimize the dim variables in the module, since this corresponds to using fewer nested loops and fewer loop counters, resulting in simpler solutions in general. We prioritize minimizing dim variables controlling writes over those controlling reads, as lower write complexity

leads to lower read complexity anyway. We formalize this as the following multi-objective optimization problem:

$$\mathcal{MOP}_1 := \{\mathcal{OP}_1, \mathcal{OP}_w^1, \ldots, \mathcal{OP}_w^{d_w}, \mathcal{OP}_r^1, \ldots, \mathcal{OP}_r^{d_r}\} :$$
$$\mathcal{OP}_1 := \langle \Sigma_i\ dim_i, A_{BV}, \preccurlyeq_{BV}, \phi, min\rangle \text{ for } i \in [1, d],$$
$$\mathcal{OP}_w^i := \langle dim_w^i, A_{BV}, \preccurlyeq_{BV}, \phi, min\rangle \text{ for } i \in [1, d_w]$$
$$\mathcal{OP}_r^i := \langle dim_r^i, A_{BV}, \preccurlyeq_{BV}, \phi, min\rangle \text{ for } i \in [1, d_r]$$

Here, A_{BV} is the domain of bit-vectors (i.e., unsigned machine integers), $\preccurlyeq_{BV}$ is the usual total order on bit-vector values, d is the number of affine sequence generators in the module, and dim_i for $i \in [1, d]$ are all of the dim variables in the module. These are further partitioned into write dimensionality variables dim_w^i, $i \in [1, d_w]$, and read dimensionality variables, dim_r^i, $i \in [1, d_r]$, with $d_w + d_r = d$. ϕ is the configuration formula.

Objective 2: we minimize the products of the range configuration variables in each loop-nest structure. The objective term corresponds to the aggregate number of reads or writes that occur to a particular memory. By minimizing this number, we eliminate unnecessary reads and writes to the memory. Formally, the optimization problem is:

$$\mathcal{OP}_2 := \langle \Sigma_{i=0}^{d-1} \Pi_{j=0}^{dim_i - 1} ranges_i[j], A_{BV}, \preccurlyeq_{BV}, \phi, min\rangle$$

Objective 3: we minimize stride variables to avoid generating configurations using unnecessarily large addresses.

Many different sets of values for strides could produce the same $vals$ stream in the end, so by choosing the smallest values, we hope to generate the simplest solution. The optimization problem simply minimizes the sum of all stride variables in the module:

$$\mathcal{OP}_3 := \langle \Sigma_i\ strides_i, A_{BV} \preccurlyeq_{BV}, \phi, min\rangle.$$

Objective 4: we also minimize $offset$ configuration variables in addressor modules. For addressor modules, minimizing the $offset$ addressor variable prevents unnecessary offsets, improving the readability of the generated configuration. Note that values of $offset$ variables in the accessors are fixed by the application. The corresponding problem is as follows, minimizing the sum of all addressor $offset$ variables in the module:

$$\mathcal{OP}_4 := \langle \Sigma_i\ offset_i, A_{BV}, \preccurlyeq_{BV}, \phi, min\rangle.$$

Combined objective: the combined optimization query includes all four objectives and captures the full set of optimization objectives for each module:

$$\mathcal{MOP}_{\mathcal{H}} := \{\mathcal{MOP}_1, \mathcal{OP}_2, \mathcal{OP}_3, \mathcal{OP}_4\}.$$

We solve and prioritize $\mathcal{MOP}_1$ by iteratively increasing the bound on the sum $\Sigma_i dim_i$, and for each bound, trying all possible assignments to the variables, in the order specified by $\mathcal{MOP}_1$. Note that this approach does not directly fit the scheme described in Figure 4, since it does not require finding a first solution that is iteratively improved. Instead, it

iteratively widens the search space until the first solution is found.

For the other objectives, we use a branch-and-bound algorithm. First, a solution is found, and the value of the term is calculated; then, the solution space is explored systematically, by iteratively constraining the value of the objective term to be better than the current best value. Each optimal solution is propagated to the next optimiziation objective as a constraint.

VI. EVALUATION

Implementation. We have implemented our framework using `Pono` [11], an open-source SMT-based model checker. `Pono` is built on `Smt-Switch` [12], a generic C++ API for interacting with SMT solvers. `Pono` provides infrastructure for reading in, unrolling, and otherwise manipulating transition systems. We use `Boolector` [13] as the underlying SMT solver. We convert the memory tile design in our case study from a SystemVerilog representation to its equivalent representation in the Btor2 format [13], which is accepted by `Pono`. We use `Yosys` [14], a Verilog synthesis suite, to do the translation. The experimental code is available at https://github.com/StanfordAHA/Configuration/.

Experimental Results. We evaluate our configuration-finding framework using the memory tile design and the four stencil applications described in Section V. For each application, we generated benchmarks for various input image sizes, from 16x16 to 60x60. For applications that require more than one memory tile (i.e., cascade and harris), we choose one representative configuration problem: conv for Cascade and lxx for Harris (more results appear in the appendix of an extended version of this paper [4]). The number of transitions required for each configuration problem is based on the number of clock cycles it takes to process an image of a given size for a given application.

For each benchmark, we first run the basic algorithm described in Section III, which finds the first satisfying configuration. We try both with and without the modular approach described in Section III-B. We then run our optimization-assisted configuration algorithm (using only the modular approach) as described in Section IV. We run our experiments on a 2x Intel Xeon E5-2620 v4 @ 2.10GHz 8-core 128GB computer. Timeout is set to 4000 seconds. Memory limit is 100 GB.

The results are shown in Figure 7. Each chart shows results for both the basic algorithm (First Configuration) and the optimization-assisted algorithm (Optimal Configuration). Within each of these categories, up to five different results are shown for each image size: *top* is the time required to configure the entire design, monolithically; *agg*, *tb*, and *sram* refer to the time required to configure each of the sub-modules independently; and *sram_agg_tb* is the time required to configure the SRAM module after first configuring AGG and TB (this is the most efficient order for these modules) and then propagating the shared configurations from those modules as described in Figure 3. Note that in the modular

approach, AGG and TB are configured independently; thus, the configuration can be performed in parallel, and the total design configuration time is the sum of *sram_agg_tb* and the maximum of *agg* and *tb*. Timeouts are represented by full bars (up to the timeout limit), and memory outs are represented by omitting the bar completely. We also omit the bar for *sram_agg_tb* if either AGG or TB is not solved within the given time-memory budget. We make several observations about the results below.

Modular Approach. As the experiments show, the full memory tile is too large to solve within the given time-memory budget—it times out for all image sizes. However, by using the modular approach, we are able to configure the design for all applications for reasonably useful image sizes. For the Identity Stream, we can configure for all image sizes (with unroll depths up to 3601) relatively easily using the modular approach. Other applications are more challenging, but we are still able to scale up to images of size 40x40 (and unroll depth up to 1939 clock cycles).

We also observe that the AGG and TB modules take comparable time for the Identity Stream, but for other applications, configuration of the TB module is more challenging. This can be explained as follows. AGG and TB are both two-port designs, comparable in size and complexity. But for all applications, AGG can be configured by exploiting only a single port, while only the Identity Stream allows a single-port configuration of TB. Thus, we quickly find a simple configuration for TB with the Identity Stream, but no comparatively simple configuration exists for the other applications.

Optimal Configurations. The right-hand side of each chart shows the results of running our optimization-assisted configuration algorithm for each application. There are several interesting observations. First of all, for the AGG and TB modules, finding optimal configurations is generally more expensive. However, once these optimal configurations are found, it is often easier to find the corresponding SRAM configuration, suggesting that optimal configurations may help improve later stages of modular configuration. The total configuration time with optimization is generally comparable to or only slightly worse than the time required to configure without optimization. Given the value of optimal configurations in terms of simplicity and readability, these results suggest that modular configuration with optimization may be the best strategy in practice.

VII. RELATED WORK

The problem of system configuration has been studied in various formulations and domains, such as software tool configuration, hardware configuration, network configuration, distributed application configuration, and deployment strategies. In one research stream, the configuration problem is to select and arrange a set of components from a given set of assets in order to construct an overall system with a desired specification [15]–[18]. Other formulations take as input a configuration database, including configuration variables, and desired requirements to be met [19], [20]. The task is to find

(a) Identity Stream

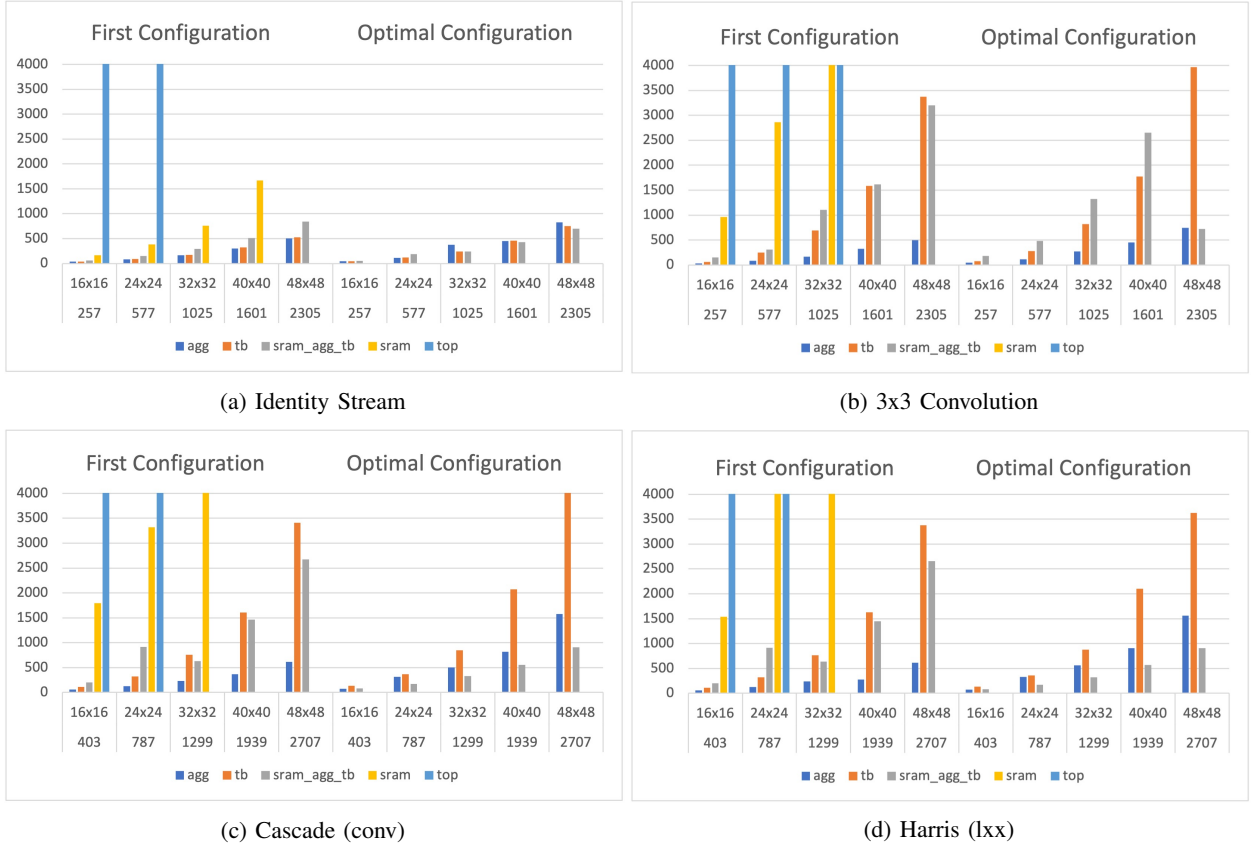(b) 3x3 Convolution

(c) Cascade (conv)

(d) Harris (lxx)

Fig. 7: Horizontal axis shows image sizes and number of clock cycles required for processing. Vertical axis shows time in seconds.

values for the configuration variables which instantiate the database so that it meets the requested requirement. The work whose problem definition is closest to ours is [21], which also uses transition systems. The authors define a configuration as an initial state of a transition system, which is very similar to our notion of configuration variables.

Constraint solving has been explored in various ways for automating system configuration. Efforts have been made to design declarative, constraint-based, object-oriented languages and policy-based tools to configure systems as well as to validate configurations [19], [22]–[24]. Early approaches were based on constraint satisfaction and constraint logic programming [18], [25], [26]. More recent approaches utilize SAT and SMT solvers [17], [19], [27], and counterexample-guided inductive synthesis and relational model finding [21], [28] for dynamic configuration. However, the way these approaches reduce configuration problems to constraint satisfaction problems is significantly different from our approach using input/output examples and unrolling.

More significantly, our work differs in its use of modularity and optimization to improve scalability and understandability. Some automated configuration efforts do employ optimization (e.g., [29]), but with a different goal, namely to configure a system in a way that maximizes its performance.

VIII. CONCLUSION

We proposed a new approach for automatically configuring systems representable as transition systems. Key contributions of our approach include its ability to leverage modularity and its use of optimization. Optimal configurations are more human-understandable, and both modularity and optimization can improve scalability. We demonstrated these claims with a case study using a CGRA memory tile.

Future directions for this work include incorporating un-bounded model checking, applying the framework to a wider variety of designs, exploring modularity for more sophisticated theories, and finding provably correct configurations for applications with repeating input/output patterns.

REFERENCES

[1] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 825–885.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 1579. Springer, 1999, pp. 193–207.

[4] N. Tsiskaridze, M. Strange, M. Mann, K. Sreedhar, Q. Liu, M. Horowitz, and C. Barrett, "Automating system configuration," 2021. [Online]. Available: https://arxiv.org/abs/2108.05987

[5] "Aha! agile hardware project at stanford university," https://aha.stanford.edu/.

[6] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3357375

[7] A. Vasilyev, "Evaluating spatially programmable architecture for imaging and vision applications," Ph.D. dissertation, Stanford University, 2019.

[8] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 137–151.

[9] R. Chandel and G. Gupta, "Image filtering algorithms and techniques: A review," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 10, 2013.

[10] C. G. Harris, M. Stephens *et al.*, "A combined corner and edge detector." in *Alvey vision conference*, vol. 15, no. 50. Citeseer, 1988, pp. 10–5244.

[11] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: a Flexible and Extensible SMT-based Model Checker," in *CAV*, ser. Lecture Notes in Computer Science. Springer, 2021.

[12] M. Mann, A. Wilson, Y. Zohar, L. Stuntz, A. Irfan, K. Brown, C. Donovick, A. Guman, C. Tinelli, and C. W. Barrett, "Smt-Switch: A Solver-agnostic C++ API for SMT Solving," in *International Conference on Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science. Springer, 2021.

[13] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_32

[14] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[15] J. P. McDermott, "R1: A rule-based configurer of computer systems," *Artif. Intell.*, vol. 19, no. 1, pp. 39–88, 1982. [Online]. Available: https://doi.org/10.1016/0004-3702(82)90021-2

[16] M. A. Mansor, M. Kasihmuddin, and S. Sathasivam, "Vlsi circuit configuration using satisfiability logic in hopfield network," *International Journal of Intelligent Systems and Applications*, vol. 8, pp. 22–29, 2016.

[17] R. Michel, A. Hubaux, V. Ganesh, and P. Heymans, "An smt-based approach to automated configuration," in *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, ser. EPiC Series in Computing, P. Fontaine and A. Goel, Eds., vol. 20. EasyChair, 2012, pp. 109–119. [Online]. Available: https://easychair.org/publications/paper/bKGs

[18] D. Sabin and E. C. Freuder, "Configuration as composite constraint satisfaction," 1996.

[19] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *J. Netw. Syst. Manag.*, vol. 16, no. 3, pp. 235–258, 2008. [Online]. Available: https://doi.org/10.1007/s10922-008-9108-y

[20] S. Narain, "Network configuration management via model finding," in *LISA*, 2005.

[21] T. Nelson, N. Danas, T. Giannakopoulos, and S. Krishnamurthi, "Synthesizing mutable configurations: Setting up systems for success," in *34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 81–85. [Online]. Available: https://doi.org/10.1109/ASEW.2019.00034

[22] J. Hewson, "Constraint-based specifications for system configuration," 2013.

[23] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," vol. 2006, 07 2006, pp. 10 pp.–.

[24] J. Hewson, "Constraint-based specifications for system configuration," Ph.D. dissertation, 11 2013.

[25] N. Sharma and R. Colomb, "Mechanising shared configuration and diagnosis theories through constraint logic programming," *The Journal of Logic Programming*, vol. 37, no. 1, pp. 255–283, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743106698100109

[26] J. Tiihonen, M. Heiskala, A. Anderson, and T. Soininen, "Wecotin–a practical logic-based sales configurator," *AI Communications*, vol. 26, no. 1, pp. 99–131, 2013.

[27] S. Peter and T. Givargis, "Component-based synthesis of embedded systems using satisfiability modulo theories," *ACM Trans. Design Autom. Electr. Syst.*, vol. 20, no. 4, pp. 49:1–49:27, 2015. [Online]. Available: https://doi.org/10.1145/2746235

[28] A. Wagner, "Where to begin? synthesizing initial configurations for cellular automata," 2020.

[29] J. A. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *Strategies, Tools , and Techniques: Proceedings of the 26th Large Installation System Administration Conference, LISA 2012, San Diego, CA, USA, December 9-14, 2012*, C. Rowland, Ed. USENIX Association, 2012, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/lisa12/technical-sessions/presentation/hewson