# Sequential Encoding for Relational Analysis

Fadi Zaraket
IBM Systems & Technology Group

Adnan Aziz Sarfraz Khurshid
The University of Texas at Austin

## ABSTRACT

We present SERA, a novel algorithm for compiling a class of *finitized relational logic* formulas into *sequential circuits*. The compiled sequential structures use much fewer variables than traditional approaches that compile to SAT, and allow us to iteratively apply powerful reduction, abstraction, and decision algorithms from *transformation-based verification* (TBV). Our SERA prototype leverages TBV to analyze formulas written in Alloy, a first-order language with transitive closure that is based on relations. The experimental results show that SERA can check formulas for *scopes*–bounds on universe of discourse–that are an order of magnitude higher than those feasible with existing combinational approaches such as the Alloy Analyzer.

## Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification, Formal methods, Model checking*; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Verifying and Reasoning about Programs*

## General Terms

Verification, Languages, Algorithms

## Keywords

Model checking, Finitization, First-order logic, Transformation-based verification, Sequential encoding

## 1. INTRODUCTION

Alloy [14] is a first-order relational logic with transitive closure that has been gaining wide acceptance for modeling software designs and specifications. Alloy is particularly suited for expressing integrity properties of structurally complex data that arise in various different contexts, such as, (1) a height-balanced binary search tree; (2) an *intentional naming system* for resource discovery and service location [1]; and (3) a Microsoft *common object modeling* (COM) interface that enables interprocess communication and dynamic object creation and sharing through mutual recursion [6].

Alloy formulas describe infinite software designs and specifications, and are undecidable [14]. To make them amenable to automated analysis, the Alloy Analyzer [14, 27] *finitizes* the models using a scope as an upper bound on the cardinality of sets of typed objects. Analyses based on finitization are inspired by the *small scope hypothesis* [7], which observes that many errors can be detected using small configurations, even in the case of large software systems. The Alloy Analyzer encodes relations between objects of arbitrary arities as bit-matrices of *atomic propositions* with *true* and *false* values. It then formalizes the specification in question as a pure combinational Boolean predicate. The Alloy Analyzer expresses the predicate in *conjunctive normal form* (CNF) and then decides its validity using an off-the-shelf *satisfiability* (SAT) solver.

SAT solvers often face an exponential blow up in the number of possible assignments to the atomic propositions. This problem, known as *state explosion*, along with the large number of variables used in the CNF encoding, often limits the SAT-based Alloy analysis to scopes below 10. By scaling Alloy to larger scopes, we increase its applicability to real-world designs.

### 1.1 Sequential encoding and TBV

The CNF encoding is inherently restricted to the use of reduction techniques that can be applied on a combinational formula such as symmetry breaking [2], redundancy removal [18, 5], and re-parameterization [18, 22].

The limitations of the current CNF encoding for Alloy formulas motivated us to develop SERA which uses sequential circuits to encode the Alloy formulas. The sequential aspect of SERA opens the door for the use of an arsenal of powerful sequential techniques such as retiming [16], target enlargement [18], localization abstraction [18], state equivalence minimization [5, 28], bounded-model checking [21], and semi-formal search [13, 18]. The synergistic application of these techniques combined in one framework substantially simplifies the sequential circuit.

*SixthSense* [18, 28] is an internal IBM TBV framework for hardware designs which leverages these and other techniques on a sequential circuit description of the system in consideration. It *iteratively* exploits the power of these transfor-

mations to reduce the search space by reducing the number of variables and the level of complexity of the sequential circuit. It then decides the simplified sequential circuit via techniques such as bounded model checking, SAT-solving, or semi-formal searches.

**Novel sequential encoding.** Given an Alloy formula with a scope, we derive a sequential circuit and a gate therein such that the gate can be set to 1 if and only if the Alloy formula is satisfiable within the scope. We use optimization heuristics to form sequential structures that are amenable to the reduction techniques available in TBV. Finally we pass the sequential circuit to our TBV framework in order to decide it. This is qualitatively different than synthesis techniques [25, 11] that detect matches for *regular expressions* including the Kleene-star [24] operator. While these techniques aim at matching a set of strings to an expression, we produce the full accepted language of an Alloy construct, which may include the transitive closure operator, within a given scope.

We make the following contributions:

1. **New encoding for Alloy:** We propose SERA, an algorithm that encodes an Alloy model as a sequential circuit. We tailor the sequential structures in order to reduce the number of Boolean variables needed to encode the design and its specifications.
2. **TBV for relational analysis:** We introduce the use of transformation-based verification in relational model checking.
3. **Implementation:** We present a prototype implementation that scales to scopes that are an order of magnitude higher compared to the Alloy Analyzer. We also conclude checks with *validity within the scope* and checks with a counter-example faster.

This paper is structured as follows. We first visit an example in Section 2 to illustrate Alloy and the Alloy Analyzer. In Section 3, we describe the existing Alloy encoding and its limitations, review sequential circuits, and introduce SERA components. We introduce the ideas behind the SERA encoding in Section 4. We present our implementation and review TBV in Section 5. We present experimental results in Section 6 and conclude in Section 7.

## 2. ALLOY EXAMPLE

We illustrate key Alloy constructs through an example; more details about Alloy are available [15]. We prefer to first present the Alloy code and then comment on the constructs that we introduce. Consider the abstract mathematical notion of a *tree*, i.e., a connected, acyclic, undirected graph. There are various equivalent ways of defining a tree. We consider five characterizations from a standard textbook [9], model them with Alloy, and check their equivalence using the Alloy Analyzer.

Let $G = (V, E)$ be an undirected graph, where $V$ is a set (of vertices) and $E$ is a binary relation on $V$. The following statements are equivalent:

1. $G$ is a tree.
2. $G$ is connected, but removing any edge from $E$ results in a disconnected graph.
3. $G$ is connected, and $|E| = |V| - 1$.
4. $G$ is acyclic, and $|E| = |V| - 1$.
5. $G$ is acyclic, but adding any edge to $E$ results in a graph that has a cycle.

An Alloy model consists of *signature* declarations that introduce basic sets and relations, as well as *formulas* that put constraints on these sets and relations. In order to model trees, we declare a `Sig`, that is a set, of vertices and a binary relation on this set to represent the edges:

```
// V: set of vertices, E: V -> V edges
sig V { E: set V }
```

Comments are prefixed with `//`. The operator `set` states that `E` is an arbitrary relation. We represent an undirected edge between vertices $u$ and $w$ as a pair of directed edges $(u, w)$ and $(w, u)$. The relation `E` is symmetric and we use the transpose operator '`~`' to express that:

```
fact UndirectedGraph { E = ~E }     // E is symmetric

// consider non empty graphs
fact NonEmpty { some V }
```

A *fact* introduces a constraint that must be satisfied by any *instance* of the model, i.e., any satisfying assignment of values to sets and relations. The fact `NonEmpty` requires the instances to have at least one vertex.[1] The formula `some e` evaluates to true if the expression `e` evaluates to a non-empty set since `some` represents existential quantification.

We express Statement 1 using a *predicate*, i.e., a formula with free relational variables that can be *invoked* elsewhere:

```
pred InCycle(v: V, c: V -> V) {
  v in v.c or
  some v': v.c | v' in v.^( c - (v -> v') - (v' -> v))}

pred Acyclic() {all v: V | not InCycle(v, E) }

pred Connected(c: V->V) { all v1, v2 : V | v2 in v1.*c }

pred Statement1() { Connected() and Acyclic() }
```

The operator `and` is a logical conjunction operator; Alloy also defines the other logical operators: `or`, `not`, `=>` (implication), and `<=>` (iff). The keywords `all` and `some` respectively represent universal and existential quantification; `in` represents subset (and membership); '`.`' denotes relational product; '`^`' denotes transitive closure, and '`*`' denotes reflexive transitive closure. The expression `v2.^E` thus denotes the set of

[1]This condition is required for equivalence of Statements 1–5.

all vertices reachable from `v2` following edges in `E`, and the predicate `Connected` states that there is a path between any two distinct vertices. The predicate `InCycle` states that a vertex `v` is a part of a cycle according to an edge relation `c` iff there is a self-loop at `v` or `v` has some neighbor `v'` such that even if we remove the edge connecting `v` and `v'`, these two vertices are still connected. The operators '`->`' and '`-`' represent pairing (more generally, Cartesian product) and set difference, respectively.

Statements 2–5 can be defined likewise:

```
pred Statement2() {
// connected, removing an edge makes it disconnected
  Connected(E) and
  all u : V | all v : u.E  |
    not Connected( E - (u->v) - (v->u)) }

pred Statement3() { // connected and |E| = |V| - 1
  Connected(E) and #E = #V + #V - 2}

pred Statement4() { // acyclic and |E| = |V| - 1
  Acyclic() and #E = #V + #V - 2 }

pred Cyclic(c: V->V) { some v : V | InCycle(v, c) }

pred Statement5() {
// acyclic, but cyclic if any edge is added
  Acyclic()
  all u,v : V | (u->v) not in E implies
    Cyclic(E + (u->v) + (v->u)) }
```

`Statement3` and `Statement4` use the set cardinality operator '`#`'; the constraint $|E| = |V| - 1$ is represented using the Alloy formula `#E = #V + #V - 2`, since each undirected edge is represented using two directed edges. The equivalence of Statements 1–5 is expressed using a chain of implications.

```
assert EquivOfTreeDefns {
  Statement1() implies Statement2()
  Statement2() implies Statement3()
  Statement3() implies Statement4()
  Statement4() implies Statement5()
  Statement5() implies Statement1() }

check EquivOfTreeDefns for 6
```

An Alloy *assertion* introduces a formula that should be checked, in this case that the equivalence holds. The command `check` instructs the analyzer to find a counterexample to the given assertion using the specified scope, specifically 4. The analyzer proceeds by looking assignments that satisfy the negation of the formula.[2] Each such assignment effectively gives a valuation to the set `V` and the relation `E` that satisfies the negation of `EquivOfTreeDefns` (and implicitly all `fact` formulas). If the analyzer fails to generate a counterexample, the formula is valid with respect to the given scope. The user can also choose to generate more satisfying assignments; this option in the analyzer exploits solution enumeration in SAT solvers such as Berkmin [12], mChaff [23] and relsat [4].

[2]Besides `check`, Alloy Analyzer also provides a command `run` that directly finds satisfying assignments for a given formula.

For reference, for this example, our SERA implementation scaled to a scope of 32 while the Alloy Analyzer was limited to a scope of 7. Readers who operate on the principle of dessert first can refer to Section 6, specifically Table 3 for more examples.

# 3. OVERVIEW OF SERA ENCODING

In this section we motivate the need for better encoding for Alloy formulas, define sequential circuits, and introduce the principle of the SERA component.

## 3.1 The need for better encodings

Alloy Analyzer encodes the model into a satisfiability problem and calls an off-the-shelf selected SAT solver to decide the problem [14]. Briefly, an Alloy relation $T$ is encoded into a bit matrix $\mathcal{T}$. If $T$ relates the $i$-th object of type $A$ to the $j$-th object of type $B$, then $\mathcal{T}$ is formed such that the $i$ and $j$-th entry of its projection over $A$ and $B$ is set to 1, i.e., $\mathcal{T}_{A,B}(i,j) = 1$. The scope limits the range of indexes and thus the matrix is finite. Transitive closure expands the relation over the involved relational product (composition) and the union operators by introducing new variables. A quantifier folds its formula over the range with either conjunction (universal) or disjunction (existential) operations. The encoding is then mapped to a CNF formula.

We faced three main limitations when we tried to apply TBV on a pure combinational circuit translated directly from the CNF formula. First, the Alloy Analyzer often failed to produce the CNF formula due to the large number of variables needed. Second, the combinational nature of the encoding precludes the power of the sequential transforms available in a TBV framework. Finally, the huge number of variables presented to the decision algorithms prevented a conclusive result. This motivated the need for a better encoding—specifically, sequential rather than combinational encodings.

## 3.2 Sequential circuits

*Definition 1.* A *sequential circuit* is a tuple $\big((V,E),G,O\big)$. The pair $(V,E)$ represents a directed graph on vertices $V$ and edges $E$. The function $G$ maps vertices to *types*. There are three types: *primary inputs*, *bit-registers* (which we often simply refer to as *registers*), and *gates*. A subset $O$ of $V$ is specified as the *primary outputs* of $V$.

We will denote the set of primary input variables by $I$, and the set of bit-register variables by $R$. For ease of exposition, we restrict gates to have 2 fanins, and compute the NAND function; since NAND is functionally complete, this is not a limitation. For the sequential circuit to be syntactically *well-formed*, vertices in $I$ should have no fanins, vertices in $R$ should have 2 fanins (*the next-state function and the initial-value function* of that register), gates should have two fanins, and every cycle in the sequential circuit should contain at least one vertex from $R$. The initial-value functions of $R$ shall have no registers in their fanins. All sequential circuits we consider will be well-formed.

A sequential circuit can naturally be associated with a *finite state machine (FSM)*, which is a graph on the states. However, the circuit is very different from its FSM; among other differences it is exponentially more succinct in almost all

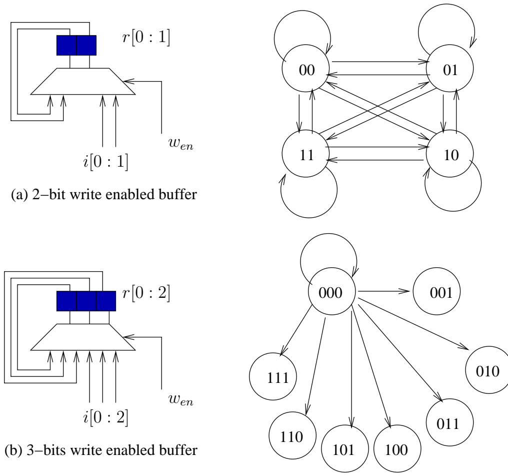



**Figure 1: Sequential circuits and FSMs**

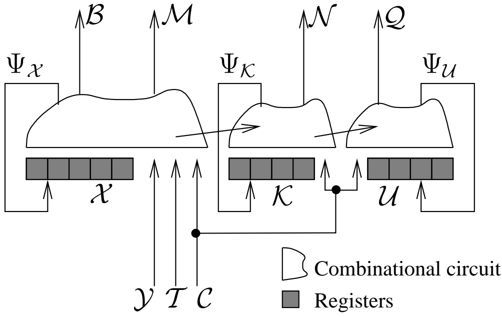



**Figure 2: Prototypical SERA component**

cases of interest [8]. Figure 1(a) shows a 2-bit write enabled buffer which uses a multiplexer enabled by $w_{en}$ to update its register bits $r$ with values in inputs $i$ when $w_{en}$ is 1 and keeps its state otherwise. It also shows the corresponding FSM with no labels on transitions for clarity of exposition purposes. Figure 1(b) shows a 3-bit version of the same circuit with the corresponding FSM. Only the arcs corresponding to state 000 are shown for clarity of exposition.

A Boolean valuation to $R$ is referred to as a *state*. Given an input, i.e., a Boolean valuation to $I$, and a state, the gates in the sequential circuit uniquely define Boolean values for each register's next-state node. The *semantics* of a sequential circuit are defined with respect to *input sequences*, where an input sequence is a sequence of input valuations. Given an input sequence and a state, the resulting *trace* is a sequence of Boolean valuations to all vertices in $V$ which is consistent with the Boolean functions at the gates. We will refer to the transition from one valuation to the next as a *step*. A node in the circuit is *justifiable* if there is an input sequence which when applied to an initial state will result in that node taking value 1. We will have occasion to refer to *targets* in the circuit; these are simply nodes in the circuit whose justifiability is of interest.

### 3.3 The SERA component

Given an Alloy formula $\Phi$ with a scope $n$, the SERA algorithm we propose will construct a sequential circuit SERA$(\Phi, n)$. The exact steps in the construction are described in Section 4. The construction process proceeds recursively on the abstract syntax graph for $\Phi$. At each node in the abstract syntax graph for $\Phi$ we construct a sequential circuit with a special structure for the formula rooted at that node. We will refer to this circuit as a *SERA component*.

Figure 2 shows a prototypical SERA component. The set of input variables $I$ is partitioned into 3 subsets: $I = \mathcal{Y} \cup \mathcal{T} \cup \mathcal{C}$. The set of outputs is partitioned into 4 subsets: $O = \mathcal{B} \cup \mathcal{M} \cup \mathcal{N} \cup \mathcal{Q}$. The set of registers is partitioned into 3 sets: $R = \mathcal{X} \cup \mathcal{K} \cup \mathcal{U}$. Depending on the formula, some of these sets may be empty.

We will use the SERA components to encode Alloy sets, relations and Boolean predicates. We will show in Section 4 how to recursively compute the SERA component SERA$(\Phi, n)$ for $\Phi$ from the SERA components for the subgraphs of the abstract syntax graph.

Readers more comfortable with software than hardware can view the SERA components as procedures that compute the value of a predicate, the membership of an element in a set, or a cardinality, depending on the sub-formula the component corresponds to. These procedures operate concurrently, and their primary inputs and outputs serve as an interface to initialize the state stored in registers, execute computations, signal completion, pass intermediate results, as well as return requested status.

More precisely, the set of primary inputs in $\mathcal{C}$ includes an input that is used to initialize the component's registers, and an input that is used to begin execution of component. The $\mathcal{T}$ inputs include relation and set data passed from other components. The $\mathcal{Q}$ outputs are used to signal other components for more data, or that computation has completed. The component uses the register sets $\mathcal{X}$, $\mathcal{K}$, and $\mathcal{U}$ to store its membership, cardinality and validity state respectively. It updates the registers through the next-state function $\Psi$, which maps a state and an input valuation to a next-state. The function $\Psi$ is decomposed into $\Psi_{\mathcal{X}}$, $\Psi_{\mathcal{K}}$, and $\Psi_{\mathcal{U}}$. The output sets $\mathcal{M}$, $\mathcal{N}$, and $\mathcal{B}$ return membership, cardinality and predicate information; The component's environment signals queries on membership and cardinality through the index inputs $\mathcal{Y}$ and control inputs $\mathcal{C}$. For example, a SERA component corresponding to a set asserts whether the element whose index is encoded by the input in $\mathcal{Y}$ belongs to the set or not. We either set the initial values of the registers to constants or connect them to primary inputs.

## 4. CONSTRUCTION OF SERA$(\Phi, N)$

We propose the use of SERA components to encode the Alloy model as a sequential circuit. This differs from techniques used in monitor circuits and high performance sequential synthesis [25, 24] since while they match a set of strings to an expression, we produce all matching assignments of a formula within a given scope. A SERA component representing a set or a relation provides an easy implementation of membership and cardinality and allows parallel access to both when the component is in a valid state. Table 1 shows the membership, cardinality and predicate functions and the interesting connections of SERA components corresponding to the different Alloy constructs and will be

**Table 1: Encoding of Alloy constructs into SERA components. The membership and cardinality columns show circuitry required to compute the membership and cardinality output and next-state functions respectively. Validity column shows the number of steps needed to complete execution of the membership and cardinality computations. Variables column shows the number of bit-registers needed.**

| Alloy Construct | Membership | Cardinality | Validity | Variables |
|---|---|---|---|---|
| 1. `Sig` | $\mathcal{M} = R[\mathcal{Y}]$ | $\mathcal{N} = \text{priority}(R)$ | 0,0 | $n$ |
| 2. `Relation` | $\mathcal{M} = R[\mathcal{Y}]$ | $\text{countOnes}(R)$ | 0,0 | $n^k$ |
| 3. $A \cup B$ | $\mathcal{M} = M_A \vee M_B$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{countOnes}(i)$ | $1, n$ | $\lg(n)$ |
| 4. $A \cap B$ | $\mathcal{M} = M_A \wedge M_B$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{countOnes}(i)$ | $1, n$ | $\lg(n)$ |
| 5. $A \setminus B$ | $\mathcal{M} = M_A \wedge \neg M_B$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{countOnes}(i)$ | $1, n$ | $\lg(n)$ |
| 6. $\neg A$ | $\mathcal{M} = \neg M_A$ | $\mathcal{N} = \lvert U_A \rvert - \lvert A \rvert$ | $1, 1$ | constant |
| 7. $\sim A$ | $\mathcal{M} = \mathcal{M}_{\mathcal{A}} \rvert_{\mathcal{Y} = \langle \mathcal{Y}[i], \mathcal{Y}[j], \mathcal{Y}[i] \rangle}$ | $\lvert A \rvert$ | 0,0 | constant |
| 8. $A.B$ | $\mathcal{M} = M_B \rvert_{\mathcal{Y} = \langle \mathcal{T}_A, Y \rangle}$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{priority}(A)$ | $n, n$ | $\lg(n)$ |
| 9. $\hat{}A$ | $\mathcal{M} = R[\mathcal{Y}]$ , $\mathcal{S}_0 = R_A$ <br> $\Psi_{R_{i+1}} = (R_i : R_i) \cup R_i$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{priority}(R_i)$ | $n, \lg(n)$ | $n \lg(n)$ |
| 10. $v \in V \mid F$ | $\mathcal{M}(a \in V) = F \rvert_{v=a}$ | $\Psi_{\mathcal{K}} = \mathcal{K} + \text{countOnes}(i)$ | $1, n$ | $\lg(n)$ |
| | Predicate | | | |
| 11. $\forall v \in V \mid F$ | $\Psi_{\mathcal{B}} = \mathcal{B} \wedge F \rvert_{v=V[i]}$ – early termination | | 1 | $n \lg(n)$ |
| 12. $\exists v \in V \mid F$ | $\Psi_{\mathcal{B}} = \mathcal{B} \vee F \rvert_{v=V[i]}$ – early termination | | 1 | $n \lg(n)$ |

described in further details in Section 4.2. It also shows the number of steps and variables needed for the computation to complete as a function of the scope $n$ and the highest arity of a relation $k$.

## 4.1 Sequential circuit example

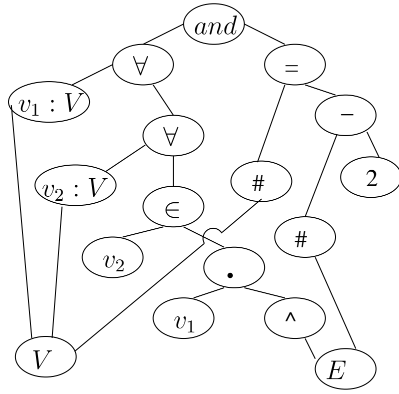
The diagram in Figure 3(a) shows an abstract syntax graph for the `Statement3` predicate. We start at the `and` node and compute the components for the two sub-formulas rooted at this node. If a component was previously instantiated, we connect to it appropriately. The sequential circuit in Figure 3(b) shows the result of this compositional encoding. The `all` statements enclose their bodies and bind the variables, the filled boxes connected to the membership output of the `V` component are register elements indexing the members of `V`. Their output is concatenated to connect appropriately to the index input $\mathcal{Y}$ of the transitive closure component. The `all` component also accumulates the membership results of the transitive closure in register elements and flag the validity once a result is available. The cardinality output of the components `E` and `V` are connected to the equality checker. Finally the predicate outputs, $\mathcal{B}$, of the equality and the quantifier components are passed as operands to the `and` component. We assume a scope of 2 and thus only one indexing register is needed per quantifier. The validity and control inputs and state bits are not shown for clarity, but the validity of the inner quantifier initializes and executes the outer one, which executes the `and` component when ready. Finally we designate the predicate output $\mathcal{B}$ of top level `and` as our target.

The 6 steps trace in Figure 4 shows an execution of the sequential circuit in Figure 3(b). Step 1 shows the graph instance, and the `V` and `E` encodings. `V` is initialized to indicate the existence of both members, and `E` is initialized to indicate the edge in the graph. The membership and cardinality state of `V` and `E` is valid immediately since it corresponds to the initial state set by primary inputs. Thus, `#E = #V +#V - 2` is true and valid immediately. In Step 2 the quantifiers $\forall v_1$ and $\forall v_2$ are executed as well as the transitive closure on `E`. Since the quantifiers depend on `*E`, they have to wait for its component to signal membership validity. Fortunately for this example, this happens in 1 step since as we will describe later in Section 4.2.6, transitive closure takes $\lg(n)$ steps to complete where $n$ is the scope. In Step 3, the relational product $v_2.*E$ is executed and it returns a vector that is immediately tested for membership of $v_1$. Since all data is valid, the $\forall v_2$ component updates its predicate state bit and increments its index $v_2$. The same happens in Step 4, and now $\forall v_2$ has completed execution and thus can signal the validity of its predicate. The $\forall v_1$ quantifier updates its predicate state, increment its index $v_1$ and initializes the $\forall v_2$ component to start execution again. Step 5 is similar to Step 3, and Step 6 witnesses the completion of execution of the $\forall v_1$ component. The propositional `and` component has now two true inputs with valid data so it evaluates to 1 and thus our target is satisfiable.

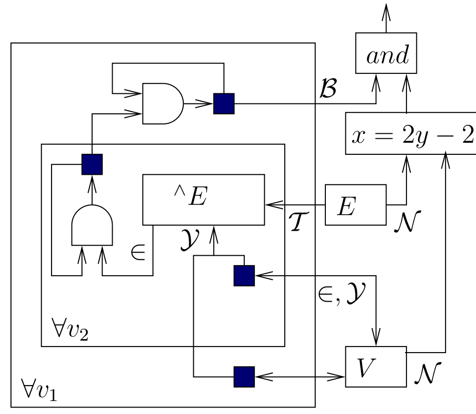
## 4.2 SERA encoding algorithm

SERA recursively traverses the abstract syntax graph for an Alloy formula $\Phi$ with a scope $n$ from its command to its `Sig`s and implicit relations. For each Alloy construct, SERA instantiates its corresponding structure as summarized in shown in Table 1. It composes each structure into the desired sequential circuit $\text{SERA}(\Phi, n)$ connecting the validity outputs to the initialization and execution inputs, and the index outputs to the query inputs. Finally, SERA designates the conjunction of the predicate output that corresponds to the asserted command, the constraints, and their validity outputs as the target of the sequential circuit.

For readers less familiar with hardware, we review two terms. A *priority encoder* is a circuit that takes $m$ inputs $x_0, \ldots, x_{m-1}$ where $m > 0$ and has outputs $v$ and $c_0, \ldots, c_{\lceil \lg m \rceil}$; if $\forall i.\, x_i = 0$ then all outputs are 0, otherwise $v = 1$ and $c_0, \ldots, c_{\lceil \lg m \rceil}$ encodes the smallest $i$ such that $x_i = 1$. The *sequential depth* of a sequential circuit is the distance from the initial state to the farthest reachable state in the underlying FSM in the absence of cycles in the path.




(a) Parse graph for Alloy constructs




(b) Sequential circuit implementation

**Figure 3: Diagram for the `Statement3` predicate**

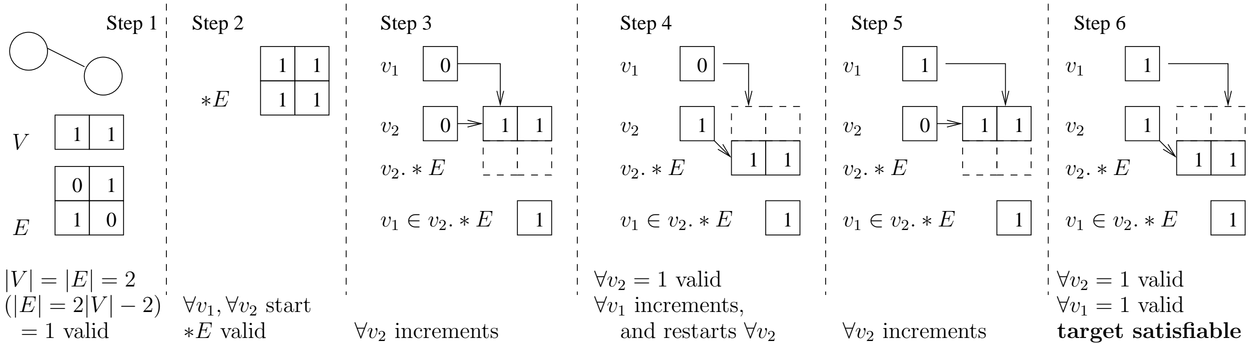



**Figure 4: SERA execution of a consistent instance of the `Statement3` predicate with a scope of 2**

### 4.2.1 `Sig`*s and relations*

The leaf nodes of the abstract syntax tree of an Alloy formula are `Sig`s and implicit relations and constitute the base case for the $\text{SERA}(\Phi, n)$ recursion. Their primary inputs are the unconstrained primary inputs of the final circuit, thereby allowing all possible models within the scope. To represent membership, we connect the initial value nodes for the registers in $\mathcal{X}$ to primary inputs. Consequently, if the $i$-th bit in the range of `Sig` $A$ is set to 1, then the $i$-th object exists. Without loss of generality, we require the set-bits to appear before the off-bits within a given range. This allows us to trivially implement cardinality by a priority encoder and membership by checking whether an index is smaller than the cardinality. As shown in Rows 1 and 2 of Table 1, we need, in general, $n^k$ variables to represent implicit relations, where $k$ is the arity of the relation.

In a tradeoff between the sequential depth and the number of variables, we allow $n$ multiple accesses to the membership functions. This requires an additional $\lg(n)$ bookkeeping variables in $\mathcal{U}$ and $n$ more indexing wires in $\mathcal{Y}$. This keeps the sequential depth of all components linear in $n$.

The *countOnes* function counts the number of matches in multiple membership checks to return the cardinality of the relation. The *priority* function implements the priority encoder to return the cardinality of the set. Membership and cardinality of `Sig`s and implicit relations depend only on the initial values, and are immediately valid as shown in the validity column in Table 1. The validity output $\mathcal{Q}$ of one component is connected to the execute control input in $\mathcal{C}$ of the next component in the hierarchy.

The internal nodes of an Alloy formula's abstract parse graph correspond to a variety of logical and relational operations, including propositional connectives, quantifiers, relational product, transitive closure, set operations, and arithmetical predicates. We now show in turn how to build the SERA component for each internal node, assuming we have SERA components for all its subnodes.

### 4.2.2 *Propositional operators*

SERA encodes each propositional operator (`and, or, not, implies, iff`) with a combinational circuit and connects $\mathcal{B}$ with the corresponding connective to the predicate outputs of the operand components. SERA also connects $\mathcal{Q}$ to a conjunction of the validity outputs of the operand components.

### 4.2.3 *Quantifiers*

Because the scope is finite, we can easily perform quantifier elimination. The universal quantification of $\theta$ by $x$ is replaced by the conjunction of $\theta$ restricted to each value $x$ can take; for existential quantification, conjunction is replaced by disjunction. The SERA component computes conjunction and disjunction sequentially, and employs an early termination mechanism where the first false or true value respectively terminates the computation. This mechanism

gives us a substantial advantage since we can abort the quantification without having to compute for the whole domain. The quantification component uses a counter to index the components embedded in its formula through their $\mathcal{Y}$ inputs, holds the current index, and accumulates a valid bit and an output bit. The latter evaluates when the valid bit indicates the quantification is done.

#### 4.2.4 Relational product

The relational product of two components $A$ and $B$ is implemented by a component that fills in the index $\mathcal{Y}_B$ of the queried component $B$ by the concatenation of tuples from the left operand $A$ and the actual argument $\mathcal{Y}$ to the relational product component. All the matches are saved in registers, which are reserved for this purpose, and then produced sequentially, one after the other, while updating the cardinality count and setting the validity bits. The cardinality is computed using a priority encoder over the matches for the concatenated index. Since we allow $n$ parallel membership checks, we can guarantee the validity of the membership and cardinality data in at most $n$ steps.

#### 4.2.5 Transpose

We define a unique variable order to respect the order in which `Sig`s were declared in $\Phi$ and we exploit that order to trivialize our type-determination functions. The transpose operator may produce a result which conflicts with the unique variable order. SERA attempts to rewrite the formula in question to normalize transposition. In cases of conflict, such as the `UndirectedGraph` constraint `E =~E`, or in cases of obvious suboptimality introduced in the variable ordering, SERA resorts to adding a redundant variable appropriately. We show the needed substitutions in Row 7 of Table 1. SERA also adds a constraint that indicates the equivalence of the redundant data so that redundancy removal techniques can easily exploit the hint.

#### 4.2.6 Transitive closure

Transitive closure in Alloy repeats one or more relational product (composition) infinitely many times. SERA encodes the transitive closure of each relational product using *iterative squaring* [8]. The details are expressed in Row 9 of Table 1. This allows us to use only $n\lg(n)$ variables and allows the computation to converge within $\lg(n)$ steps. Due to the Alloy context based type system, we may need to apply multiple compositions.

#### 4.2.7 Set operations

As shown in Rows 3 through 6 of Table 1 the membership function and its validity can be encoded as combinational functions of the membership and validity of the operand components. The cardinality and the subset check require $n$ steps to converge. In a sequential run of a union or an intersection, duplicates can easily be eliminated by relaxing one of the indexes of the operand components. The component iterates the execution of the *countOnes* function with the indexes resulting in true and valid membership outputs from the operand components.

#### 4.2.8 Arithmetic predicates

Another source for Boolean values in Alloy is integer arithmetic comparisons, which may involve cardinality values. The Alloy Analyzer uses the scope to allocate a finite number of integers to model the integer space and simplify the arithmetic predicates. We encode arithmetic operators with combinational circuits in addition to a valid state bit that propagates the validity bits of the operand components. Note that all integer valuations, less the cardinality values, are considered valid by default.

THEOREM 1. *Let $\Phi$ be an Alloy formula and let $n$ be a given scope. There exists a finite trace $\mathcal{Z}$ that sets the target of $P = SERA(\Phi, n)$ to 1 at its last cycle iff there exists an assignment that satisfies $\Phi$ within the scope $n$.*

**Sketch of proof.** Theorem 1 follows from a straightforward induction on the length of $\Phi$. We use the validity entries of Table 1 to establish a tight upper bound on the length of $\mathcal{Z}$ if $\mathcal{Z}$ exists. The bound depends on the length of the formula and the scope.

The base cases, where $\Phi$ is either `Sig` or an implicit relation, are trivial and always satisfiable. We can construct a trace $\mathcal{Z}$ of length 1 that assigns the primary inputs initializing the registers of the corresponding component appropriately, for instance set them all to 1. The proofs for the correctness of the remaining constructs are similar to one another. We will illustrate the key ideas for existential formulas; the rest follow similarly.

If $\Phi = \exists v \in \Gamma.F(v)$, and $\Phi$ is satisfiable within the scope $n$, then there is a model $\sigma \subseteq \text{range}(\Gamma)$ and there is an element $\alpha \in \sigma$ such that $F(v = \alpha)$ is statisfiable. Let $C_\Phi$ by the SERA component corresponding to the $\exists v$ statement and let $C_\Gamma$ and $C_F$ be the SERA circuits corresponding to $\Gamma$ and $F$ respectively. By the induction hypothesis, there exists a trace $\mathcal{Z}_\Gamma$ that sets $C_\Gamma$ to a state matching $\sigma$ in $k_\Gamma$ steps, and there exists a trace $\mathcal{Z}_\alpha$ that sets the predicate and validity outputs of $C_F$ to true in $k_F$ steps after presenting $C_F$ with $\alpha$. The component $C_\Phi$ enumerates all the possible elements in range($\Gamma$), queries $C_\Gamma$ for their membership and concurrently presents them to $C_F$ for evaluation in case the membership test was valid. After at most $i \leq |\text{range}(\Gamma)|$ steps from the point it starts, it is guaranteed to find $\alpha$. Since it accumulates a disjunction of the results of $C_F$, the first valid true predicate output of $C_F$ terminates the computation. Following the above steps we can construct a concatenated trace $\mathcal{Z}_\Phi$ of length $k_\Gamma + i \times k_F$ that sets the Boolean predicate $\mathcal{B}$ of $C_\Phi$ to true as well as its predicate validity output. Since range($\Gamma$) is bound to be either a set, or a relation with arity $a$, then $|\text{range}(\Gamma)| \leq n^a$ and $\mathcal{Z}_\Phi$ is finite. In case $\Phi$ is not satisfiable, then $C_\Phi$ is guaranteed to try all models of range($\Gamma$) and complete execution in a finite number of steps ($2^{n^a}$). By the induction hypothesis, all elements in models that match $C_\Gamma$ will not satisfy $C_F$. At the end of the iteration the output predicate of $C_\Phi$ will be set to false and its corresponding validity output will be asserted. After that the $C_\Phi$ component will stabilize and never change state. Consequently there exists no trace that satisfies the predicate output of $C_\Phi$.

### 4.3 Optimizations

Without loss of generality we support the same scope value for all `Sigs`. This allows us to keep our type-determination

**Table 2: TBV algorithms in order of their runtimes.**

| | |
|---|---|
| **COM** | merges functionally equivalent gates using low complexity analysis [17]. |
| **EQV** | makes intelligent guesses on equivalency [5] and performs expensive checks that allow huge gate merging reductions when they pass. It also exploits structural symmetry detection. |
| **RET** | reduces the number of registers by shifting them across combinational gates [16]. |
| **BMC** | attempts a conclusive result within a given limit on steps or resources [21]. |
| **PRE** | performs compositional minimization [28] by isolating a component in the sequential circuit and detecting equivalent states within the component, and then reduces the input space of the component. |
| **BIG** | replaces a target by a re-encoding, i.e., a set of states which will hit that target within $k$ time steps [18]. |
| **LOC** | overapproximates the target by replacing the gates on a boundary with free variables [18]. If the target is proved by subsequent transforms, the proof holds for the original sequential circuit. Otherwise, LOC checks whether the counter-example is an actual one. If not, it refines the approximation. |
| **CUT** | replaces a set of gates with a simpler yet equivalent sequential circuit; reduces input count via defining inputs as functions of each other [18]. |
| **SCH** | conducts a semi-formal search for a target using a hybrid approach of random simulation, symbolic simulation, and induction [13, 17]. |

as simple as checking whether an index lies within a range. We also simplify the counters embedded in all components by restricting the scope to be *a power of two*. With this restriction we allow the counters to start at any state and terminate when they reach that state again. The corresponding component can then call the counter cycle state its idle state. The type-determination of a certain index is now simplified to an appropriate shift operation. In most cases some SERA components are guaranteed to finish their production before other components even begin. We use this fact to allow memory sharing between non-overlapping components. Note also that we separated variables based on the functions they are used for. This allows huge *cone of influence* reductions if for example the cardinality of a components is not checked. Furthermore, this introduces redundancy which can be exploited by redundancy removal transforms in TBV.

# 5. THE SERA PROTOTYPE

Based on Theorem 1, we developed a prototype implementation of SERA. As illustrated in Figure 3, we parse the Alloy model into a DAG of `Sig`s, relations, and operators with the root as the command to be executed and the leaves as the `Sig`s and implicit relations. Note that we generate a DAG since our analysis tries to reuse syntax-equivalent nodes. We also wrote *generic* parameterized SERA components for each of the constructs as VHDL templates. SERA recursively traverses the DAG of Alloy formula $\Phi$ with scope $n$ from the command down to the signatures and relations, and instantiates the appropriate VHDL entities with the appropriate *wiring* connections and generic parameters. We end up with a hierarchical VHDL design with an asserted signal designated as target. We pass the VHDL to an internal IBM frontend compiler that generates the sequential circuit SERA$(\Phi, n)$. The TBV framework provides a satisfying trace for the target of the reduced sequential circuit. The trace is translated to the original sequential circuit using *tracelifting* algorithms specific to each transform. We map the trace corresponding to SERA$(\Phi, n)$ via a simple decoding of values assigned to the initial-value functions of all `Sig` and implicit relation components. Step 1 of Figure 4 illustrates how trivial the mapping is. We map the initial values of the membership state of the `V` component as two vertices of type `V` and the initial values of the membership state of the `E` component as the existence or absence of arcs with label `E` between the vertices. Our approach can be readily implemented with public domain tools with TBV capabilities such as SIS [10] and VIS [19]. We next review ideas underlying TBV.

## 5.1 Transformation-based verification

TBV applies reduction techniques iteratively on a sequential circuit to reduce the verification complexity by reducing the number of gates, registers, and inputs in the circuit. Then it attempts to solve the problem via decision techniques such as bounded model checkers, circuit SAT solvers, or semi-formal searches. The decision techniques aim to find a satisfying *trace*, that is an assignment to initial value functions of registers and a sequence of input valuations that result in asserting the target gate to a true value at the last cycle of the trace. In Table 2, we briefly describe various transforms that were used in the context of this work and comment on their efficiency. We had to manually experiment with various techniques in order to find a successful flow of transforms for each of the problems. This effort, although nontrivial, requires only general knowledge of the available transforms and the synergies between them and can be automated as reported in [18].

# 6. EXPERIMENTS

We chose three examples when we began this research. The tree integrity entries in Table 3 show results for checking

**Table 3: Detailed comparison of SERA results and the Alloy Analyzer**

| | Tree integrity | | | | File System | | | | LISP lists | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Alloy Analyzer results** | | | | | | | | | | | | |
| Scope | 5 | 6 | 7 | 8 | 5 | 6 | 7 | 8 | 10 | 8 | 9 | 10 |
| Satisfiable | NO | | | | YES | | | | YES | NO | | |
| Vars | 212 K | 641 K | 1,682 K | SO | 4.7 | 7.2 | 10 | 13 | 17 | 2.9 | 18.9 | 23.7 |
| Clauses | 641 K | 2,014 K | 5,428 K | | 14.5 | 14 | 33 | 43.8 | 98 | 68.8 | 103 | 129 |
| SAT (sec) | 156 | 658 | SO | | 5 | 6 | 21 | 28 | 36 | 55 | 2 K | >14 K |
| **SERA: sequential circuit original size** | | | | | | | | | | | | |
| Scope | 4 | 8 | 16 | 32 | 4 | 8 | 16 | 32 | 32 | 8 | 16 | 32 |
| Inputs | 12 | 32 | 80 | 192 | 22 | 41 | 68 | 143 | 178 | 46 | 88 | 178 |
| Registers | 36 | 94 | 780 | 3.3 K | 112 | 728 | 4.3 K | 27.7 K | 22.3 K | 1 K | 5.2 K | 25.7 K |
| NANDs | 354 | 1,087 | 11 K | 22 K | 1,054 | 4.5 K | 20 K | 422 K | 285 K | 14 K | 111 K | 328 K |
| **SixthSense: size after TBV reductions** | | | | | | | | | | | | |
| Inputs | 0 | 32 | 76 | 98 | 18 | 22 | 38 | 76 | 111 | 42 | 53 | 121 |
| Registers | 0 | 35 | 67 | 216 | 44 | 151 | 1 K | 1.4 K | 1.8 K | 324 | 198 | 2.4 K |
| NANDs | 0 | 987 | 2.9 K | 7.1 K | 768 | 2.9 K | 7.9 K | 9 K | 14.3 K | 2.9 K | 1.5 K | 17.9 K |
| Time (sec) | 12 | 16 | 87 | 4.1 K | 19 | 32 | 292 | 1.5 K | 163 | 217 | 287 | 473 |
| **Resources to solve the reduced sequential circuit** | | | | | | | | | | | | |
| Time (sec) | 1 | 2 | 133 | 209 | 7 | 7 | 125 | 212 | 8 | 12 | 54 | 102 |
| MEM (MB) | 12 | 24 | 91 | 134 | 10 | 16 | 45 | 101 | 34 | 21 | 44 | 88 |

the `EquivOfTreeDefns` assertion. The other two examples are representatives from the standard Alloy distribution that have been the subject of research by multiple Alloy related papers for the past 6 years. The *file system* example describes relations between directories, files, and a root directory in a Unix-like file system and asserts alias consistency and acyclicity. The *LISP list* example describes empty and non-empty nested lists of objects, defines equivalency between lists, and asserts symmetry and reflexiveness properties of the equivalence definition; it also asserts that all empty lists are equivalent.

For the unsatisfiable formula from the tree example, the Alloy Analyzer could not perform checks with a scope larger than 6; SixthSense applied to SERA-generated sequential circuits was able to check these formulas for scopes upto 32. For unsatisfiable formulas from the list example, the Alloy Analyzer failed beyond a scope of 9, whereas SERA could check these formulas for scopes upto 32. For satisfiable formulas from the list and file suites, we specified lower bounds on the minimum size of the list and file examples as Alloy `facts`, and were able to find counterexamples in scopes 3× larger than the Alloy Analyzer.

The entries in the first set of rows of Table 3 show the number of Boolean variables and clauses that were used in the Alloy encoding and the time it took the SAT solver to decide the CNF formula for a given scope. The satisfiable row indicates whether the predicate tested is satisfiable or not. We ran all experiments on a 1.7 GHz Pentium 4 machine with 1 GB memory. For our examples, the Berkmin solver consistently outperformed all the other solvers that come with the standard Alloy distribution, so we tabulate the results for Alloy using Berkmin. The Alloy Analyzer was able to validate the tree equivalence for scopes up to 6. For a scope of 7, the SAT solvers spaced out (SO), and for a scope of 8 the Alloy Analyzer ran out of memory and could not generate the SAT problem.

The second set of rows in Table 3 shows the size of the sequential circuits produced by SERA. The third set of rows shows the size of the sequential circuit after successful TBV flows simplified the structure of the problem and made it possible to deploy a resource-bounded decision algorithm. They also show the time needed to achieve these reductions. In the fourth set of rows we show the running time taken to solve the problem and the total memory used to both reduce and decide the problem.

In general, we noticed that the number of needed memory elements grew quadratically with the exponential growth of the scope and this agrees with the highest complexity of SERA. Using sequential encoding, we were able to scale Alloy analysis to a scope of 32 with relatively acceptable computational resources and time limitations.

All cases required applying iterative reduction transformations. In the case of the tree equivalence example, LOC was instrumental in reducing the problem, also COM and EQV did a good job of merging the common parts of the different equivalent tree characterizations. In the file system case, the counter-examples happened to be relatively sequentially deep since they depended on comparisons between distinctively initialized transitive closures, high cardinality comparisons, and conflicting transpose statements. SCH was able to detect counter-examples once the design was reduced using EQV.

# 7. CONCLUSION AND FUTURE WORK

We proposed an automated sequential encoding method for Alloy models and verified the resulting sequential circuit within a TBV framework. SERA traded complexity in space with complexity in time. We encoded our problem with less variables and enabled TBV power. Thus more transformations are now available to attack the problem of checking finitized Alloy models. We were able to show that a scope of 32 is feasible using reasonable resources, and that in prac-

tice the number of variables in the sequential circuit grew quadratically with the growth of the scope of the problem.

In the future we plan to optimize SERA to allow reuse of variables. We also plan to explore how to determine an upper bound on the scope for a given Alloy formula as this may allow to conclude complete checks. Moreover, we would like to explore how our approach of sequential encoding may be extended to other logic specifications such as MACE/OTTER [20], S1S [3], and Presburger Arithmetic [26].

# 8. REFERENCES


[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, Dec. 1999.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Design automation conference*. ACM Press, 2002.

[3] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis using S1S. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[4] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *National Conference on Artificial Intelligence*, 1997.

[5] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, November 2000.

[6] D. Box. *Essential COM*. Addison Wesley, 1998.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *International Symposium on Software Testing and Analysis, Rome, Italy*, 2002.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2), 1992.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[10] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[11] R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3), 1982.

[12] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

[13] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *Int'l Conference on Computer-Aided Design*, Nov. 2000.

[14] D. Jackson. Automating first-order relational logic. In *ACM-SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, 2000.

[15] D. Jackson. *Alloy 3.0 Reference Manual*, May 2004. http://alloy.mit.edu/reference-manual.pdf.

[16] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer-Aided Verification*, July 2001.

[17] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, 21(12), 2002.

[18] H. Mony et al. Scalable automated verification via expert-system guided transformations. In *Formal Methods in Computer-Aided Design*, Nov. 04.

[19] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Computer-Aided Verification*, July 1996.

[20] W. McCue. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, May 1994.

[21] I.-H. Moon, G. D. Hachtel, and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Formal Methods in Computer-Aided Design*, Nov. 2000.

[22] I.-H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley. Simplifying circuits for formal verification using parametric representation. In *Formal Methods in Computer-Aided Design*, Nov. 2002.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Conference on Design Automation*, June 2001.

[24] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *Design Automation Conference*, pages 129–134. ACM Press, 2002.

[25] A. Seawright and F. Brewer. High-level symbolic construction technique for high performance sequential synthesis. In *DAC '93*. ACM Press, 1993.

[26] T. Shiple, J. Kukula, and R. Ranjan. A comparison of presburg engines for efsm reachability. In *Computer-Aided Verification*, June 1998.

[27] I. A. Shlyakhter. *Declarative Symbolic Pure-Logic Model Checking*. PhD thesis, MIT, February 2005.

[28] F. Zaraket, J. Baumgartner, and A. Aziz. Scalable compositional minimization via static analysis. In *International Conference on Computer Aided Design*, Nov. 2005.