Sequential Circuits for Relational Analysis

Fadi Zaraket
IBM Systems & Technology Group

Adnan Aziz Sarfraz Khurshid
The University of Texas at Austin

Abstract

The Alloy tool-set has been gaining popularity as an alternative to traditional manual testing and checking for design correctness. Alloy uses a first-order relational logic for modeling designs. The Alloy Analyzer translates Alloy formulas for a given scope*, i.e., a bound on the universe of discourse, to Boolean formulas in conjunctive normal form (CNF), which are subsequently checked using propositional satisfiability solvers.*

We present SERA, a novel algorithm that compiles a relational logic formula for a given scope to a sequential circuit*. There are two key advantages of sequential circuits: they form a more succinct representation than CNF formulas, sometimes by several orders of magnitude. Also sequential circuits are amenable to a range of powerful automatic analysis techniques that have no counterparts for CNF formulas. Our experiments show that SERA, used in conjunction with a sequential circuit analyzer, can check formulas for scopes that are an order of magnitude higher than those feasible with the Alloy Analyzer.*

1 Introduction

As software systems steadily grow in complexity and size, designing such systems manually becomes more and more error-prone. The last few years have seen a new generation of lightweight design tools that allow formulating designs formally, as well as checking their correctness to detect crucial flaws that, if not corrected, could lead to massive failures. The Alloy tool-set is one such design tool that is rapidly gaining prominence [19, 30]. The user formulates their design in the Alloy language, which is a first-order logic (with transitive closure) based on relations, and checks the correctness properties using the Alloy Analyzer.

The Alloy tool-set has been used successfully to check designs of various applications, such as Microsoft's Common Object Modeling interface for interprocess communication [8], the Intentional Naming System for resource discovery in mobile networks [1], and avionics systems [13], as well as designs of cancer therapy machines [20].

The Alloy language provides a convenient notation based on path expressions and quantifiers, which allow a succinct and intuitive formulation of a range of useful properties, including rich structural properties of software.

Much of Alloy's utility, however, comes from its fully automatic analyzer, which performs a bounded exhaustive analysis using propositional satisfiability (SAT) solvers. Given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy formula into a Boolean formula in conjunctive normal form (CNF), and solves it using an off-the-shelf SAT solver [27, 15, 23].

1.1 The case for sequential circuits

While recent advances in SAT have enabled the Alloy Analyzer to check designs of real systems, these designs often need to be partial, leaving out important aspects of the systems, to enable the analysis to complete. Moreover, the analysis is typically limited to relatively small scopes, e.g., fewer than 10 entities in a file system.

There are two limiting aspects of the current Alloy analysis. **(1.)** The translation to CNF depends on the scope; a small increase in the scope can cause a large increase in the size of the translated CNF formula due to quantifier elimination and transitive closure unrolling, e.g., for an undirected 7-node tree the translation generates a formula with over 1 million variables and 5 million clauses. **(2.)** The SAT solver is restricted to using optimizations that apply at the level of CNF formulas.Often times when the analyzer successfully generates a large CNF formula, the underlying solver chokes.

To increase Alloy's applicability to a wider class of systems as well as to checking more sophisticated properties of designs and gaining more confidence in the results, we need to scale Alloy's analysis to significantly larger scopes.

The limitations of the CNF encoding motivated us to develop *sequential encoding for relational analysis* (SERA), an algorithm which encodes Alloy formulas as *sequential circuits* and decides them using a sequential circuit solver. A sequential circuit can be viewed as a restricted C++ program, specifically a multi-threaded program in which all variables are either integers, whose range is statically bounded, or Boolean-valued, and dynamic allocation is forbidden [14].

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

Given an Alloy formula and a scope, SERA automatically derives a sequential circuit and a Boolean variable therein that serves as an *invariant*, i.e., the variable can be set to $true$ if and only if the Alloy formula is satisfiable within the scope. (For ease of exposition, we will sometimes refer to the output of SERA as a circuit, with the invariant being implicit.)

We use *SixthSense* [24, 31], a tool developed at IBM, to automatically check invariants on sequential circuits. SixthSense reads designs expressed in the VHDL design language [2]. Sequential circuits, as described in the preceding paragraph, can be efficiently translated into VHDL using inlining [12, 14].

There are two key advantages to compiling Alloy formulas into sequential circuits rather than CNF formulas:

Advantage 1 Our encodings are much more succinct than those generated by Alloy—in cases, Alloy's encoding algorithms produce a data structure that uses several orders of magnitude more memory to represent.

Advantage 2 Casting the decision problem for an Alloy formula as an invariant check on a sequential circuit allows us to make use of a number of powerful automated analysis techniques that have no counterpart in CNF analysis. Empirically, our implementation scales to scopes that are an order of magnitude higher compared to the Alloy Analyzer; it also concludes satisfiable and unsatisfiable checks faster.

Intuitively, Advantage 1 holds because sequential circuits are imperative and stateful while CNF formulas are declarative and state-free. For example, sequential circuits can represent quantification using loops without the need for an expansion with respect to the scope. Moreover, sequential circuits can store and reuse intermediate results in local variables.

We justify Advantage 2 by illustrating some automatic analysis techniques that SixthSense implements. These directly apply to sequential circuits generated by SERA.

Abstraction. Consider the verification of library code L which uses a sophisticated memory allocator for performance.

Let the library $L*$ be L with L's allocator abstracted to a simpler allocator that nondeterministically selects a block from the set of free blocks. Since the simpler allocator uses nondeterminism, if an invariant holds of $L*$, it holds of L. The simpler allocator in $L*$ makes verifying invariants on $L*$ easier than verifying the same invariants on L.

While there exist efficient algorithms for automatically identifying components for abstraction in sequential circuits [22, 24], abstraction for CNF formulas is much harder. This is because there is no structure in a CNF formula to guide the abstraction algorithm—the clauses are unordered.

Note that an invariant may fail on $L*$, but hold of L, e.g., L's code makes use of details from the implementation of the allocator beyond those exported from the abstract interface. SixthSense will automatically identify a negative as false, and roll back the abstraction [22, 24].

Compositional minimization. Consider the verification of a spanning tree algorithm T which uses a balanced search tree (BST) to manipulate sets.

With respect to its abstract interface, a BST implementation of sets is functionally equivalent to a list implementation of sets. Let spanning tree algorithm T' be T with sets implemented using lists. Because a BST is more complex than a list, verification of T' is easier than verification of T. Since the list and BST representations of sets are equivalent with respect to their abstract interface, an invariant holds of T iff it holds of T'.

There exist several techniques for automatically identifying components and minimizing them in sequential circuits [7, 31, 4]. These techniques are based on the notion of equivalent states [17]. Analogous techniques do not exist for CNF formulas, as there is no notion of state.

Logic optimizations. There are literally hundreds of techniques for automatically optimizing sequential circuits; examples include variable minimization via retiming [21], common subexpression extraction [9], and exploiting reduced observability and controllability at internal components [3, 28].

SixthSense automatically iterates through the techniques described above as well as some other techniques, to simplify its input sequential circuit. It decides the simplified circuit using a variety of techniques, such as bounded model checking, circuit SAT solving, invariant enlargement, and semi-formal search [26, 27, 16, 5, 24].

We make the following key contributions:

1. **New encoding for Alloy:** We propose SERA, an algorithm that encodes an Alloy formula and scope as a sequential circuit.

2. **Relational analysis:** We enable the use of sequential circuit verification including many powerful reduction techniques for relational model checking.

This paper is structured as follows. We first visit an example in Section 2 to illustrate Alloy and its Analyzer. In Section 3, we describe the existing Alloy encoding, review sequential circuits, and introduce SERA components. We introduce the SERA encoding in Section 4. We evaluate our approach in Section 5 and conclude in Section 6.

2 Alloy Example

We illustrate the key Alloy constructs through an example; more details are available elsewhere [19]. Consider a *tree*, i.e., a connected, acyclic, undirected graph. There are various equivalent ways of defining trees. We take five textbook definitions [11], model them in 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 for non-empty graphs:

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 constrain them. For tree, we declare signature `V` to model vertices and binary relation `E` to model edges:

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

The keyword `set` makes `E` an arbitrary relation. We represent an undirected edge between vertices u and w as a pair of directed edges (u, w) and (w, u). `E` is a symmetric relation, which we express using the transpose operator '`~`':

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

fact NonEmpty { #V >= 1 } // consider non empty graphs
```

A *fact* introduces a constraint on the declared sets and relations. The fact `NonEmpty` uses the cardinality operator `#` to state there is at least one vertex.

We express Statement 1 using a *predicate*, i.e., a formula that may have free variables and 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(E) and Acyclic() }
```

The operator `and` is logical conjunction; Alloy also provides `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 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)) }
```

We represent the constraint $|E| = |V| - 1$ using the formula `#E = #V + #V - 2`, since each undirected edge is represented using two directed edges. We express the equivalence of Statements 1–5 using a chain of implications:

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

//final check is subject to facts being true
check EquivOfTreeDefns for 4
```

An Alloy *assertion* introduces a formula that should be checked, in this case whether the equivalence holds. The command `check` instructs the analyzer to find a counterexample to the given assertion using the specified scope, specifically 4.

Besides `check`, Alloy Analyzer also provides a command `run` that directly finds *instances*, i.e., valuations to `V` and `E` that satisfy a given formula as well as the facts. The user can also choose to enumerate satisfying assignments by selecting an enumerating solver [15], mChaff [27] and relsat [6].

To check `EquivOfTreeDefns`, the analyzer searches for a counterexample, an instance which satisfies the negation of the assertion while satisfying all the facts. If the analyzer fails to generate a counterexample, the formula is valid with respect to the given scope.

For `EquivOfTreeDefns`, the Alloy Analyzer failed to complete its check for a scope of 7; we timed it out after 14,000 seconds. In contrast, SERA successfully checked the assertion for a scope of 32. Section 5.2, specifically Table 4, presents detailed results, including those for other Alloy designs.

3 SERA Overview

We first review the Alloy encoding to introduce necessary terminology and then introduce the SERA component.

3.1 Alloy encoding

Alloy Analyzer encodes the problem of checking validity of an Alloy formula within a given scope into a CNF satisfiability problem; it then calls an off-the-shelf SAT solver to decide the problem [18]. 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 $true$, i.e., $\mathcal{T}_{A,B}(i,j) = 1$. The scope limits the range of indices, 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.

3.2 The SERA component

For the reasons given in Section 1.1 we developed SERA, an algorithm for encoding Alloy formulas into sequential circuits. Given an Alloy formula Φ with a scope n, the SERA algorithm constructs a sequential circuit SERA(Φ, n). The exact steps in the construction are described in Section 4. The construction proceeds recursively on the abstract syntax directed acyclic graph (DAG) for Φ. At each node in the DAG for Φ we construct a sequential circuit with a special structure for the formula rooted at that node. We refer to each such circuit as a *SERA component*.

We illustrate the SERA components using C++ classes and objects. For ease of exposition we omit access modifiers and trivial constructors. The abstract class `Component` in Table 2(a) describes the generic interface of all SERA components. `Component` inherits from `Thread` to denote that all components run concurrently and its member function `nextState` is the thread's entry point.

```
vector<Component> compVec;
bool circuit() {
  int depth = 0;
  for(int i=0; i<compVec.size(); i++)
    compVec[i].initialState();
  while( (!compVec[0].predValid()) &&
    (depth++ < compVec[0].depth())) {
    // run all threads concurrently
    for( int j=0; j<compVec.size(); j++)
      compVec[j].nextState();
    waitForAllThreads();}
  return compVec[0].pred();};
```

The vector `compVec` contains all the components SERA generated, and `compVec[0]` is the top level component which corresponds to Φ. The `while` loop models time where each iteration is a *step*, and the number of steps it takes the loop to terminate is the *depth* of the circuit. The function `circuit` makes sure to call all `nextState` functions synchronously at every step. The `for` loop spawns all the threads concurrently and then `circuit` waits for all `nextState` calls to finish. The invariant we need to check is the `pred` function of `compVec[0]`—Φ will be satisfiable in scope n iff `compVec[0].pred()` returns $true$ on termination of `circuit()`.

We classify the functions `in`, `card`, and `pred` as output functions. The functions `memberValid`, `cardValid`, and `predValid` are Boolean validity functions and their return values signal whether a value returned by the corresponding output function is valid or not.

1. Function `in` takes index arguments and returns whether the set or relation described by the component contains the variable or tuple denoted by the indices.
2. Function `card` returns the cardinality of the set or the relation described by the component.
3. Function `pred` returns the Boolean value of a predicate if the component corresponds to a Boolean expression.

Depending on the Alloy sub-formula the component corresponds to, some of these functions may never be invoked, and thus may be left unimplemented.

The component contains references to other components. A function in a component uses the references to execute other components if their output functions are not valid yet and query them once valid. The other data elements of a component constitute its *state*. We refer to the values of the non-reference data elements of a component at a specific step as the state of the component. The `intialState` function initializes the component to its initial state, and the `nextState` function updates the state. The `evaluate` and `terminate` functions are control functions. They start the execution of the component if it was not in a valid or running state, and force it to stop execution if a top hierarchy does not need the result anymore.

We next define several terms related to the *semantics* of sequential circuits. We use these terms to prove SERA's correctness. *Inputs* are arguments passed to a component's functions and non-deterministic assignments generated by calls to the function `choose` in `initialState` functions. The semantics of a sequential circuit are defined with respect to *input sequences*, where an input sequence is a sequence of input assignments. Given an input sequence and an initial state, the resulting *trace* is a sequence of Boolean assignments to all functions returning a Boolean value in the circuit. A *transition* happens when an assignment changes at a specific step. A predicate in the circuit is *justifiable* if there is an input sequence which when applied to an initial state will result in that predicate taking the value $true$.

4 Construction of SERA(Φ, n)

We use SERA components to encode the Alloy model as a sequential circuit. Tables 2 and 3 show C++ classes

Table 2. Pseudo C++ description for abstract component, `sig`, and union components.

```
class Component:public Thread{
public:

  bool predicate();
  bool in(iter u);
  bool in(iter u,iter v);
  int card();

  bool memberValid();
  bool cardValid();
  bool predValid();

  void evaluate();
  void terminate();

  void initialState();
  void nextState();
  int depth();};
```

(a) Sequential component interface

```
template<int scope>
class Sig:public Component{
  class iter{/*omitted details*/};
  int size;
  bool in(iter v){
    return v < size;};
  int card() {
    return size;};

  bool memberValid() {
    return true;};
  bool cardValid(){
    return true;};

  void initialState() {
    //non-deterministic choice
    size = choose()%scope;};
  void nextState() {
    size = size;};
  int depth(){return 1;};};
```

(b) Sig-set component

```
template<class S1, class S2>
class Union : public Component{
  class iter{/*omitted details*/};
  S1 & V1; S2 & V2;

  bool in(iter u){
    return V1.in(u) || V2.in(u);};
  int card(){
    bitVector in;
    for(iter v;v.valid();v++){
      in[v] = V1.in(v) ||
        V2.in(v);};
    return countOnes(in); };

  bool memberValid(){
    return V1.memberValid() &&
      V2.memberValid();};
  bool cardValid(){
    return memberValid();};
  int depth(){return
    max(V1.depth(), V2.depth());};};
```

(c) Set union operator component

Table 1. Time steps and Boolean state variables for the sequential SERA components.

Alloy Construct	Validity depth		Number of variables
	Membership	Cardinality	
Signature	1	1	$\lg(n)$
Relation	1	1	n^k
Relational Product	n	n	$\lg(n)$
Transitive Closure	$\lg(n)$	$\lg(n)$	n^2
	Predicate		
Universal Quantifier	1 to n		$\lg(n)$
Existential Quantifier	1 to n		$\lg(n)$

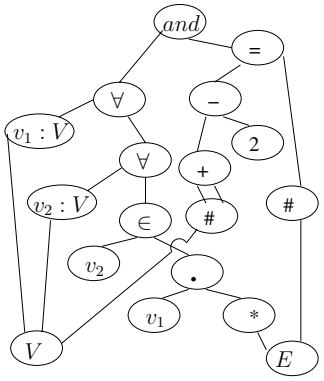

Figure 1. `Statement3` **predicate diagram**

describing some of the components corresponding to Alloy constructs. Section 4.2 describes the constructs and their translation in detail. The component's computation is complete when its `predValid` function returns $true$. Table 1 shows the number of state variables as well as an upper bound on the number of steps needed for the computation of an individual component to complete as a function of the scope n. The final depth of the SERA circuit depends on the scope n, and the formula itself and thus it can be computed at compile time via calling the `depth` function of the top level component.

4.1 Sequential circuit example

The diagram in Figure 1 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 code below describes the resulting SERA sequential circuit for the `Statement3` predicate.

```
typedef Sig<2> S1;
typedef S1::iter S1Var;

S1 V;
Relation<S1,S1> E;
TClosure<S1,S1> Et(E);
S1Var v2, v1;
Product P(v1, Et);
Belongs B(v2, P);
ForAll<S1> A1(v1, V, B);
ForAll<S1> A2(v2, V, A1);

IntPlus P1(V, V);
IntMinus M1(P1, 2);
IntEqual E1( M1, E);
And A3(A2, E1);
```

We pass 2 as a template parameter for `S1` to set its scope at compile time. The constructor of each component initializes its references appropriately and adds itself to the global vector of components. Classes `Sig`, `Relation`, `TClosure`, `Product`, `Belongs`, `ForAll`, `IntPlus`, `IntMinus`, `IntEqual`, and `And` inherit all from `Component` and each implements the Alloy construct its name suggests.

The 6-step trace in Figure 2 shows an execution of the sequential circuit corresponding to `Statement3`. Step 1 shows the graph instance, and the `V` and `E` encodings. `V` is initialized to indicate the existence of both members, and `E`

Table 3. Pseudo C++ description for relation, universal quantifier, and transitive closure components.

```
template<class Sig1, class Sig2>
class Relation:public Component{
  class iter{/*omitted details*/};
  Sig1 & V1;
  Sig2 & V2;
  bitMatrix R[Sig1::scope]
   [Sig2::scope];
  bool in(iter v, iter u){
   return R[v][u];};
  int card(){
   return countOnes(R);};

  bool memberValid(){
   return true;};
  bool cardValid(){
   return true;};

  void initialState(){
   Sig1::iter u;
   Sig2::iter v;
   for(;u<V1.card();u++)
    for(v.start();v<V2.card();v++)
      //non-deterministic choice
      R[u][v]=choose()%2;
  void nextState(){
   R = R;};
  int depth(){return 1;};};
```

(a) Arbitrary binary relation component

```
template<class Sig>
class ForAll:public Component{
  Sig & V;
  Sig::iter & v;
  Component & F;
  bool value, valid;

  bool predicate() {
    return value;};
  bool predValid{
    return valid;};

  void initialState(){
    value = true;
    valid = false;
    v = 0;};
  void nextState(){
    if(!valid && v.isValid()){
      if(F.predValid()){
        value &= F.predicate();
        if(v.isLast() ){
          valid == true; }
        if(!value){
          valid = true;
          F.terminate();}
        v++;
        F.evaluate();
      } } };
  int depth(){
    return Sig::scope*F.depth();
  };};
```

(b) Universal quantifier component

```
template<class S1, class S2>
class TClosure:public Relation{
  Relation<S1,S2> & T;
  bitMatrix E;
  bool valid;
  int count;

  bool in(S1::iter u,S2::iter v){
    return E[u][v];};
  int card(){
    return countOnes(E);};
  bool memberValid()
    {return valid;};
  Boolean cardValid()
    {return valid;};
  void initialState(){
    E = T.R;
    count = 0;
    valid = false;};
  void nextState(){
    if((!valid) && T.cardValid() &&
      T.V1.cardValid() &&
      T.V2.cardValid()){
        E = E*E + E;//iterative squaring
      if(count++ ==
        max(lg(T.V1.card()),
         lg(T.V2.card()))
          valid = true; } };
  int depth(){
    int n=max(S1::scope, S2::scope);
    return lg(n)+T.depth();};};
```

(c) Transitive closure component

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. 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`, their validity has to wait for the `Et` component to signal membership validity. Fortunately for this example, this happens in one step since as we will describe Section 4.2.2, transitive closure takes $\lg(n)$ steps to complete where n is the scope. In Step 3, the validity of `Et` is propagated to the relational product `P` corresponding to $v_1.*E$ and `P` is immediately tested for membership of v_2. Since all data is valid, the $\forall v_1$ component updates its predicate state and increments its iterator v_1. The same happens in Step 4, and now $\forall v_1$ completed execution and thus can signal the validity of its predicate. The $\forall v_2$ quantifier updates its predicate state, increments its iterator v_2 and initializes the $\forall v_1$ component to start execution again. Step 5 is similar to Step 3, and Step 6 witnesses the completion of execution of the $\forall v_2$ component. The propositional `and` component has now two valid true inputs so it evaluates to true and thus our predicate can be set to $true$.

4.2 SERA encoding algorithm

SERA recursively traverses the abstract syntax DAG for an Alloy formula Φ with a scope n from its command to its signatures and implicit relations. For each Alloy construct, SERA instantiates its corresponding `Component` object. It composes each component into the desired sequential circuit SERA(Φ, n); the `pred` function of the top level component corresponds to the invariant.

4.2.1 Leaf nodes: signatures and relations

The leaf nodes of the abstract syntax tree of an Alloy formula are signatures and implicit relations and constitute the base case for the SERA(Φ, n) recursive construction. The template class `Sig` in Table 2(b) takes a template parameter as its scope and thus the scope is part of the structure of the class. It uses the return value of a nondeterministic function, `choose`, modulo its scope, to initialize its size. Without loss of generality, the size of the set is enough to represent it since its members are indexed arbitrarily and any set can be re-indexed appropriately. The functions `card` and `in` are valid immediately and `in` returns true if the index is smaller than the size. The template class `Relation` in Table 3(a) implements the binary implicit Alloy relations. It takes two set types, `Sig1` and `Sig2` as template parameters and references the two sets, `V1` and `V2`, it relates. It uses the scope parameters of `Sig1` and `Sig2` to declare the bitMatrix `R` where it stores its membership state. The bit-matrix `R` is initialized nondeterministically modulo the cardinality of

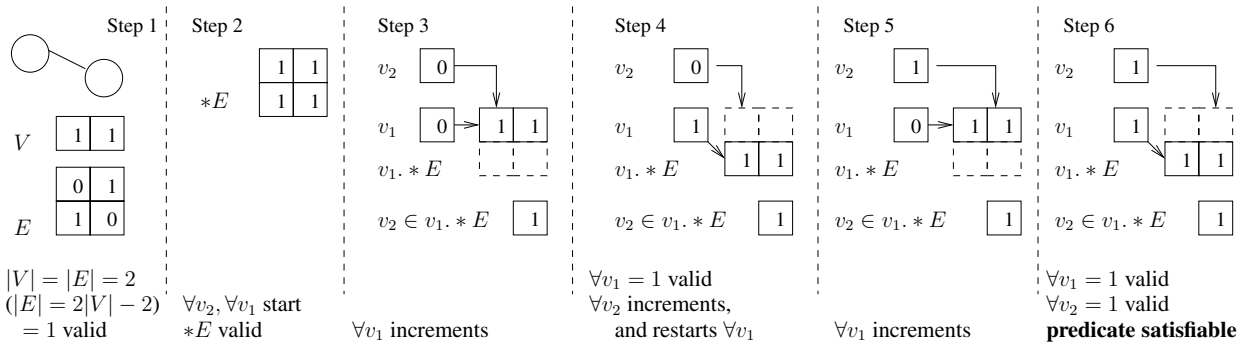

Figure 2. SERA execution of a consistent instance of the `Statement3` predicate with a scope of 2

`V1` and `V2`. The `countOnes` function returns the number of set bits in `R` to compute the cardinality of the relation.

In a trade off between the depth of the circuit and the number of variables, we allow n multiple accesses to the membership functions. This requires an additional $\lg(n)$ bookkeeping variables and keeps the sequential depth of all components linear in n.

4.2.2 Internal nodes

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

Propositional operators. SERA encodes each propositional operator (`and, or, not, implies, iff`) with a combinational circuit component. The component holds references to its operand components and uses their `pred` and `predValid` functions. We show next a logically complete `Nand` component and the rest of the operators can be described in terms of the `Nand` component.

```
class Nand:public Component{
  Component & F1, & F2;
  bool predicate(){
    return !(F1.predicate() && F2.predicate());};
  bool predValid(){
    return (F1.predValid() && !F1.predicate())||
      (F2.predValid() && !F2.predicate())||
      (F1.predValid() && F2.predValid());};
  int depth(){
    return max(F1.depth(), F2.depth());};};
```

Quantifiers. Because the scope is finite, we can easily perform quantifier elimination. The universal quantification of θ by x is replaced by the conjunction of θ restricted to each value x can take; for existential quantification, conjunction is replaced by disjunction. The `ForAll` component shown in Table 3(b) implements a universal quantifier. It takes a set type, `Sig`, a set, `V`, as its quantification domain, a reference to an iterator, `v`, as the quantified variable, and a reference to the formula component, `F`. It computes conjunction (disjunction) sequentially, and employs an *early termination mechanism* where the first false (true) value 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 the `v` iterator to evaluate `F`, and accumulates its Boolean valid and value members.

Relational product. The relational product of two components A and B is implemented by a component that fills in the index of the right operand B by the concatenation of tuples from the left operand A and the actual index argument to the relational product component. All the matches are saved in a bitMatrix, and then produced sequentially, one after the other, while updating the cardinality count and the validity bits. Since we allow n parallel membership checks, we can guarantee the validity of the membership and cardinality data in at most n steps.

Transpose. We define a unique variable order to respect the order in which signatures were declared in Φ 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 suboptimality introduced in the variable ordering, SERA resorts to adding a redundant variable appropriately. SERA also adds a constraint that indicates the equivalence of the redundant data so that logic optimization techniques can easily exploit the hint.

Transitive closure. Transitive closure in Alloy repeats one or more compositions infinitely many times. SERA's implementation of transitive closure, the `TClosure` tem-

plate class shown in Table 3(c), takes two set types as template parameters and a reference to the component corresponding to the original binary relation. The `nextState` function encodes the transitive closure using *iterative squaring* [10]. This allows us to use only $n \lg(n)$ variables and allows the computation to complete within $\lg(n)$ steps.

Set operations. As shown in Table 2(c) the `in` and `card` functions and their validity can be encoded as a combination of the `in` and `memberValid` functions of the operand components. The `Union` class takes two set types and constructs its own appropriate iterator that maps indices correctly in case the types were different. It also takes references to the components representing its operands. The `card` function inlines membership checks in both its operands and counts the matches in an intermediary bit vector. The rest of the set operators can be easily described in a similar fashion.

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 circuit components in addition to validity propagation of the operand components. Note that all integer valuations, less the cardinality values, are considered valid by default.

Theorem 1. Let Φ be an Alloy formula and let n be a given scope. The sequential circuit $C = \text{SERA}(\Phi, n)$ always terminates and on termination, the invariant evaluates to $true$ iff Φ is satisfiable in scope n.

Sketch of proof. Theorem 1 follows from a straightforward induction on the length of Φ. We use the validity entries Column 1 of Table 1 and the `depth` function in all components to establish a tight upper bound on the depth of C. The bound depends on the length of the formula and the scope and can be computed by calling the function `depth` of the top level component. Note that given a scope, the call `compVec[0].depth()` can be computed at compile time and thus is a constant.

The base cases, where Φ is either a `Sig` or an implicit relation, are trivial and always satisfiable since they are initialized with non-deterministic values. 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 follows similarly.

If $\Phi = \exists v \in \Gamma.F(v)$, and Φ 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 satisfiable. Let C_Φ be the SERA component corresponding to the $\exists v$ statement and let C_Γ and C_F be the SERA components corresponding to Γ and F respectively. By the induction hypothesis, there exists a trace $\mathcal{Z}_\Gamma$ that sets C_Γ to a state matching σ in k_Γ steps, and there exists a trace $\mathcal{Z}_\alpha$ that sets the predicate and validity output functions of C_F to true in k_F steps after presenting C_F with α. The component C_Φ enumerates all the possible elements in range(Γ), queries C_Γ 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 α. Since it accumulates a disjunction of the results of C_F, the first valid $true$ return value of the predicate output function 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 function of C_Φ to true as well as its predicate validity function. Since range(Γ) 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 Φ is not satisfiable, then C_Φ is guaranteed to try all models of range(Γ) and complete execution in a finite number of steps (2^{n^a}). By the induction hypothesis, all elements in models that match C_Γ will not satisfy C_F. At the end of the iteration the output predicate of C_Φ will be set to false and its corresponding validity output will be asserted and the `while` loop will terminate. Consequently there exists no trace that satisfies the predicate output function of C_Φ.

4.3 Optimizations

Without loss of generality we support the same scope value for all signatures. This allows us to keep our type-determination as simple as checking whether an index lies within a range. We also simplify the counters embedded in the iterators in some 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 Boolean shift operation. In most cases some SERA components are guaranteed to complete execution before other components even begin. We use this fact to allow memory sharing between non-overlapping components. Note also that we separate variables based on the functions they are used for. This allows huge *cone of influence* [22] reductions if for example the cardinality of a component is not checked. Furthermore, this introduces redundancy which can be exploited by redundancy removal transforms.

5 Evaluation of SERA

We introduce our SERA implementation and we present the results of SERA in comparison to the Alloy Analyzer.

Table 4. Results of SERA and Alloy Analyzer. Tree example: SERA validated a scope of 32, Alloy Analyzer failed on 7. File system: SERA found counter examples for scope 32, Alloy Analyzer did not pass 8. LISP lists: SERA validated a scope of 32, Alloy Analyzer timed out for a scope of 10.

	Tree integrity				File System				LISP lists			
Alloy Analyzer: results for **scopes 5 to 10**												
Scope	5	**6**	7	8	5	6	7	**8**	10	8	**9**	10
Satisfiable	NO				YES				YES	NO		
Variables	212,032	**641,983**	1,682,479	∞	4,703	7,204	10,377	**13,448**	17,035	2,990	**18,921**	23,704
Clauses	641,644	**2,014,005**	5,428,222		14,448	22,640	33,232	**43,875**	98,381	68,858	**103,440**	129,628
SAT solver time(sec)	156	**658**	∞		5	6	21	**28**	36	55	**2,062**	∞
SERA: sequential circuit for **scopes 4 to 32**												
Scope	4	8	16	**32**	4	8	16	**32**	32	8	16	**32**
Boolean inputs	12	32	80	**192**	22	41	68	**143**	178	46	88	**178**
Boolean state variables	36	94	780	**3,272**	112	728	4,264	**27,720**	22,305	1,020	5,226	**25,790**
Nand Connectives	354	1,087	10,786	**21,841**	1,054	4,548	19,545	**421,681**	285,046	13,859	110,734	**328,065**
SixthSense: resources to automatically solve the sequential circuit												
Solver time (sec)	13	18	220	**4,309**	26	39	417	**1,712**	171	229	341	**575**
Solver MEM (MB)	12	24	91	**134**	10	16	45	**101**	34	21	44	**88**

5.1 Implementation

Our implementation of SERA mirrors the description in Section 4. As illustrated in Figure 1, we parse the Alloy model into a DAG of signatures, relations, and operators with the root as the command to be executed and the leaves as the signatures and implicit relations. Note that we generate a DAG since our analysis tries to reuse syntax-equivalent nodes. The C++ subset we used to describe SERA along with the implied concurrency semantics and the bounds on integers guaranteed by scope finitization can be directly synthesized to sequential circuits described in VHDL similar to [12, 14]. We end up with a hierarchical VHDL design with an asserted signal designated as the invariant. We pass the VHDL to SixthSense which, in case of satisfiability, provides a trace that satisfies the invariant. Step 1 of Figure 2 illustrates the mapping of the trace to an Alloy instance. We map the initial values of the membership state of component `V` as two vertices and the initial values of the `bitMatrix` membership state of component `E` as the existence or absence of arcs with label `E` between the vertices.

5.2 Results

To evaluate SERA we chose three examples when we began this research. The *tree integrity* entries in Table 4 show results for checking 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 six 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 automatically check these formulas for scopes upto 32. The ∞ in the Alloy entries denotes a timeout with a time limit of $14,000$ seconds. 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 $3\times$ larger than the Alloy Analyzer.

The entries in the first set of rows of Table 4 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 (∞), and for a scope of 8 the Alloy Analyzer ran out of memory and could not generate the CNF formula.

The second set of rows in Table 4 shows the size of the sequential circuits produced by SERA. In the third 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 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 re-

sources and time limitations.

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

6 Conclusion and Future Work

We developed the use of sequential circuits for checking the validity of Alloy formulas. By doing so we used far fewer variables and enabled sophisticated automatic reduction techniques to be applied. We were able to show that a scope of 32 is feasible using reasonable resources.

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 [25], S1S [3], and Presburger Arithmetic [29].

Acknowledgments. We would like to thank Darko Marinov for his detailed comments on a previous draft. We also would like to thank Jason Baumgartner, Viresh Paruthi, Geert Janssen, Jessie Xu, Mark Williams, Hari Mony, Robert Kanzelman, and Ali El-Zein for contributions to the verification framework used in the experiments.

References

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

[2] P. Ashenden. *The Designers Guide to VHDL*. Morgan Kaufmann, 2002.

[3] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Synthesis Using `S1S`. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, Oct. 2000.

[4] A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli. Formula Dependent Equivalence for Compositional CTL Model Checking. *Journal of Formal Methods in System Design*, 21(2):193–224, 2002.

[5] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification*, July 2002.

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

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

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

[9] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *International Symposium on Circuits and Systems*, May 1982.

[10] 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.

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

[12] G. De Micheli. Hardware synthesis from C/C++ models. In *Design Automation and Test in Europe*, Mar. 1999.

[13] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Master's thesis, Massachusetts Institute of Technology, 2003.

[14] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design Automation and Test in Europe*, 2005.

[15] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Design Automation and Test in Europe*, 2002.

[16] 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.

[17] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

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

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

[20] D. Jackson and M. Jackson. *Separating Concerns in Requirements Analysis: An Example*. Springer-Verlag. (To appear).

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

[22] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993.

[23] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

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

[25] 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.

[26] 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.

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

[28] A. Saldanha, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-Level Logic Simplification using Don't Cares and Filters. In *Conference on Design Automation*, pages 277–282, 1989.

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

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

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