Symbolic Automata: The Toolkit

Margus Veanes and Nikolaj Bjørner

Microsoft Research, Redmond, WA

Abstract. The symbolic automata toolkit lifts classical automata analysis to work modulo rich alphabet theories. It uses the power of state-of-the-art constraint solvers for automata analysis that is both expressive and efficient, even for automata over large finite alphabets. The toolkit supports analysis of finite symbolic automata and transducers over strings. It also handles transducers with registers. Constraint solving is used when composing and minimizing automata, and a much deeper and powerful integration is also obtained by internalizing automata as theories. The toolkit, freely available from Microsoft Research[1], has recently been used in the context of web security for analysis of potentially malicious data over Unicode characters.

Introduction. The distinguishing feature of the toolkit is the use and operations with symbolic labels. This is unlike classical automata algorithms that mostly work assuming a finite alphabet. Adtantages of a symbolic representation are examined in [4], where it is shown that the symbolic algorithms consistently outperform classical algorithms (often by orders of magnitude) when alphabets are large. Moreover, symbolic automata can also work with infinite alphabets. Typical alphabet theories can be *arithmetic* (over integers, rationals, bit-vectors), *algebraic data-types* (for tuples, lists, trees, finite enumerations), and *arrays*. Tuples are used for handling alphabets that are cross-products of multiple sorts. In the following we describe the core components and functionality of the tool. The main components are $\mathsf{Automaton}\langle T\rangle$, basic automata operations modulo a Boolean algebra T; $\mathsf{SFA}\langle T\rangle$, symbolic finite automata as theories modulo T; and $\mathsf{SFT}\langle T\rangle$, symbolic finite transducers as theories modulo T. We illustrate the tool's API using code samples from the distribution.

$\mathsf{Automaton}\langle T\rangle$. The main building block of the toolkit, that is also defined as a corresponding generic class, is a (symbolic) automaton over T: $\mathsf{Automaton}\langle T\rangle$.

The type T is assumed to be equipped with effective Boolean operations over T: $\underline{\wedge}$, $\underline{\vee}$, $\underline{\neg}$, $\underline{\bot}$, $is_\bot$ that satisfy the standard axioms of Boolean algebras, where $is_\bot(\varphi)$ checks if a term φ is false (thus, to check if φ is true, check $is_\bot(\underline{\neg}\varphi)$). The main operations over $\mathsf{Automaton}\langle T\rangle$ are $\cap$ (intersection), $\cup$ (union) $\complement$ (complementation), $A \equiv \emptyset$ (emptiness check). As an example of a simple symbolic operation consider products: when A, B are of type $\mathsf{Automaton}\langle T\rangle$, then $A \cap B$ has the transitions $\langle (p,q), \varphi\underline{\wedge}\psi, (p',q')\rangle$ for each transition $\langle p, \varphi, p'\rangle \in A$, and

[1] The binary release is available from `http://research.microsoft.com/automata`

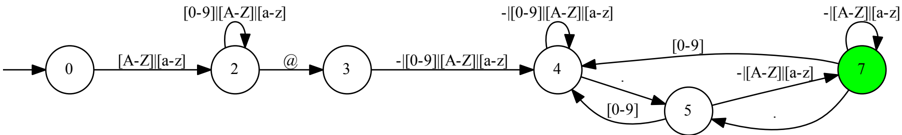
C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 472–477, 2012.
© Springer-Verlag Berlin Heidelberg 2012

$\langle q, \psi, q' \rangle \in B$. Infeasible and unreachable transitions are pruned by using the $is_\perp$ tester. Note that Automaton$\langle T \rangle$ is also a Boolean algebra (using the operations $\cap, \cup, \complement, \equiv \emptyset$). Consequently, the tool supports building and analyzing nested automata Automaton$\langle$Automaton$\langle T \rangle\rangle$.

The tool provides a Boolean algebra solver CharSetSolver that uses specialized BDDs (see [4]) of type CharSet. This solver is used to efficiently analyze .Net regexes with Unicode character encoding. The following code snippet illustrates its use, as well as some other features like visualization.

```
CharSetSolver solver = new CharSetSolver(CharacterEncoding.Unicode);       // charset solver
string a = @"^[A-Za-z0-9]+@(([A-Za-z0-9\-])+\.)+([A-Za-z\-])+$";                 // .Net regex
string b = @"^\d.*$";                                                             // .Net regex
Automaton<CharSet> A = solver.Convert(a);                      // create the equivalent automata
Automaton<CharSet> B = solver.Convert(b);
Automaton<CharSet> C = A.Minus(B, solver);                           // construct the difference
var M = C.Determinize(solver).Minimize(solver);    // determinize then minimize the automaton
solver.ShowGraph(M, "M.dgml");                                             // save and visualize
string s = solver.GenerateMember(M); //generate some member, e.g. "HV7@9.2.8.-d2bVu0YH.z1f.R"
```

The resulting graph from line 8 is shown below.



SFA$\langle T \rangle$. A symbolic finite automaton SFA$\langle T \rangle$ is an extension of Automaton$\langle T \rangle$ with a logical evaluation context of an SMT (Satisfiability Modulo Theories) solver that supports operations that go beyond mere Boolean algebraic operations. The main additional solver operations are: *assert* (to assert a logical formula), *push/pop* (to manage scopes of assertions), *get_model*: $T \to \mathcal{M}$ to obtain a model for a satisfiable formula. A model $\mathcal{M}$ is a dictionary from the free constants in the asserted formulas to values. The method *assert_theory* takes an SFA$\langle T \rangle$ A and adds the *theory of* A to the solver. It relies on a built-in theory of *lists* and uses it to define a *symbolic language acceptor* for A that is a unary relation symbol acc_A such that $acc_A(s)$ holds iff s is accepted by A.

The following code snippet illustrates the use of SFAs together with Z3 as the constraint solver. The class Z3Provider is a conservative extension of Z3 that extends its functionality for use in the automata toolkit. The sample is similar (in functionality) to the one above, but uses the Z3 Term type rather than CharSet for representing predicates over characters.

```
Z3Provider Z = new Z3Provider();
string a = @"^[A-Za-z0-9]+@(([A-Za-z0-9\-])+\.)+([A-Za-z\-])+$";                 // .Net regex
string b = @"^\d.*$";                                                             // .Net regex
var A = new SFAz3(Z, Z.CharacterSort, Z.RegexConverter.Convert(a));             //  SFA for a
var B = new SFAz3(Z, Z.CharacterSort, Z.RegexConverter.Convert(b));             //  SFA for b
A.AssertTheory(); B.AssertTheory();                            // assert both SFA theories to Z3
Term inputConst = Z.MkFreshConst("input", A.InputListSort);    // declare List<char> constant
var assertion   = Z.MkAnd(A.MkAccept(inputConst),              // get solution for inputConst
                      Z.MkNot(B.MkAccept(inputConst)));      //  accepted by A but not by B
var model       = Z.GetModel(assertion, inputConst);           //   retrieve satisfying model
string input    = model[inputConst].StringValue;               //    the witness in L(A)-L(B)
```

SFA acceptors can be combined with arbitrary other constraints. This feature is used in Pex[2] for path analysis of string manipulating .Net programs that use regex matching in branch conditions.

SFT⟨*T*⟩. A symbolic finite transducer over labels in T (SFT) is a finite state symbolic input/output* automaton. A transition of an $\mathsf{SFT}\langle T\rangle$ has the form (p, φ, out^*, q) where φ is an input character predicate and $out*$ is a sequence of output terms that may depend on the input character. For example, a transition $(p, x > 10, [x+1, x+2], q)$ means that, in state p, if the input symbol is greater than 10, then output $x+1$ followed by $x+2$ and continue from state q.

An SFT is a generalization of a classical finite state transducer to operate modulo a given label theory. The core operations overs SFTs are the following: *union* $T \cup T$, (relational) *composition* $T \circ T$, *domain restriction* $T \restriction A$, *subsumption* $T \preceq T$ and *equivalence* $T \equiv T$. These operation form (under some conditions, such as *single-valuedness* of SFTs) a decidable algebra over SFTs. The theory and the algorithms of SFTs are studied in [7].

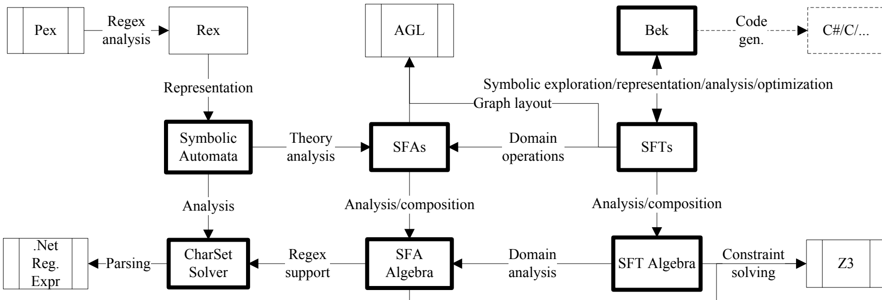
Bek is a domain specific language for string-manipulating functions. It is to SFTs as regular expressions are to SFAs. The toolkit includes a parser for Bek as well as customized visualization support using the graph viewer of Visual Studio. Key scenarios and applications of Bek for security analysis of sanitation routines are discussed in [3]. The following snippet illustrates using the library for checking idempotence of a decoder P (it decodes any consecutive digits d_1 and d_2 between '5' and '9' to their ascii letter $dec(d_1, d_2)$, e.g. $dec($`'7'`, `'7'`$)$ = `'M'`, thus $P($`"7777"`$)$ = `"MM"`. The Bek program decoder is first converted to an equivalent SFT (where the variable b is eliminated).

```
string bek = @"program P(input) {                                        // The Bek program P
   return iter(c in input) [b := 0;] {                           // P decodes certain digit pairs
      case (b == 0): if ((c>='5')&&(c<='9')) { b:=c; } else { yield(c); }
      case (true): if ((c>='5')&&(c<='9')) { yield(dec(b,c));b:=0; } else { yield(c); }
   } end { case (b != 0): yield (b);};}";
Z3Provider Z = new Z3Provider();                                  // analysis uses the Z3 provider
var f = BekConverter.BekToSTb(Z, bek).ToST().Explore();           //         convert P to an SFT f
var fof = f + f;                                                  //          self-compostion of f
if (!f.Eq1(fof)) {                                                //        check idempotence of f
    var w = f.Diff(fof);                                  // find a witness where f and fof differ
    string input = w.Input.StringValue;                   //                            e.g. "5555"
    string output1 = w.Output1.StringValue;               //                 e.g. f("5555") == "77"
    string output2 = w.Output2.StringValue; }             //               e.g. f(f("5555")) == "M"
```

Users and tool availability. This is the first public release of the toolkit. It has so far been used at Microsoft, and part of the tool (Rex) is also an integrated part of the parameterized unit testing tool Pex. Applications that illustrate some key usage scenarios, are also used from the web services `http://www.rise4fun.com/rex` and `http://www.rise4fun.com/bek`. The tool has been used in numerous experiments, some of which are described in [4,3], that show scalability and applicability to concrete real-life scenarios.

[2] `http://research.microsoft.com/pex/`

Tool Overview. An overview of the toolkit is illustrated in the diagram below. The core components are in bold. Arrows indicate dependencies between the components. They are labeled by the main relevant functionality.



Related Tools. String analysis has recently received increased attention, with several automata-based analysis tools. We make a systematic comparison of related techniques in [4]. Tools include the Java String Analyzer [1], with the `dk.brics.automaton` library as a constraint solver for finite alphabets. It compresses contiguous character ranges. Hampi [5] solves bounded length string constraints over finite alphabets using a reduction to bit-vectors. Kaluza extends Hampi to systems of constraints with multiple variables and concatenation [6]. MONA [2] uses MTBDDs for encoding transitions. BDDs are used in the PHP string analysis tool in [8].

References

1. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
2. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic Second-Order Logic in Practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
3. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with bek. In: USENIX Security Symposium (August 2011)
4. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011)
5. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: ISSTA (2009)
6. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A Symbolic Execution Framework for JavaScript (March 2010)
7. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: Algorithms and applications. In: POPL 2012 (January 2012)
8. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An Automata-Based String Analysis Tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010)

A Bek

Bek is a domain specific language for writing common string functions. With Bek, you can answer questions like: Do these two programs output the same strings? Given a target string, is there an input string such that the program produces the target string? Does the composition of two programs produce a desired result? Does the order of composition matter? Bek has been specifically tailored to capture common idioms in string manipulating functions.

A.1 UTF8Encode Example

Bek includes a fairly complete set of arithmetic operations that are, by default, over 16-bit bit-vectors, since the most common case of analysis is over strings that use UTF-16 encoding. The example shows a concrete representation of a UTF8 encoding routine written in Bek. It takes a UTF-16 encoded string and transforms it into the corresponding UTF8 encoded string. The encoder "raises an exception" when invalid surrogate pairs are detected. These exception cases define, in terms of the generated SFTs, partial behavior, i.e., that the input is not accepted by the SFT.

```
// UTF8 encoding from UTF16 strings, hs is the lower two bits of the previous high surrogate
                   // this encoder raises an exception when an invalid surrogate is detected
program UTF8Encode(input){
return iter(c in input)[HS:=false; hs:=0;]
{
case (HS):                                    // the previous character was a high surrogate
  if (!IsLowSurrogate(c)) { raise InvalidSurrogatePairException; }
  else {
    yield ((0x80|(hs << 4))|((c>>6)&0xF), 0x80|(c&0x3F));
    HS:=false; hs:=0;
  }
case (!HS):                               // the previous character was not a high surrogate
  if (c <= 0x7F) { yield(c); }                                       // one byte: ASCII case
  else if (c <= 0x7FF) {                                                        // two bytes
    yield(0xC0 | ((c>>6) & 0x1F), 0x80 | (c & 0x3F)); }
  else if (!IsHighSurrogate(c)) {
    if (IsLowSurrogate(c)) { raise InvalidSurrogatePairException; }
    else { //three bytes
      yield(0xE0| ((c>>12) & 0xF), 0x80 | ((c>>6) & 0x3F), 0x80 | (c&0x3F));} }
  else {
    yield (0xF0|(((1+((c>>6)&0xF))>>2)&7), (0x80|(((1+((c>>6)&0xF))&3)<<4))|((c>>2) & 0xF));
    HS:=true; hs:=c&3; }
} end {
case (HS): raise InvalidSurrogatePairException;
case (true): yield();
};
}
```

The following code is a unit test from the automata toolkit. Assume that the above Bek program is in the file "UTF8Encode.bek". The code does the following. First, it converts the Bek program into a symbolic transducer *stb* (that allows branching conditions in rules). It then eliminates the registers *hs* and *HS* by fully exploring *stb*. Then the domain of the resulting sft is restricted with the regular expression that excludes the empty input string. The theory of the resulting sft is asserted as a background theory extension of the solver. New uninterpreted

constants are defined for input and output lists of the sft. Then the Z3 provider is used to generate (50) solutions. Old solutions are pruned from iterated calls to the solver.

```
public void TestUTF8Encode() {
    Z3Provider solver = new Z3Provider();
    var stb = BekConverter.BekFileToSTb(solver, "UTF8Encode.bek");
    var sft = stb.Explore();
    //sft.ShowGraph(); //saves the sft in DGML format and opens it in Visual Studio.

    var restr = sft.ToST().RestrictDomain(".+");
    restr.AssertTheory();

    Term inputConst = solver.MkFreshConst("input", restr.InputListSort);
    Term outputConst = solver.MkFreshConst("output", restr.OutputListSort);

    solver.AssertCnstr(restr.MkAccept(inputConst, outputConst));

    //validate correctness for some values against the actual UTF8Encode
    int K = 50;
    for (int i = 0; i < K; i++) {
      var model = solver.GetModel(solver.True, inputConst, outputConst);
      string input = model[inputConst].StringValue;
      string output = model[outputConst].StringValue;

      Assert.IsFalse(string.IsNullOrEmpty(input));
      Assert.IsFalse(string.IsNullOrEmpty(output));

      byte[] encoding = Encoding.UTF8.GetBytes(input);
      char[] chars = Array.ConvertAll(encoding, b => (char)b);
      string output_expected = new String(chars);

      Assert.AreEqual<string>(output_expected, output);

      // exclude this solution, before picking the next one
      solver.AssertCnstr(solver.MkNeq(inputConst, model[inputConst].Value));
    }
}
```

The whole unit test takes a few seconds to complete. In this case the unit test simply tests on 50 random samples that the encoder does not differ from the built-in implementation.

A.2 Bek on Rise4Fun.com

The web-site `http://rise4fun.com/bek` illustrates several examples of Bek programs. It runs the Symbolic Automata Toolkit with the Bek extensions and converts Bek programs into ECMA script (Java script) and also shows a graphical representation of an graph representation of the transducer.

B Rex on Rise4Fun.com

The web-site `http://rise4fun.com/rex` illustrates several examples of Rex as a game. The game is to guess a secret regular expression. The user enters a candidate expression, and the Symbolic Automata Toolkit is used to find strings that are (1) accepted by both languages (if any), (2) accepted by one and rejected by the other (if any), and (3) rejected by both languages.