OpenNWA: A Nested-Word Automaton Library*

Evan Driscoll[1], Aditya Thakur[1], and Thomas Reps[1,2]

[1] Computer Sciences Department, University of Wisconsin – Madison
{driscoll,adi,reps}@cs.wisc.edu
[2] GrammaTech, Inc., Ithaca, NY

Abstract. Nested-word automata (NWAs) are a language formalism that helps bridge the gap between finite-state automata and pushdown automata. NWAs can express some context-free properties, such as parenthesis matching, yet retain all the desirable closure characteristics of finite-state automata.

This paper describes OpenNWA, a C++ library for working with NWAs. The library provides the expected automata-theoretic operations, such as intersection, determinization, and complementation. It is packaged with WALi—the *W*eighted *A*utomaton *Li*brary—and interoperates closely with the weighted pushdown system portions of WALi.

1 Introduction

Many problems in computer science are solved by modeling a component as an automaton. Traditionally, this means either confronting several undecidable problems that arise with the use of pushdown automata or giving up expressivity, and usually precision, by using finite-state automata. Recently, the development of nested word automata (NWAs) and related formalisms [2,3] has revealed a fertile middle ground between these two extremes. NWAs are powerful enough to express some "context-free"-style properties, such as parenthesis matching, and yet retain the decidability properties that make it convenient to work with regular languages. In particular, NWAs and their languages are closed under operations such as intersection and complementation. NWAs have been applied in areas such as modeling programs and XML documents. When modeling programs, NWAs can eliminate spurious data flows along paths with mismatched calls and returns. In XML documents, NWAs can model the matching between opening and closing tags.

We have created a C++ implementation of NWAs, called OpenNWA. OpenNWA is packaged with the *W*eighted *A*utomata *Li*brary, WALi [7]. WALi also provides implementations for weighted finite-state automata and weighted pushdown systems (WPDSs). The OpenNWA library

* Supported by NSF under grants CCF-{0540955, 0810053, 0904371}; by ONR under grants N00014-{09-1-0510, 10-M-0251}; by ARL under grant W911NF-09-1-0413; and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 665–671, 2012.

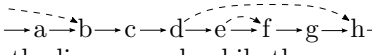
- implements (in the terminology of [3]) linearly-accepting, weakly-hierarchical NWAs, along with standard automata-theoretic operations. (See §3.1.)
- is extensible via a mechanism that allows the user to attach arbitrary client information to each node in the automaton. (See §3.2.)
- inter-operates with WALi's WPDS library and allows the user to issue queries about an NWA's configuration space. (See §3.3.)
- provides utilities for operating on a textual NWA format [5, §5].
- provides extensive documentation [5] and a test suite.

OpenNWA is currently used by three projects [11,4,6]; see §4. OpenNWA is available at `http://research.cs.wisc.edu/wpis/OpenNWA`.

2 NWAs

This section describes nested-word automata [2] and related terms at an intuitive level, and gives an example of how they are used in program analysis. For the formal definitions that we use, see our technical report [5, App. A].

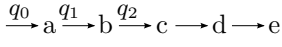
A nested word is an ordinary (linear) string of symbols over some alphabet Σ paired with a *nesting relation*. The nesting relation describes a hierarchical relation between input positions, for instance between matched parentheses. Graphically, a nested word can be depicted as illustrated to the right. In this image, following just the horizontal arrows illustrates the linear word, while the curved edges ("*nesting edges*") indicate positions that are related by the nesting relation. For a nesting relation to be valid, nesting edges must only point forward in the word and may not share a position or cross.



Positions in the word that appear at the left end of a nesting edge are called *call positions*, those that appear at the right end are called *return positions*, and the remaining are *internal positions*. It is possible to have *pending* calls and returns, which are not matched within the word itself. For a given return, the source of the incoming nesting edge is called that return's *call predecessor*.

Nested-word automata (NWAs) are a generalization of ordinary finite-state automata (FA). An NWA's transitions are split into three sets—call transitions, internal transitions, and return transitions. Call and internal transitions are defined the same as transitions in ordinary FAs, while return transitions also have a call-predecessor state as an additional element.

To understand how an NWA works, consider first the case of an ordinary FA M. We can think of M's operation as labeling each edge in the input word with the state that M is in after reading the symbol at that edge's source. For instance, shown to the right is a partial run. To find the next state, M looks for a transition out of q_2 with the symbol c—say with a target of q_3—and labels the next edge with q_3.



The operation of an NWA proceeds in a fashion similar to a standard FA, except that the machine also labels the nesting edges. When the NWA reads an internal position, it chooses a transition and labels the next linear edge the same way an FA would. When the NWA reads a call position, it picks a matching call

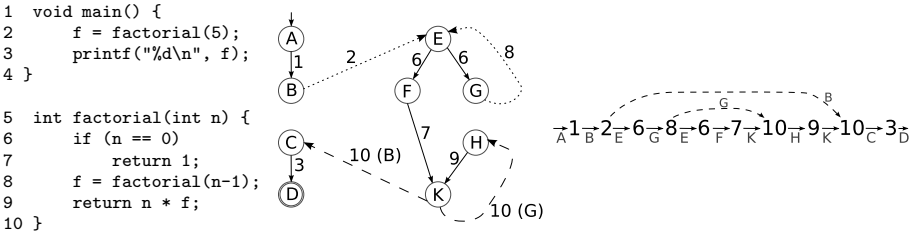


Fig. 1. An example program, corresponding NWA, and accepted word. State labels are arbitrary; transition symbols give the line number of the corresponding statement. Some nodes are elided. Dotted lines indicate call transitions, and dashed lines are return transitions. The state in parentheses on a return transition is the call predecessor.

transition and labels the next linear edge in the same way, but also labels the outgoing *nesting* edge with the state that the NWA is leaving. When the NWA reads a return position, it looks not only at the preceeding linear state but also at the state on the incoming nesting edge. It then chooses a return transition that matches both, and labels the next linear edge with the target state. An example NWA run is shown in Fig. 1.

OpenNWA supports ε internal transitions, which operate in an analogous way to ε rules in ordinary FAs. It also supports something we call *wild* transitions, which match any single symbol. Wilds can appear on any transition type.

Example 1. NWAs can be used to encode the interprocedural control-flow graph (ICFG) of a program. Intraprocedural ICFG edges become internal transitions, interprocedural call edges become call transitions, and interprocedural return edges become return transitions. For an ICFG return edge (*exit-site*, *return-site*), we use the call site that corresponds to *return-site* in the call-predecessor position of the NWA's transition. The symbols on a transition depend on the application, but frequently are the corresponding statement.

An example program, the corresponding NWA, and an example word accepted by that NWA are shown in Fig. 1. Using an NWA allows us to exclude paths such as *1–2–5–6–7–8–9–...* that do not follow a matched path of calls and returns. Fewer paths can allow a client analysis to obtain increased precision.

3 The OpenNWA library

OpenNWA provides a C++ class called `NWA` for representing NWAs. For information about constructing NWAs and actual API information, see the OpenNWA documentation [5]. In this section, we briefly describe some things a user can do with an NWA (or NWAs) once it is built.

3.1 Automata-Theoretic Operations

As mentioned in §1, OpenNWA supports most automata-theoretic operations:

- intersection
- union
- Kleene star
- reversal
- concatenation
- determination
- complement
- emptiness checking
- example word generation

For the most part, we use Alur and Madhusudan's algorithms [2,3]; however we note three exceptions. First, we implemented emptiness checking and example generation using WPDSs, discussed below. Second, we found and corrected a minor error in Kleene star [5, §6.4]. Third, we expressed Alur and Madhusudan's determinization algorithm using relational operators [5, App. B].

OpenNWA supports determining whether an NWA's language is empty, and if it is not, OpenNWA can return an arbitrary word from the NWA's language. It is also possible to ask specifically for a shortest accepted word. The library performs these operations by converting the NWA into a WPDS and running a post* query from the set of initial configurations (see §3.3). To find an example word, OpenNWA uses a witness-tracing version of post* [10, §3.2.1], and extracts the word from the resulting witness. For the shortest word, we simply use weights that track the length of the path from the initial state.

The ability to obtain an example word is important in program analysis. It can show the end user of an analysis tool a program trace that may violate a property. Moreover, this feature is fundamental to counterexample-guided refinement: in CEGAR-based model checkers, the counterexample is typically an example word from the automaton that models the program.

3.2 Client Information

OpenNWA provides a facility that we call *client information*. This feature allows the user of the library to attach arbitrary information to each state in the NWA. For instance, as discussed in §4, McVeto uses NWAs internally, and uses client information to attach a formula to each state in the NWA.

The library tracks this information as best as it can through each of the operations discussed in the previous section, and supports callback functions to compute new client information when it does not have the information it needs.

3.3 Inter-operability with WPDSs

Weighted pushdown systems (WPDSs) can be used to perform interprocedural static analysis of programs [10]. The PDS proper provides a model of the program's control flow, while the weights on PDS rules express the dataflow transformers. Algorithms exist to query the configuration space of WPDSs, which corresponds to asking a question about the data values that can arise at a set of configurations in the program's state space. A configuration consists of a control location and a list of items on the stack.

OpenNWA supports converting an NWA into a WALi WPDS. This feature allows an OpenNWA client to issue queries about the configuration space of an NWA. The WPDS's stack corresponds to the states that label the as-yet-unmatched nesting edges in a prefix of an input nested word. When viewed in program-analysis terms, the WPDS that results from this conversion reflects the same control structure as the original NWA. Answers to WPDS queries tend to have a natural interpretation in the NWA world, as well. For NWA operations that can be expressed naturally in this way (e.g., emptiness checking and post*), OpenNWA employs this conversion. Other NWA operations, such as determinization, do not have an equivalent WPDS operation.

NWAs themselves are not weighted, but OpenNWA provides a facility for determining the weights of the WPDS rules during conversion. The user provides an instance of class `WeightGen`, which acts as a factory function. The function is called with the states in question and returns the weight of the resulting WPDS rule. (The weight can depend on the client information of the states.)

4 Uses of OpenNWA

I/O Compatibility Checking. We used OpenNWA as the primary component of a tool called the Producer-Consumer Conformance Analyzer (PCCA) [4]. Given two programs that operate in a producer/consumer relationship over a stream, PCCA's goal is to determine whether the consumer is prepared to accept all messages that the producer can emit, and find a counterexample if not. PCCA infers a model of the output language of the producer, infers a model of the input language of the consumer, and determines whether the models are compatible.

PCCA uses NWAs for its models, building them from the ICFG, as discussed in Ex. 1. Edges corresponding to statements that perform I/O are labeled with the type of the I/O, and all other internal transitions are labeled with ε. Conceptually what we want to check is whether the producer's output language is a subset of the consumer's input language, which is an operation NWAs and our library support. In practice, this check is likely to be too strict, and we need an additional step, detailed in [4, §2.3 and §3.2].

Machine-Code Model Checking. McVeto is a machine-code verification engine [11] that, given a binary and description of a bad target state, tries to find either (i) an input that forces the bad state to be reached, or (ii) a proof that the bad state is impossible to reach.

McVeto uses a model of the program called a *proof graph*, which is an NWA that overapproximates the program's behavior. States in a proof graph are labeled with formulas; edges are labeled with program statements. McVeto starts with a very coarse overapproximation, which it then refines. One refinement technique uses symbolic execution to produce a concrete trace of the program's behavior, performs *trace generalization* [11, §3.1] to convert the trace into an overapproximating NWA (the "folded trace"), and intersects the current proof graph with the folded trace to obtain the new proof graph. The formula on a state in the new proof graph is the conjunction of the formulas on the states that are being paired from the current proof graph and the folded trace.

McVeto's implementation uses OpenNWA's client-information feature to store the formula for each state. During intersection, the callback functions mentioned in §3.2 compute the conjunction of the input formulas, which are then used in the new proof graph.

To determine whether the target state is not reachable in the proof graph (and thus is definitely not reachable in the actual program), McVeto calls `prestar()` (see §3.3). The result of this call is also used to determine which "frontier" to extend next during directed test generation [11, §3.3].

JavaScript Security-Policy Checking and Weaving. The JAM tool [6] checks a JavaScript program against a security policy, either verifying that the program is already correct with respect to that policy or inserting dynamic checks into the program to ensure that it will behave correctly. JAM builds *two* models of the input program, one that overapproximates the control flow of the program and one that overapproximates the data flow. The policy is also expressed as an NWA of forbidden traces. By intersecting the policy automaton with both program models, JAM obtains an NWA that expresses traces that possibly violate the policy.

Once JAM has the combined NWA, it asks OpenNWA for a shortest word in the language. If the language is empty (i.e., there is no shortest word), the program always respects the policy. If OpenNWA returns an example word w, JAM checks whether w corresponds to a valid trace through the program. If w is valid, then JAM inserts a dynamic check to halt concrete executions corresponding to w that would violate the policy. If w is not valid, than JAM can either refine the abstraction and repeat, or insert a dynamic check to detect and halt concrete executions corresponding to w for which the policy would be violated.

5 Related work

Alur and Madhusudan each maintain a page giving a significant bibliography of papers that present theoretical results, practical applications, and tools related to NWAs and visibly pushdown automata (VPAs) [1,8]. VPAs and their languages are another formalism which can be seen as an alternative encoding of NWAs and nested-word languages [3].

VPAlib [9] is a general-purpose library implementing VPAs. However, OpenNWA's implementation is far more complete. For instance, VPAlib does not support concatenation, complementation (although it does support determinization), checking emptiness, or obtaining an example word.

References

1. Alur, R.: Nested words (2011), `http://www.cis.upenn.edu/~alur/nw.html`
2. Alur, R., Madhusudan, P.: Adding Nesting Structure to Words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006)
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3) (May 2009)
4. Driscoll, E., Burton, A., Reps, T.: Checking conformance of a producer and a consumer. In: FSE (2011)
5. Driscoll, E., Thakur, A., Burton, A., Reps, T.: WALi: Nested-word automata. TR-1675R, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (September 2011)
6. Fredrikson, M., Joiner, R., Jha, S., Reps, T., Porras, P., Saïdi, H., Yegneswaran, V.: Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 548–563. Springer, Heidelberg (2012)
7. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007), `http://www.cs.wisc.edu/wpis/wpds/download.php`

8. Madhusudan, P.: Visibly pushdown automata – automata on nested words (2009), http://www.cs.uiuc.edu/~madhu/vpa/
9. Nguyen, H.: Visibly pushdown automata library (2006), http://www.emn.fr/z-info/hnguyen/vpa/
10. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP 58(1-2) (October 2005)
11. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. TR 1669, UW-Madison (April 2010); Abridged version published in CAV 2010