OpenNWA: Nested-Word Automata

Evan Driscoll$^1$  Aditya Thakur$^1$  Amanda Burton$^1$  Thomas Reps$^{1,2}$

$^1$University of Wisconsin — Madison
Computer Sciences Department
{driscoll,adi,burtona,reps}@cs.wisc.edu

$^2$GrammaTech, Inc.

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 document describes OpenNWA, a C++ implementation of NWAs. OpenNWA provides the expected automata-theoretic operations, such as intersection, determinization, and complementation, as well as query operations. It is packaged with WALi—Weighted Automata Library—and interoperates closely with the weighted pushdown system portions of WALi.

The OpenNWA library is available from http://www.cs.wisc.edu/wpis/OpenNWA/. It is packaged with the Weighted Automata Library, WALi. WALi 4.0 is paired with the first public release of OpenNWA, which is what this document describes. Building instructions can be found in README.txt in the root directory of the OpenNWA/WALi distribution.

We assume the reader is familiar with nested words and nested-word automata; if not, App. A provides the definition we use and Alur et al. [2, 3] describe them in detail. Our implementation corresponds to the definition from the DLT 2006 paper [2]; relative to follow-up work [3], this definition removes the distinction between the machine’s linear and hierarchical states, and on a call transition always stacks the source state. In addition, we only require that the final linear state be accepting for an NWA to accept a word. (Following [3], this specific variant is a “weakly-hierarchical, linearly-accepting” NWA.)

The terminology used in the interface of the package is geared toward program analysis, so we use terms such as “call site” and “return site” to refer to states of the NWA even though they take on different meanings in different contexts.

In contrast to the standard definitions, we allow internal $\varepsilon$ transitions (but not call or return $\varepsilon$ transitions). We also allow “wild” transitions, which match any single symbol. (Wilds can appear on calls and returns.)
Contents

1 Library Overview 3
  1.1 NWA core classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.2 NWA non-member functions . . . . . . . . . . . . . . . . . . . . . . . . . 4
  1.3 Supporting features from WAL . . . . . . . . . . . . . . . . . . . . . . . 4
  1.4 Simple example use of the library . . . . . . . . . . . . . . . . . . . . . . 5

2 The NestedWord class 8

3 The Nwa class 9
  3.1 Construction, copying, assignment, and clearing . . . . . . . . . . . . . . 9
  3.2 Simple manipulations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  3.3 Client Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

4 The opennwa::query namespace 11
  4.1 Querying information about an automaton’s transitions . . . . . . . . . . 11
  4.2 Querying other structural aspects of an automaton . . . . . . . . . . . . . 11
  4.3 Querying properties of an automaton’s language . . . . . . . . . . . . . . 11

5 NWA serialization 13
  5.1 Parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  5.2 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
  5.3 NWA description format . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

6 Building NWAs from other NWAs (namespace opennwa::construct) 18
  6.1 Union . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  6.2 Intersection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  6.3 Concatenation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
  6.4 Kleene star . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  6.5 Reverse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  6.6 Determinize . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  6.7 Complement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

7 Conversions beween WPDSs and NWAs (namespace opennwa::nwa_pds) 32
  7.1 WPDS to NWA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
  7.2 NWA to WPDS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
    7.2.1 Forwards flow stacking calls . . . . . . . . . . . . . . . . . . . . . . 35
    7.2.2 Backwards flow stacking calls . . . . . . . . . . . . . . . . . . . . . 36
    7.2.3 Forwards flow stacking returns . . . . . . . . . . . . . . . . . . . . 36
    7.2.4 Backwards flow stacking returns . . . . . . . . . . . . . . . . . . . 37

A Nested-Word Automata 40

B Determinize 42

C Tables 44
1 Library Overview

The core of the NWA library is in the namespace opennwa. It includes the classes NestedWord, Nwa, and others.

There are a large number of non-member functions that operate on NWAs. These are divided into three sub-namespaces: opennwa::query, opennwa::construct, and opennwa::nwa.pds. There is also an experimental parser in opennwa.

Finally, there are some operations and classes provided by WALi that OpenNWA uses. This mostly consists of the key-handling code, but also includes the WPDS portions of WALi.

Throughout this document, include paths are relative to the Source directory from the WALi/OpenNWA distribution.

Section 1.4 illustrates a simple sample use of the library.

1.1 NWA core classes

These types live in namespace opennwa:

NestedWord

Implements a single nested word. Currently the only operation that can be performed using one is checking whether a NestedWord is a member of an Nwa’s language. Defined in opennwa/NestedWord.hpp. See §2.

Nwa

Implements a nested-word automaton. Defined in opennwa/Nwa.hpp with a forward declaration in opennwa/NwaFwd.hpp. See §3. (§5 also discusses member functions related to serializing an NWA.)

ref_ptr<T>

This is a reference-counted, intrusive smart-pointer template. (“Intrusive” means that the pointed-to type must be modified to include a count field used by the pointer. This is most conveniently done by subclassing wali::Countable.) Defined in wali/ref_ptr.hpp. However, a using wali::ref_ptr declaration brings this type into the opennwa namespace as well; we suggest preferring the latter version. In addition, there are typedefs of both ref_ptr<Nwa> and ref_ptr<NestedWord>, and we suggest preferring those names.

NwaRefPtr

NwaRefPtr is a typedef of ref_ptr<Nwa>. Defined in opennwa/NwaFwd.hpp.

State and Symbol

These are both typedefs of wali::Key (see below). Both are defined in opennwa/NwaFwd.hpp. Note: the use of keys for both states and symbols means that they can be confused from a types perspective, so use caution that this does not happen. In addition, a future version of the library may change State and Symbol to be distinct types.

StateSet and SymbolSet

These are typedefs of std::sets of the corresponding type. (Client code should not rely on this fact; it could change to be an unordered_map or other similar container.) Both are defined in opennwa/NwaFwd.hpp.

ClientInfo

This class holds client-specific information that is associated with a state. Client code can subclass ClientInfo and use functions of the Nwa class to attach instances to states. To use, include opennwa/ClientInfo.hpp. See §3.3.

WeightGen

This abstract class describes how to assign weights to transitions of the NWA, and is used both when converting an NWA to a PDS and when doing prestar/poststar queries on the NWA directly. To use, include opennwa/WeightGen.hpp. See §7.2.
1.2 NWA non-member functions

The library provides a large number of non-member functions for operating on NWAs. These are partitioned into the following namespaces:

`opennwa::query`

This namespace provides functions for querying an automaton. The functions in this namespace are:

- Functions that query transitions of the NWA, returning information about one of the states or the symbol in transitions that meet some criteria. (For example, “give me all states that appear as the target state of an internal transition with this source state.”) See §4.1.
- Determining whether two NWAs have any states in common. See §4.2.
- Determining whether an NWA is deterministic. See §4.2.
- Determining whether a `NestedWord` is a member of the language of an NWA. See §4.3.
- Determining whether the language of an NWA is empty. See §4.3.
- Determining whether the languages of two NWAs are equal, or if one is a subset of the other. See §4.3.
- Prestar and poststar operations on an NWA. See §4.3.

`opennwa::construct`

This namespace provides functions for constructing an NWA. Functions in this namespace are standard, automata-theoretic operations such as intersection, union, concatenation, Kleene star, and reversal. See §6.

`opennwa::nwa_pds`

This namespace provides functions for converting between NWAs and WPDSs, and related functions. (Note that construction of an NWA from a WPDS is in this namespace, not in `construct`.) See §7.

`opennwa`

This namespace, in addition to the items already discussed, provides an experimental parser for a serialization format of NWAs. See §5.

1.3 Supporting features from WALi

The NWA library uses these portions of the standard WALi library. Unless otherwise specified, they are in namespace `wali`. See §5.

**Key**

A `Key` is a unique identifier naming states and symbols in an NWA or NestedWord. It is actually a typedef of an integer, though client code should not rely on this precise type. Declared in `wali/Key.hpp`.

`getKey(...)`

This function produces a key from some input. If the input has not been seen before, it returns a new key; if it has, then it returns the same key as before. (One can think of this as translating whatever unique identifier is known by the client code to the Key needed by OpenNWA and WALi.) Overloads of this function are provided for the following types:

- `std::string const &`
- `char *`
- `int`
- `std::set<Key> const &`
- `Key, Key` (this is a two-argument version of `getKey`)
- `key_src_t`

The final overload, for `key_src_t`, is a `ref_ptr` to a `KeySource` object. `KeySource` is an abstract class that client code can subclass to provide a translation to `Keys` for arbitrary types.
All overloads are declared in `wali/Key.hpp`.

```cpp
std::string key2str(Key k)
```

`key2str` is an “inverse” of `getKey`, except that it always returns a string representation. If the key was created with `getKey(std::string const &)` in the first place, this returns a copy of the original string. 

```cpp
wali::wfa::WFA
```

This is a weighted finite automaton. It is used in NWA poststar and prestar queries in much the same manner as in WPDS poststar and prestar queries. Defined in `wali/wfa/WFA.hpp`.

```cpp
wali::wpds::WPDS
```

This is a weighted pushdown system. NWAs use WPDSs behind the scenes when doing poststar and prestar queries, and the library provides conversion routines between the two automata types. Defined in `wali/wpds/WPDS.hpp`.

### 1.4 Simple example use of the library

The following is an illustration of basic operations of the library. The full code is available in the file `Doc/NWA.tex/Example/example.cpp` in the OpenNWA distribution.

```cpp
// This program will walk through creating the NWA shown in Figure 4 of the // associated documentation, and reverse it (to get the result of Figure 12). // We then make a NestedWord of the one word in Figure 4’s language and // another NestedWord of the one word in Figure 12’s language, then test that // each is a member of just the appropriate NWA.

#include <iostream>
using std::cout;

#include "wali/nwa/NWA.hpp"
#include "wali/nwa/NestedWord.hpp"
#include "wali/nwa/construct/reverse.hpp"
#include "wali/nwa/query/language.hpp"
using wali::getKey;
using namespace wali::nwa;
using wali::nwa::construct::reverse;
using wali::nwa::query::languageContains;

// These symbols are used in the NWA and both words
Symbol const sym_a = getKey("a");
Symbol const sym_b = getKey("b");
Symbol const sym_call = getKey("call");
Symbol const sym_ret = getKey("ret");

/// Creates the NWA shown in Figure 4 of the Wali NWA documentation
NWA create_figure_4()
{
    NWA out;

    // Translate the names of the states then symbols to Wali identifiers
    State start = getKey("Start");
```

---

1For `ints`, it is the textual representation of the original number. For a `set<Key>`, it is the set printed in standard set notation. For a pair of keys, it is the pair printed as an ordered pair. In general, it is the result of calling `key_src_t::print`. See the `print` function in each of the files `wali/*Source.cpp`.

---
State call = getKey("Call");
State entry = getKey("Entry");
State state = getKey("State");
State exit = getKey("Exit");
State return_ = getKey("Return");
State finish = getKey("Finish");

// Add the transitions
out.addInternalTrans(start, sym_a, call);
out.addCallTrans(call, sym_call, entry);
out.addInternalTrans(entry, sym_b, state);
out.addInternalTrans(state, sym_b, exit);
out.addReturnTrans(exit, call, sym_ret, return_);
out.addInternalTrans(return_, sym_a, finish);

// Set the initial and final states
out.addInitialState(start);
out.addFinalState(finish);

return out;

/// Creates a (the one) word in the language of Figure 4's NWA
NestedWord create_forwards_word()
{
    NestedWord out;
    out.appendInternal(sym_a);
    out.appendCall(sym_call);
    out.appendInternal(sym_b);
    out.appendInternal(sym_b);
    out.appendReturn(sym_ret);
    out.appendInternal(sym_a);
    return out;
}

/// Creates a (the one) word in the language of the reverse of Figure 4’s NWA
NestedWord create_backwards_word() { ... }

int main()
{
    // Create the NWA and reversed NWA
    NWA fig4 = create_figure_4();
    NWARefPtr fig4_reversed = reverse(fig4);

    // These are the words we are testing
    NestedWord forwards_word = create_forwards_word();
    NestedWord backwards_word = create_backwards_word();

    // Now do the tests
    cout << "fig4 contains:

```cpp
<< "  forwards_word [expect 1] : "
<< languageContains(fig4, forwards_word) << "\n"
<< "  backwards_word [expect 0] : "
<< languageContains(fig4, backwards_word) << "\n"

<< "fig4_reversed contains:\n"
<< "  forwards_word [expect 0] : "
<< languageContains(*fig4_reversed, forwards_word) << "\n"
<< "  backwards_word [expect 1] : "
<< languageContains(*fig4_reversed, backwards_word) << "\n";
```
2 The NestedWord class

The class `opennwa::NestedWord` provides support for building and iterating over nested words; currently there is no support for modifying them (other than appending).

The representation used by this class is closer to that of a word in a visibly-pushdown language [3]. It holds the linear contents of a word, but does not store the nesting relation explicitly. Instead, each position in the word is annotated with whether it is an internal, call, or return position. The nesting relation is induced by the matchings between calls and returns.

A position in a word is represented by the (nested) structure `NestedWord::Position`. Position itself declares an enumeration `Type`, which has the possible values `CallType`, `InternalType`, or `ReturnType`. A Position object has two (public) fields: Symbol `symbol` and `Position::Type` `type`; these hold the symbol at that position and the type of the position.

The `NestedWord` class has just seven functions:

```cpp
void NestedWord::appendInternal(Symbol s)
void NestedWord::appendCall(Symbol s)
void NestedWord::appendReturn(Symbol s)
void NestedWord::append(Position p)
size_t NestedWord::size() const
NestedWord::const_iterator NestedWord::begin() const
NestedWord::const_iterator NestedWord::end() const
```

These append the symbol `s` to the linear word, and annotate that position as an internal, call, or return position, respectively.

```cpp
void NestedWord::append(Position p)
```

Appends `p` to the word.

```cpp
size_t NestedWord::size() const
```

Returns the length of the word.

```cpp
NestedWord::const_iterator NestedWord::begin() const
NestedWord::const_iterator NestedWord::end() const
```

These return an iterator to the start or end of the nested word. The type returned by dereferencing these iterators is a `Position` object. (There is no non-const version of these functions.)

The only operation the library currently supports on NestedWords, besides building them, is checking whether a nested word is in an NWA’s language. This is done by the function `opennwa::query::languageContains(Nwa const &, NestedWord const &);`
3 The Nwa class

3.1 Construction, copying, assignment, and clearing

The NWA provides two constructors: the default constructor and the copy constructor. The default constructor creates an empty NWA. Thereafter, client code can add or remove states, add or remove symbols, add or remove transitions, and set the status of certain states as initial or final.

The following functions are basic NWA operations:

\[ \text{Nwa::Nwa}() \]

Constructs an empty NWA.

\[ \text{Nwa::Nwa}(\text{Nwa const} \ & \ \text{other}) \]

Copies \text{other}; the automata will not share structure. Any client information that is present is cloned.

\[ \text{Nwa}& \text{Nwa::operator=}(\text{Nwa const} \ & \ \text{other}) \]

Assigns \text{other} to \text{this}; the automata will not share structure. Client information is cloned.

\[ \text{bool Nwa::operator==}(\text{Nwa const} \ & \ \text{other}) \]

Determines whether the two automata are structurally equal — that is, they contain exactly the same set of states (including initial and final states), symbols, and transitions — and returns the result. (To test language equality, use the \text{languageEquals} function covered in \S 4.)

\[ \text{void Nwa::clear}() \]

Removes all states, symbols, and transitions from the NWA.

3.2 Simple manipulations

An \text{Nwa} object tracks the set of states, symbols, and transitions in the automaton. It also tracks the set of initial and accepting states. Counting each kind of transition separately, this gives 7 kinds of “entities” that client code may need to manipulate.

For each kind of entity, there are \text{Nwa} member functions to add and remove a single entity, check whether a particular entity is in the automaton, count the number of entities of a particular type, remove all entities of a particular type, and retrieve the set of entities in the NWA. In addition, there are member functions for counting and clearing all transitions, regardless of the type.

The names of these functions are very regular, but Tab. 1 (App. C) gives a list.

The only potential difficulties are in the interactions between the functions. Naturally, removing a state or a symbol also removes all transitions involving it; likewise, clearing all states or clearing all symbols also clears all transitions. (Calling \text{removeInitialState} or \text{removeFinalState} does not cause ripple effects.)

Adding a transition implicitly adds the states and symbol involved in that transition if they are not already present; hence it is not necessary to explicitly add all states and symbols before adding transitions. (It is quite reasonable to build an NWA by just adding initial states, final states, and transitions.)

The system supports two meta-symbols: \text{opennwa::EPSILON} (ε) and \text{opennwa::WILD} (تا). \text{EPSILON} is the standard ε from automata theory; it denotes that a transition can be traversed without matching and consuming an input symbol. Epsilon symbols cannot label call or return transitions. \text{WILD} is the ‘any’ symbol; it matches any single symbol. Because the NWA alphabet is not fixed, the actual symbols that \text{WILD} stands for is fluid. (This is useful if you do not know all possible symbols at the time you construct the NWA.) Note: \text{EPSILON} and \text{WILD} are not explicit elements of Σ; see Tab. 1 footnote 5.
3.3 Client Information

Each state in the NWA can be associated with some client-specific information. To utilize this functionality, client code must subclass `ClientInfo` and override `ClientInfo * clone() const`.

Once done, client information can be attached or retrieved using the following two functions (both members of `Nwa`):

```cpp
ref_ptr<ClientInfo> Nwa::getClientInfo(State st) const
Returns the client-specific information associated with the given state, st.

void Nwa::setClientInfo(State st, ref_ptr<ClientInfo> ci)
Sets the client-specific information, ci, associated with the given state, st.
```

Client information is tracked through the use of `ref_ptr`s, so the programmer must consider the standard aliasing and lifetime considerations imposed by using smart pointers. (`ClientInfo` supplies the `count` field needed by `ref_ptr`.) Operations that clone client information, such as copying an NWA, can also result in changes in aliasing: if the client information for states `p` and `q` are aliased in NWA `N` that is then copied to NWA `N'`, the “copies” of `p` and `q` in `N'` will have separate copies of the client information.

In addition, you may wish to subclass `Nwa` itself. There are several virtual helper functions that are called during intersection and determinization to compute the client info for the resulting automaton; these can be overridden to customize the behavior. (This design is an instance of the “template method” design pattern.)

The list of these helper methods is:

- `intersectClientInfoInternal`
- `intersectClientInfoCall`
- `intersectClientInfoReturn`
- `mergeClientInfo`
- `mergeClientInfoInternal`
- `mergeClientInfoCall`
- `mergeClientInfoReturn`

(The first three are used by intersection and the remaining four by determinization.) In addition, `stateIntersect` and `transitionIntersect` can further customize the behavior of intersection, including computation of client information. §6.2 and §6.6 have more information on intersection and determinization, and discuss these functions further.
The `opennwa::query` namespace provides functions related to querying both the structure and language of an automaton.

In this section, we first discuss how to retrieve information from an automaton’s transitions (§4.1) then move on to other queries about an automaton proper. Following that, we discuss queries about the language of an automaton (§4.3).

### 4.1 Querying information about an automaton’s transitions

The `opennwa::query` namespace provides a large number of functions for retrieving information about an Nwa’s transitions. The questions answered by these functions are of the form, e.g., “what are all symbols that appear on a return transition where the call predecessor state is $s_1$ and the return site state is $s_1$?”

Functions return information either about all transitions or about transitions of a particular type (internal, call, or return). The names of these functions have regular patterns based on the information that is known and desired, but the rules for producing the names are perhaps not as simple as they could be. Because of this, Tabs. 2–5 in App. C provide a quick reference for the functions.

The tables do not have all the information one needs to know in order to call them, and the entries use some shorthands; but they provide enough information for specifics to be looked up in the corresponding header or Doxygen documentation. The caption of each table gives the header that contains the functions listed. In the interest of space, the types of the function arguments are omitted, but they are likely to be what you expect. (The number of arguments is also always sufficient to uniquely identify the function because all types are just `wali::Keys` anyway.) Almost all functions return a `StateSet`, `SymbolSet`, or a `std::set` of pairs of a state and a symbol. (Those that return a set of pairs are marked specially.)

### 4.2 Querying other structural aspects of an automaton

The NWA library two additional functions for querying attributes of NWAs themselves. These functions are declared in the header `opennwa/query/automaton.hpp` (and, of course, are defined in the namespace `opennwa::query`).

```cpp
def isDeterministic(Nwa const & nwa):
    Computes whether the given NWA is deterministic and returns the result. Warning:
The result is not cached in the NWA, and as presently-implemented, this function
is very inefficient. Improvements to this will come in a later version of the library.

def statesOverlap(Nwa const & a, Nwa const & b):
    Computes whether the two NWAs share any states and returns the result.

def numCallSites(Nwa const & a):
    Returns the indicated statistic of the given automaton. A call site is a state which
is in the call position of a call transition, etc.
```

### 4.3 Querying properties of an automaton’s language

The NWA library has several functions for querying properties of the language of an NWA (or how the languages of two NWAs relate), and it also supports the poststar and prestar queries used in program analysis.
The following functions perform standard language-theoretic queries. They are located in namespace `opennwa::query` and are declared in `opennwa/query/language.hpp`:

```cpp
bool languageContains(Nwa const & nwa, NestedWord const & word)
Computes whether word is a member of nwa's language, and returns the result. (It simulates the NWA in a nondeterministic fashion; the NWA is not determinized to answer this query.) The worst-case running time is \(O(|\text{word}| \cdot w \log w)\), where \(w\) is the “width” of the nondeterminism of the NWA (i.e. the maximum number of simultaneous configurations that are possible when reading word).  
```

```cpp
bool languageIsEmpty(Nwa const & nwa)
Computes whether the NWA's language is empty, and returns the result.  
```

```cpp
ref_ptr<NestedWord> getSomeAcceptedWord(Nwa const & nwa)
If nwa's language is empty, returns NULL. Otherwise returns an arbitrary word in \(L(nwa)\). (If nwa accepts unbalanced words, the return value may be unbalanced.)  
```

```cpp
ref_ptr<NestedWord> getSomeShortestAcceptedWord(Nwa const & nwa)
If nwa's language is empty, returns NULL. Otherwise returns an arbitrary shortest word in \(L(nwa)\). (If nwa accepts unbalanced words, the return value may be unbalanced.)  
```

```cpp
bool languageSubsetEq(Nwa const & left, Nwa const & right)
Computes whether \(L(left) \subseteq L(right)\) and returns the result. Since this procedure must determinize right, the worst-case running time is \(O(2^m n^2 m)\) where \(n\) is the number of states in right and \(m\) is the number of states in left.  
```

```cpp
bool languageEquals(Nwa const & a, Nwa const & b)
Performs languageSubsetEq(a, b) && languageSubsetEq(b, a). This determinizes both machines; the worst-case running time is \(O(2^m n^2 m + 2^{m^2} m^2 n)\).  
```

There are also functions for performing a poststar and prestar query on an NWA. These work by converting the NWA to a WPDS (see \(\S\) and, in particular, \(\S\)) using the function `NwaToWpdsCalls` then running the query on the PDS.

Conceptually, the transition system queried by prestar and poststar is the space of configurations of the NWA. A configuration is a pair \(\langle q, s \rangle\), where \(q\) is the current state and \(s\) is the stack of unmatched call sites. (More exactly, it is the stack of states the machine was in before reading a symbol in a call position that has not yet been matched.) In reality, the transition system is the configuration space of the WPDS resulting from calling `NwaToWpdsCalls` (see \(\S\) and \(\S\)): that space is \(\langle p, t \rangle\), where \(p\) is a WPDS state and \(t\) is the stack of unmatched call sites with the NWA’s “current” state \(q\) added on top.

There are two variants of each function. One returns the WFA that results from the query, and the other stores the WFA in an output parameter. The WFA type is `wali::wfa::WFA`.

The functions that perform poststar and prestar are members of `Nwa`. (This interface will likely change in a later release, in which case the equivalent functionality will be moved into namespace `query` and these versions will be deprecated.) They are:

```cpp
WFA Nwa::prestar(WFA const & input, WeightGen & wg)
void Nwa::prestar(WFA const & input, WFA & output, WeightGen & wg)
WFA Nwa::poststar(WFA const & input, WeightGen & wg)
void Nwa::poststar(WFA const & input, WFA & output, WeightGen & wg)
```

Computes the prestar or poststar of the configurations specified by `input`, using the weights generated by `wg`. Either returns the result or stores it in the parameter `output`.

---

\(^2\)We count state lookups, additions, etc. as constant time even though they are actually logarithmic, and occasionally linear.
5 NWA serialization

It is possible to serialize an NWA in two different supported formats.

The first is a description language of our own creation, described in most of the rest of this section. We also provide a parser for this format, with minor caveats. The grammar that the parser uses is defined formally in §5.3.

The second is output in Graphviz Dot format. The output is natural. The color of an edge denotes the type of a transition: black edges are internal transitions, green edges are call transitions, and red edges are sets of return transitions with the same exit site, return site, and symbol. Return transitions are labeled with the symbol for its transitions and the list of call predecessors. Because it is possible to wind up with extremely long key names (especially in a determinized NWA) that can essentially destroy the ability of Dot to render useful output, by default any state names longer than 20 characters are replaced by the numerical value of the key. A call to print_dot can specify different behavior, however. All figures later in the document have been created via the Dot output (with minimal hand editing in some cases).

There is a member function to produce each type of output:

```cpp
std::ostream& Nwa::print(std::ostream & stream) const
Writes the custom output described in this section to stream, returning stream.
(This method is inherited from the wali::Printable abstract class.)
```

```cpp
std::ostream& Nwa::print_dot(ostream & os, std::string const & n, bool abbrev = true) const
Outputs a Dot description of the NWA to the stream os, returning os. The name of the graph (digraph "n" {...}) is given by n. Finally, if abbrev is set to false, the aforementioned abbreviation step above for state names is not done. Use this freedom at your peril.
```

5.1 Parser

The NWA library contains a semi-experimental parser for descriptions of NWAs. The format is described below, and matches the output of Nwa::print(std::ostream &), provided that the result of calling key2str on the state and symbol keys produces strings that match the description of ⟨actual-names⟩ in this grammar.

*Note:* This is experimental code; in particular, the error-handling in it is basically non-existent. We detect when the input does not match the grammar, but we sometimes signal such input failures with assertion violations. In a future version of the library, we will report errors more gracefully, almost certainly through exceptions.

There are two functions that parse NWA descriptions, both declared in opennwa/NwaParser.hpp in namespace opennwa:

```cpp
NwaRefPtr read_nwa(std::istream & is, std::string * name = NULL)
Reads a description of a single NWA from is and returns it. The serialization format allows an optional name for an NWA; if one is specified and name is non-null, then the name is stored in the string pointed to by name.
```

```cpp
std::map<std::string, NwaRefPtr> read_nwa_proc_set(std::istream &)
Reads the entire input stream, returning a map from the name of each automaton description to the parsed NWA. (This form was originally created to read NWAs for an entire program formatted with each procedure in a separate NWA. The name of each NWA was the name of the procedure in the original program.) The return type is typedefed to the name ProcedureMap, defined in the same header and namespace.
```
Q0: Start
Qf: Finish

Delta_i: (Start, a, Call)
Delta_i: (Entry, b, State)
Delta_i: (State, b, Exit)
Delta_i: (Return, a, Finish)

Delta_c: (Call, call, Entry)
Delta_r: (Exit, Call, ret, Return)

Figure 1: Serialized form of the NWA depicted in Fig. 3.

5.2 Examples
Figs. 1 and 2 show examples of the NWA serialization format, illustrating both the general form and some of the flexibilities in what precise format is accepted.

5.3 NWA description format
This section describes the grammar of the file format.

Note: Some characteristics in the description below have the following phrase as part of their description: “you should not rely on this.” In such cases, we reserve the right to change this behavior in future versions.

The grammar for an NWA is as follows. In order to accept a couple of slightly different output formats we have used in the past, there are some choices in whether braces are present and other aspects.

A single input stream can contain either a single NWA (just a series of blocks) or multiple NWAs. If you would like to describe multiple NWAs, each individual starts with the literal nwa.

\[
\langle \text{nwa-description} \rangle \ ::= \ (\text{'nwa'} \ \langle \text{name} \rangle? \ \text{':'}? \ ?')? \ ?'\}\langle \text{block} \rangle+ \ ?'
\]

Braces are required if “nwa” is present and the name is absent in order to distinguish the first block header (Q:, Q0:, or Qf:) from the name of the NWA.

An NWA is a sequence of blocks; each block specifies one or more states, symbols, or transitions.

\[
\langle \text{block} \rangle \ ::= \ \langle \text{state-block} \rangle
\ |
\langle \text{sigma-block} \rangle
\ |
\langle \text{delta-block} \rangle
\]

There can be more than one block of a given type; e.g. it is perfectly fine to specify all transitions in one block (as in Fig. 2), one block per transition (as in Fig. 1), or any mixture. The “block header” specifies what kind of block it is, and the body is a comma-separated list of whatever entity the block header specifies (e.g., states). The body may be surrounded by curly braces, but they are not required unless the body is empty.

State blocks can specify states, initial states, or accepting states; these are denoted by the block headers Q:, Q0:, and Qf:, respectively.
Q: {  
    Begin (=2),  
    End (=3),  
    Entry (=4),  
    Exit (=6),  
    (Begin, prime) (=10),  
    (End, prime) (=11),  
    (Entry, prime) (=12),  
    (Exit, prime) (=13)}

Q0: {  
    Begin (=2),  
    (Begin, prime) (=10)}

Qf: {  
    End (=3),  
    (Begin, prime) (=10)}

Sigma: {  
    call,  
    a,  
    ret}

Delta_c: {  
    (Begin (=2), call, Entry (=4)),  
    ((Begin, prime) (=10), call, Entry (=4))}

Delta_i: {  
    (Entry (=4), a, Exit (=6)),  
    ((Entry, prime) (=12), a, (Exit, prime) (=13))}

Delta_r: {  
    (Exit (=6), Begin (=2), ret, End (=3)),  
    (Exit (=6), Begin (=2), ret, (Begin, prime) (=10)),  
    (Exit (=6), (Begin, prime) (=10), ret, (Begin, prime) (=11)),  
    (Exit (=6), (Begin, prime) (=10), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), Begin (=2), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), Begin (=2), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), End (=3), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), End (=3), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), Entry (=4), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), Entry (=4), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), Exit (=6), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), Exit (=6), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), (Begin, prime) (=10), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), (Begin, prime) (=10), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), (End, prime) (=11), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), (End, prime) (=11), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), (Entry, prime) (=12), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), (Entry, prime) (=12), ret, (End, prime) (=11)),  
    ((Exit, prime) (=13), (Exit, prime) (=13), ret, (Begin, prime) (=10)),  
    ((Exit, prime) (=13), (Exit, prime) (=13), ret, (End, prime) (=11))}

Figure 2: Serialized form of the NWA depicted in Fig. 14.
\[
\langle \text{state-block} \rangle ::= \text{‘Q:’} \ (\langle \text{name-list} \rangle) \\
| \text{‘Q0:’} \ (\langle \text{name-list} \rangle) \\
| \text{‘Qf:’} \ (\langle \text{name-list} \rangle)
\]

(It is not necessary to explicitly list all states; if a state is listed in a transition, it is implicitly added to the machine as needed. Thus, it is quite reasonable to have a machine description with no Q: blocks.)

A name-list is simply a comma-separate list of names of states.

\[
\langle \text{name-list} \rangle ::= \{? \ (\langle \text{name} \rangle, \langle \text{name} \rangle)* \}?
\]

Recall that if the list is empty, it must be followed by a } (or EOF) to distinguish the next block header from the name of a state.

The grammar for \langle \text{name} \rangle will be specified below.

Symbol blocks are just like name blocks, except that they specify symbol names. The block header is simply ‘sigma’.

\[
\langle \text{sigma-block} \rangle ::= \text{‘sigma:’} \ (\langle \text{name-list} \rangle)
\]

A transition block can list internal, call, or return transitions, denoted by the block headers \text{Delta}_i, \text{Delta}_c, and \text{Delta}_r, respectively.

\[
\langle \text{delta-block} \rangle ::= \text{‘Delta}_i:’ \ (\langle \text{triple-list} \rangle) \\
| \text{‘Delta}_c:’ \ (\langle \text{triple-list} \rangle) \\
| \text{‘Delta}_r:’ \ (\langle \text{quad-list} \rangle)
\]

The bodies in each case are simply a comma-separated list of triples or quads (like \langle \text{name-list} \rangle); if these are empty, they must be followed by ‘}’:

\[
\langle \text{triple-list} \rangle ::= \{? \ (\langle \text{triple} \rangle, \langle \text{triple} \rangle)* \}?
\]
\[
\langle \text{quad-list} \rangle ::= \{? \ (\langle \text{quad} \rangle, \langle \text{quad} \rangle)* \}?
\]

and each triple (resp., quad) is a 3-tuple (4-tuple) of names:

\[
\langle \text{triple} \rangle ::= \langle \langle \text{name} \rangle, \langle \text{name} \rangle, \langle \text{name} \rangle \rangle
\]
\[
\langle \text{quad} \rangle ::= \langle \langle \text{name} \rangle, \langle \text{name} \rangle, \langle \text{name} \rangle, \langle \text{name} \rangle \rangle
\]

All that remains is to define the grammar for ‘name’. Because the format grew organically and we wished to keep compatibility with existing log files (which could previously not be automatically parsed), the ‘name’ terminal is a bit convoluted.

The name consists of the actual name of the state or symbol in question, followed by an optional parenthesis-delimited token that is ignored. (In the output of print(), the \langle \text{name} \rangle is the result of key2str and the optional token is the actual numeric value of the key.)

Before we discuss the grammar of \langle \text{name} \rangle, the following shows some examples of strings that are names:

\[
\begin{align*}
123 & \quad \text{xyz} & \quad \text{a2y} \\
123 \ (=4) & \quad \text{xyz} \ (=42) & \quad \text{a2y} \ (=030th) \\
\text{a,b} \ (=32) & \quad \text{x,y} & \quad <1, 2>
\end{align*}
\]

and strings which are not names:

\[
\begin{align*}
\text{hello world} & \quad \text{hello world} \ (=32) & \quad \text{(unmatched}
\end{align*}
\]
The actual grammar is:

\[
\begin{align*}
\langle \text{name} \rangle & \ ::= \langle \text{actual-name} \rangle \langle \text{dummy-token} \rangle ? \\
\langle \text{dummy-token} \rangle & \ ::= \ ' ( \neg \{ ' \} ) * ' \\
\end{align*}
\]

The \( \langle \text{actual-name} \rangle \) portion generally behaves like a standard programming-language identifier, but with one important difference: it can contain balanced parentheses, and any characters are allowed within a set of parentheses. (We actually allow a larger set of characters than is typical, but you should not rely on this. Surround your names with parentheses if you use “special” characters.)

We can now define the grammar of \textit{‘actual-name’}.

\[
\begin{align*}
\langle \text{actual-name} \rangle & \ ::= \langle \text{unit} \rangle + \\
\langle \text{unit} \rangle & \ ::= \langle \text{normal-char} \rangle \\
& \quad | \langle \text{paren-group} \rangle \\
\langle \text{normal-char} \rangle & \ ::= \text{character other than whitespace, ‘,’} \text{, or a paren} \\
\langle \text{std::isspace} \rangle & \ ::= \text{used to determine whether a character is whitespace.} \\
\langle \text{left-paren} \rangle & \ ::= \ ' ( | \ ' { | \ ' [ | \ '<' \\
\langle \text{right-paren} \rangle & \ ::= \ ' ) | \ ' } | \ ' ] | \ '> \\
\langle \text{non-paren} \rangle & \ ::= \text{a character other than any of the above} \\
\langle \text{paren-group} \rangle & \ ::= \langle \text{left-paren} \rangle ( \langle \text{non-paren} \rangle | \langle \text{paren-group} \rangle ) * \langle \text{right-paren} \rangle \\
\end{align*}
\]

(Finally, we do not currently demand that the types of parens match up: e.g., \(1, 2, 3\) is a valid name. You should not rely on this feature.)
6 Building NWAs from other NWAs (namespace opennwa::construct)

The library provides functions for performing automata-theoretic operations upon NWAs. The supported operations are union, intersection, concatenation, reversal, Kleene star, complement, and determinization.

The library supports two interfaces to each of these operations. In one, the operation allocates an NWA with new, performs the construction, and returns a NwaRefPtr to the result. In the other, the operation takes a reference to an NWA, clears it, and constructs the result in-place. The first form is usually more convenient to use, and creates a mini language for set expressions; the second form makes it possible to store the result of an operation in a subclass of Nwa, or an existing Nwa object.

Each of these functions is in the namespace opennwa::construct:

NwaRefPtr unionNwa(Nwa const & a, Nwa const & b)
void unionNwa(Nwa & out, Nwa const & a, Nwa const & b)
Computes the union of the NWAs a and b, either returning the result or storing it in out. See Section 6.1. (This function is called unionNwa instead of union because the latter is a C++ keyword.) The two NWAs must not have any states in common, and the output will be nondeterministic even if both input NWAs are deterministic. The running time is $O(|Q_a| + |Q_b| + |δ_a| + |δ_b|)$.

NwaRefPtr intersect(Nwa const & a, Nwa const & b)
void intersect(Nwa & out, Nwa const & a, Nwa const & b)
Computes the intersection of the NWAs a and b, either returning the result or storing it in out. See Section 6.2. If both input NWAs are deterministic, the output will be deterministic. The worst-case running time is $O(|Q_a||Q_b|d_ad_b)$, where $d_a$ is the maximum out-degree of a state in a and $d_b$ is the maximum out-degree of a state in b.

NwaRefPtr concat(Nwa const & left, Nwa const & right)
void concat(Nwa & out, Nwa const & left, Nwa const & right)
Computes the concatenation of the NWAs left and right, either returning the result or storing it in out. See Section 6.3. The output automaton will be nondeterministic. The running time is $O(|Q_a| + |Q_b| + |δ_a| + |δ_b| + |Q_a||δ_b|)$.

NwaRefPtr star(Nwa const & orig)
void star(Nwa & out, Nwa const & orig)
Computes the Kleene star of the NWA orig, either returning the result or storing it in out. See Section 6.4. The output automaton will be nondeterministic. The running time is $O(|Q| + |δ| + |Q_0||Q_f||δ_c|)$.

NwaRefPtr reverse(Nwa const & orig)
void reverse(Nwa & out, Nwa const & orig)
Computes the NWA that accepts the reverse of each nested word accepted by the NWA orig, either returning the result or storing it in out. See Section 6.5. The running time is $O(|Q| + |δ|)$.

NwaRefPtr determinize(Nwa const & orig)
NwaRefPtr determinize(Nwa & out, Nwa const & orig)
Computes a determinization of orig, either returning the result or storing it in out. See Section 6.6. The worst-case running time for this algorithm is at least $O(|Q|^3 \cdot 2^{|Q|^2})$.

---

3We count state lookups, additions, etc. as constant time even though they are actually logarithmic, and occasionally linear.
NwaRefPtr complement(Nwa const & orig)
    void complement(Nwa & out, Nwa const & orig)

    Computes the complement of the determinization of orig, either returning the result or storing it in out. See Section 6.7. complement() determinizes orig, so its worst-case running time is also at least \( O(|Q|^3 \cdot 2^{|Q|^2}) \).

    As mentioned above, these functions create a small language of set expressions. For instance, to compute an automaton \( M \) whose language is \( A \cup (B \cap C)^\ast \) (where \( A, B, \) and \( C \) are NwaRefPtrs), one can write

    \[
    \text{NwaRefPtr } M = \text{unionNwa}(\ast A, \ast \text{star}(\ast \text{intersect}(\ast B, \ast C)))
    \]

6.1 Union

The union of two NWAs is constructed by taking the union of each of the components of the NWAs. (In particular, it does not do a cross-product construction, and will always produce a nondeterministic automaton as a result as long as both machines have at least one initial state.)

Formally, the union of NWAs \( N = (Q_N, \Sigma_N, Q_{0,N}, \delta_N, F_N) \) and \( M = (Q_M, \Sigma_M, Q_{0,M}, \delta_M, F_M) \) is \( N \cup M = (Q_N \cup Q_M, \Sigma_N \cup \Sigma_M, Q_{0,N} \cup Q_{0,M}, \delta_N \cup \delta_M, F_N \cup F_M) \).

As an example, Fig. 5 illustrates the union of the NWA in Figure 3 and the NWA in Figure 4. Note that there are two initial states.
Figure 6: Simple NWA to intersect with the NWA in Figure 3.

Figure 7: The NWA resulting from the intersection of the NWA in Figure 3 and the NWA in Figure 6. Note that because each of the input NWAs only accepts a single word and those words are different, the language of the intersection is empty. The NWA built up by intersect() got as far as it could. (key#2,3) is the pair of states (start,start) from the two input automata.

The state sets of the NWAs must not overlap, i.e., \( Q_1 \cap Q_2 = \emptyset \). Client code should not rely on this condition being checked or any particular behavior occurring if it does not hold.

Client information is copied directly from the original NWAs using ClientInfo::clone().

### 6.2 Intersection

The intersection of two NWAs is computed in the standard cross-product fashion, using a worklist algorithm to only compute those states that are reachable.

The algorithm traverses the original NWAs starting at the initial states and incrementally adds transitions for each pair of “intersectable” transitions that are encountered. By default, transitions are intersectable when the transitions are the same kind (internal, call, or return) and the symbols on the edge are identical or one is wild.

For example, Fig. 6 shows the intersection of Fig. 3 and Fig. 6 and Fig. 9 shows the intersection of the two NWAs in Fig. 8.

It is possible to customize what symbols are considered equivalent, or otherwise impose constraints on what transitions can be intersected, by overriding the transitionIntersect function in a subclass of Nwa. In addition, it is possible to impose additional constraints on what states can be combined by overriding stateIntersect. Both functions also produce the result of the intersection: transitionIntersect produces the symbol that will be used on the resulting edge, and stateIntersect produces the state that will be used as the target.
Figure 8: Two complex NWAs to intersect.

Figure 9: The NWA resulting from the intersection of the NWAs in Figure 8.
The default behavior of `transitionIntersect` is that two transitions are intersectable if neither symbol is epsilon and either the symbols are the same or at least one of the symbols is a wild. (Epsilon transitions are dealt with in `intersect()` itself. If you override `transitionIntersect`, it should return `false` if either input is epsilon.)

The default behavior of `stateIntersect` is that any two states can be combined, the resulting state is labeled with a key that is uniquely generated from the pair of the keys of the two states under consideration, and the client information associated with the resulting state is `null`.

Client information is initially generated by the helper method `stateIntersect`, but can be altered through the use of the helper methods `intersectClientInfoInternal`, `intersectClientInfoCall`, and `intersectClientInfoReturn`, which are invoked by `intersect()` as transitions of the three different kinds involving the associated state are added. The default behavior of these three functions is to perform no changes to the `ClientInfo`. These methods can be overridden in subclasses to specify alternative behaviors.

The following operations are virtual functions in class `Nwa` and are intended to be overridden to customize behavior:

```cpp
bool stateIntersect(Nwa const & first, State state1,
                    Nwa const & second, State state2,
                    State & resSt, ref_ptr<ClientInfo> & resCI)
```

Determines whether the given states can be combined and, if so, creates the combined state. If the two states are incompatible, returns `false`. If the two states are compatible, it computes the key of the combined state (storing it in `resSt`) and the client information (storing it in `resCI`), then returns `true`.

```cpp
bool transitionIntersect(Nwa const & first, Symbol sym1,
                         Nwa const & second, Symbol sym2,
                         Symbol & resSym)
```

Determines whether the given symbols are considered to be equivalent for the purposes of intersection. If so, it computes the symbol that should be associated with the combined transition (storing it in `resSym`) and returns `true`. If not, returns `false`.

```cpp
void intersectClientInfoInternal(Nwa const & first, State src1, State tgt1,
                                  Nwa const & second, State src2, State tgt2,
                                  Symbol resSym, State resSt)
```

```cpp
void intersectClientInfoCall(Nwa const & first, State call1, State entry1,
                              Nwa const & second, State call2, State entry2,
                              Symbol resSym, State resSt)
```

```cpp
void intersectClientInfoReturn(Nwa const & first, State exit1, State call1, State ret1,
                                Nwa const & second, State exit2, State call2, State ret2,
                                Symbol resSym, State resSt)
```

Called after a transition of the corresponding type is added to the automaton. It is intended to be used to alter the client information associated with `resSt` given the endpoints of the new transition.

### 6.3 Concatenation

The concatenation of two NWAs is constructed by taking the union of all states and transitions of the two automata, and adding internal epsilon transitions from each final state of the first NWA to each initial state of the second NWA. In the resulting NWA, the initial states are the initial states from the first NWA, and the final states are the final states of the second NWA.
However, the concatenation construction is a bit more complicated than just this, because we need to address the issue of an unbalanced-left word being concatenated with an unbalanced-right word. (Recall the definition of what happens when an NWA reads a pending return: it is allowed to take a return transition where the call predecessor is an initial state of the automaton.)

To deal properly with the case where the first operand generates strings with pending calls and the second operand generates strings with pending returns, the transitions from the second automaton are augmented in the following manner. When computing the concatenation of \textit{left} and \textit{right}, for each transition \((q, p_0, a, q') \in \delta_\text{right}^r\) where \(p_0 \in Q_0^\text{right}\) (these are transitions that \textit{right} can take when reading a pending return) and each \(p \in Q^\text{left}\), we add \((q, p, a, q')\) to \(\delta_\text{right}\) of the result. (It is actually only necessary to add such a transition for states \(p\) that either appear in the call position of a call transition in \textit{left} or are in \(Q_0^\text{left}\).

The state sets of the NWAs must not overlap, i.e., \(Q_1 \cap Q_2 = \emptyset\). \textit{Client code should not rely on this condition being checked.}

Client information is copied directly from the original NWAs using \texttt{ClientInfo::clone}().

Fig. 12 shows the result of concatenating Fig. 3 and Fig. 10.

### 6.4 Kleene star

Like concatenation, Kleene-star is complicated in the case of NWAs because of the ability to have unbalanced words in the automaton’s language. Relative to standard FAs, in the concatenation construction we only needed to add extra transitions; in this construction, we must add additional states as well.

The NWA resulting from performing Kleene-Star on the NWA shown in Fig. 13 is shown in Fig. 14.

The construction presented in \cite{3} has a minor error. The error is analogous to not adding a distinguished start state in the traditional Thompson construction, and in fact can be exhibited using the same example (it is not necessary to use NWA calls or returns). Alur confirmed that our interpretation of the construction in \cite{3} is correct \cite{1}.

Below, we present a version that uses \(\varepsilon\)-transitions, and thus it looks a bit different from the version in \cite{3}.

When computing \(R = A^*\) for some NWA \(A\), the resulting NWA has two “copies” of \(A\). These are denoted by primed and unprimed version of states from \(A\) in the definition below.

Suppose that \(R\) is reading a word \(w = w_1w_2\cdots w_n\), where each \(w_i \in L(A)\). \(R\) begins in a start state of \(A'\). Henceforth it maintains the following invariant on the state that \(R\) is in with respect to the portion of \(w\) read so far: if the next symbol \(\sigma\) is in a return position, then that symbol is a pending return in the current \(w_i\) iff \(R\) is in the \(A\)’ portion, i.e., if the current state is primed. (Note that this return only needs to be pending in the current \(w_i\). In the full string \(w\), \(\sigma\) may match a call in an earlier \(w_j\), or it may be pending in the whole string.)

Internal transitions thus keep \(R\) in the same copy of \(A\), and call transitions always take \(R\) to the unprimed copy of \(A\) (because if it then reads a return, the return will match that call). Return transitions can target either copy of \(A\): if the call predecessor is unprimed, then the target will be unprimed; if the call predecessor is primed, then the target will be primed.

The description above describes \(R\)’s operation under “normal” conditions. If \(R\) is in a final state (either primed or unprimed) of the automaton \(A\), it is also allowed to guess.

---

\(^4\)The initial state of the automaton \(A^*\) must accept, because \(\varepsilon\) is in \(L^*\); and because of this property it is incorrect to merely add epsilon transitions from the old final states to the old initial states. If you do this and there is a cycle from the initial state back to itself (for example, a self-loop on the initial state), the word corresponding to that path would be accepted even though it should not be.
Figure 10: Simple NWA to concatenate onto the NWA in Figure 3.

Figure 11: The NWA resulting from performing reverse on the NWA in Figure 3.

Figure 12: The NWA resulting from the concatenation of the NWA in Figure 3 with the NWA in Figure 10.
Figure 13: An NWA on which to perform Kleene-Star.

Figure 14: The NWA resulting from performing Kleene-Star on the NWA in Figure 13.
that it should “restart” by taking an epsilon transition to a distinguished start and final
state \(q_0\). This guess is correct if it just read the last character in \(w_i\) (making the next
character the first one in \(w_{i+1}\)). Note that \(q_0\) only has transitions to the \(A'\) portion of \(R\),
maintaining the invariant.

The reason for the two copies of \(A\) comes into play when \(R\) reads a return \(\sigma\) while
in the \(A'\) portion. By the invariant, \(\sigma\) is pending in the current \(w_i\). In the original
automaton \(A\), the transitions that the machine can use are return transitions \((q, r, \sigma, p)\)
where the call predecessor \(r\) is in \(Q_0\). We need to make sure that \(R\) can take those same
transitions. There are two cases we need to consider:

1. For the cases where \(\sigma\) is pending in the whole string \(w\), we need to have a version of
the return transition with \(q_0\) in the call-predecessor position, so we add \((q', q_0, \sigma, p')\).
2. For the cases where \(\sigma\) is matched with a call in some earlier \(w_j\), it does not matter
what state the machine was in before that call; thus we add \((q', s, \sigma, p')\) for each
state \(s\) in \(Q \cup Q'\).

“Normal” operation corresponds to the transitions introduced by the \textsc{Internal},
\textsc{Call}, and \textsc{Locally-Matched-Return} inference rules given below. The source states
of both transitions added by \textsc{Locally-Matched-Return} are unprimed, because if the
current symbol is a return that matches a call in the same \(w_i\), \(R\) will be in the \(A'\) portion
by the invariant. The \textsc{Restart} rule allows \(R\) to restart, and the \textsc{Start} rule allows \(R\) to
get from \(q_0\) to the \(A'\) portion; these transitions target the \(A'\) portion because there have
been no calls read in the current \(w_i\), and thus a return symbol would be pending. The
\textsc{Globally-Pending-Return} rule addresses the situation where the current symbol is
a return that is pending in the whole string \(w\). (This is the first case in the previous
paragraph.) The \textsc{Locally-Pending-Return} rule addresses the situation where the
current symbol is a return that is pending in the current \(w_i\) but matches a call from a
previous \(w_j\).

Formally, if the original NWA is \((Q, \Sigma, Q_0, \delta, Q_f)\), then the result of performing
Kleene-Star on that NWA is \((Q^*, \Sigma, Q_0^*, \delta^*, Q^*_f)\). The sets of states are defined by
\(Q^* = Q \cup Q' \cup \{q_0\}\) (with \(Q' = \{q' \mid q \in Q\}\) and \(q_0 \notin Q\), and \(Q_0^* = Q_f^* = \{q_0\}\). The
transitions in \(\delta^*\) are defined by the following rules:

\[
\begin{align*}
\text{INTERNAL} & \quad (q, \sigma, p) \in \delta_i \quad (q, \sigma, p) \in \delta_i^* \quad (q, \sigma, p) \in \delta_c \quad (q, \sigma, p) \in \delta_c^* \\
\text{CALL} & \quad (q, \sigma, p) \in \delta_r \quad (q, \sigma, p) \in \delta_r^* \\
\text{LOCALLY-MATCHED-RETURN} & \quad (q, r, \sigma, p) \in \delta_r \quad (q, r, \sigma, p) \in \delta_r^* \\
\text{START} & \quad q \in Q_0 \quad (q, \varepsilon, q_0) \in \delta_i \quad (q, \varepsilon, q_0) \in \delta_i^* \\
\text{RESTART} & \quad q \in Q_f \quad (q', \varepsilon, q_0) \in \delta_i \quad (q', \varepsilon, q_0) \in \delta_i^* \\
\text{GLOBALLY-PENDING-RETURN} & \quad (q, r, \sigma, p) \in \delta_r \quad r \in Q_0 \\
\text{LOCAL-PENDING-RETURN} & \quad (q, r, \sigma, p) \in \delta_r \quad r \in Q_0 \quad s \in Q \cup Q' \\
& \quad (q', s, \sigma, p') \in \delta_r^* \\
\end{align*}
\]

Client information is copied directly from the original NWA (using \texttt{ClientInfo::clone()})
such that for each \(q \in Q\), \(q\) and \(q'\) have (different copies of)
the same client information.

\textit{Note:} The key for state \(q'\) is generated from the key for state \(q\) using the expression
\texttt{getKey(q, getKey("prime"))}. The input automaton to the Kleene star function must
not already contain both \(q\) and \(q'\) for any \(q\).
6.5 Reverse

A nested word \( n = (w, \llra) \) is reversed by reversing the linear word \( w \) and exchanging calls and returns. Formally, \( n^{rev} = (w^{rev}, \{(|w|+1-r, |w|+1-c)|(c,r) \in \llra}) \). (Pending calls and returns are handled by defining \(|w|+1-(-\infty) = -\infty \) and \(|w|+1-(+\infty) = +\infty \).) Roughly speaking, call transitions in \( A \) correspond to return transitions in \( A^{rev} \) and vice versa, and we reverse the direction of all transitions as in the standard FA construction.

We describe the construction from the perspective of \( A^{rev} \) — that is, a “call transition” is a call transition in \( A^{rev} \), and a “call” is a call in the reversed string.

Perhaps unsurprisingly, pending returns pose a problem because the role of initial and final states are exchanged. Because of this complication, the algorithm for reversing an NWA has a similar flavor to that of the Kleene-star procedure. The automaton \( A^{rev} \) has two “copies” of \( A \) (primed and unprimed), and maintains the same invariant as the Kleene-star construction: if the next symbol \( \sigma \) is in a return position, then that symbol is a pending return iff \( A^{rev} \) is in the \( A^\prime \) portion. (For those familiar with the construction in [3], ours is more complicated because the version in [3] will not work as stated with a weakly-hierarchical NWA.)

If the original NWA is \( (Q, \Sigma, Q_0, \delta, Q_f) \), then the result of reversing that NWA is \( (Q \cup Q^\prime, \Sigma, Q_f^\prime, \delta^{rev}, Q_0) \) obtained using the following rules:

\[
\text{INTERNAL} \quad \frac{(p, \sigma, q) \in \delta_i}{(q, \sigma, p) \in \delta_i^{rev}}
\]

\[
\text{CALL-RETURN} \quad \frac{(q_c, \sigma_c, q_e) \in \delta_c}{(q_r, \sigma_r, q_e) \in \delta_r} \quad \frac{(q_r, \sigma_r, q_e) \in \delta_r^{rev}}{(q_c, \sigma_c, q_e) \in \delta_c^{rev}}
\]

\[
\text{PENDING-RETURN} \quad \frac{(q_c, \sigma, q_e) \in \delta_c}{(q_r, Q_f, q, q_c) \in \delta_c^{rev}}
\]

The NWA resulting from performing reverse on the NWA shown in Figure 3 is shown in Figure 11.

Client information is copied directly from the original NWA using \texttt{ClientInfo::clone()}.

6.6 Determinize

\textbf{Definition.} An NWA, \( (Q, \Sigma, Q_0, \delta, Q_f) \), is \textbf{deterministic} iff

1. \( |Q_0| \leq 1 \),
2. For all \( q \in Q \), there is never a choice between reading \( \sigma \) and following a \( \sigma \) transition or following a wild (\( \emptyset \)) transition:
   - if \( (q, \emptyset, q') \in \delta \) then \( \{|q'(q, \sigma, q') \in \delta, \sigma \neq \emptyset\}| = 0 \);
   - otherwise, for all \( \sigma \in \Sigma - \{\emptyset\} \), \( \{|q'(q, \sigma, q') \in \delta, \sigma \neq \emptyset\}| \leq 1 \),
3. There are no \( \varepsilon \) transitions:
   - for all \( (q, \varepsilon, q') \in \delta, \varepsilon \neq \varepsilon \),
   - for all \( (q, \sigma, q') \in \delta, \sigma \neq \varepsilon \), and
   - for all \( (q, \sigma, q') \in \delta, \sigma \neq \varepsilon \).

If an NWA is not deterministic, then it is \textbf{non-deterministic}.
Determinizing an NWA operates like a generalization of the classical subset construction. Instead of the states in the determinized NWA being subsets of states in the original NWA, states of the determinized NWA are sets of state pairs (i.e., binary relations on states) [3]. To support determinization, the library provides a typedef of `std::set<pair<State, State>>` as `Nwa::BinaryRelation`. See also `opennwa/RelationOps.hpp`.

We present the algorithm we use for determinization in App. [1] The result of determinizing the automaton in Fig. [15] is shown in Fig. [16].

Client information is generated through the use of the helper method `mergeClientInfo`, but can be altered through the use of the helper methods `mergeClientInfoInternal`, `mergeClientInfoCall`, and `mergeClientInfoReturn`, which are invoked by `determinize` as transitions of the three kinds involving the associated state are added. The default behavior of `mergeClientInfo` is that the `ClientInfo` associated with the resulting state is `null`. The default behavior of `mergeClientInfoInternal`, `mergeClientInfoCall`, and `mergeClientInfoReturn` is to make no changes to the `ClientInfo`. These methods can be overridden to specify alternative behaviors. As determinization is performed, `mergeClientInfo` is called each time a new state is created. Then, as each transition is added, `mergeClientInfoInternal`, `mergeClientInfoCall`, or `mergeClientInfoReturn` is called (depending on the type of transition being added) to update the `ClientInfo` associated with the target state of the transition being added.

The following functions can be overridden in a subclass of `Nwa` to customize the behavior of determinization:

```cpp
void mergeClientInfo (Nwa const & nondet, BinaryRelation const & binRel,
                      St resSt, ref_ptr<ClientInfo>& resCI) {
    Callback that gets called when a new state `resSt` (representing the binary relation `binRel`) is added to the determinized automaton. Intended to provide a hook for computing the client information that should be associated with the new state; the client information should be set in the output parameter `resCI` (i.e., `setClientInfo` should not be called directly). `nondet` is the NWA being determinized.
}

void mergeClientInfoInternal (Nwa const & nondet,
                               BinaryRelation const & binRelSource,
                               BinaryRelation const & binRelTarget,
                               State sourceSt, Symbol resSym, State resSt,
                               ref_ptr<ClientInfo>& resCI) {
                               Callbacks that get called when a new transition is added to the given automaton. The endpoints and their associated binary relations are given. Alters the client information associated with `resSt` given information about the transition being added to the determinized automaton.
```
6.7 Complement

Complementing an NWA is performed by determinizing the automaton and then complementing the set of final states. The result of complementing the NWA shown in Figure 15 is shown in Figure 17.

Client information is computed during determinization as described in the previous section, then cloned during the complement step.
Figure 15: Simple nondeterministic NWA.

Figure 16: The NWA resulting from determinizing the NWA in Figure 15. As mentioned in the text, states in the determinized NWA are relations on the states in the original NWA. The state $\emptyset$ has been removed from this diagram.
Figure 17: The complement of the NWA in Figure 15 (the determinization of which is shown in Figure 16). We omit transitions to the state \{\}; any action that does not appear in the diagram goes to the state \{\}. 
Conversions between WPDSs and NWAs (namespace opennwa::nwa_pds)

It is possible to convert an NWA into a WALi WPDS and vice versa. However, the construction of an NWA from a WPDS is not the inverse of constructing a WPDS from an NWA, i.e., one cannot perform the two conversions in sequence and obtain the identity conversion.

At a high level, the WPDS to NWA conversion works by making the NWA encode both the state of the WPDS and its top-of-stack symbol. A WPDS rule of the form \( \langle p, q_1 \rangle \rightarrow \langle p, q_2 \rangle \) leaves the stack height unchanged, and is thus associated with an internal NWA transition; in this case, that transition goes from the state \( (p, q_1) \) to \( (p, q_2) \). The symbol of a transition is associated with the top-of-stack symbol of the source state, so in this example, the symbol labeling that transition would be \( q_1 \). In other words, the WPDS rule \( \langle p, q_1 \rangle \rightarrow \langle p, q_2 \rangle \) is translated to the NWA internal transition \( ((p, q_1), q_1, (p, q_2)) \).

WPDS push rules correspond to NWA call transitions, and WPDS pop rules correspond to NWA return transitions.

The conversion in the other direction creates a WPDS with one primary state and one “helper” state for each NWA state that appears in the exit position of a return transition. The NWA’s state is encoded by the symbol at the top of the WPDS’s stack – essentially the inverse of the encoding described in the previous paragraph. A slight complication arises in the case of return transitions. The NWA is able to look at both the exit node and the call predecessor. In the WPDS this would correspond to looking at the top two stack symbols – but the WPDS is only allowed to look at the top one. Hence each NWA return transition becomes two WPDS rules: the first pops the top symbol (which corresponds to the current NWA state) and remembers what it was using the helper state; the second rule looks at the call predecessor and the helper state to dispatch to the corresponding return site.

The library also offers two kinds of variants of this conversion. First, there is a backwards variant that can be used for backwards dataflow-analysis problems. Second, the resulting WPDS can stack either calls or returns. The “stacking-calls” version turns a call transition \( (c, \sigma, e) \) into a WPDS rule \( \langle p, c \rangle \rightarrow \langle p, ec \rangle \) – leaving the call predecessor \( c \) on the stack. (This is the translation described in the previous paragraph.) The “stacking-returns” version has, in general, several WPDS push rules for each NWA call transition. Each push rule leaves a potential return site on the stack. (For example, if there is a call transition \( (c, \sigma, e) \) and a return transition \( (x, c, \sigma', r) \), then the WPDS will have a rule \( \langle p, c \rangle \leftarrow \langle p, er \rangle \).

The following functions are in the namespace opennwa::nwa_pds and, except for plusWpds(), are declared in the header opennwa/nwa_pds/conversions.hpp. The WPDS type is wali::wpds::WPDS, and is declared in wali/wpds/WPDS.hpp.

void WpdsToNwa(Nwa & out, const WPDS& pds)
NwaRefPtr WpdsToNwa(const WPDS& pds)

Converts pds to an NWA, either storing the result in nwa or returning it.

---

5 This encoding is motivated by our uses of both WPDSs and NWAs in program analysis. It is common for WPDSs to have just one state, \( p \), and to use the top-of-stack symbol to encode the “current” program point. Pushing something onto the stack corresponds to a call, and popping corresponds to a return. For NWAs, we use the states themselves to encode the current program point. (The function that converts a WPDS into an NWA supports multi-state WPDSs, however; a WPDS rule of the form \( \langle p_1, q_1 \rangle \rightarrow \langle p_2, q_2 \rangle \) is translated to the NWA internal transition \( ((p_1, q_1), q_1, (p_2, q_2)) \).)
WPDS NwaToWpdsCalls(Nwa const & nwa, WeightGen const & wg)
WPDS NwaToWpdsCalls(Nwa const & nwa, WeightGen const & wg, ref_ptr<Wrapper> wr)
WPDS NwaToBackwardsWpdsCalls(Nwa const & nwa, WeightGen const & wg)
WPDS NwaToWpdsReturns(Nwa const & nwa, WeightGen const & wg)
WPDS NwaToBackwardsWpdsReturn(Nwa const & nwa, WeightGen const & wg)

These functions each construct a WPDS that is equivalent to nwa using the appropriate method (backwards or forwards flow, and stacking calls or stacking returns), returning the result. Uses wg to determine weights for the WPDS’s transitions. The second variant of NwaToWpdsCalls takes a wali::wpds::Wrapper reference wr, and the WPDS is constructed by passing wr to the constructor. This feature can be used, for instance, if you would like the resulting WPDS to support witness tracing. (If wr is NULL, then the second version is equivalent to the first.)

State getProgramControlLocation()
Returns the program state p used as the primary WPDS state in the result of the NwaToWpds* variants.

State getControlLocation(State exit, State call, State return)
Returns the WPDS state p, q or p q 3 used as a “helper” state for return transitions from exit to return with call as a predecessor.

WPDS plusWpds(Nwa const & nwa, const WPDS& base)
This function returns a WPDS that is the product of the NWA nwa and WPDS base, as described in the “Explicit NWA plus PDS” construction from § 6. This function is declared in the header opennwa/nwa_pds/plusWpds.hpp.

7.1 WPDS to NWA

The WpdsToNwa functions convert a WPDS into an NWA in a manner faithful to the encoding sketched out in the introduction to this section.

Assume that we have a WPDS (P, Γ, ∆) where ∆ = (∆0, ∆1, ∆2). This WPDS is converted into an NWA (Q, Σ, {}, δ, {}) using the following rules:

\[
\text{States} \quad \begin{array}{c}
p \in P \\
(p, q) \in Q \\
q \in \Gamma
\end{array} \quad \begin{array}{c}
q \in \Sigma
\end{array} \\
\text{Internal} \quad \begin{array}{c}
\langle p, q \rangle \mapsto \langle p', q' \rangle \in \Delta_1 \\
(p, q), (p', q') \in \delta_i
\end{array} \\
\text{Call} \quad \begin{array}{c}
\langle p, q_c \rangle \mapsto \langle p', q_c, q_r \rangle \in \Delta_2 \\
(p, q_c), (p', q_c), (p', q_r) \in \delta_c
\end{array} \\
\text{Return} \quad \begin{array}{c}
\langle p'', q_x \rangle \mapsto \langle p''' , \epsilon \rangle \in \Delta_0 \\
\langle p, q_c \rangle \mapsto \langle p', q_c, q_r \rangle \in \Delta_2 \\
(p'', q_x), (p, q_c), (p', q_c), (p', q_r) \in \delta_r
\end{array}
\]

Note that these rules generate an NWA return transition for each pair of WPDS pop and push rules; there is no constraint between the two rules. This is because, with the exception of the “revealed” stack symbol q_r, everything that the push rule talks about concerns the call predecessor (p, q_c) and entry node (q', q_c); nothing that the pop rule talks about — p'', q_x, or p''' — has any relation to those. (A consequence is that the number of NWA transitions may be quadratic in the number of WPDS rules.)

In the resulting NWA, Q_0 and Q_f are empty; client code must set the initial and final states as appropriate (with addInitialState(State) and addFinalState(State). The keys that are generated for the names of the NWA states are part of the interface of this function; they are generated by getKey(p, q) where p and q are the keys of the WPDS state and stack symbol being converted.
Figure 18: An example WPDS.

Figure 19: The NWA resulting from converting the WPDS in Fig. 18 into an NWA.

All weights on WPDS rules are ignored, and do not survive in any way in the resulting NWA. The client information for all states in the resulting NWA are set to null.

For example, the NWA created from the WPDS shown in Fig. 18 is shown in Fig. 19.

7.2 NWA to WPDS

An NWA can also be converted into a WPDS. Weights for the rules of the resulting WPDS are provided using a mechanism described below. In this way it is possible to use the WPDS reachability queries that are a part of the main WALi library on NWAs. As mentioned in the introduction to §7 there are four variations on the NWA-to-WPDS conversion: forward flow with call states on the stack, backward flow with call states on the stack, forward flow with return states on the stack, and backward flow with return states on the stack. All four variations use \texttt{WeightGen} to determine weights for WPDS rules.

\texttt{WeightGen} is an abstract class that client code must subclass to calculate the weights of the rules in the generated WPDS. It allows the underlying NWA to be decoupled from the weight domain used in the WPDS. See [5, §4-5] for details about weight domains.

There is a trivial weight domain (containing \(1\) and \(0\) only) implemented in the class \texttt{wali::Reach}, defined in \texttt{wali/Reach.hpp}. A \texttt{WeightGen} subclass that returns elements from this reachability domain is provided as the \texttt{ReachGen} class, defined in \texttt{opennwa/WeightGen.hpp}. \texttt{ReachGen} returns \(1\) for all transitions.
In addition, we provide shortest-path weights and two variants on a WeightGen subclass for them. The shortest path weights are implemented in the class wali::ShortestPathSemiring in wali/ShortestPathSemiring.hpp; a semiring element of this class consists of just a single integer. The two WeightGen implementations we provide (both defined in opennwa/WeightGen.hpp) are ShortestPathGen and ShortestWordGen. Both set the weight of each WPDS rule to have a length 1 (note that this is different from the semiring 1 element), except that ShortestWordGen uses length 0 for rules that correspond to ε transitions. Thus ShortestPathGen results in a WPDS with weights that describe the length of the shortest path through the NWA, from the initial configurations, while ShortestWordGen results in a WPDS with weights that describe the length of the shortest word.

The following operations are virtual methods of WeightGen intended to be overridden:

- `sem_elem_t WeightGen::getOne() const = 0` Returns an instance of the 1 element of the weight domain.
- `sem_elem_t getWeight(State source, ClientInfoRefPtr sourceInfo, Symbol symbol, Kind k, State target, ClientInfoRefPtr targetInfo ) const` Computes and returns the weight for the rule corresponding to the transition from source to target (of kind k) labeled with symbol symbol. By default, returns `getOne()`.
- `sem_elem_t getWildWeight(State source, ClientInfoRefPtr sourceInfo, State target, ClientInfoRefPtr targetInfo ) const` Computes and returns the weight for the WPDS rule corresponding to the transition from source to target labeled with the meta-symbol @. By default, returns `getOne()`.
- `sem_elem_t getExitWeight(State src, ClientInfoRefPtr srcInfo) const` This method computes the weight (in the desired semiring) for the return rule of the WPDS corresponding to the exit src. Note: the value is generally the same as `getOne()`, which is what the default implementation returns.

### 7.2.1 Forwards flow stacking calls

The conversion is performed by:

\[
\begin{align*}
(p, q_c, \sigma, q_r) & \in \Delta_2 & q \in Q & \quad (q, \sigma, q') \in \delta_i \\
(p, q_c) & \stackrel{w_2}{\rightarrow} (p, q_c, q_r) & q \in \Gamma & \quad (p, q) & \stackrel{w_1}{\rightarrow} (p, q', \epsilon) \in \Delta_0 \end{align*}
\]

where

\[
\begin{align*}
w_0 &= \begin{cases} 
\text{wg.getWildWeight}(q_c, CI_{q_c}, q_r, CI_{q_r}), & \text{if } \sigma = @ \\
\text{wg.getWeight}(q_c, CI_{q_c}, \sigma, EXIT_TO_RET, q_r, CI_{q_r}), & \text{otherwise}
\end{cases} \\
w_1 &= \begin{cases} 
\text{wg.getWildWeight}(q, CI_q, q', CI_{q'}), & \text{if } \sigma = @ \\
\text{wg.getWeight}(q, CI_q, \sigma, INTRA, q', CI_{q'}), & \text{otherwise}
\end{cases} \\
w_2 &= \begin{cases} 
\text{wg.getWildWeight}(q_c, CI_{q_c}, q_r, CI_{q_r}), & \text{if } \sigma = @ \\
\text{wg.getWeight}(q_c, CI_{q_c}, \sigma, CALL_TO_ENTRY, q_r, CI_{q_r}), & \text{otherwise}
\end{cases} \\
w_3 &= \text{wg.getOne()}
\end{align*}
\]
For example, the WPDS resulting from converting the NWA shown in Fig. 20 into a WPDS is shown in Fig. 21.

### 7.2.2 Backwards flow stacking calls

The backwards-flow conversions are equivalent to calling `wali::nwa::construct::reverse` and then the corresponding forwards flow version. When reversing, call transitions become return transitions (and vice versa), and so call sites become return sites (and vice versa). Return sites in the original automaton behave as call sites in the reversed automaton, and thus this version of the NWA-to-WPDS conversion stacks return states. (In other words, the Calls and Returns part of `NwaToBackwardsWpdsCalls` and `NwaToBackwardsWpdsReturns` refer to the behavior of the states in the reversed automaton, not the role they play in the original.)

For example, the result of converting the NWA in Fig. 20 into a backwards flow WPDS is shown in Fig. 22.

Formally, the conversion is performed by:

\[
\begin{align*}
\forall p \in P & \quad (q_c, \sigma, q_e) \in \delta_c & \quad q \in Q & \quad (q, \sigma, q') \in \delta_i \\
\forall p_{qe} \in P & \quad q \in \Gamma & \quad \langle p, q' \rangle \xrightarrow{w_1} \langle p, q \rangle \in \Delta_1
\end{align*}
\]

where \( w_0 \) =

\[
\begin{align*}
\text{if } \sigma = \text{ENTRY} & \quad \text{wg.getWildWeight}(q_c, CI_{q_c}, q_e, CI_{q_e}) \\
\text{otherwise} & \quad \text{wg.getWeight}(q_c, CI_{q_c}, \text{ENTRY}, q_e, CI_{q_e})
\end{align*}
\]

\( w_1 \) =

\[
\begin{align*}
\text{if } \sigma = \text{ENTRY} & \quad \text{wg.getWildWeight}(q, CI_{q}, q', CI_{q'}) \\
\text{otherwise} & \quad \text{wg.getWeight}(q, CI_{q}, \text{ENTRY}, q', CI_{q'})
\end{align*}
\]

\( w_2 \) =

\[
\begin{align*}
\text{if } \sigma = \text{ENTRY} & \quad \text{wg.getWildWeight}(q_x, CI_{q_x}, q_r, CI_{q_r}) \\
\text{otherwise} & \quad \text{wg.getWeight}(q_x, CI_{q_x}, \text{EXIT}, q_r, CI_{q_r})
\end{align*}
\]

\( w_3 = \text{wg.getOne}() \)

### 7.2.3 Forwards flow stacking returns

As an example, converting the NWA in Fig. 20 into a WPDS results in the WPDS shown in Fig. 23.

The conversion is performed by:

\[
\begin{align*}
\forall p \in P & \quad (q_x, q_c, \sigma, q_r) \in \delta_r & \quad q \in Q & \quad (q, \sigma, q') \in \delta_i \\
\forall p_{qe} \in P & \quad q \in \Gamma & \quad \langle p, q' \rangle \xrightarrow{w_1} \langle p, q \rangle \in \Delta_1
\end{align*}
\]

\[
\begin{align*}
\forall p_{qe} \in P & \quad \langle q_c, \sigma, q_e \rangle \in \delta_c & \quad (q_x, q_c, \gamma, q_r) \in \delta_r \\
\forall q \in Q & \quad \langle p, q' \rangle \xrightarrow{w_2} \langle p, q \rangle \in \Delta_1
\end{align*}
\]

\[
\begin{align*}
\forall p_{qe} \in P & \quad \langle q_c, \sigma, q_e \rangle \in \delta_c & \quad (q_x, q_c, \gamma, q_r) \in \delta_r \\
\forall q \in Q & \quad \langle p, q' \rangle \xrightarrow{w_2} \langle p, q \rangle \in \Delta_1
\end{align*}
\]

\[
\begin{align*}
\forall p_{qe} \in P & \quad \langle q_c, \sigma, q_e \rangle \in \delta_c & \quad (q_x, q_c, \gamma, q_r) \in \delta_r \\
\forall q \in Q & \quad \langle p, q' \rangle \xrightarrow{w_2} \langle p, q \rangle \in \Delta_1
\end{align*}
\]
where $w_0 = \texttt{wg.getOne}()$

\[
\begin{align*}
    w_1 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q, CI_q, q', CI_{q'}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q, CI_q, \sigma, \text{INTRA}, q', CI_{q'}), & \text{otherwise}
    \end{cases} \\

    w_2 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q_x, CI_{q_x}, q_r, CI_{q_r}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q_x, CI_{q_x}, \sigma, \text{EXIT_TO_RET}, q_r, CI_{q_r}), & \text{otherwise}
    \end{cases} \\

    w_3 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q_c, CI_{q_c}, q_e, CI_{q_e}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q_c, CI_{q_c}, \sigma, \text{CALL_TO_ENTRY}, q_e, CI_{q_e}), & \text{otherwise}
    \end{cases}
\end{align*}
\]

### 7.2.4 Backwards flow stacking returns

As an example, converting the NWA in Fig. 20 into a backwards flow WPDS results in the WPDS shown in Fig. 24.

The conversion is performed by:

\[
\begin{align*}
\text{States} & \quad p \in P \\
(p, q) \in \delta_c & \quad q \in Q \\
(p, q) \in \Gamma & \quad (p, q') \in \Delta_1
\end{align*}
\]

\[
\begin{align*}
\langle p, q_c, \sigma, q_e \rangle \in \Delta_0 & \quad \langle p, q_c, \sigma, q_r \rangle \in \Delta_r \\
\langle p, q_c \rangle \xrightarrow{w_0} \langle p, q_c, \epsilon \rangle & \quad \langle p, q_c \rangle \xrightarrow{w_3} \langle p, q_c \rangle \in \Delta_1 \\
\langle p, q_r \rangle \xrightarrow{w_2} \langle p, q_x \rangle & \quad \langle p, q_x \rangle \xrightarrow{w_1} \langle p, q \rangle \in \Delta_1
\end{align*}
\]

where $w_0 = \texttt{wg.getOne}()$

\[
\begin{align*}
    w_1 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q, CI_q, q', CI_{q'}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q, CI_q, \sigma, \text{INTRA}, q', CI_{q'}), & \text{otherwise}
    \end{cases} \\

    w_2 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q_x, CI_{q_x}, q_r, CI_{q_r}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q_x, CI_{q_x}, \sigma, \text{EXIT_TO_RET}, q_r, CI_{q_r}), & \text{otherwise}
    \end{cases} \\

    w_3 &= \begin{cases} 
        \texttt{wg.getWildWeight}(q_c, CI_{q_c}, q_e, CI_{q_e}), & \text{if } \sigma = @ \\
        \texttt{wg.getWeight}(q_c, CI_{q_c}, \sigma, \text{CALL_TO_ENTRY}, q_e, CI_{q_e}), & \text{otherwise}
    \end{cases}
\end{align*}
\]
Figure 20: An example NWA.

- \( \langle p,\text{main} \rangle \rightarrow \langle p, q_1 \rangle \)
- \( \langle p, q_1 \rangle \rightarrow \langle p, c_1 \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, q_2 \rangle \)
- \( \langle p, q_2 \rangle \rightarrow \langle p, q_3 \rangle \)
- \( \langle p, q_3 \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, q_4 \rangle \)
- \( \langle p, q_4 \rangle \rightarrow \langle p, q_5 \rangle \)
- \( \langle p, q_5 \rangle \rightarrow \langle p, c_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, q_6 \rangle \)
- \( \langle p, q_6 \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, e \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, r_1 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)

Figure 21: The WPDS resulting from calling NwaToWpdsCalls on Fig. 20.

- \( \langle p, \text{main} \rangle \rightarrow \langle p, q_1 \rangle \)
- \( \langle p, q_1 \rangle \rightarrow \langle p, c_1 \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, q_2 \rangle \)
- \( \langle p, q_2 \rangle \rightarrow \langle p, q_3 \rangle \)
- \( \langle p, q_3 \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, q_4 \rangle \)
- \( \langle p, q_4 \rangle \rightarrow \langle p, q_5 \rangle \)
- \( \langle p, q_5 \rangle \rightarrow \langle p, c_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, q_6 \rangle \)
- \( \langle p, q_6 \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, e \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, r_1 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)

Figure 22: The result of calling NwaToBackwardsWpdsCalls on Fig. 20.

- \( \langle p, q_1 \rangle \rightarrow \langle p, \text{main} \rangle \)
- \( \langle p, c_1 \rangle \rightarrow \langle p, q_1 \rangle \)
- \( \langle p, q_2 \rangle \rightarrow \langle p, e \rangle \)
- \( \langle p, q_2 \rangle \rightarrow \langle p, q_3 \rangle \)
- \( \langle p, q_3 \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, q_4 \rangle \)
- \( \langle p, q_4 \rangle \rightarrow \langle p, q_5 \rangle \)
- \( \langle p, q_5 \rangle \rightarrow \langle p, c_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, q_6 \rangle \)
- \( \langle p, q_6 \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, e \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, r_1 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)

Figure 23: The WPDS resulting from calling NwaToWpdsReturns on Fig. 20.

- \( \langle p, \text{main} \rangle \rightarrow \langle p, q_1 \rangle \)
- \( \langle p, q_1 \rangle \rightarrow \langle p, c_1 \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, q_2 \rangle \)
- \( \langle p, q_2 \rangle \rightarrow \langle p, q_3 \rangle \)
- \( \langle p, q_3 \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, q_4 \rangle \)
- \( \langle p, q_4 \rangle \rightarrow \langle p, q_5 \rangle \)
- \( \langle p, q_5 \rangle \rightarrow \langle p, c_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, q_6 \rangle \)
- \( \langle p, q_6 \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, c \rangle \rightarrow \langle p, c \rangle \)
- \( \langle p, e \rangle \rightarrow \langle p, e \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, x \rangle \rightarrow \langle p, x \rangle \)
- \( \langle p, r_1 \rangle \rightarrow \langle p, r_1 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)
- \( \langle p, r_2 \rangle \rightarrow \langle p, r_2 \rangle \)

Figure 24: The result of calling NwaToBackwardsWpdsReturns on Fig. 20.

38
References

A Nested-Word Automata

Nested-word automata (NWAs)\footnote{2,3} are a generalization of finite-state automata that can capture the matched-parenthesis structure that is exhibited by, for example, opening and closing tags in XML and the call/return structure in execution traces in multi-procedure programs. Their languages represent somewhat of a middle-ground between standard regular languages and context-free languages. They are strictly more powerful than finite automata (FAs), but can also accept some languages that are context-free once the nesting structure (call/return, etc.) is dropped. (For instance, there is an NWA that accepts exactly the language of properly-balanced parentheses, with the associated matching structure.) Importantly, nested-word languages also retain all the closure properties that makes standard regular languages attractive; in particular, they are closed under complementation and intersection. However, they are not directly comparable to languages of linear words, because the nesting structure is an explicit part of each nested word.

Definition. A nested word \((w, \rightsquigarrow)\) over alphabet \(\Sigma\) is an ordinary (linear) word \(w \in \Sigma^*\) together with a nesting relation \(\rightsquigarrow\).

The relation \(\rightsquigarrow\) is a collection of edges (over the positions in \(w\)) that do not cross. Formally, \(\rightsquigarrow \subseteq \{−\infty, 1, 2, \ldots, |w|\} \times \{1, 2, \ldots, |w|, +\infty\}\) such that:

- Nesting edges only go forward: if \(i \rightsquigarrow j\) then \(i < j\).
- No two edges share a position unless one is \(±\infty\): for \(1 \leq i \leq |w|\), either \(i = ±\infty, j = ±\infty\), or there is at most one \(j\) such that \(i \rightsquigarrow j\) or \(j \rightsquigarrow i\).
- Edges do not cross: if \(i \rightsquigarrow j\) and \(i' \rightsquigarrow j'\), then one cannot have \(i < i' \leq j < j'\).

A nested-word language is any set of nested words; such a language is a regular nested-word language if it is accepted by an NWA as defined below.

When \(i \rightsquigarrow j\) holds, for \(1 \leq i \leq |w|\), \(i\) is called a call position. If \(i \rightsquigarrow +\infty\), then \(i\) is a pending call; otherwise \(i\) is a matched call, and the (unique) position \(j\) such that \(i \rightsquigarrow j\) is called its return successor. (Note that these terms refer to positions within \(w\) and not the to symbol itself, which is what you may expect if you are familiar with visibly pushdown languages\footnote{3}.)

Similarly, when \(i \rightsquigarrow j\) holds, for \(1 \leq j \leq |w|\), \(j\) is a return position. If \(−\infty \rightsquigarrow j\), then \(j\) is a pending return, otherwise \(j\) is a matched return, and the (unique) position \(i\) such that \(i \rightsquigarrow j\) is called its call predecessor.

A position \(1 \leq i \leq |w|\) that is neither a call nor a return is an internal position. A nested word is balanced if it has no pending calls or returns. A nested word is unbalanced-left (or a nested-word prefix) if it has only pending calls, and it is unbalanced-right (or a nested-word suffix) if it has only pending returns.

Definition. A nested-word automaton (NWA) \(A\) is a 5-tuple \((Q, \Sigma, Q_0, \delta, F)\), where \(Q\) is a finite set of states, \(\Sigma\) is a finite alphabet, \(Q_0 \subseteq Q\) is the initial state, \(F \subseteq Q\) is a set of final states, and \(\delta\) is a transition relation. The transition relation \(\delta\) consists of three components, \((\delta_c, \delta_i, \delta_r)\), where:

- \(\delta_c: (Q \times \Sigma) \times Q\) is the transition relation for internal positions of the input word.
- \(\delta_i: (Q \times \Sigma) \times Q\) is the transition relation for call positions.
- \(\delta_r: (Q \times Q \times \Sigma) \times Q\) is the transition relation for return positions.

Starting from a state in \(Q_0\), an NWA \(A\) reads a nested word \((w, \rightsquigarrow)\) from left to right, and performs transitions according to the current input symbol and \(\rightsquigarrow\). If \(A\) is in state \(q\) when reading input symbol \(\sigma\) at position \(i\) in \(w\), and \(i\) is an internal (resp. call) position in \(\rightsquigarrow\), \(A\) makes a transition to a state \(q'\) (if
one is available) such that \((q, \sigma, q') \in \delta_i\) (resp, \((q, \sigma, q') \in \delta_c\)). If \(i\) is a return position, let \(k\) be the call predecessor of \(i\) (so \(k \rightarrow i\)) and \(q_c\) be the state \(A\) was in just before the transition it made on the \(k^{th}\) symbol; \(A\) changes to a state \(q'\) such that \((q, q_c, \sigma, q') \in \delta_r\). If there is a computation of \(A\) on input \((w, \rightarrow)\) that terminates in a state \(q \in F\), then \(A\) accepts \((w, \rightarrow)\).

NWAs can be either deterministic or nondeterministic, and these variations have equivalent power.

We extend the above definition to allow two meta-symbols, \(\varepsilon\) and \(\@\) (wild). Call and return transitions are prohibited from being labeled with \(\varepsilon\), but allowing \(\varepsilon\)'s on internal transitions can be done in an analogous fashion to standard FAs. The wild symbol, \(\@\), can label any transition, and it matches any input symbol. For example, the internal rule from the definition would be reworded as follows:

If \(A\) is in state \(q\) when reading input symbol \(\sigma\) at position \(i\) in \(w\), and \(i\) is an internal position, \(A\) makes a transition to a state \(q'\) (if one is available) such that either \((q, \sigma, q') \in \delta_i\) or \((q, \@, q') \in \delta_i\).

The other rules are modified similarly.

To distinguish among the different roles for states in an internal transition \((q, \sigma, q')\), we say that \(q\) is the source and \(q'\) is the target. Similarly, to distinguish among the roles for states in a call transition \((q_c, \sigma, q_e)\), we say that \(q_c\) is the call state and \(q_e\) is the entry state. To distinguish among the roles for states in a return transition \((q_x, q_c, \sigma, q_r)\), we say that \(q_x\) is the exit state, \(q_c\) is the call state (or call predecessor), and \(q_r\) is the return state.
B Determinize

We found the explanations of how to determinize NWAs that are given in [2, 3] to be confusing (and contradictory between the two accounts), and so we reformulated it using relational operations.

We use the following notation in the determinize algorithm:

\[(Q, \Sigma, \delta, Q_0, Q_f)\] The components of the input automaton nwa

\[\delta_i|\sigma\] The binary relation \(\{(p, q) | (p, \sigma, q) \in \delta_i\}\)

\[\delta_c|\sigma\] The binary relation \(\{(p, q) | (p, \sigma, q) \in \delta_c\}\)

\[\delta_r|\sigma\] The binary relation \(\{(p, q) | \exists c. (p, c, \sigma, q) \in \delta_r\}\)

\[R \circ S\] Relational composition of the binary relations \(R\) and \(S\)

\[R^*\] Transitive closure of the binary relation \(R\)

\[Q_{new}, \delta_{new}\] Components of the determinized NWA

We use the following auxiliary function to compute the target of a return transition:

\[
\text{Merge}(R^{exit}, R^{call}, \delta) = \{(q, q') \mid \exists q_1, q_2. (q, q_1) \in R^{call} \text{ and } (q_1, q) \in R^{exit} \text{ and } (q_2, q_1, q') \in \delta}\]

Each state \(R\) in the determinized automaton is a binary relation on states in the original. In a standard deterministic FA, a state \(\{q_0, q_1, \cdots, q_n\}\) means the automaton can be in state \(q_0\) of the original FA, or in state \(q_1\) of the original, etc. For NWAs, a state \(\{(p_0, q_0), (p_1, q_1), \cdots, (p_n, q_n)\}\) means that the NWA is one of the states \(\{q_0, q_1, \cdots, q_n\}\), but the relation carries around extra meaning.

If a state in the determinized automaton contains a pair \((p, q)\), then this means the input automaton can begin in the state \(p\), immediately perform a call transition, follow a path with balanced calls and returns, and finally arrive in state \(q\). In such a configuration, if the input automaton then reads a return symbol, \(q\) is the exit site and \(p\) is the call predecessor. These two pieces of information are exactly what the automaton needs to decide what return transitions it can take. The call predecessor \(p\) needs to be stored explicitly because it is possible to arrive at the same state \(q\) with different call predecessors.

At the start of the run, and any time the automaton has not read any pending calls, the first component of each pair in the current state will be some \(q \in Q_0\); this is because the initial states act as call predecessors in that situation.
determinize(NWA nwa)

\[ \text{Close} = (\delta_i|_\epsilon)^* \]

\[ R_0 = Q_0 \times Q_0 \circ \text{Close} \]

\[ Q_{\text{new}} = \{R_0\} \]

Insert \( R_0 \) in WL

while \( WL \neq \emptyset \) do

select and remove a relation \( R \) from WL

// Note that \( R \) is a state in \( Q_{\text{new}} \)

mark \( R \)

for \( \sigma \in \Sigma \) do

// Compute internal transitions

\[ R^i = R \circ \delta_i|_\sigma \circ \text{Close} \]

\[ Q_{\text{new}} = Q_{\text{new}} \cup \{R^i\} \]

Insert \( R \xrightarrow{\sigma} R^i \) into \( \delta^i_{\text{new}} \)

if \( R^i \) unmarked then

\[ WL = WL \cup \{R^i\} \]

// Compute call transitions

\[ R^c = \text{Close} \circ \delta_c|_\sigma \circ \text{Close} \]

\[ Q_{\text{new}} = Q_{\text{new}} \cup \{R^c\} \]

Insert \( R \xrightarrow{\sigma} R^c \) into \( \delta^c_{\text{new}} \)

if \( R^c \) unmarked then

\[ WL = WL \cup \{R^c\} \]

// Compute return transitions where \( R \) appears as the exit node

for \( R^{\text{call}} \in Q_{\text{new}} \) do

\[ R^r = \text{Merge}(R, R^{\text{call}}, \delta_r|_\sigma) \circ \text{Close} \]

\[ Q_{\text{new}} = Q_{\text{new}} \cup \{R^r\} \]

Insert \( (R, R^{\text{call}}, \sigma, R^r) \) into \( \delta^r_{\text{new}} \)

if \( R^r \) unmarked then

\[ WL = WL \cup \{R^r\} \]

// Compute return transitions with \( R \) as the call predecessor

for \( R^{\text{exit}} \in Q_{\text{new}} \) do

\[ R^r = \text{Merge}(R^{\text{exit}}, R, \delta_r|_\sigma) \circ \text{Close} \]

\[ Q_{\text{new}} = Q_{\text{new}} \cup \{R^r\} \]

Insert \( (R^{\text{exit}}, R, \sigma, R^r) \) into \( \delta^r_{\text{new}} \)

if \( R^r \) unmarked then

\[ WL = WL \cup \{R^r\} \]

// end worklist while loop

\[ Q_{f_{\text{new}}} = \{R \in Q_{\text{new}} | \text{there is } (p, q) \in R \text{ with } q \in Q_f \} \]

return \((Q_{\text{new}}, \Sigma, \delta_{\text{new}}, \{R_0\}, Q_{f_{\text{new}}})\)
This section provides a number of quick-reference tables.

The tables do not have *all* the information one needs to know in order to call the functions, and the entries use some shorthands; but they provide enough information for specifics to be looked up in the corresponding header or Doxygen documentation. The caption of each table gives the header that contains the functions listed. In the interest of space, the types of the function arguments are sometimes omitted, but they are likely to be what you expect.

**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Accessors and mutators of NWA components</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>Query functions for all transition types</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>Query functions for internal transitions</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>Query functions for call transitions</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>Query functions for return transitions</td>
<td>49</td>
</tr>
<tr>
<td>Table 1: Accessors and mutators of NWA components. All functions are members of the base class, and thus declared in opennwa/Nwa.hpp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add and remove</td>
<td>States addState(State st) removeState(State st)</td>
<td></td>
</tr>
<tr>
<td>and return</td>
<td>states isState(State st) sizeStates() clearStates()</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>states getStates() or {begin,end}States()</td>
<td></td>
</tr>
<tr>
<td>initial states</td>
<td>addInitialState(State st) removeInitialState(State st) isInitialState(State st) sizeInitialStates() clearInitialStates() getInitialStates() or {begin,end}InitialStates()</td>
<td></td>
</tr>
<tr>
<td>final states</td>
<td>addFinalState(State st) removeFinalState(State st) isFinalState(State st) sizeFinalStates() clearFinalStates() getFinalStates() or {begin,end}FinalStates()</td>
<td></td>
</tr>
<tr>
<td>symbols</td>
<td>addSymbol(Symbol sym) removeSymbol(Symbol sym) isSymbol(Symbol sym) sizeSymbols() clearSymbols() getSymbols() or {begin,end}Symbols()</td>
<td></td>
</tr>
<tr>
<td>all transitions</td>
<td>— findTrans(State s1, Symbol sym, State s2)</td>
<td></td>
</tr>
<tr>
<td>internal transitions</td>
<td>addInternalTrans( ... ) removeInternalTrans( ... ) sizeInternalTrans() internalTrans()</td>
<td></td>
</tr>
<tr>
<td>call transitions</td>
<td>addCallTrans( ... ) removeCallTrans( ... ) sizeCallTrans() callTrans()</td>
<td></td>
</tr>
<tr>
<td>return transitions</td>
<td>addReturnTrans( ... ) removeReturnTrans( ... ) sizeReturnTrans() returnTrans()</td>
<td></td>
</tr>
</tbody>
</table>

These functions return a bool indicating whether the item was added/removed. Adding a transition implicitly adds all states and symbols in that transition if they are not already present. Removing a state or symbol removes all transitions involving it. (Hence clearing all states or clearing all symbols also clears all transitions.)

EPSILON and WILD are not explicit members of the symbol set. The other functions (getKind()) return a StateSet or SymbolSet as appropriate. Removing a state or a symbol also removes all transitions involving it. (Hence clearing all states or clearing all symbols also clears all transitions.)
Table 2: Query functions for all transition types. These functions are in the namespace opennwa::query; include the file opennwa/query/transitions.hpp.

For return transitions, the “source” is the first component of the transition; nothing involving call predecessors (the second component of return transitions) appears in this table. A table entry of “—” means that the combination of arguments does not make sense.

<table>
<thead>
<tr>
<th>What you know</th>
<th>What you want</th>
</tr>
</thead>
<tbody>
<tr>
<td>source (nothing)</td>
<td>getSuccessors(nwa, src)</td>
</tr>
<tr>
<td>symbol (nothing)</td>
<td>getSuccessors(nwa, sym, tgt)</td>
</tr>
<tr>
<td>target (nothing)</td>
<td>getSuccessors(nwa, tgt)</td>
</tr>
<tr>
<td>this... ... and this</td>
<td>getSuccessors(nwa, src, sym)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>symbols</th>
<th>source</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSymbol(nwa, src, tgt, &amp;sym)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>getSuccessors(nwa, sym, tgt)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>getSuccessors(nwa, tgt)</td>
<td>(none)</td>
<td>—</td>
</tr>
<tr>
<td>getSuccessors(nwa, src, sym)</td>
<td>(none)</td>
<td>—</td>
</tr>
</tbody>
</table>

...and this
### Table 3: Query functions for internal transitions.

These functions are in the namespace `opennwa::query`; include the file `opennwa/query/internals.hpp`.

<table>
<thead>
<tr>
<th>What you know</th>
<th>What you want</th>
<th>How you get this...</th>
<th>... and this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>source (nothing)</td>
<td>(nothing)</td>
<td>getSources(nwa)</td>
<td>this...</td>
</tr>
<tr>
<td>target (nothing)</td>
<td>(nothing)</td>
<td>getInternalSym(nwa)</td>
<td>...</td>
</tr>
<tr>
<td>source (nothing)</td>
<td>target (nothing)</td>
<td>getInternalSym(nwa, src)</td>
<td>getSources(nwa, src)</td>
</tr>
<tr>
<td>target (nothing)</td>
<td>source (nothing)</td>
<td>target (nothing)</td>
<td>source (nothing)</td>
</tr>
<tr>
<td>target (nothing)</td>
<td>target (nothing)</td>
<td>target (nothing)</td>
<td>target (nothing)</td>
</tr>
<tr>
<td>target (nothing)</td>
<td>source (nothing)</td>
<td>source (nothing)</td>
<td>source (nothing)</td>
</tr>
<tr>
<td>sources symbols</td>
<td>target (nothing)</td>
<td>sources symbols</td>
<td>target (nothing)</td>
</tr>
</tbody>
</table>

A table entry of “—” means that combinations of arguments do not make sense.

Returns a set of pairs (either source/symbol or symbol/target).
Table 4: Query functions for call transitions. The "call site" is the source of the transition (and uses the argument name \texttt{call}) and the "entry" of the transition is the target (and uses the argument name \texttt{ent}).

<table>
<thead>
<tr>
<th>What you know</th>
<th>What you want</th>
</tr>
</thead>
<tbody>
<tr>
<td>- getCallSites(nwa)</td>
<td>getEntries(nwa, call)</td>
</tr>
<tr>
<td>getEntries(nwa, sym)</td>
<td>getCallSym(nwa, call, ent)</td>
</tr>
<tr>
<td>getEntries(nwa, sym)</td>
<td>getEntries(nwa, sym)</td>
</tr>
</tbody>
</table>

1 Returns a set of pairs (either call site/symbol or symbol/entry).
Table 5: Query functions for return transitions. These functions are in the namespace `opennwa::query` and include the file `opennwa/query/returns.hpp`. The “exit site” is the source of the transition (the first component) and uses the argument name `exit` in this table; the “call predecessor” is the second component and uses the argument name `call`; the symbol is the third component and uses the argument name `sym`; the “return site” is the fourth component and uses the argument name `ret`.

<table>
<thead>
<tr>
<th>What you know</th>
<th>What you want</th>
<th>Symbol</th>
<th>Return Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit sites</td>
<td>(nothing)</td>
<td>exit</td>
<td>getExits(nwa)</td>
</tr>
<tr>
<td>call pred</td>
<td>(nothing)</td>
<td>call</td>
<td>getReturnSym(nwa, exit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or getReturns(nwa, exit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or getCalls(nwa, exit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>symbol</td>
<td>getEntries(nwa, call, sym)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or getReturns(nwa, call, sym)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return site</td>
<td>getReturnSym(nwa, exit, call, ret)</td>
</tr>
</tbody>
</table>

Note: The table entries represent the output of the query functions for return transitions. Each function takes a set of pairs (a symbol with one of the states, in the order of the raw transition).