WALi: Nested-Word Automata [*]

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

[1] University of Wisconsin
[2] GrammaTech, Inc.
{burtona,adi,driscoll,reps}@cs.wisc.edu

1 Nested-Word Automata and Overview of the Library's Organization

WALi-NWA is a library for constructing and querying nested-word automata. It is implemented in C++.

1.1 Definitions

Definition 1. *A **nested-word** $(w, \rightsquigarrow)$ over an alphabet, Σ, is an ordinary word w, together with a **nesting relation** $\rightsquigarrow$ of length $|w|$. $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ such that*
  - *Nesting edges only go forwards: if $i \rightsquigarrow j$ then $i < j$.*
  - *No two edges share a position: for $1 \leq i \leq l$, $|\{j | i \rightsquigarrow j\}| \leq 1$ and $|\{j | j \rightsquigarrow i\}| \leq 1$.*
  - *Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.*
*When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, 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**. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, 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 $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.*

*A **balanced nested-word** is a nested word that has no pending calls or returns. An **unbalanced-left nested word** (or **nested-word prefix**) is a nested word that has no pending returns. An **unbalanced-right nested word** (or **nested-word suffix**) is a nested word that has no pending calls.*

The library supports unbalanced-left, unbalanced-right, and balanced nested-words, but has no direct support for general nested words. For the effects of boolean operations on the four kinds of nested-words see Figure 1.

Intersection	B	UL	UR	NW
B	B	B	B	B
UL	B	UL	B	UL
UR	B	B	UR	UR
NW	B	UL	UR	NW

Concatenation	B	UL	UR	NW
B	B	UL	UR	NW
UL	UL	UL	NW	NW
UR	UR	NW	UR	NW
NW	NW	NW	NW	NW

Union	B	UL	UR	NW
B	B	UL	UR	NW
UL	UL	UL	NW	NW
UR	UR	NW	UR	NW
NW	NW	NW	NW	NW

Operation	B	UL	UR	NW
Star	B	UL	UR	NW
Reverse	B	UR	UL	NW
Complement	UL	UL	UR	NW

Fig. 1. Table of Operations on Nested Words. Note: B = balanced nested-word, UR = unbalanced-right nested-word, UL = unbalanced-left nested-word, NW = nested-word

Definition 2. *A **nested-word automaton** (NWA), A, is a tuple of the form $A = (Q, \Sigma, Q_0, \delta, Q_f)$, where Q is a finite set of states, Σ is a finite alphabet, $Q_0 \subseteq Q$ is a set of initial states, $Q_f \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_i, \delta_c, \delta_r)$, where:*

- *$\delta_i \subseteq Q \times \Sigma \times Q$ defines the transition relation for internal positions.*
- *$\delta_c \subseteq Q \times \Sigma \times Q$ defines the transition relation for call positions.*
- *$\delta_r \subseteq Q \times Q \times \Sigma \times Q$ defines the transition relation for return positions.*

Starting from some $q_0 \in Q_0$, an NWA A reads a nested word $nw = (w, v)$ from left to right and performs transitions (possibly non-deterministically) according to the input symbol and the nesting relation. That is, if A is in state q when reading input symbol σ at position i in w, then if i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. Otherwise, i is a return position. Let k be the call-predecessor of i, and let q_c be the state A was in just before the transition it made on the kth symbol; then A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in Q_f$, then A accepts nw [1].

To distinguish among the different roles for states in an internal transition, (q, σ, 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, σ, q_e), we say that q_c is the call state and q_e is the entry state, and to distinguish among the roles for states in a return transition, (q_x, q_c, σ, q_r), we say that q_x is the exit state, q_c is the call state, and q_r is the return state.

1.2   Classes

NWA
  Models nested-word automata. This is the main class of the WALi-NWA package.
NWS
  Models a nested-word suffix, i.e., an unbalanced-right nested word. This can also model balanced nested words.
NWP
  Models a nested-word prefix, i.e., an unbalanced-left nested word. This can also model balanced nested words.
ClientInfo
  Additional information that can be attached to NWA states.
WeightGen
  Weight-generation algorithms for the NWA to WPDS conversion and prestar and poststar reachability queries.
Key
  A special kind of identifier used for uniquely identifying states and symbols. See [4].
WPDS
  Models weighted pushdown systems. See [4].
WFA
  Models weighted finite automata. See [4].
ref_ptr<T>
  A reference counting pointer class. See [4].

1.3   Construction

To construct an NWA, a constructor is invoked that creates either an empty NWA or a one-state NWA. Thereafter, method calls are performed to 1. add or remove states, 2. add or remove symbols, 3. add or remove transitions, 4. set the status of certain states as initial or final, and 5. combine component NWAs (via union, intersection, etc).

One complication arises due to stuck states. A stuck state is a state that can never be a final state and has no outgoing transitions other than to itself. The NWA class supports modeling NWAs with a designated stuck state; in such an NWA, there may be implicit transitions to this stuck state, whereas an NWA without a stuck state can hove only explicit transitions. The remainder of this document uses "stuck state" to refer to this explicitly-designated stuck state. This means that in an NWA without a stuck state there must exist a transition of each kind out of every state (out of every pair of states for return transitions) for every symbol. Therefore, we distinguish between two types of NWAs: type 1 - an NWA with a stuck state, and type 2 - an NWA without a stuck state. Each state in an NWA of type 2 must be complete; that is, each state must have outgoing transitions for each symbol of the alphabet. Operations that

add or remove individual states and individual transitions would cause state-completeness to be violated, and hence they are disallowed on NWAs of type 2. If such an operation is applied to an NWA of type 2, the operation causes an assertion violation. In NWAs of type 1, completeness is implicit: a given state has some number (possibly zero) of explicit outgoing transitions on some number of symbols; for all other symbols, for each of the three kinds of transitions, there are implicit transitions to the stuck state.

Important Note: Due to the requirement of the completeness of an NWA of type 2, states (except the stuck state), symbols, and transitions can only be added to an NWA of type 1.

NWAs can be created in multiple ways:

1. The basic constructor (`NWA()`) constructs an NWA of type 2 with no states, symbols, or transitions. An NWA created in this way cannot have states, symbols, or transitions added to it until it has a stuck state (which can be added using `setStuckState`).
2. If the constructor is supplied with a state (`NWA( Key stuckSt )`), it constructs an NWA of type 1 (having stuck state `stuckSt`) with $Q = \{\texttt{stuckSt}\}$, and no symbols or transitions.
3. The copy constructor constructs an NWA of the same type as the NWA passed in.

The type of an NWA can be checked (using `hasStuckState`) or changed (using `realizeImplicitTrans`, `clear` for changing an NWA of type 1 to an NWA of type 2, or `setStuckState` for changing an NWA of type 2 to an NWA of type 1) at any time after its construction.

The following operations are methods of class NWA:

`bool hasStuckState()`
  Tests whether the NWA is of type 1 (returns `true`) or type 2 (returns `false`).

`void setStuckState( Key stuckSt )`
  Allows the user to specify a stuck state, `stuckSt`, thereby making an NWA of type 2 into an NWA of type 1. The state specified must not already exist in the NWA (this guarantees that the stuck state will not be final or have any outgoing transitions). Furthermore, because a stuck state already exists in anNWA of type 1, if this method is invoked on an NWA of type 1 it triggers an assertion violation.

`void realizeImplicitTrans()`
  See Section 2.3. Note: Converts an NWA of type 1 into an NWA of type 2 having the same behavior.

`clear()`
  Removes all states (including the stuck state if there is one, so the resulting NWA will always be of type 2), symbols, and transitions from the NWA.

The only ways to remove a stuck state from an NWA are to 1. clear all states, symbols, and transitions from the NWA using `clear`, or 2. materialize all implicit transitions via `realizeImplicitTrans` (which changes the type of

the NWA to type 2), set a new stuck state (which changes the type of the NWA back to type 1), and remove the old stuck state (which is no longer the stuck state of the NWA).

1.4   Examples

As an example, consider the nested word automaton $A = (Q_A, \Sigma_A, Q_{0A}, \delta_A, F_A)$ where $Q_A = \{Start, Call, Entry, State, Exit, Return, End, Stuck\}$ (with $Stuck$ set as the stuck state), $\Sigma_A = \{a, call, b, ret\}$, $Q_{0A} = \{Start\}$, $F_A = \{End\}$, and

$$\delta_A = \begin{cases} \delta_i = \{(Start, a, Call), (Entry, b, State), (State, b, Exit), (Return, a, End)\}, \\ \delta_c = \{(Call, call, Entry)\}, \\ \delta_r = \{(Exit, Call, ret, Return)\}, \end{cases}$$

as seen in Figure 2. This NWA is of type 1, since there is a stuck state, $Stuck$, and some transitions are implicit (in fact, in this NWA, all transitions to the stuck state are implicit.
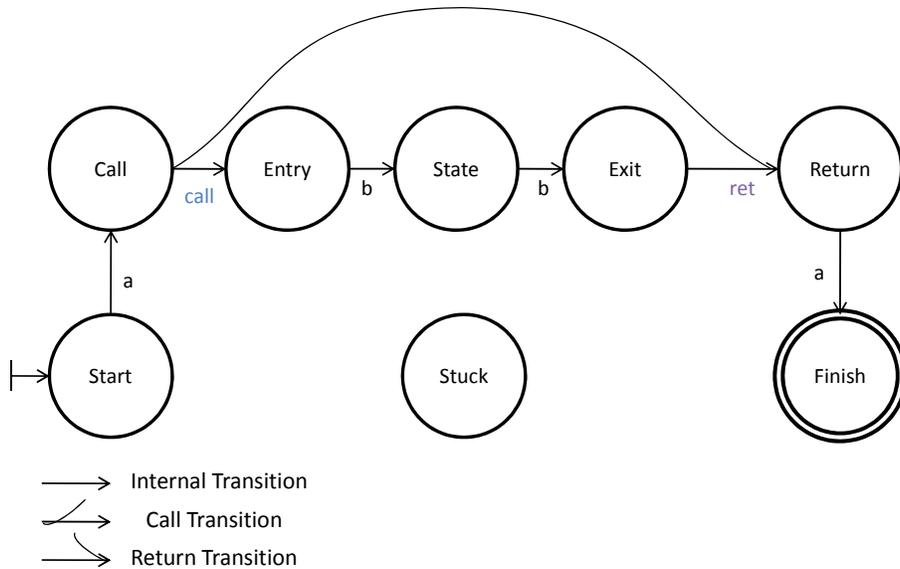

Fig. 2. An example of an NWA of type 1.

The equivalent nested word automaton of type 2 is shown in Figure 3. Here $Stuck$ is just an ordinary state. For ease of reading the lines connecting calls and returns are left out of this diagram. In reality every return transition to $Stuck$ in Figure 3 represents as many as 8 return transitions, one for each possible call state, i.e., the transitions $(Start, Start, a, Stuck)$, $(Start, Call, a, Stuck)$, $(Start, Entry, a, Stuck)$, $(Start, State, a, Stuck)$, $(Start, Exit, a, Stuck)$, $(Start, Return, a, Stuck)$, $(Start, Finish, a, Stuck)$, and

5

($Start, Stuck, a, Stuck$) are the return transitions from $Start$ to $Stuck$ on the symbol a. The number of transitions that each represents appears in parentheses following the symbol. Note: If `realizeImplicitTrans` is called on the NWA in Figure 2 the result is the NWA in Figure 3.
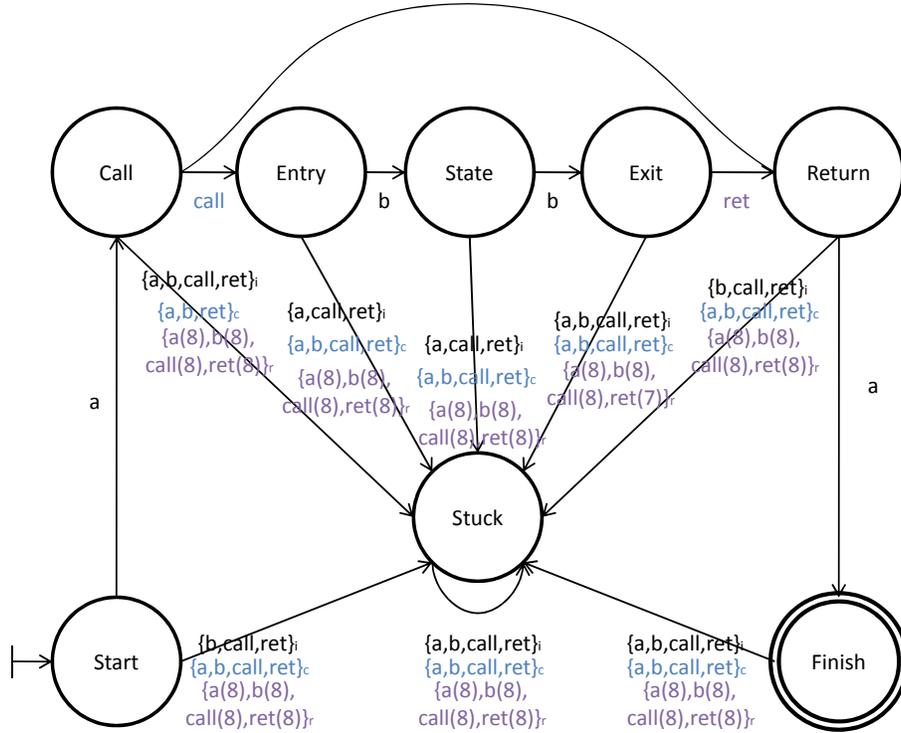

Fig. 3. An example of an NWA of type 2. Note: The edge from $Entry$ to $Stuck$ labeled $\{a, call, ret\}_i$ is a shorthand for three internal transitions in δ_i: ($Entry, a, Stuck$), ($Entry, call, Stuck$), and ($Entry, ret, Stuck$). Similarly, the edges labeled $\{\cdots\}_c$ are shorthand for call transitions and the edges labeled $\{\cdots\}_r$ are shorthand for return transitions.

2 Building NWAs from Scratch

2.1 States

The NWA has the possiblility for a special state called the stuck state (recall that an NWA of type 1 has a stuck state). The stuck state is a state that can never be a final state and has no outgoing transitions except to itself and, to reduce the size of the transition sets of the NWA, has implicit incoming

transitions. Any transition to the stuck state, i.e., any internal transition in which the target is the stuck state, any call transition in which the entry state is the stuck state, or any return transition in which the return state is the stuck state, need not be explicitly specified (this occurs only in an NWA of type 1). In addition, no transition can leave the stuck state unless it is going to the stuck state as well (see Section 2.3).

The following operations are methods of class NWA:

`Key getStuckState()`
> Allows the user to access the `Key` for the stuck state. This method should not be called on an NWA of type 2 because there is no stuck state to retrieve; otherwise there is an assertion violation.

`bool isStuckState( Key st )`
> Tests whether the given `Key`, `st`, is the stuck state's `Key`. If yes, returns true. Otherwise, returns false. If called on an NWA of type 2, returns false.

States are WALi `Keys` and can be added to an NWA of type 1 individually (using `addState`, `addInitialState`, or `addFinalState`) or as a part of a transition (see Section 2.3). Note that in NWAs of type 1, the addition of an individual state automatically introduces an implicit internal transition to the stuck state for each symbol (except meta-symbols[3]) in the NWA; an implicit call transition to the stuck state for each symbol (except meta-symbols) in the NWA; and two return transitions to the stuck state for each state/symbol pair (except meta-symbols) pair in the NWA, one in which the state added plays the role of the exit state and one in which the state added plays the role of the call state. Because NWAs of type 2 have no stuck state, such transitions cannot be added automatically and thus adding a state would violate the completeness of the states of the NWA. Therefore, **no states can be added to an NWA of type 2**.

The following operations are methods of class NWA:

`bool addState( Key st )`
> Adds the given state, `st`, to Q. Returns false if state `st` already exists in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool addInitialState( Key st )`
> Adds the given state, `st`, to Q_0 (and also to Q if `st` $\notin Q$). Returns false if state `st` already exists in Q_0, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool addFinalState( Key st )`
> Adds the given state, `st`, to Q_f (and also to Q if `st` $\notin Q$). Returns false if state `st` already exists in Q_f, otherwise returns true. If `st` is the NWA's stuck state, then the NWA is automatically transformed to one of type 2 (by materializing all the implicit transitions) prior to making `st` a final

---

[3] meta-symbols will be described in greater detail in Section 2.2

state. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

States can be removed from an NWA individually (using `removeState`) or by clearing the state set of the NWA. Note that in NWAs of type 1, the removal of an individual state automatically eliminates all transitions for each internal transition that has the state being removed as the target, each call transition that has the state being removed as the entry state, and each return transition that has the state being removed as the return state or call state. Transitions that are removed are redirected to the stuck state as implicit transitions. Because NWAs of type 2 have no stuck state, such transitions cannot be added automatically and thus removing a state would violate the completeness of the states of the NWA. Therefore, **no states can be removed from an NWA of type 2**.

Initial (final) states can be removed from Q_0 (Q_f) (but not from Q) individually (using `removeInitialState`, `removeFinalState`) or by clearing the initial (final) state set of the NWA. These operations can be performed on NWAs of both types.

The following operations are methods of class NWA:

`bool removeState( Key st )`
  Removes the given state, `st`, from Q (and also removes `st` from Q_0 if `st` $\in Q_0$ and removes `st` from Q_f if `st` $\in Q_f$). Returns false if state `st` did not exist in the NWA, otherwise returns true. This method cannot remove the stuck state from the NWA, because that would create implicit transitions to nowhere. Any transitions (internal, call, and return) associated with `st` are also removed from the NWA. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`void clearStates()`
  Removes all states (including the stuck state) from Q, Q_0, and Q_f. As a side effect it also removes all transitions in the NWA. The symbols in the NWA are not affected.

`bool removeInitialState( Key st )`
  Removes the given state, `st`, from Q_0 (but not from Q or Q_f). Returns false if state `st` did not exist in Q_0, otherwise returns true. Nothing else in the NWA is altered.

`void clearInitialStates()`
  Removes all states from Q_0 (but not from Q or Q_f). Nothing else in the NWA is altered.

`bool removeFinalState( Key st )`
  Removes the given state, `st`, from Q_f (but not from Q or Q_0). Returns false if state `st` did not exist in Q_f, otherwise returns true. Nothing else in the NWA is altered.

```
void clearFinalStates()
```
Removes all states from Q_f (but not from Q or Q_0). Nothing else in the NWA is altered.

The status of individual states (i.e., existence in an NWA, in the initial state set, or in the final state set) can be checked for an NWA of either type (using isState/is_nwa_state, isInitialState, and isFinalState). In addition, states, initial states, and final states can be requested and examined as collections (using getStates/get_states, getInitialStates, getFinalStates, sizeStates/num_nwa_states, sizeInitialStates, and sizeFinalStates).

The following operations are methods of class NWA:

```
bool isState( Key st ) const
```
Tests whether the given Key, st, is a state in the NWA. If yes, returns true. Otherwise, returns false.
```
bool isInitialState( Key st ) const
```
Tests whether the given Key, st, is an initial state in the NWA. If yes, returns true. Otherwise, returns false.
```
bool isFinalState( Key st ) const
```
Tests whether the given Key, st, is a final state in the NWA. If yes, returns true. Otherwise, returns false.
```
const set<Key>& getStates() const
```
Returns a set consisting of the Keys of all states in the NWA (including the stuck state).
```
const set<Key>& getInitialStates() const
```
Returns a set consisting of the Keys of all initial states in the NWA.
```
const set<Key>& getFinalStates() const
```
Returns a set consisting of the Keys of all final states in the NWA.
```
size_t sizeStates() const
```
Yields the number of states in the NWA. If the NWA is of type 1, this count includes the stuck state.
```
size_t sizeInitialStates() const
```
Yields the number of initial states in the NWA.
```
size_t sizeFinalStates() const
```
Yields the number of final states in the NWA.

Each state in the NWA can be associated with some client-specific information. To utilize this functionality, the user must create a subclass of the ClientInfo class (making sure to supply a copy constructor for the subclass). In addition, the helper methods for intersection and determinization in the NWA class (intersectClientInfoInternal, mergeClientInfoInternal, etc.) must be over-ridden (see Section 3.2 and Section 3.6). Client information can then be associated with each state in the NWA (using setClientInfo) and accessed given the associated state (using getClientInfo). For details on automatic

generation of client information in NWA operations, see Section 3.

The following operations are methods of class NWA:

`ref_ptr<ClientInfo> getClientInfo( Key st ) const`
    Allows the user to access the client-specific information associated with the
    given state, `st`.

`void setClientInfo( Key st, const ref_ptr<ClientInfo> ci )`
    Allows the user to specify the client-specific information, `ci`, associated
    with the given state, `st`.

2.2 Symbols

The system supports two meta-symbols: 1. `Epsilon` (ϵ) and 2. `Wild` (`*`) which
are globally available. `Epsilon` is the 'absence' of a symbol, it denotes the
situation in which a transition can be traversed without matching and consum-
ing an input symbol. `Epsilon` symbols cannot label call or return transitions
(see Section 2.3). `Wild` is the 'any' symbol, it denotes the situation in which a
transition can be traversed by consuming a single input symbol of any kind.
Since the NWA alphabet is not fixed, the actual symbols that `Wild` stands for is
fluid. Note: `Epsilon` **and** `Wild` **are not elements of** Σ**.** Let $\Sigma^* = \Sigma \cup \{*, \epsilon\}$.

The following operations are methods of class NWA:

`Key getEpsilon()`
    Allows the user to access the `Key` for the `Epsilon` meta-symbol.

`bool isEpsilon( Key sym )`
    Tests whether the given `Key`, `sym`, is the `Key` for the `Epsilon` meta-symbol.
    If yes, returns true. Otherwise, returns false.

`Key getWild()`
    Allows the user to access the `Key` for the `Wild` meta-symbol.

`bool isWild( Key sym )`
    Tests whether the given `Key`, `sym`, is the `Key` for the `Wild` meta-symbol. If
    yes, returns true. Otherwise, returns false.

Symbols are WALi `Keys` and can be added to an NWA of type 1 individually
(using `addSymbol`) or as a part of a transition (see Section 2.3). Note that in
NWAs of type 1, the addition of a symbol adds an implicit internal transition
from each state to the stuck state on the symbol being added, an implicit call
transition from each state to the stuck state on the symbol being added, and
an implicit return transition from each pair of states to the stuck state on
the symbol being added. Because NWAs of type 2 have no stuck state, such
transitions cannot be added automatically and thus adding a symbol would
violate the completeness of the states of the NWA. Therefore, **no symbols can
be added to an NWA of type 2**.

The following operations are methods of class NWA:

```
bool addSymbol( Key sym )
```
Adds the given symbol, `sym`, to Σ. Returns false if symbol `sym` already exists in the NWA, otherwise returns true. Note: `Wild` and `Epsilon` are meta-symbol, not symbols of the alphabet, so if `sym` is the `Key` for either of these a failure code of false is returned. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

Symbols can be removed from an NWA of either type individually (using `removeSymbol`) or by clearing the alphabet of the NWA (using `clearSymbols`). Note that the removal of a symbol removes all internal, call, and return transitions having that symbol (both implicit and explicit) from each state, thus preserving the completeness of the states of the NWA.

The following operations are methods of class NWA:

```
bool removeSymbol( Key sym )
```
Removes the given symbol, `sym`, from Σ. Returns false if symbol `sym` did not already exist in the NWA, otherwise returns true. Any transitions associated with this symbol are also removed, but no states are altered. Note: `Wild` and `Epsilon` are meta-symbols, not symbols of the alphabet, so if `sym` is the `Key` for either of these a failure code of false is returned and no transitions are removed.

```
void clearSymbols()
```
Removes all symbols from Σ. As a side effect it also removes all transitions in the NWA. The state set is unaffected.

The status of individual symbols (i.e., existence in an NWA) can be checked for an NWA of either type (using `isSymbol`). In addition, symbols can be requested and examined as a collection (using `getSymbols` and `sizeSymbols`).

The following operations are methods of class NWA:

```
bool isSymbol( Key sym ) const
```
Tests whether the given `Key`, `sym`, is a symbol in the NWA. If yes, returns true. Otherwise, returns false. Note: `Wild` and `Epsilon` are meta-symbols, not symbols of the alphabet, so if `sym` is the `Key` for either of these, false is returned.

```
const set<Key>& getSymbols() const
```
Returns a `set` consisting of the `Keys` of all symbols in the NWA. Note this does not include meta-symbols.

```
size_t sizeSymbols() const
```
Yields the number of symbols in the NWA. This count does not include the meta-symbols ϵ and `*` even if they have been used in transitions added to the NWA.

2.3 Transitions

Transitions connect a pair (or triple) of states and are labeled with symbols or meta-symbols. There are three types of transitions in an NWA: 1. Internal 2. Call 3. Return. An internal transition is a transition of the form $(source, sym, target)$ in which sym corresponds to the symbol at an internal position in a nested word accepted by the NWA. A call transition is a transition of the form $(callSite, sym, entryPoint)$ in which sym corresponds to the symbol at a call position in a nested word accepted by the NWA. A return transition is a transition of the form $(exitPoint, callSite, sym, returnSite)$ in which sym corresponds to the symbol at a return position in a nested word accepted by the NWA. The number of transitions of each kind associated with a deterministic NWA is bounded by the number of states and symbols in the NWA. There can only be $(number\_of\_states) * (number\_of\_symbols)$ internal transitions, $(number\_of\_states) * (number\_of\_symbols)$ call transitions, and $(number\_of\_states)^2 * (number\_of\_symbols)$ return transitions. The number of transitions of each kind associated with a nondeterministic NWA is bounded in a slightly different way by the number of states and symbols in the NWA. There can only be $(number\_of\_states)^2 * (number\_of\_symbols)$ internal transitions, $(number\_of\_states)^2 * (number\_of\_symbols)$ call transitions, and $(number\_of\_states)^3 * (number\_of\_symbols)$ return transitions.

Transitions can be added to an NWA of type 1 via the overloaded functions `addInternalTrans`, `addCallTrans`, and `addReturnTrans`. Those functions can either take three `Keys` that define the transition as input, or an input of type `KeyTriple&` or `KeyQuad&`. `KeyTriple` and `KeyQuad` are WALi data structures containing 3 and 4 `Keys`, respectively. Alternatively, the set of transitions of a state (or the set of outgoing transitions of a state) can be duplicated for another state (using `duplicateState` or `duplicateStateOutgoing`) or all implicit transitions (to the stuck state) can be added as explicit transitions (using `realizeImplicitTrans`). As a side effect of all of these functions, if any component (state or symbol) of a transition does not already exist in the NWA it will be added to the NWA. Thus states and symbols need not be added independently prior to adding transitions. Note that, in NWAs of type 1, the addition of a state or symbol adds to the NWA implicit transitions to the stuck state (see Section 2.1 and Section 2.2). Because NWAs of type 2 have no stuck state, such transitions cannot be added automatically, and thus adding a transition would violate the completeness of the states in the NWA. Therefore, **no transitions can be added to an NWA of type 2**. Recall that the stuck state has no outgoing transitions except to itself (see Section 2.1). Thus, no transition leaving the stuck state will be added to the NWA unless it is going to the stuck state as well. Recall also that the ϵ symbol cannot label a call transition or a return transition (see Section 2.2). Thus, no call or return transition with a symbol of ϵ will be added.

The following operations are methods of class NWA:

```
bool addInternalTrans( Key source, Key sym, Key target )
```
Adds the internal transition $(\mathtt{source}, \mathtt{sym}, \mathtt{target})$ to the NWA. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, `source` and `target` are added as states (if $\mathtt{source} \notin Q$ and $\mathtt{target} \notin Q$, respectively) and `sym` is added as a symbol (if $\mathtt{sym} \notin \Sigma^*$). This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

```
bool addInternalTrans( KeyTriple& intTrans )
```
Adds an internal transition to the NWA for the triple specified by `intTrans`. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, the states of `intTrans` are added as states of the NWA (if not already states in the NWA) and the symbol of `intTrans` is added as a symbol of the NWA (if not a meta-symbol or already a symbol in the NWA). This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

```
bool addCallTrans( Key callSite, Key sym, Key entryPoint )
```
Adds the call transition $(\mathtt{callSite}, \mathtt{sym}, \mathtt{entryPoint})$ to the NWA. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, `callSite` and `entryPoint` are added as states of the NWA (if $\mathtt{callSite} \notin Q$ and $\mathtt{entryPoint} \notin Q$, respectively) and `sym` is added as a symbol of the NWA (if $\mathtt{sym} \notin \Sigma^*$). This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

```
bool addCallTrans( KeyTriple& callTrans )
```
Adds a call transition to the NWA for the triple specified by `callTrans`. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, the states of `callTrans` are added as states of the NWA (if not already states in the NWA) and the symbol of `callTrans` is added as a symbol of the NWA (if not a meta-symbol or already a symbol in the NWA). This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

```
bool addReturnTrans( Key exitPoint, Key callSite,
                     Key sym, Key returnSite )
```
Adds the return transition $(\mathtt{exitPoint}, \mathtt{callSite}, \mathtt{sym}, \mathtt{returnSite})$ to the NWA. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, `exitPoint`, `callSite`, and `returnSite` are added as states of the NWA (if $\mathtt{exitPoint} \notin Q$, $\mathtt{callSite} \notin Q$, and $\mathtt{returnSite} \notin Q$, respectively) and `sym` is added as a symbol (if $\mathtt{sym} \notin \Sigma^*$). This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

```
bool addReturnTrans( KeyQuad& retTrans )
```
Adds a return transition to the NWA for the 4-tuple specified by `retTrans`. Returns false if this transition already exists in the NWA, otherwise returns true. In addition, the states of `retTrans` are added as states (if not already states in the NWA) and the symbol of `retTrans` is added as a symbol (if

not a meta-symbol or already a symbol in the NWA). This method should
not be called on an NWA of type 2, otherwise there is an assertion violation.

void duplicateState(Key orig, Key dup)

Duplicates all the transitions containing the state `orig` and adds the tran-
sitions to the NWA with `orig` replaced by `dup`. Self-loops are duplicated
by adding a transition for all possible combinations of some occurrence of
`orig` replaced by `dup`, i.e., if $(\text{orig}, a, \text{orig})$ is an internal (or call) tran-
sition, then the transitions $(\text{dup}, a, \text{dup})$, $(\text{dup}, a, \text{orig})$, and $(\text{orig}, a, \text{dup})$
are added and if $(\text{orig}, \text{orig}, a, \text{orig})$ is a return transition, then the
transitions $(\text{dup}, \text{dup}, a, \text{dup})$, $(\text{dup}, \text{dup}, a, \text{orig})$, $(\text{orig}, \text{dup}, a, \text{dup})$,
$(\text{orig}, \text{dup}, a, \text{orig})$, $(\text{dup}, \text{orig}, a, \text{dup})$, $(\text{dup}, \text{orig}, a, \text{orig})$, and
$(\text{orig}, \text{orig}, a, \text{dup})$ are added. This method should not be called on
an NWA of type 2, otherwise there is an assertion violation.

void duplicateStateOutgoing(Key orig, Key dup)

Duplicates all the transitions originating from `orig` and adds the transitions
to the NWA with `orig` replaced by `dup`. Self-loops are duplicated by adding
a transition for all possible combinations of some occurrence of `orig` re-
placed by `dup` while maintaining that the transitions are outgoing from `dup`,
i.e., if $(\text{orig}, a, \text{orig})$ is an internal (or call) transition, then the transitions
$(\text{dup}, a, \text{dup})$ and $(\text{dup}, a, \text{orig})$ are added and if $(\text{orig}, \text{orig}, a, \text{orig})$ is a
return transition, then the transitions $(\text{dup}, \text{dup}, a, \text{dup})$, $(\text{dup}, \text{dup}, a, \text{orig})$,
$(\text{dup}, \text{orig}, a, \text{dup})$, and $(\text{dup}, \text{orig}, a, \text{orig})$ are added. This method should
not be called on an NWA of type 2, otherwise there is an assertion violation.

void realizeImplicitTrans()

Adds all implicit transitions explicitly to the stuck state. This should only
be called on an NWA of type 1, and it changes the NWA to an NWA of
type 2. Otherwise there is an assertion violation.

Transitions can be removed from an NWA of type 1 individually or by
specifying component parts (using `removeInternalTrans`, `removeCallTrans`,
or `removeReturnTrans`), by clearing all transitions from the NWA, or as a
side effect of removing a state or symbol from the NWA (see Section 2.1 and
Section 2.2). Note that, in NWAs of type 1, the removal of a transition will
add an implicit transition with the same source/call-state/exit- and call- states,
the same symbol, and the stuck state as the target/entry-state/return-state.
Because NWAs of type 2 have no stuck state, such transitions cannot be added
automatically, and thus removing a transition would violate the completeness
of the states in the NWA. Therefore, **no transitions can be removed from
an NWA of type 2**.

The following operations are methods of class NWA:

bool removeInternalTrans(Key source, Key sym, Key target)

Removes the internal transition $(\text{source}, \text{sym}, \text{target})$ from the NWA (if
$(\text{source}, \text{sym}, \text{target}) \in \delta_i$). Returns false if this transition did not exist in

the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeInternalTrans( KeyTriple& intTrans )`

Removes an internal transition from the NWA for the triple specified by `intTrans` (if `intTrans` $\in \delta_i$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeCallTrans( Key callSite, Key sym, Key entryPoint )`

Removes the call transition ($\text{callSite}, \text{sym}, \text{entryPoint}$) from the NWA (if ($\text{callSite}, \text{sym}, \text{entryPoint}$) $\in \delta_c$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeCallTrans( KeyTriple& callTrans )`

Removes a call transition from the NWA for the triple specified by `callTrans` (if `callTrans` $\in \delta_c$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeReturnTrans( Key exitPoint, Key callSite,`
`                        Key sym, Key returnSite )`

Removes the return transition ($\text{exitPoint}, \text{callSite}, \text{sym}, \text{returnSite}$) from the NWA (if ($\text{exitPoint}, \text{callSite}, \text{sym}, \text{returnSite}$) $\in \delta_r$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeReturnTrans( Key exitPoint, Key sym, Key returnSite )`

Removes the return transitions ($\text{exitPoint}, st, \text{sym}, \text{returnSite}$) for all $st \in Q$ from the NWA (if ($\text{exitPoint}, st, \text{sym}, \text{returnSite}$) $\in \delta_r$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`bool removeReturnTrans( KeyQuad& retTrans )`

Removes a return transition from the NWA for the quadruple specified by `retTrans` (if `retTrans` $\in \delta_r$). Returns false if this transition did not exist in the NWA, otherwise returns true. This method should not be called on an NWA of type 2, otherwise there is an assertion violation.

`void clearTrans()`

Removes all transitions from the NWA, but does not remove any states or symbols.

The set of supported queries about an NWA's explicit transitions includes: the existence of a specified transition, the number of (internal, call, return, or total) transitions in the NWA, the set of states that are predecessors (sources of internal transitions, call states of call transitions, or exit states of return transitions) of a given state by some transition, the set of states that are call-predecessors of a given state by some return transition, the set of states that

are successors (targets of internal transitions, entry states of call transitions, or return states of return transitions) of a given state by some transition, the set of states that are successors of a given call-predecessor (return states) by some return transition, the set of symbol/state pairs for which there exist internal transitions with a particular source, the set of symbol/state pairs for which there exist call transitions with a particular call state, the set of symbol/state pairs for which there exist return transitions with a particular exit state and call state, the set of sources that are associated with a given target, the set of targets that are associated with a given source, the set of call states that are associated with a given entry state, the set of entry states that are associated with a given call state, the set of exit states that are associated with a given call state, the set of exit states that are associated with a given return state, the set of call states that are associated with a given exit state, the set of call states that are associated with a given return state, the set of return states that are associated with a given exit state, the set of return states that are associated with a given call state, etc. Many of these functions are available in a version that restricts the symbol.

The following operations are methods of class NWA:

`bool findTrans( Key from, Key sym, Key to )`
    Tests whether there exists an internal transition $(\texttt{from}, \texttt{sym}, \texttt{to}) \in \delta_i$, a call transition $(\texttt{from}, \texttt{sym}, \texttt{to}) \in \delta_c$, or a return transition $(\texttt{from}, st, \texttt{sym}, \texttt{to}) \in \delta_r$ for some $st \in Q$.

`bool getSymbol( Key from, Key to, Key& sym )`
    Tests whether there exists an internal transition $(\texttt{from}, \alpha, \texttt{to}) \in \delta_i$ for some $\alpha \in \Sigma^*$, a call transition $(\texttt{from}, \alpha, \texttt{to}) \in \delta_c$ for some $\alpha \in \Sigma \cup \{*\}$, or a return transition $(\texttt{from}, st, \alpha, \texttt{to}) \in \delta_r$ for some $st \in Q$ and some $\alpha \in \Sigma \cup \{*\}$. In addition, the α for the first such transition found is stored in the reference parameter `sym`.

`size_t sizeInternalTrans()`
    Yields the number of explicit internal transitions in the NWA.

`size_t sizeCallTrans()`
    Yields the number of explicit call transitions in the NWA.

`size_t sizeReturnTrans()`
    Yields the number of explicit return transitions in the NWA.

`size_t sizeTrans()`
    Yields the total number of explicit transitions (internal, call, and return) in the NWA.

`const std::set<Key> getPredecessors( Key state )`
    Yields all states that are predecessors of a given state by some transition; i.e., for each internal transition, $(source, sym, \texttt{state})$, $source$ is added to the set of predecessors, for each call transition, $(callSite, sym, \texttt{state})$, $callSite$ is added to the set of predecessors, and for each return transition, $(exitPoint, callSite, sym, \texttt{state})$, $exitPoint$ is added to the set of predecessors.

```
const std::set<Key> getPredecessors( Key symbol, Key state )
```
Yields all states that are predecessors of a given state by some transition having the given symbol; i.e., for each internal transition, $(source, \texttt{symbol}, \texttt{state})$, $source$ is added to the set of predecessors, for each call transition, $(callSite, \texttt{symbol}, \texttt{state})$, $callSite$ is added to the set of predecessors, and for each return transition, $(exitPoint, callSite, \texttt{symbol}, \texttt{state})$, $exitPoint$ is added to the set of predecessors.

```
const std::set<Key> getSuccessors( Key state )
```
Yields all states that are successors of a given state by some transition; i.e., for each internal transition, $(\texttt{state}, sym, target)$, $target$ is added to the set of successors, for each call transition, $(\texttt{state}, sym, entryPoint)$, $entryPoint$ is added to the set of successors, and for each return transition, $(\texttt{state}, callSite, sym, returnSite)$, $returnSite$ is added to the set of successors.

```
const std::set<Key> getSuccessors( Key state, Key symbol )
```
Yields all states that are successors of a given state by some transition having the given symbol; i.e., for each internal transition, $(\texttt{state}, \texttt{symbol}, target)$, $target$ is added to the set of successors, for each call transition, $(\texttt{state}, \texttt{symbol}, entryPoint)$, $entryPoint$ is added to the set of successors, and for each return transition, $(\texttt{state}, callSite, \texttt{symbol}, returnSite)$, $returnSite$ is added to the set of successors.

```
const std::set<Key> getCallPredecessors( Key state )
```
Yields all states that are call-predecessors of a given state by some return transition; i.e., for each return transition, $(exitPoint, callSite, sym, \texttt{state})$, $callSite$ is added to the set of call predecessors.

```
const std::set<Key> getCallPredecessors( Key symbol, Key state )
```
Yields all states that are call-predecessors of a given state by some return transition having the given symbol; i.e., for each return transition, $(exitPoint, callSite, \texttt{symbol}, \texttt{state})$, $callSite$ is added to the set of call predecessors.

```
const std::set<Key> getCallSuccessors( Key state )
```
Yields all states that are call-successors of a given state by some return transition; i.e., for each return transition, $(exitPoint, \texttt{state}, sym, returnSite)$, $returnSite$ is added to the set of call successors.

```
const std::set<Key> getCallSuccessors( Key state, Key symbol )
```
Yields all states that are call-successors of a given state by some return transition having the given symbol; i.e., for each return transition, $(exitPoint, \texttt{state}, \texttt{symbol}, returnSite)$, $returnSite$ is added to the set of call successors.

```
const std::set<Key> getSources_Sym( Key symbol )
```
Yields the set of all states, $source$, such that the internal transition $(source, \texttt{symbol}, target)$ is in δ_i for some $target \in Q$.

```
const std::set<Key> getSources( Key symbol, Key target )
```
Yields the set of all states, *source*, such that the internal transition $(source, \mathtt{symbol}, \mathtt{target})$ is in δ_i.

```
const std::set<std::pair<Key,Key>> getSources( Key target )
```
Yields the set of all state/symbol pairs, $(source, sym)$, such that the internal transition $(source, sym, \mathtt{target})$ is in δ_i.

```
const std::set<Key> getSources( )
```
Yields the set of all states, *source*, such that the internal transition $(source, symbol, target)$ is in δ_i for some $symbol \in \Sigma$ and $target \in Q$.

```
const std::set<Key> getTargets_Sym( Key symbol )
```
Yields the set of all states, *target*, such that the internal transition $(source, \mathtt{symbol}, target)$ is in δ_i for some $source \in Q$.

```
const std::set<Key> getTargets( Key source, Key symbol )
```
Yields the set of all states, *target*, such that the internal transition $(\mathtt{source}, \mathtt{symbol}, target)$ is in δ_i.

```
const std::set<std::pair<Key,Key>> getTargets( Key source )
```
Yields the set of all symbol/state pairs, $(sym, target)$, such that the internal transition $(\mathtt{source}, sym, target)$ is in δ_i.

```
const std::set<Key> getTargets( )
```
Yields the set of all states, *target*, such that the internal transition $(source, symbol, target)$ is in δ_i for some $source \in Q$ and $symbol \in \Sigma$.

```
const std::set<Key> getInternalSym( )
```
Yields the set of all symbols, *symbol*, such that the internal transition $(source, symbol, target)$ is in δ_i for some $source \in Q$ and $target \in Q$.

```
const std::set<Key> getInternalSym( Key source, Key target )
```
Yields the set of all symbols, *symbol*, such that the internal transition $(\mathtt{source}, symbol, \mathtt{target})$ is in δ_i.

```
const std::set<Key> getInternalSym_Source( Key source )
```
Yields the set of all symbols, *symbol*, such that the internal transition $(\mathtt{source}, symbol, target)$ is in δ_i for some $target \in Q$.

```
const std::set<Key> getInternalSym_Target( Key target )
```
Yields the set of all symbols, *symbol*, such that the internal transition $(source, symbol, \mathtt{target})$ is in δ_i for some $source \in Q$.

```
const std::set<Key> getCallSites_Sym( Key symbol )
```
Yields the set of all states, *callSite*, such that the call transition $(callSite, \mathtt{symbol}, entryPoint)$ is in δ_c for some $entryPoint \in Q$.

```
const std::set<Key> getCallSites( Key symbol, Key entryPoint )
```
Yields the set of all states, *callSite*, such that the call transition $(callSite, \mathtt{symbol}, \mathtt{entryPoint})$ is in δ_c.

```
const std::set<std::pair<Key,Key>> getCallSites( Key entryPoint )
```
Yields the set of all state/symbol pairs, $(callSite, sym)$, such that the call transition $(callSite, sym, \mathtt{entryPoint})$ is in δ_c.

```
const std::set<Key> getCallSites( )
```
Yields the set of all states, *callSite*, such that $(callSite, symbol, entryPoint)$ is in δ_c for some $symbol \in \Sigma$ and $entryPoint \in Q$.

```
const std::set<Key> getEntries_Sym( Key symbol )
```
Yields the set of all states, $entryPoint$, such that $(callSite, \mathsf{symbol}, entryPoint)$ is in δ_c for some $callSite \in Q$.

```
const std::set<Key> getEntries( Key callSite, Key symbol )
```
Yields the set of all states, $entryPoint$, such that the call transition $(\mathsf{callSite}, \mathsf{symbol}, entryPoint)$ is in δ_c.

```
const std::set<std::pair<Key,Key>> getEntries( Key callSite )
```
Yields the set of all symbol/state pairs, $(sym, entryPoint)$, such that the call transition $(\mathsf{callSite}, sym, entryPoint)$ is in δ_c.

```
const std::set<Key> getEntries( )
```
Yields the set of all states, $entryPoint$, such that the call transition $(callSite, symbol, entryPoint)$ is in δ_c for some $callSite \in Q$ and $symbol \in \Sigma$.

```
const std::set<Key> getCallSym( )
```
Yields the set of all symbols, $symbol$, such that the call transition $(callSite, symbol, entryPoint)$ is in δ_c for some $callSite \in Q$ and $entryPoint \in Q$.

```
const std::set<Key> getCallSym( Key callSite, Key entryPoint )
```
Yields the set of all symbols, $symbol$, such that the call transition $(\mathsf{callSite}, symbol, \mathsf{entryPoint})$ is in δ_c.

```
const std::set<Key> getCallSym_Call( Key callSite )
```
Yields the set of all symbols, $symbol$, such that the call transition $(\mathsf{callSite}, symbol, entryPoint)$ is in δ_c for some $entryPoint \in Q$.

```
const std::set<Key> getCallSym_Entry( Key entryPoint )
```
Yields the set of all symbols, $symbol$, such that the call transition $(callSite, symbol, \mathsf{entryPoint})$ is in δ_c for some $callSite \in Q$.

```
const std::set<Key> getExits_Sym( Key symbol )
```
Yields the set of all states, $exitPoint$, such that the return transition $(exitPoint, callSite, \mathsf{symbol}, returnSite)$ is in δ_r for some $callSite \in Q$ and $returnSite \in Q$.

```
const std::set<Key> getExits( Key callSite, Key symbol,
                             Key returnSite )
```
Yields the set of all states, $exitPoint$, such that the return transition $(exitPoint, \mathsf{callSite}, \mathsf{symbol}, \mathsf{returnSite})$ is in δ_r.

```
const std::set<std::pair<Key,Key>> getExits( Key callSite,
                             Key returnSite )
```
Yields the set of all state/symbol pairs, $(exitPoint, sym)$, such that the return transition $(exitPoint, \mathsf{callSite}, sym, \mathsf{returnSite})$ is in δ_r.

```
const std::set<Key> getExits( )
```
Yields the set of all states, $exitPoint$, such that the return transition $(exitPoint, callSite, symbol, returnSite)$ is in δ_r for some $callSite \in Q$, $symbol \in \Sigma$, and $returnSite \in Q$.

```
const std::set<Key> getExits_Call( Key callSite, Key symbol )
```
Yields the set of all states, $exitPoint$, such that the return transition $(exitPoint, \mathsf{callSite}, \mathsf{symbol}, returnSite)$ is in δ_r for some $returnSite \in Q$.

```
const std::set<std::pair<Key,Key>> getExits_Call( Key callSite )
```
Yields the set of all state/symbol pairs, $(exitPoint, sym)$, such that the return transition $(exitPoint, \texttt{callSite}, sym, returnSite)$ is in δ_r for some $returnSite \in Q$.

```
const std::set<Key> getExits_Ret( Key symbol, Key returnSite )
```
Yields the set of all states, $exitPoint$, such that the return transition $(exitPoint, callSite, \texttt{symbol}, \texttt{returnSite})$ is in δ_r for some $callSite \in Q$.

```
const std::set<std::pair<Key,Key>> getExits_Ret( Key returnSite )
```
Yields the set of all state/symbol pairs, $(exitPoint, sym)$, such that the return transition $(exitPoint, callSite, sym, \texttt{returnSite})$ is in δ_r for some $callSite \in Q$.

```
const std::set<Key> getCalls_Sym( Key symbol )
```
Yields the set of all states, $callSite$, such that the return transition $(exitPoint, callSite, \texttt{symbol}, returnSite)$ is in δ_r for some $exitPoint \in Q$ and $returnSite \in Q$.

```
const std::set<Key> getCalls( Key exitPoint, Key symbol,
                       Key returnSite )
```
Yields the set of all states, $callSite$, such that the return transition $(\texttt{exitPoint}, callSite, \texttt{symbol}, \texttt{returnSite})$ is in δ_r.

```
const std::set<std::pair<Key,Key>> getCalls( Key exitPoint,
                       Key returnSite )
```
Yields the set of all state/symbol pairs, $(callSite, sym)$, such that the return transition $(\texttt{exitPoint}, callSite, sym, \texttt{returnSite}$ is in δ_r.

```
const std::set<Key> getCalls( )
```
Yields the set of all states, $callSite$, such that the return transition $(exitPoint, callSite, symbol, returnSite)$ is in δ_r for some $exitPoint \in Q$, $symbol \in \Sigma$, and $returnSite \in Q$.

```
const std::set<Key> getCalls_Exit( Key exitPoint, Key symbol )
```
Yields the set of all states, $callSite$, such that the return transition $(\texttt{exitPoint}, callSite, \texttt{symbol}, returnSite)$ is in δ_r for some $returnSite \in Q$.

```
const std::set<std::pair<Key,Key>> getCalls_Exit( Key exitPoint )
```
Yields the set of all state/symbol pairs, $(callSite, sym)$, such that the return transition $(\texttt{exitPoint}, callSite, sym, returnSite)$ is in δ_r for some $returnSite \in Q$.

```
const std::set<Key> getCalls_Ret( Key symbol, Key returnSite )
```
Yields the set of all states, $callSite$, such that the return transition $(exitPoint, callSite, \texttt{symbol}, \texttt{returnSite}$ is in δ_r for some $exitPoint \in Q$.

```
const std::set<std::pair<Key,Key>> getCalls_Ret( Key returnSite )
```
Yields the set of all state/symbol pairs, $(callSite, sym)$, such that the return transition $(exitPoint, callSite, sym, \texttt{returnSite}$ is in δ_r for some $exitPoint \in Q$.

```
const std::set<Key> getReturns_Sym( Key symbol )
```
Yields the set of all states, $returnSite$, such that the return transition $(exitPoint, callSite, \texttt{symbol}, returnSite)$ is in δ_r for some $exitPoint \in Q$ and $callSite \in Q$.

```
const std::set<Key> getReturns( Key exitPoint ,Key callSite,
                        Key symbol )
```
Yields the set of all states, $returnSite$, such that the return transition $(\texttt{exitPoint}, \texttt{callSite}, symbol, returnSite)$ is in δ_r.

```
const std::set<std::pair<Key,Key>> getReturns( Key exitPoint,
                        Key callSite )
```
Yields the set of all symbol/state pairs, $(sym, returnSite)$, such that the return transition $(\texttt{exitPoint}, \texttt{callSite}, sym, returnSite)$ is in δ_r.

```
const std::set<Key> getReturns( )
```
Yields the set of all states, $returnSite$, such that the return transition $(exitPoint, callSite, symbol, returnSite)$ is in δ_r for some $exitPoint \in Q$, $callSite \in Q$, and $symbol \in \Sigma$.

```
const std::set<Key> getReturns_Exit( Key exitPoint, Key symbol )
```
Yields the set of all states, $returnSite$, such that the return transition $(\texttt{exitPoint}, callSite, \texttt{symbol}, returnSite)$ is in δ_r for some $callSite \in Q$.

```
const std::set<std::pair<Key,Key>>
  getReturns_Exit( Key exitPoint )
```
Yields the set of all symbol/state pairs, $(sym, returnSite)$, such that the return transition $(\texttt{exitPoint}, callSite, sym, returnSite)$ is in δ_r for some $callSite \in Q$.

```
const std::set<Key> getReturns_Call( Key callSite, Key symbol )
```
Yields the set of all states, $returnSite$, such that the return transition $(exitPoint, \texttt{callSite}, \texttt{symbol}, returnSite)$ is in δ_r for some $exitPoint \in Q$.

```
const std::set<std::pair<Key,Key>> getReturns_Call(
                        Key callSite )
```
Yields the set of all symbol/state pairs, $(sym, returnSite)$, such that the return transition $(exitPoint, \texttt{callsite}, sym, returnSite)$ is in δ_r for some $exitPoint \in Q$.

```
const std::set<Key> getReturnSites( Key callSite )
```
Yields the set of all states, $returnSite$, such that the return transition $(exitPoint, \texttt{callSite}, sym, returnSite)$ is in δ_r for some $exitPoint \in Q$ and $sym \in \Sigma$.

```
const std::set<Key> getReturnSym(
                        Key exitPoint, Key callSite, Key returnSite )
```

Yields the set of all symbols, $symbol$, such that the return transition $(\texttt{exitPoint}, \texttt{callSite}, symbol, \texttt{returnSite})$ is in δ_r.

```
const std::set<Key> getReturnSym( )
```
Yields the set of all symbols, $symbol$, such that the return transition $(exitPoint, callSite, symbol, returnSite)$ is in δ_r for some $exitPoint \in Q$, $callSite \in Q$, and $returnSite \in Q$.

```
const std::set<Key> getReturnSym_Exit( Key exitPoint )
```
Yields the set of all symbols, $symbol$, such that the return transition $(\texttt{exitPoint}, callSite, symbol, returnSite)$ is in δ_r for some $callSite \in Q$ and $returnSite \in Q$.

```
const std::set<Key> getReturnSym_Call( Key callSite )
```
Yields the set of all symbols, *symbol*, such that the return transition $(exitPoint, \texttt{callSite}, symbol, returnSite)$ is in δ_r for some $exitPoint \in Q$ and $returnSite \in Q$.

```
const std::set<Key> getReturnSym_Ret( Key returnSite )
```
Yields the set of all symbols, *symbol*, such that the return transition $(exitPoint, callSite, symbol, \texttt{returnSite})$ is in δ_r for some $exitPoint \in Q$ and $callSite \in Q$.

```
const std::set<Key> getReturnSym_ExitCall(
                       Key exitPoint, Key callSite )
```
Yields the set of all symbols, *symbol*, such that the return transition $(\texttt{exitPointcallSite}, symbol, returnSite)$ is in δ_r for some $returnSite \in Q$.

```
const std::set<Key> getReturnSym_ExitRet(
                       Key exitPoint, Key returnSite )
```
Yields the set of all symbols, *symbol*, such that the return transition $(\texttt{exitPoint}, callSite, symbol, \texttt{returnSite})$ is in δ_r for some $callSite \in Q$.

```
const std::set<Key> getReturnSym_CallRet(
                       Key callSite, Key returnSite )
```
Yields the set of all symbols, *symbol*, such that the return transition $(exitPoint, \texttt{callSite}, symbol, \texttt{returnSite})$ is in δ_r for some $exitPoint \in Q$.

3 Building NWAs from other NWAs

NWAs do not need to be built state-by-state and transition-by-transition, they can also be built by performing language-theoretic operations over component NWAs. These operations include: union, intersection, concatenation, reversal, Kleene-star, complement, and determinize. The library supports two interfaces to the Boolean operations. In one, each operation returns a `ref_ptr<NWA>` to a freshly-built NWA. In the other, each operation transforms the NWA on which it was called.

The following operations are methods of class NWA:
```
static ref_ptr<NWA> unionNWA( ref_ptr<NWA> first,
                       ref_ptr<NWA> second, Key stuck )
```
The result is the union of the NWAs `first` and `second`. The stuck state of the resulting NWA is `stuck`. See Section 3.1.
```
static ref_ptr<NWA> intersect( ref_ptr<NWA> first,
                       ref_ptr<NWA> second, Key stuck )
```
The result is the intersection of the NWAs `first` and `second`. The stuck state of the resulting NWA is `stuck`. See Section 3.2.
```
static ref_ptr<NWA> concat( ref_ptr<NWA> first,
                       ref_ptr<NWA> second, Key stuck )
```
The result is the concatenation of the NWAs `first` and `second`. The stuck state of the resulting NWA is `stuck`. See Section 3.3.

```
static ref_ptr<NWA> star( ref_ptr<NWA> first, Key stuck )
```
The result is the Kleene-Star of the NWA `first`. The stuck state of the resulting NWA is `stuck`. See Section 3.4.

```
static ref_ptr<NWA> reverse( ref_ptr<NWA> first, Key stuck )
```
The result is the NWA which accepts the reverse of each nested word accepted by the NWA `first`. The stuck state of the resulting NWA is `stuck`. See Section 3.5.

```
static ref_ptr<NWA> determinize( ref_ptr<NWA> nondet, Key stuck )
```
The result is the deterministic NWA that is equivalent to the NWA `nondet`. The stuck state of the resulting NWA is `stuck`. See Section 3.6.

```
static ref_ptr<NWA> complement( ref_ptr<NWA> first, Key stuck )
```
The result is the NWA which is the complement of the NWA `first`. The stuck state of the resulting NWA is `stuck`. See Section 3.7.

```
void unionNWA( ref_ptr<NWA> first, ref_ptr<NWA> second )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA resulting from the union of the NWAs `first` and `second`. See Section 3.1. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.unionNWA(B,C);

```
void intersect( ref_ptr<NWA> first, ref_ptr<NWA> second )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA resulting from the intersection of the NWAs `first` and `second`. See Section 3.2. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.intersect(B,C);

```
void concat( ref_ptr<NWA> first, ref_ptr<NWA> second )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA resulting from concatenating the NWAs `first` and `second`. See Section 3.3. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.concat(B,C);

```
void star( ref_ptr<NWA> first )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA resulting from performing the Kleene-Star operation on the NWA `first`. See Section 3.4. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.star(B);

```
void reverse( ref_ptr<NWA> first )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA which accepts the reverse of each nested word accepted by the NWA `first`. See Section 3.5. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.reverse(B);

```
void determinize( ref_ptr<NWA> nondet )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the deterministic NWA that is equivalent to the NWA `nondet`. See Section 3.6. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A with a stuck state; A.determinize(B);

```
void complement( ref_ptr<NWA> first )
```
This operation is an instance method; the NWA referred to by `this` is transformed into the NWA resulting from complementing the NWA `first`. See Section 3.7. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct Awith a stuck state; A.complement(B);

3.1 Union

Consider an example of computing the union of two NWAs, the NWA shown in Figure 2 and the NWA shown in Figure 4. The union is constructed by combining all functionality of both component NWAs. If the component NWAs are $(Q_1, \Sigma_1, Q_{01}, \delta_1, Q_{f_1})$ and $(Q_2, \Sigma_2, Q_{02}, \delta_2, Q_{f_2})$, then the resulting NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$ where $Q = Q_1 \cup Q_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $Q_0 = Q_{01} \cup Q_{02}$, $\delta = \delta_1 \cup \delta_2$, and $Q_f = Q_{f_1} \cup Q_{f_2}$. The NWA resulting from the union of the NWA shown in Figure 2 and the NWA shown in Figure 4 is shown in Figure 5. Note that the resulting NWA is nondeterministic (because it has multiple initial states).
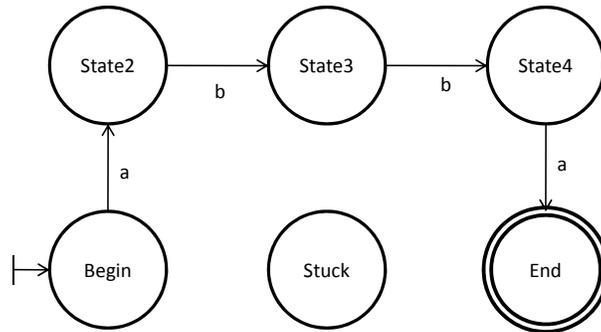
Fig. 4. Simple NWA to union with the NWA in Figure 2.

The stuck state of the resulting NWA cannot appear in either of the component NWAs, except as a stuck state. Furthermore, to construct the union of two NWAs, the state sets of the NWAs cannot overlap (except by the stuck state), i.e. $Q_1 \cap Q_2 = \emptyset$ or $Q_1 \cap Q_2 = Stuck$, where $Stuck$ is the stuck state of both NWAs. Client information is copied directly from the component NWAs using the `ClientInfo` copy constructor.
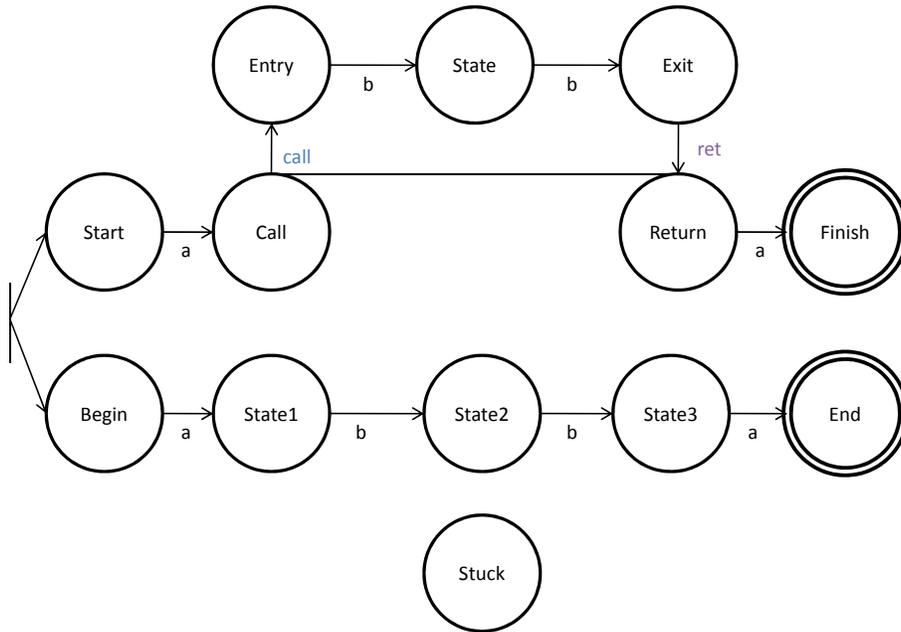
Fig. 5. The NWA resulting from the union of the NWA in Figure 2 and the NWA in Figure 4.

3.2 Intersection

Consider an example of computing the intersection of two NWAs, the NWA shown in Figure 2 and the NWA shown in Figure 6. The intersection is constructed by traversing both component NWAs starting at the initial states and incrementally adding transitions for each pair of "intersectable" transitions that are encountered. Transitions are "intersectable" when 1. the transitions are the same kind (internal, call, or return), 2. the symbols of the transitions are identical,[4] and 3. the states at each position in the transitions can be joined. The resulting NWA is shown in Figure 7.

It is possible to customize what states and symbols are considered equivalent by overriding the `stateIntersect` and `transitionIntersect` methods. The default behavior of `stateIntersect` is that any two states can be intersected, the resulting state to add 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`. The default behavior of `transitionIntersect` is that two transitions are intersectable if the symbols

---

[4] Strictly speaking, the symbols do not have to be identical when the meta-symbols ϵ and $*$ are involved. In addition, the client has the flexibility to change how identity of symbols is handled by overriding the method transitionIntersect.
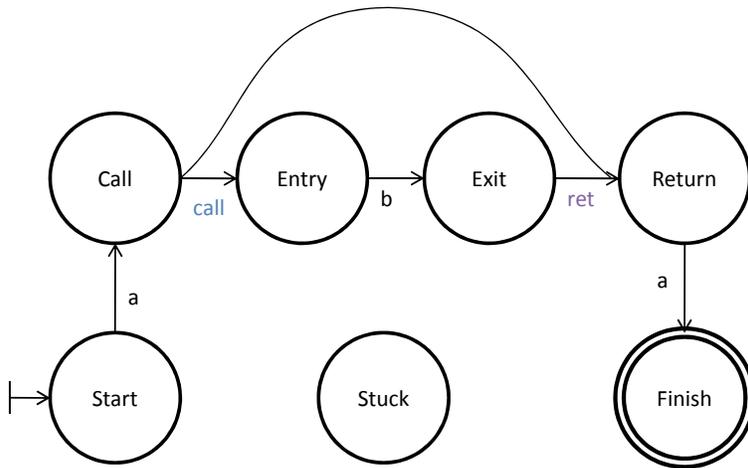
Fig. 6. Simple NWA to intersect with the NWA in Figure 2.
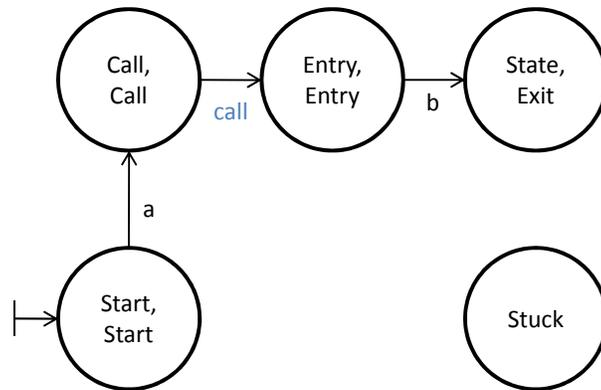
Fig. 7. The NWA resulting from the intersection of the NWA in Figure 2 and the NWA in Figure 6.

that label the transitions are not `epsilon` (ϵ) and either the symbols are the same or at least one of the symbols is a `wild` (∗).

Client information is generated by the helper method `stateIntersect`, but can be altered through the use of the helper methods `intersectClientInfoInternal`, `intersectClientInfoCall`, and `intersect-ClientInfoReturn`, 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 to specify alternative behaviors.

As intersection is performed, and the transitions of the component NWAs are traversed, `transitionIntersect` is called to determine whether a tran-

sition should be added to the resulting NWA. If so, `stateIntersect` is called to determine whether the target states of the transitions in question can be joined. Finally, if a transition is going to be added to the resulting NWA, `intersectClientInfoInternal`, `intersectClientInfoCall`, or `intersectClientInfoReturn` is called (as appropriate) to update the `ClientInfo` associated with the target state of the transition being added.

The following operations are methods of class NWA and are intended to be overridden to customize behavior:

```
bool stateIntersect( ref_ptr<NWA> first, Key state1,
                     ref_ptr<NWA> second,Key state2,
                     Key& resSt, ref_ptr<ClientInfo>& resCI )
```
Determines whether the given states are considered to be equivalent for the purposes of intersection and, if so, creates the combined state, together with the client information that should be associated with the combined state and returns true. Otherwise, returns false.

```
bool transitionIntersect( ref_ptr<NWA> first, Key sym1,
                          ref_ptr<NWA> second, Key sym2,
                          Key& resSym )
```
Determines whether the given symbols are considered to be equivalent for the purposes of intersection and, if so, computes the symbol which should be associated with the combined transition and returns true. Otherwise, returns false.

```
void intersectClientInfoInternal(
                     ref_ptr<NWA> first, Key src1, Key tgt1,
                     ref_ptr<NWA> second, Key src2, Key tgt2,
                     Key resSym, Key resSt )
```
Alters the client information associated with `resSt` given the sources and targets (as well as access to the client information associated with each state) of the two internal transitions that are being intersected.

```
void intersectClientInfoCall(
                     ref_ptr<NWA> first ,Key call1, Key entry1,
                     ref_ptr<NWA> second, Key call2, Key entry2,
                     Key resSym, Key resSt )
```
Alters the client information associated with `resSt` given the call sites and entry points (as well as access to the client information associated with each state) of the two call transitions that are being intersected.

```
void intersectClientInfoReturn(
              ref_ptr<NWA> first, Key exit1, Key call1, Key ret1,
              ref_ptr<NWA> second, Key exit2, Key call2, Key ret2,
              Key resSym, Key resSt )
```
Alters the client information assoicated with `resSt` given the exit point, call site, and return site (as well as access to the client information associated with each state) of the two return transitions that are being intersected.

Consider the slightly more complex example of computing the intersection of the NWAs shown in Figure 8. The resulting NWA is shown in Figure 9. Note that a state is only made final if both of its component states were final in their respective NWAs (this can be seen in the state 'Call_1,Dead_End' where 'Dead_End' was final, but 'Call_1' was not).

3.3 Concatenation

Consider an example of computing the concatenation of two NWAs, the NWA shown in Figure 2 and the NWA shown in Figure 10. The concatenation is constructed by combining all states and transitions of the first NWA (shown in Figure 2) and all states and transitions of the second NWA (shown in Figure 10) then 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; the final states are the final states of the second NWA. If the component NWAs are $(Q_1, \Sigma_1, Q_{0_1}, \delta_1, Q_{f_1})$ and $(Q_2, \Sigma_2, Q_{0_2}, \delta_2, Q_{f_2})$, then the resulting NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$ where $Q = Q_1 \cup Q_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $Q_0 = Q_{0_1}$, $\delta = \delta_1 \cup \delta_2 \cup \delta_\epsilon$ (where $\delta_\epsilon = \{(q, \epsilon, q') | q \in Q_{f_1}, q' \in Q_{0_2}\}$, and $Q_f = Q_{f_2}$. The NWA resulting from the concatenation of the NWA in Figure 2 and the NWA shown in Figure 10 is shown in Figure 11.

The stuck state of the NWA on which the concat method is called cannot appear in either of the component NWAs xcept as a stuck state. Furthermore, in order to construct the concatenation of two NWAs, the state sets of the NWAs cannot overlap (except by the stuck state), i.e. $Q_1 \cap Q_2 = \emptyset$ or $Q_1 \cap Q_2 = Stuck$, where $Stuck$ is the stuck state of both NWAs. Client information is copied directly from the component NWAs using the `ClientInfo` copy constructor.

3.4 Kleene-Star

Consider an example of computing the Kleene-Star of the NWA shown in Figure 12. If the component NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$, then the result of performing Kleene-Star on that NWA is $(Q \cup Q', \Sigma, Q'_0, \delta', Q'_0)$ where $Q' = \{q' | q \in Q\}$ and $\delta' = \{\delta'_i, \delta'_c, \delta'_r\}$ such that

1. For all $(q, a, p) \in \delta_i$, the result NWA has $(q, a, p) \in \delta'_i$ and $(q', a, p') \in \delta'_i$, and if $p \in Q_f$, then $(q, a, r') \in \delta'_i$ and $(q', a, r') \in \delta'_i$ for each $r \in Q_0$,
2. For all $(q, a, p) \in \delta_c$, the result NWA has $(q, a, p) \in \delta'_c$ and $(q', a, p) \in \delta'_c$, and if $p \in Q_f$, then $(q, a, r') \in \delta'_c$ and $(q', a, r') \in \delta'_c$ for each $r \in Q_0$,
3. For all $(q, r, a, p) \in \delta_r$, the result NWA has $(q, r, a, p) \in \delta'_r$ and $(q, r', a, p') \in \delta'_r$, and if $p \in Q_f$, then $(q, r, a, s') \in \delta'_r$ and $(q, r', a, s') \in \delta'_r$ for each $s \in Q_0$, and
4. For all $(q, r, a, p) \in \delta_r$ with $r \in Q_0$, the result NWA has $(q', s, a, p') \in \delta'_r$ for each $s \in Q \cup Q'$, and if $p \in Q_f$, then $(q', s, a, t') \in \delta'_r$ for each $s \in Q \cup Q'$ and $t \in Q_0$

The NWA resulting from performing Kleene-Star on the NWA shown in Figure 12 is shown in Figure 13. Note: The primed states, q' are constructed using `getKey(q,getKey(prime))` where $q \in Q$ and `prime` is the string "prime".
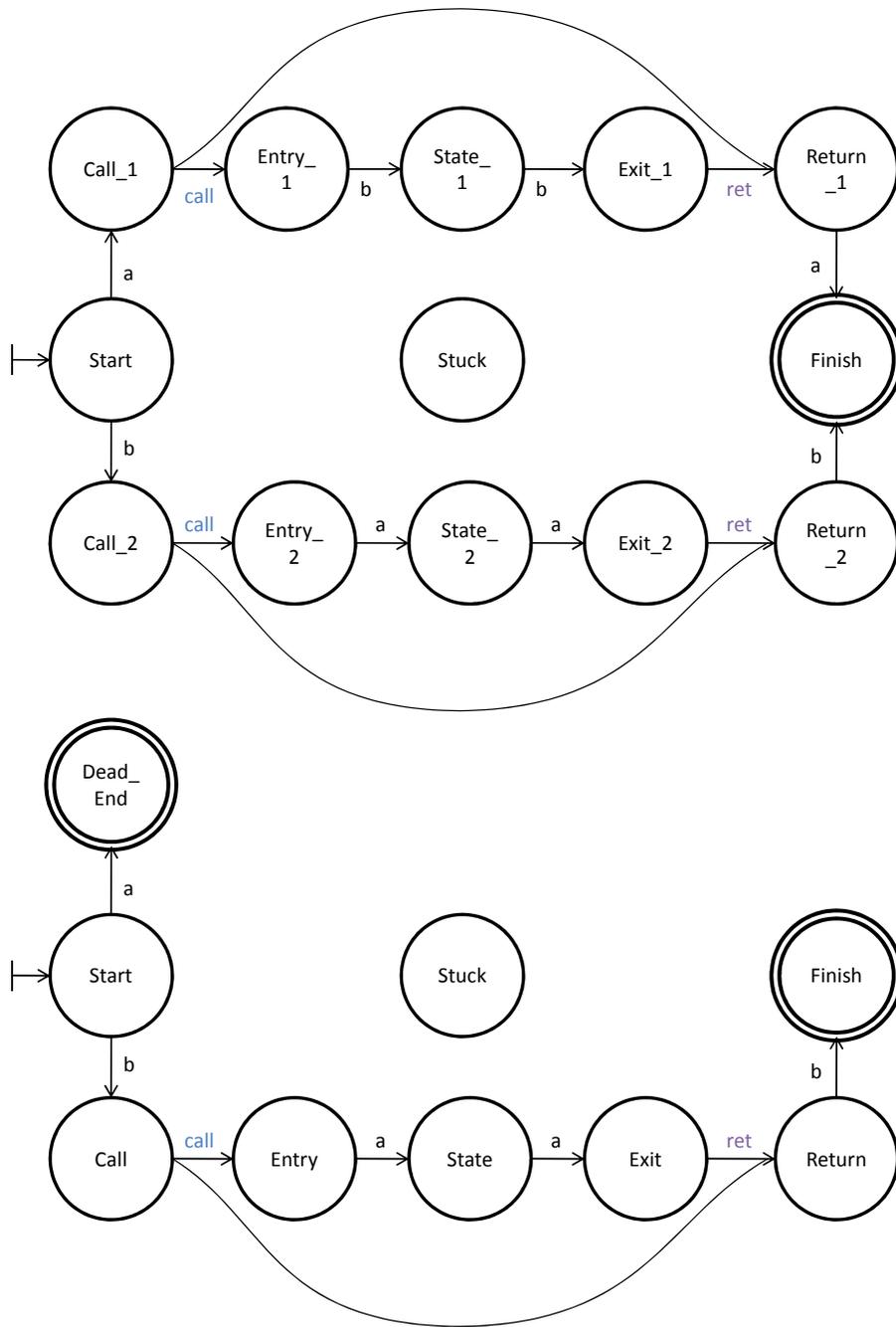
Fig. 8. Complex NWAs to intersect.

The stuck state of the NWA on which the Kleene-Star method is called cannot appear in the component NWA nor can it be the prime of any state in the NWA (i.e., $stuck \notin Q \cup Q'$) unless it is the stuck state. Client information is copied directly from the component NWA (using the `ClientInfo` copy constructor) such that for each $q \in Q$, q and q' have the same client information.

3.5 Reverse

If the component NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$, then the result of reversing that NWA is $(Q, \Sigma, Q_0, \delta_{rev}, Q_f)$ such that

1. For all $(q, \sigma, q') \in \delta_i$, the result NWA has $(q', \sigma, q) \in \delta_{revi}$
2. For all $(q_c, \sigma, q_e) \in \delta_c$, the result NWA has $(q_e, \sigma, q_c) \in \delta_{revc}$
3. For all $(q_x, q_c, \sigma, q_r) \in \delta_r$ and $(q_c, \sigma, q_e) \in \delta_c$ (where the q_cs are the same), the result NWA has $(q_r, q_e, \sigma, q_x) \in \delta_{revr}$

The NWA resulting from performing reverse on the NWA shown in Figure 2 is shown in Figure 14.

The stuck state of the NWA on which the reverse method is called cannot be any state in the component NWA (including the stuck state). Client information is copied directly from the component NWA using the `ClientInfo` copy constructor.

3.6 Determinize

Definition 3. *An NWA, $(Q, \Sigma, Q_0, \delta, Q_f)$, is **deterministic** iff*

1. $|Q_0| = 1$,
2. *For all $q \in Q$:*
    - *if $(q, *, q') \in \delta_i$ then $|\{q'|(q, \sigma, q') \in \delta_i\}| = 0$; otherwise, for all $\sigma \in \Sigma - \{*\}, |\{q'|(q, \sigma, q') \in \delta_i\}| \leq 1$,*
    - *if $(q, *, q') \in \delta_c$ then $|\{q'|(q, \sigma, q') \in \delta_c\}| = 0$; otherwise for all $\sigma \in \Sigma - \{*\}, |\{q'|(q, \sigma, q') \in \delta_c\}| \leq 1$, and*
    - *for all $q' \in Q$, if $(q, q', *, q'') \in \delta_r$ then $|\{q''|(q, q', \sigma, q'') \in \delta_r\}| = 0$; otherwise for all $\sigma \in \Sigma - \{*\}, |\{q''|(q, q', \sigma, q'') \in \delta_r\}| \leq 1$, and*
3. *For all $(q, \sigma, q') \in \delta_i, \sigma \neq \epsilon$,*
    for all $(q, \sigma, q') \in \delta_c, \sigma \neq \epsilon$, and
    for all $(q, q', \sigma, q'') \in \delta_r, \sigma \neq \epsilon$.

*If an NWA is not deterministic, then it is **non-deterministic**.*

Consider an example of determinizing a simple nondeterministic NWA, the NWA shown in Figure 15. The deterministic NWA is computed by a generalization of the classical subset construction. Instead of the states in the determinized NWA being subsets of the original NWA, states of the determinized NWA are sets of state pairs (i.e., binary relations on states) [2]. To support determinization, the library provides class `BinaryRelation`, which is a set of `Key` pairs. The resulting NWA is shown in Figure 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 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 operations are methods of class NWA and are intended to be overridden to customize behavior:

```
void mergeClientInfo( ref_ptr<NWA> first,
                      BinaryRelation const& binRel,
                      St resSt, ref_ptr<ClientInfo>& resCI )
```
  Computes the client information that should be associated with the given state.
```
void mergeClientInfoInternal( ref_ptr<NWA> first,
                      BinaryRelation const& binRelSource,
                      BinaryRelation const& binRelTarget,
                      Key sourceSt, Key resSym, Key resSt,
                      ref_ptr<ClientInfo>& resCI )
```
  Alters the client information associated with `resSt` given the details of the internal transition to be added to the NWA.
```
void mergeClientInfoCall( ref_ptr<NWA> first,
                      BinaryRelation const& binRelCall,
                      BinaryRelation const& binRelEntry,
                      Key callSt, Key resSym, Key resSt,
                      ref_ptr<ClientInfo>& resCI )
```
  Alters the client information associated with `resSt` given the details of the call transition to be added to the NWA.
```
void mergeClientInfoReturn( ref_ptr<NWA> first,
                      BinaryRelation const& binRelExit,
                      BinaryRelation const& binRelCall,
                      BinaryRelation const& binRelReturn,
```

```
                    Key exitSt, Key callSt, Key resSym,
                    Key resSt, ref_ptr<ClientInfo>& resCI )
```
Alters the client information associated with `resSt` given the details of the return transition to be added to the NWA.

3.7   Complement

The complement is performed by first determinizing the NWA and then complementing the final-state set of the determinized NWA. The NWA resulting from performing the complement of the NWA shown in Figure 15 is shown in Figure 17.

Because all non-final states (including the stuck state) become final states in the process of complementation, all implicit transitions are materialized and the resulting NWA is of type 2. Client information is copied directly from the determinization of the component NWA using the `ClientInfo` copy constructor.

4   Conversions

It is possible to both convert from a WALi WPDS to an NWA and from an NWA to a WPDS. **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.**

The following operations are methods of class NWA:

`WPDS plusWPDS( const WPDS& base )`
   This operation is an instance method. It returns the WPDS that is the product of the NWA referred to by `this` and the given WPDS [3].

`void PDStoNWA( const WPDS& pds )`
   This operation is an instance method; the NWA referred to by `this` is converted to the NWA that is equivalent to the given PDS. This method should not be called on an NWA of type 2, otherwise there is an assertion violation. Typical sequence of operations: construct A; A.PDStoNWA(pds);

`static ref_ptr<NWA> PDStoNWA( const WPDS& pds, Key stuck )`
   Constructs and returns the NWA that is equivalent to the given PDS, having the stuck state `stuck`.

`WPDS NWAtoPDScalls( WeightGen<Client>& wg ) const`
   This operation is an instance method. It constructs the WPDS that keeps call states on the stack that is equivalent to the NWA referred to by `this`. It uses `wg` to determine weights for WPDS rules (see Section 4.2).

`WPDS NWAtoBackwardsPDScalls( WeightGen<Client>& wg ) const`
   This operation is an instance method. It constructs the backwards WPDS that keeps call states on the stack that is equivalent to the NWA referred to by `this`. It uses `wg` to determine weights for WPDS rules (see Section 4.2).

```
WPDS NWAtoPDSreturns( WeightGen<Client>& wg ) const
```
This operation is an instance method. It constructs the WPDS that keeps return states on the stack thatis equivalent to the NWA referred to by `this`. It uses `wg` to determine weights for WPDS rules (see Section 4.2). (Deprecated)
```
WPDS NWAtoBackwardsPDSreturns( WeightGen<Client>& wg ) const
```
This operation is an instance method. It constructs the backwards WPDS that keeps return states on the stack that is equivalent to the NWA referred to by `this`. It uses `wg` to determine weights for WPDS rules (see Section 4.2). (Deprecated)

4.1 PDS to NWA

The last way to build an NWA is to convert a PDS into an NWA. The conversion is performed by:

1. For each $\langle p, q \rangle \hookrightarrow \langle p', q' \rangle \in \Delta_1$ in the PDS, adding $((p, q), q, (p', q'))$ to δ_i of the NWA.
2. For each $\langle p, q_c \rangle \hookrightarrow \langle p', q_e\ q_r \rangle \in \Delta_2$ in the PDS, adding $((p, q_c), q_c, (p', q_e))$ to δ_c of the NWA.
3. For each $\langle p'', q_x \rangle \hookrightarrow \langle p''', \epsilon \rangle \in \Delta_0$ in the PDS, adding $((p'', q_x), (p, q_c), q_x, (p''', q_r))$ to δ_r of the NWA for all (q_c, q_r) pairs such that $\langle p, q_c \rangle \hookrightarrow \langle p', q_e\ q_r \rangle \in \Delta_2$.

For example, the NWA resulting from converting the PDS shown in Figure 18 into an NWA is shown in Figure 19.

The client information for all states in the resulting NWA are set to `null`.

4.2 NWA to PDS

An NWA can also be converted into a WPDS. In this way it is possible to use the reachability queries that are a part of the main WALi library on NWAs. There are four variations on the NWA to WPDS conversion: 1. forward flow with call states on the stack 2. backward flow with call states on the stack 3. forward flow with return states on the stack 4. backward flow with return states on the stack. All four variations use `WeightGen` to determine weights for WPDS rules.

`WeightGen` provides an interface that you **must** implement for calculating the weights for NWA transitions which are subsequently used in calculating the weights for corresponding WPDS rules. It allows the underlying NWA to be decoupled from the weight domain to be used. See [4, §4-§5] for details about weight domains. To convert from an NWA to a WPDS, the user must first subclass WeightGen to provide the weights for the WPDS (an instance of this subclass is passed to the WPDS-generation functions).

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

```
sem_elem_t getOne()
```
Returns an instance of the $\bar{1}$ element of the weight domain.
```
sem_elem_t getWeight( Key source, ref_ptr<ClientInfo> sourceInfo,
                      Key symbol, Kind k,
                      Key target, ref_ptr<ClientInfo> targetInfo )
```
Computes and returns the weight (in the desired semiring) for a (`kind`) NWA transition from `source` to `target` labeled with symbol `symbol`.
```
sem_elem_t getWildWeight(
                      Key source, ref_ptr<ClientInfo> sourceInfo,
                      Key target, ref_ptr<ClientInfo> targetInfo )
```
Computes and returns the weight (in the desired semiring) for an NWA transition from `source` to `target` labeled with the meta-symbol `*`.

1. Forwards Flow Stacking Calls

The conversion is performed by:

1. For each $(q, \sigma, q') \in \delta_i$ in the NWA, adding $\langle p, q \rangle \hookrightarrow \langle p, q' \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight`$(q, CI_q, \sigma, \texttt{INTRA}, q', CI_{q'})$ or `wg.getWildWeight`$(q, CI_q, q', CI_{q'})$, depending on σ).
2. For each $(q_c, \sigma, q_e) \in \delta_c$ in the NWA, adding $\langle p, q_c \rangle \hookrightarrow \langle p, q_e\, q_c \rangle$ to Δ_2 of the WPDS (with weight `wg.getWeight`$(q_c, CI_{q_c}, \sigma, \texttt{CALL\_TO\_ENTRY}, q_e, CI_{q_e})$ or `wg.getWildWeight`$(q_c, CI_{q_c}, q_e, CI_{q_e})$, depending on σ).
3. For each $(q_x, q_c, \sigma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_x \rangle \hookrightarrow \langle p_{q_x}, \epsilon \rangle$ to Δ_0 of the WPDS (with weight `wg.getWeight`$(q_x, CI_{q_x}, \sigma, \texttt{EXIT\_TO\_RET}, q_r, CI_{q_r})$ or `wg.getWildWeight`$(q_x, CI_{q_x}, q_r, CI_{q_r})$, depending on σ) and $\langle p_{q_x}, q_c \rangle \hookrightarrow \langle p, q_r \rangle$ to Δ_1 of the WPDS (with weight `wg.getOne()`).

Consider, as an example, converting the NWA in Figure 22 into a WPDS. The WPDS resulting from converting the PDS shown in Figure 20 into a WPDS is shown in Figure 21.

2. Backwards Flow Stacking Calls

The conversion is performed by:

1. For each $(q, \sigma, q') \in \delta_i$ in the NWA, adding $\langle p, q' \rangle \hookrightarrow \langle p, q \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight`$(q, CI_q, \sigma, \texttt{INTRA}, q', CI_{q'})$ or `wg.getWildWeight`$(q, CI_q, q', CI_{q'})$, depending on σ).
2. For each $(q_c, \sigma, q_e) \in \delta_c$ and $(q_x, q_c, \gamma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_e \rangle \hookrightarrow \langle p_{q_e}, \epsilon \rangle$ to Δ_0 of the WPDS (with weight `wg.getWeight`$(q_c, CI_{q_c}, \sigma, \texttt{CALL\_TO\_ENTRY}, q_e, CI_{q_e})$ or `wg.getWildWeight`$(q_c, CI_{q_c}, q_e, CI_{q_e})$, depending on σ) and $\langle p_{q_e}, q_r \rangle \hookrightarrow \langle p, q_c \rangle$ to Δ_1 of the WPDS (with weight `wg.getOne()`).
3. For each $(q_x, q_c, \sigma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_r \rangle \hookrightarrow \langle p, q_x\, q_r \rangle$ to Δ_2 of the WPDS (with weight `wg.getWeight`$(q_x, CI_{q_x}, \sigma, \texttt{EXIT\_TO\_RET}, q_r, CI_{q_r})$ or `wg.getWildWeight`$(q_x, CI_{q_x}, q_r, CI_{q_r})$, depending on σ).

Consider, as an example, converting the NWA in Figure 20 into a backwards flow WPDS. The WPDS resulting from converting the PDS shown in Figure 20 into a backwards flow WPDS is shown in Figure 22. Note: `NWAtoBackwardsPDScalls(A) = NWAtoPDScalls(reverse(A))`.

3. Forwards Flow Stacking Returns

The conversion is performed by:

1. For each $(q, \sigma, q') \in \delta_i$ in the NWA, adding $\langle p, q \rangle \hookrightarrow \langle p, q' \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight(`$q, CI_q, \sigma,$`INTRA,`$q', CI_{q'}$`)` or `wg.getWildWeight(`$q, CI_q, q', CI_{q'}$`)`, depending on σ).

2. For each $(q_c, \sigma, q_e) \in \delta_c$ and $(q_x, q_c, \gamma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_c \rangle \hookrightarrow \langle p, q_e q_r \rangle$ to Δ_2 of the WPDS (with weight `wg.getWeight(`$q_c, CI_{q_c}, \sigma,$`CALL_TO_ENTRY,`q_e, CI_{q_e}`)` or `wg.getWildWeight(`$q_c, CI_{q_c}, q_e, CI_{q_e}$`)`, depending on σ) .

3. For each $(q_x, q_c, \sigma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_x \rangle \hookrightarrow \langle p_{q_x}, \epsilon \rangle$ to Δ_0 of the WPDS (with weight `wg.getOne()`) and $\langle p_{q_x}, q_r \rangle \hookrightarrow \langle p, q_r \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight(`$q_x, CI_{q_x}, \sigma,$`EXIT_TO_RET,`q_r, CI_{q_r}`)` or texttttwg.getWildWeight($q_x, CI_{q_x}, q_r, CI_{q_r}$), depending on σ).

Consider, as an example, converting the NWA in Figure 20 into a WPDS. The WPDS resulting from converting the PDS shown in Figure 20 into a WPDS is shown in Figure 23.

4. Backwards Flow Stacking Returns

The conversion is performed by:

1. For each $(q, \sigma, q') \in \delta_i$ in the NWA, adding $\langle p, q' \rangle \hookrightarrow \langle p, q \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight(`$q, CI_q, \sigma,$`INTRA,`$q', CI_{q'}$`)` or `wg.getWildWeight(`$q, CI_q, q', CI_{q'}$`)`, depending on σ).

2. For each $(q_c, \sigma, q_e) \in \delta_c$ in the NWA, adding $\langle p, q_e \rangle \hookrightarrow \langle p_{q_e}, \epsilon \rangle$ to Δ_0 of the WPDS (with weight `wg.getOne()`) and $\langle p_{q_e}, q_c \rangle \hookrightarrow \langle p, q_c \rangle$ to Δ_1 of the WPDS (with weight `wg.getWeight(`$q_c, CI_{q_c}, \sigma,$`CALL_TO_ENTRY,`q_e, CI_{q_e}`)` or `wg.getWildWeight(`$q_c, CI_{q_c}, q_e, CI_{q_e}$`)`, depending on σ).

3. For each $(q_x, q_c, \sigma, q_r) \in \delta_r$ in the NWA, adding $\langle p, q_r \rangle \hookrightarrow \langle p, q_x q_c \rangle$ to Δ_2 of the WPDS (with weight `wg.getWeight(`$q_x, CI_{q_x}, \sigma,$`EXIT_TO_RET,`q_r, CI_{q_r}`)` or `wg.getWildWeight(`$q_x, CI_{q_x}, q_r, CI_{q_r}$`)`, depending on σ).

Consider, as an example, converting the NWA in Figure 24 into a backwards flow WPDS. The WPDS resulting from converting the PDS shown in Figure 20 into a backwards flow WPDS is shown in Figure 24.

5 Queries

Once an NWA is built, there are a multitude of queries that can be made, including: 1. whether the NWA is deterministic or non-deterministic,

2. language-inclusion queries, and 3. reachability queries. All of these queries can be made for an NWA of either type.

5.1 Deterministic

See Section 3.6 for the definition of deterministic.
The following operations are methods of class NWA:

`bool isDeterministic( )`
Determines whether the NWA is deterministic (returns `true`) or non-deterministic (returns `false`). If the NWA is not deterministic, it can be determinized using `determinize` (see Section 3.6).

5.2 Language Inclusion

The following operations are methods of class NWA:

`bool isEmpty( )`
Determines whether the language accepted by this NWA is empty (returns `true`) or non-empty (returns `false`).

`static bool isMember( NWS word, ref_ptr<NWA> aut )`
Determines whether the given nested word is a member of the language accepted by the given NWA (returns `true`) or not (returns `false`).

`static bool inclusion( ref_ptr<NWA> first, ref_ptr<NWA> second )`
Determines whether the language accepted by the first NWA is included in the language accepted by the second NWA (returns `true`) or not (returns `false`).

`static bool equal( ref_ptr<NWA> first, ref_ptr<NWA> second )`
Determines whether the languages accepted by the given NWAs are equal (returns `true`) or not (returns `false`).

Class `NWS` represents nested-word suffixes, i.e., unbalanced-right nested words. In an `NWS` each position of the nested-word suffix corresponds to one `NWSNode`. Using the following methods, a nested-word suffix can be constructed and used to determine whether a specific nested word is accepted by an NWA (using `isMember`). Note: The left-hand side of an `NWS` is referred to as the working end of the `NWS` as nodes can be added to and removed from this end only.

The following operations are methods of class NWS:

`NWSNode * stackTop( )`
Returns the `NWSNode` at the top of the nesting stack.

`void pushStack( NWSNode * exit )`
Adds `exit` to the top of the nesting stack.

`void popStack( )`
Removes the next return node from the nesting stack.

`size_t stackSize( )`

  Returns the current level of nesting at the working end of the NWS.

`void addNode( Key sym )`

  Appends an `NWSNode` with the symbol `sym` to the working end of the NWS.

`bool addNode( Key sym, NWSNode * returnNode )`

  Appends an `NWSNode` with the symbol `sym` and return node `returnNode` to the working end of the NWS. If this `NWSNode` is already a call node with a different return node, the call/return link is not created, the `NWSNode` is not appended, and false is returned.

`NWSNode * nextNode( )`

  Returns the `NWSNode` at the working end of the NWS.

`NWSNode * removeNode( )`

  Removes the `NWSNode` at the working end of the NWS, adds any newly opened nesting to the nesting stack, and returns the `NWSNode` that was removed from the nested-word suffix.

`bool isEmpty( )`

  Tests whether this nested-word suffix is empty (returns true if empty, false otherwise).

Note that class NWS can also represent balanced nested words by simply having an empty nesting stack.

Similarly, class `NWP` represents nested-word prefixes, i.e., unbalanced-left nested words. In an `NWP` each position of the nested-word prefix corresponds to one `NWPNode`. Using the following methods, a nested-word prefix can be constructed and used. Note: The right-hand side of an `NWP` is referred to as the working end of the `NWP` as nodes can be added to and removed from this end only.

The following operations are methods of class NWP:

`NWPNode * currCall( )`

  Returns the `NWPNode` at the top of the nesting stack.

`size_t nestSize( )`

  Returns the current level of nesting at the working end of the NWP.

`bool addIntraNode( Key sym )`

  Appends an `NWPNode` with the symbol `sym` to the working end of the NWP.

`bool addCallNode( Key sym )`

  Appends an `NWPNode` with the symbol `sym` to the working end of the NWP and adds that `NWPNode` to the nesting stack.

`bool addReturnNode( Key sym )`

  Appends an `NWPNode` with the symbol `sym` to the working end of the NWP and removes the `NWPNode` from the top of the nesting stack.

`NWPNode * endNode( )`

  Returns the `NWPNode` at the working end of the NWP.

`NWPNode * removeNode( )`

  Removes the `NWPNode` at the working end of the NWP and returns the `NWPNode` that was removed from the nested-word prefix.

```
bool isEmpty( )
```
Tests whether this nested-word prefix is empty (returns true if empty, false otherwise).

Note that class `NWP` can also represent balanced nested words by simply having an empty nesting stack.

5.3 Reachability

The following operations are methods of class NWA:
```
WFA prestar( WFA& input, WeightGen<Client>& wg )
```
Performs the prestar reachability query defined by the given WFA [4].
```
void prestar( WFA& input, WFA& output, WeightGen<Client>& wg )
```
Performs the prestar reachability query defined by the given WFA [4].
```
WFA poststar( WFA& input, WeightGen<Client>& wg )
```
Performs the poststar reachability query defined by the given WFA [4].
```
void poststar( WFA& input, WFA& output, WeightGen<Client>& wg )
```
Performs the poststar reachability query defined by the given WFA [4].

For the `prestar` (or `poststar`) reachability query, `NWAtoPDScalls` is used to create a WPDS on which the WALi `prestar` (or `poststar`) method is called. For details about `WeightGen` and `NWAtoPDScalls` see Section 4.2.

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), May 2009.
3. N. Kidd, A. Lal, and T. Reps. Advanced querying for property checking. Technical Report TR-1624, University of Wisconsin, Madison, Oct 2007.
4. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
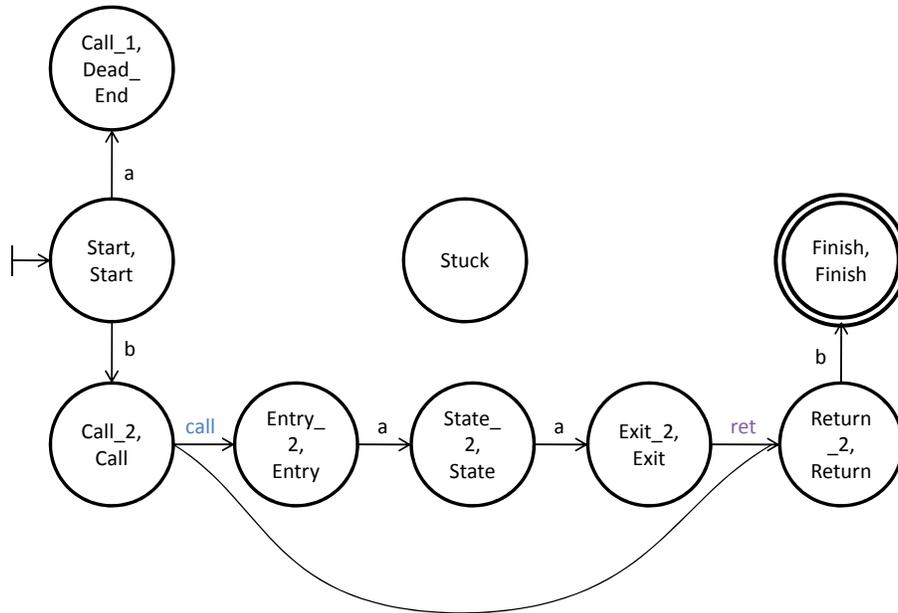
Fig. 9. The NWA resulting from the intersection of the NWAs in Figure 8.
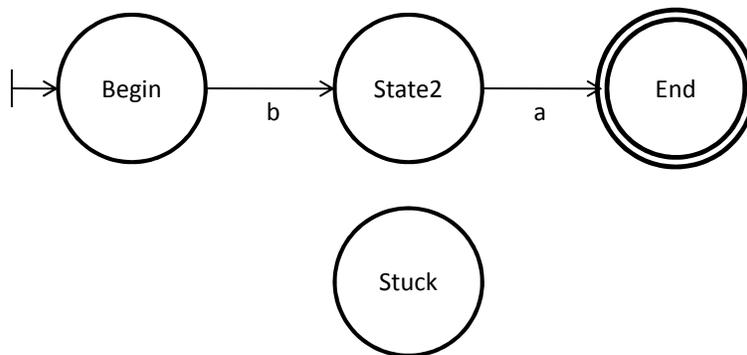
Fig. 10. Simple NWA to concatenate onto the NWA in Figure 2.
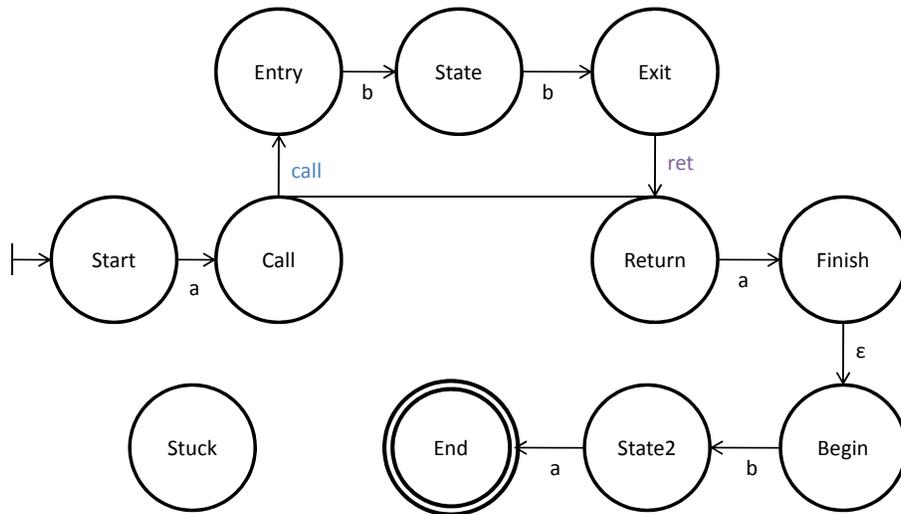
Fig. 11. The NWA resulting from the concatenation of the NWA in Figure 2 with the NWA in Figure 10.
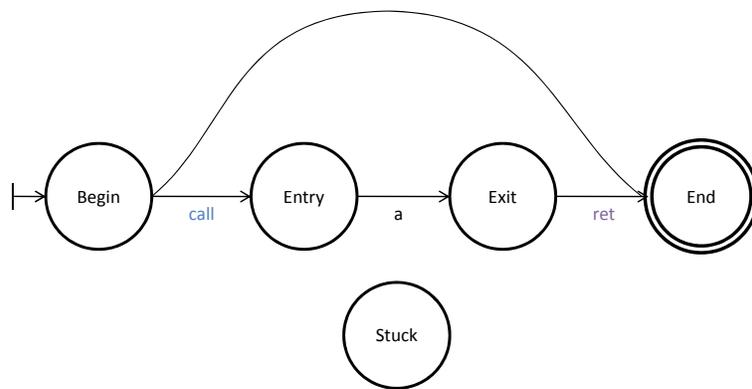
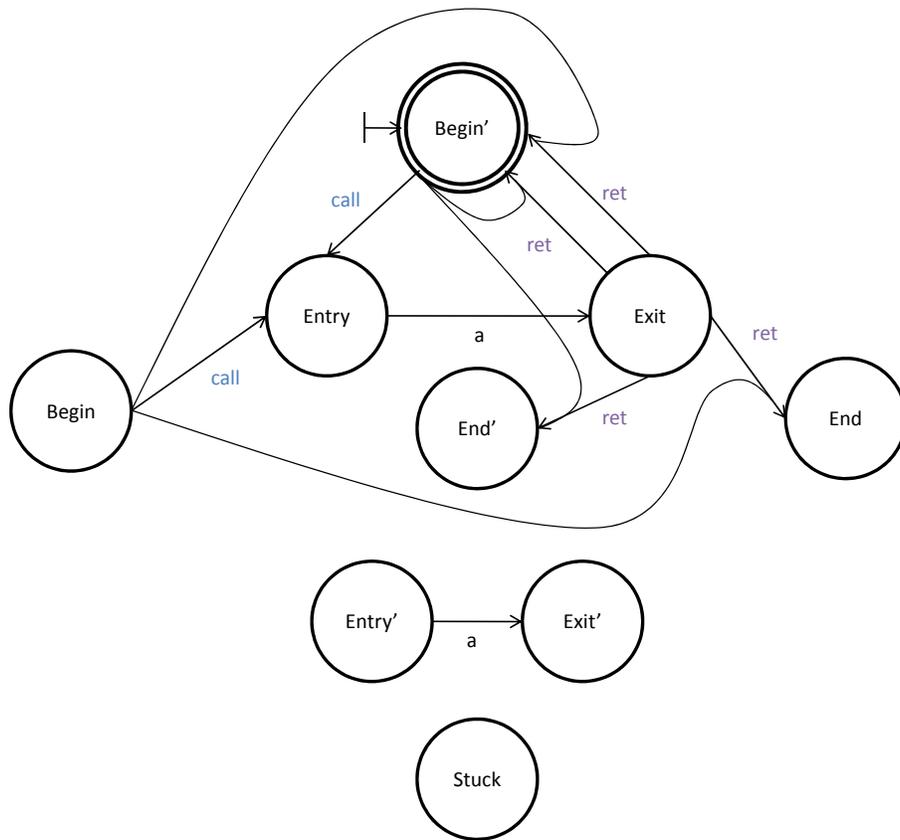

Fig. 12. An NWA on which to perform Kleene-Star.

Fig. 13. The NWA resulting from performing Kleene-Star on the NWA in Figure 12.
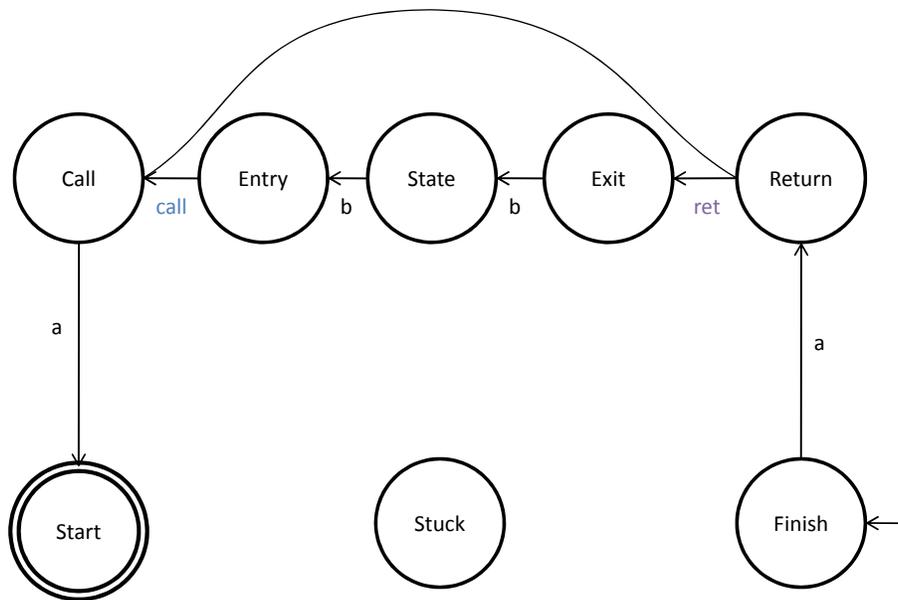
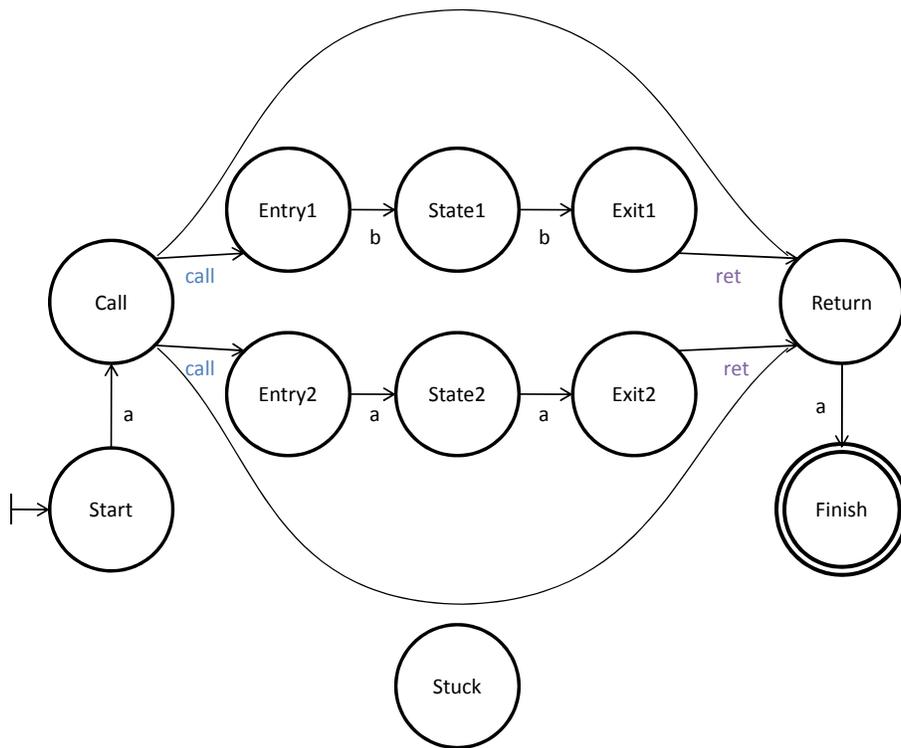**Fig. 14.** The NWA resulting from performing reverse on the NWA in Figure 2.

Fig. 15. Simple nondeterministic NWA.
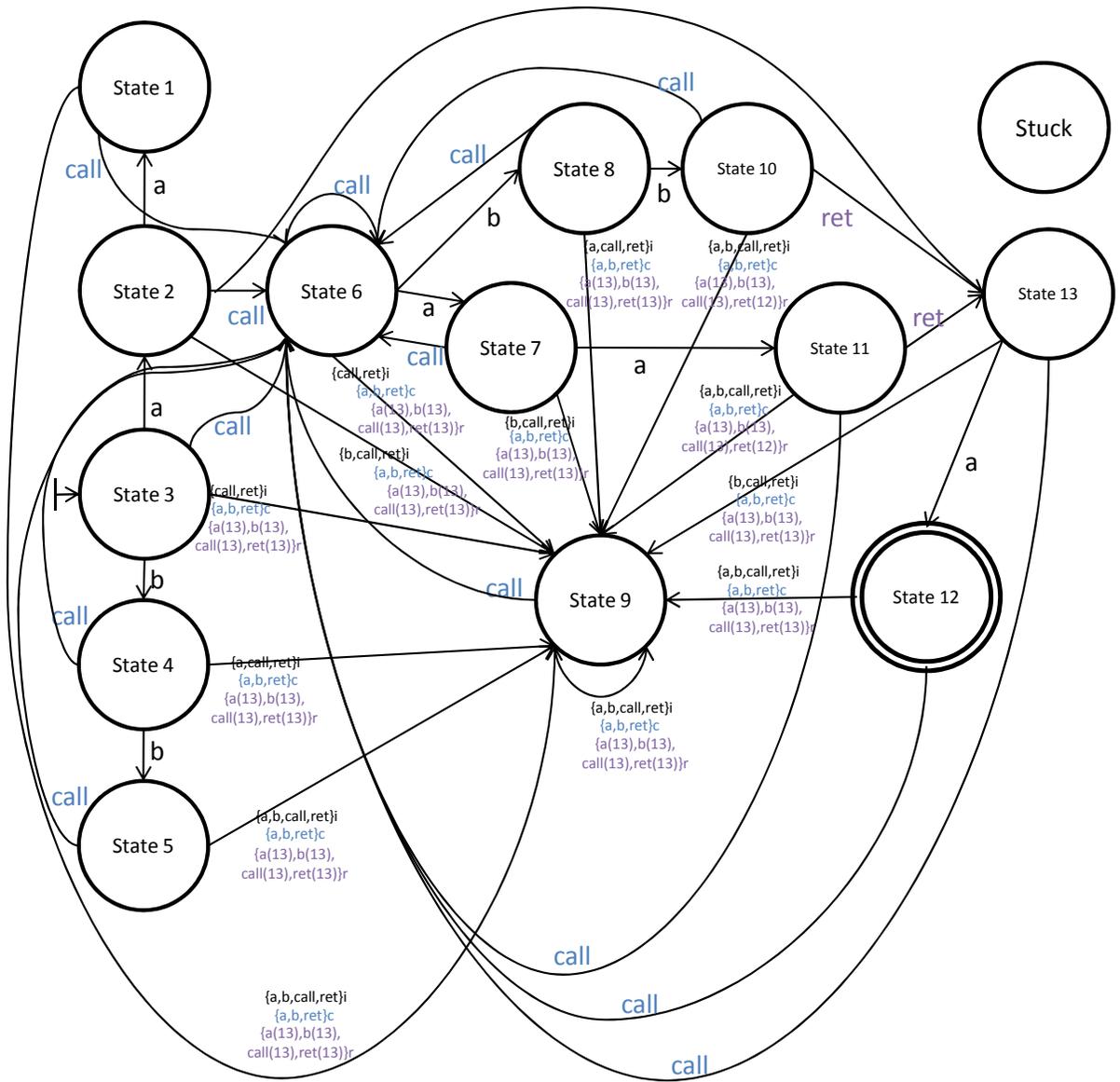
Fig. 16. The NWA resulting from determinizing the NWA in Figure 15. As mentioned in the text, states in the determinized NWA are relations on the states in the original NWA. For example, state 1 above corresponds to the relation {(Start,Stuck), (Call,Stuck), (Entry1,Stuck), (Entry2,Exit2), (State1,Stuck), (State2,Stuck), (Exit1,Stuck), (Exit2,Stuck), (Return,Stuck), (Fishish,Stuck), (Stuck,Stuck)}.
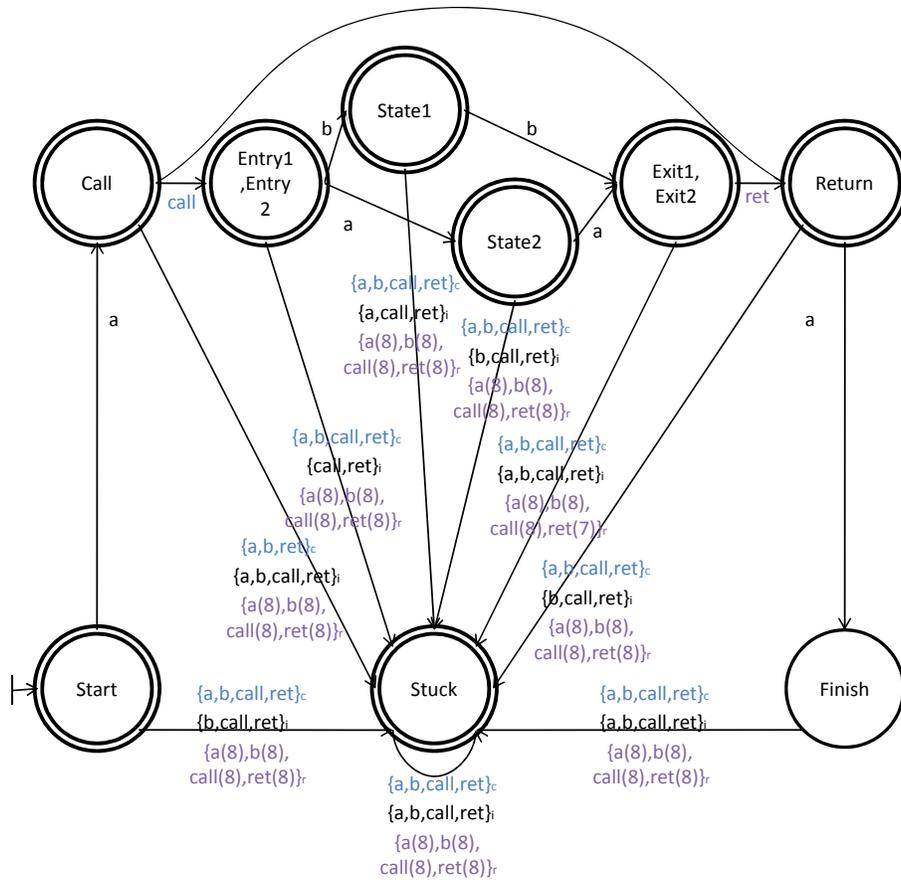
Fig. 17. The complement of the NWA in Figure 15 (the determinization of which is shown in Figure 16).

$$\langle p, main \rangle \hookrightarrow \langle p, q_1 \rangle$$
$$\langle p, q_1 \rangle \hookrightarrow \langle p, c_1 \rangle$$
$$\langle p, c_1 \rangle \hookrightarrow \langle p, e\ r_1 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p, q_2 \rangle$$
$$\langle p, q_2 \rangle \hookrightarrow \langle p, q_3 \rangle$$
$$\langle p, q_3 \rangle \hookrightarrow \langle p, x \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p, \epsilon \rangle$$
$$\langle p, r_1 \rangle \hookrightarrow \langle p, q_4 \rangle$$
$$\langle p, q_4 \rangle \hookrightarrow \langle p, q_5 \rangle$$
$$\langle p, q_5 \rangle \hookrightarrow \langle p, c_2 \rangle$$
$$\langle p, c_2 \rangle \hookrightarrow \langle p, e\ r_2 \rangle$$
$$\langle p, r_2 \rangle \hookrightarrow \langle p, q_6 \rangle$$
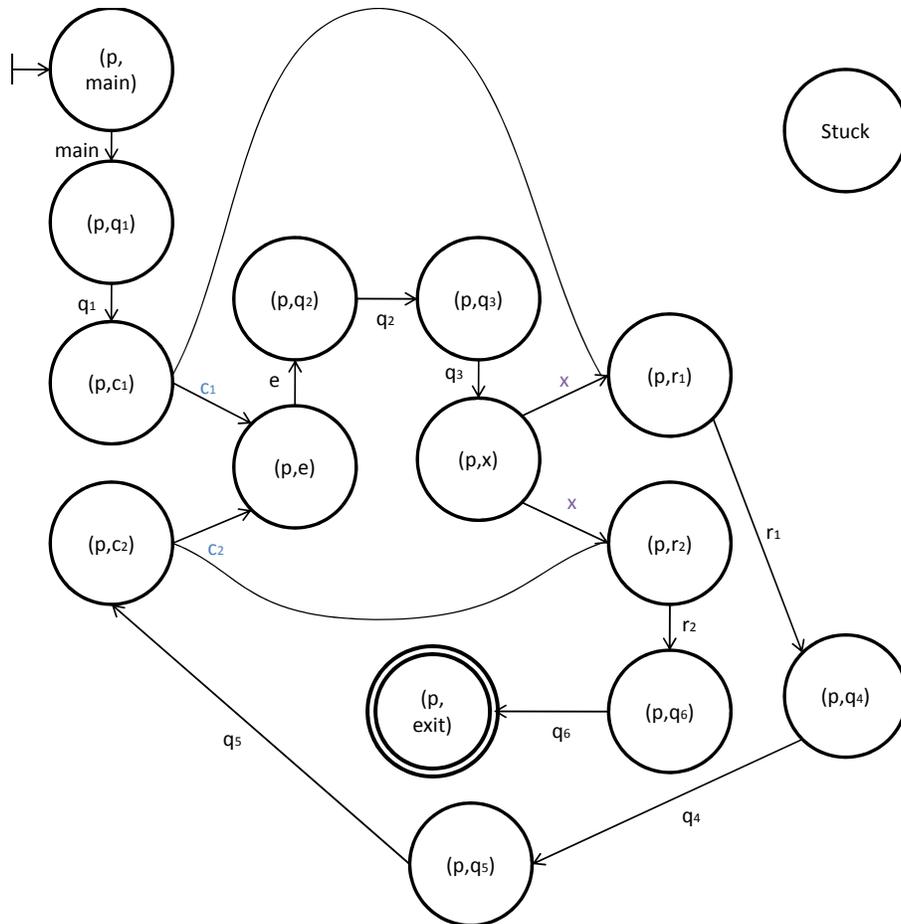$$\langle p, q_6 \rangle \hookrightarrow \langle p, exit \rangle$$

Fig. 18. An example PDS.

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

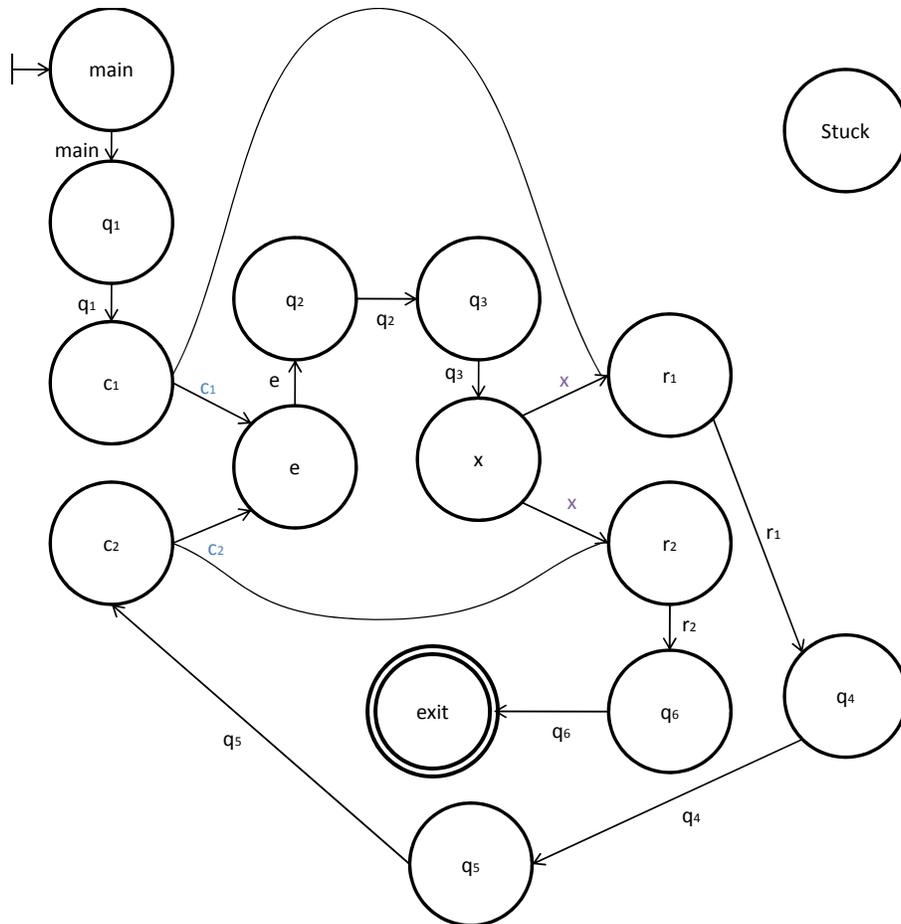**Fig. 20.** An example NWA.

$$\langle p, main \rangle \hookrightarrow \langle p, q_1 \rangle$$
$$\langle p, q_1 \rangle \hookrightarrow \langle p, c_1 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p, q_2 \rangle$$
$$\langle p, q_2 \rangle \hookrightarrow \langle p, q_3 \rangle$$
$$\langle p, q_3 \rangle \hookrightarrow \langle p, x \rangle$$
$$\langle p, r_1 \rangle \hookrightarrow \langle p, q_4 \rangle$$
$$\langle p, q_4 \rangle \hookrightarrow \langle p, q_5 \rangle$$
$$\langle p, q_5 \rangle \hookrightarrow \langle p, c_2 \rangle$$
$$\langle p, r_2 \rangle \hookrightarrow \langle p, q_6 \rangle$$
$$\langle p, q_6 \rangle \hookrightarrow \langle p, exit \rangle$$
$$\langle p, c_1 \rangle \hookrightarrow \langle p, e\ c_1 \rangle$$
$$\langle p, c_2 \rangle \hookrightarrow \langle p, e\ c_2 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p_x, \epsilon \rangle$$
$$\langle p_x, c_1 \rangle \hookrightarrow \langle p, r_1 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p_x, \epsilon \rangle$$
$$\langle p_x, c_2 \rangle \hookrightarrow \langle p, r_2 \rangle$$

Fig. 21. The PDS resulting from converting the NWA shown in Figure 20 into a PDS.

$$\langle p, q_1 \rangle \hookrightarrow \langle p, main \rangle$$
$$\langle p, c_1 \rangle \hookrightarrow \langle p, q_1 \rangle$$
$$\langle p, q_2 \rangle \hookrightarrow \langle p, e \rangle$$
$$\langle p, q_3 \rangle \hookrightarrow \langle p, q_2 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p, q_3 \rangle$$
$$\langle p, q_4 \rangle \hookrightarrow \langle p, r_1 \rangle$$
$$\langle p, q_5 \rangle \hookrightarrow \langle p, q_4 \rangle$$
$$\langle p, c_2 \rangle \hookrightarrow \langle p, q_5 \rangle$$
$$\langle p, q_6 \rangle \hookrightarrow \langle p, r_2 \rangle$$
$$\langle p, exit \rangle \hookrightarrow \langle p, q_6 \rangle$$
$$\langle p, r_1 \rangle \hookrightarrow \langle p, x\ r_1 \rangle$$
$$\langle p, r_2 \rangle \hookrightarrow \langle p, x\ r_2 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p_e, \epsilon \rangle$$
$$\langle p_e, r_1 \rangle \hookrightarrow \langle p, c_1 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p_e, \epsilon \rangle$$
$$\langle p_e, r_2 \rangle \hookrightarrow \langle p, c_2 \rangle$$

Fig. 22. The PDS resulting from converting the NWA shown in Figure 20 into a PDS.

$$\langle p, main \rangle \hookrightarrow \langle p, q_1 \rangle$$
$$\langle p, q_1 \rangle \hookrightarrow \langle p, c_1 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p, q_2 \rangle$$
$$\langle p, q_2 \rangle \hookrightarrow \langle p, q_3 \rangle$$
$$\langle p, q_3 \rangle \hookrightarrow \langle p, x \rangle$$
$$\langle p, r_1 \rangle \hookrightarrow \langle p, q_4 \rangle$$
$$\langle p, q_4 \rangle \hookrightarrow \langle p, q_5 \rangle$$
$$\langle p, q_5 \rangle \hookrightarrow \langle p, c_2 \rangle$$
$$\langle p, r_2 \rangle \hookrightarrow \langle p, q_6 \rangle$$
$$\langle p, q_6 \rangle \hookrightarrow \langle p, exit \rangle$$
$$\langle p, c_1 \rangle \hookrightarrow \langle p, e\ r_1 \rangle$$
$$\langle p, c_2 \rangle \hookrightarrow \langle p, e\ r_2 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p_x, \epsilon \rangle$$
$$\langle p_x, r_1 \rangle \hookrightarrow \langle p, r_1 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p_x, \epsilon \rangle$$
$$\langle p_x, r_2 \rangle \hookrightarrow \langle p, r_2 \rangle$$

Fig. 23. The PDS resulting from converting the NWA shown in Figure 20 into a PDS.

$$\langle p, q_1 \rangle \hookrightarrow \langle p, main \rangle$$
$$\langle p, c_1 \rangle \hookrightarrow \langle p, q_1 \rangle$$
$$\langle p, q_2 \rangle \hookrightarrow \langle p, e \rangle$$
$$\langle p, q_3 \rangle \hookrightarrow \langle p, q_2 \rangle$$
$$\langle p, x \rangle \hookrightarrow \langle p, q_3 \rangle$$
$$\langle p, q_4 \rangle \hookrightarrow \langle p, r_1 \rangle$$
$$\langle p, q_5 \rangle \hookrightarrow \langle p, q_4 \rangle$$
$$\langle p, c_2 \rangle \hookrightarrow \langle p, q_5 \rangle$$
$$\langle p, q_6 \rangle \hookrightarrow \langle p, r_2 \rangle$$
$$\langle p, exit \rangle \hookrightarrow \langle p, q_6 \rangle$$
$$\langle p, r_1 \rangle \hookrightarrow \langle p, x\ c_1 \rangle$$
$$\langle p, r_2 \rangle \hookrightarrow \langle p, x\ c_2 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p_e, \epsilon \rangle$$
$$\langle p_e, c_1 \rangle \hookrightarrow \langle p, c_1 \rangle$$
$$\langle p, e \rangle \hookrightarrow \langle p_e, \epsilon \rangle$$
$$\langle p_e, c_2 \rangle \hookrightarrow \langle p, c_2 \rangle$$

Fig. 24. The PDS resulting from converting the NWA shown in Figure 20 into a PDS.