# Safira: Implementing the Set Algebra for Service Behavior

Kathrin Kaschner

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
`safira@service-technology.org`

**Abstract.** An important property of interacting services is their observable behavior. There exist different formalisms to describe service behavior (e.g. BPEL, BPMN, Petri nets, automata). Based on an extension of automata, in previous work we proposed a compact representation to characterize the behavior of sets of services and introduced a set algebra on it. In this paper, we present with Safira a tool which implements the fundamental set operations for such set of services.

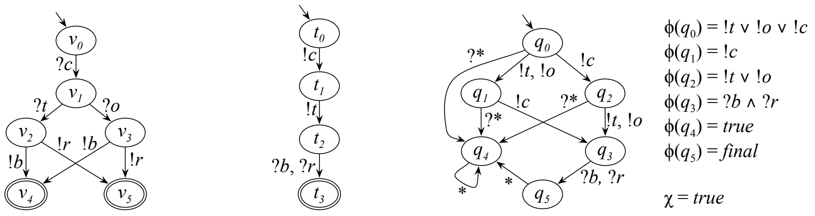
## 1 Introduction

The observable behavior is an important aspect of interacting *services*. For studying correct interaction, the concept of *operating guidelines* [1] was introduced. An operating guideline is an *annotated automaton* which represents the set of all correctly interacting partners of a given service. In [2] the concept of annotated automata has been extended such that fundamental set operations can be realized for sets of services. These operations have a number of useful applications, including reasoning about substitutability, behavioral constraints, and organizing a service registry [2].

This paper is devoted to our tool Safira which implements the fundamental set operations *complement*, *intersection* and *union* for sets of services. Section 2 introduces the basic formalisms. In Sect. 3 we define the algorithms of these set operations and highlight interesting issues of their implementation in Safira. Section 4 gives an overview of how to obtain and use Safira. Section 5 presents experimental results. Finally, we conclude the paper and give directions to future work in Sect. 6.

## 2 Background

To model the behavior of a single service, we use *service automata* [1]. A service automaton is a finite state automaton whose edges are labeled with asynchronous message events. As an example, Fig. 1(a) and Fig. 1(b) show two service automata. In the graphical representation, sending events are preceded by "!" and receiving events are preceded by "?". Multiple labels on one edge are a short hand notation for multiple transitions – each one being labeled with one of the events. Initial states have an incoming arc from nowhere. Final states are double-lined. The service automaton $V$ models a simple vending machine expecting a customer to insert $\underline{c}$oins ($?c$) and to press a button for iced $\underline{t}$ea ($?t$), or $\underline{o}$range juice ($?o$). If enough money has been inserted, the vending machine returns the $\underline{b}$everage



(a) $V$ (b) $T$ (c) $Q$

Fig. 1: The service automata in (a) and (b) model the behavior of a vending machine and a single customer. The extended annotated service automaton in (c) represents a set of customers for the vending machine in (a).

($!b$). If not enough money has been inserted, the customer does not receive the beverage. Instead, the money is returned ($!r$). The service automaton $T$ represents the behavior of a customer who orders iced tea. Note that sending events of $T$ are receiving events of $V$ and vice versa.

For the finite representation of a (possibly infinite) *set* of service automata, we use *extended annotated automata* [2]. They are also finite automata whose edges are labeled with asynchronous message events. To be able to characterize a set of services, we additionally annotate each state with a local Boolean formula $\phi$ and add a global Boolean formula $\chi$ to the automaton. In comparison to service automata, an extended annotated automaton does not have final states. Instead, the proposition *final* in the local formulas may define final states of the represented service automata.

Each service automaton $S$ belonging to the set of service automata characterized by the extended annotated automaton $A$ fulfills the following requirements. First, $S$ is a subautomaton of $A$ (including an initial state). Further, each state $q$ of $S$ satisfies the Boolean formula $\phi$ of its corresponding state in $A$ and the global formula $\chi$ is evaluated to *true*. Thereby, $\phi$ determines which outgoing edges must be present in state $q$ and $\chi$ defines which combinations of states are allowed in $S$. The interested reader is referred to [2].

Figure 1(c) depicts an extended annotated service automaton $Q$ which characterizes the set of all customers of the vending machine $V$. An edge labeled with "*" means that there is a transition for each label of $Q$. "?*" is a placeholder for all receiving events of $Q$. The customer $T$ in Fig. 1(b) is a subautomaton of $Q$. The corresponding states are $(t_0, q_0)$, $(t_1, q_2)$, $(t_2, q_3)$ and $(t_3, q_4)$. Furthermore, all states of $T$ satisfy the local formulas of the corresponding states in $A$. For example, $t_0$ evaluates the formula $!c \vee ?t \vee ?o$ of $q_0$ to *true*, because there is a leaving edge labeled with $!c$. Since the global formula $\chi$ is equal to *true*, there are no additional constraints to the states of $T$. Thus, $T$ is characterized by $Q$. As *all* possible customers of $V$ are represented, $Q$ is also called an *operating guideline* [1] of $V$.

## 3 Set Operations on extended annotated automata

In this section, we introduce the algorithms for the fundamental set operations. This means, given two extended annotated service automata representing sets

$M_1$ and $M_2$ of services, we show how to compute an extended annotated service automaton that represents the *complement* $\overline{M_1}$, the *intersection* $M_1 \cap M_2$ and the *union* $M_1 \cup M_2$. Since the result of each operation is again an extended annotated automaton, arbitrary nested structures are possible.

Due to the page limit we only sketch the algorithms (see [2] for a detailed description), but focus on the interesting details of their actual implementation in our tool Safira. At the end of each subsection, we also discuss the complexity of the operation.

### 3.1 Complement

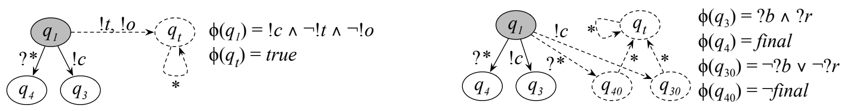
**Theory.** The most challenging of the three operations is the complement. To compute the complement, we first normalize the extended annotated automaton by applying two transformations. Both transformations modify the shape of the extended annotated automaton, but do not change the represented set of service automata. Having a normalized extended annotated automaton the actual complement operation turns out to be very simple.

The first transformation *totalizes* the extended annotated automaton $A$. That means, in each state of the totalized $A$ there exists at least one outgoing edge for each label of $A$. To compute a total extended annotated automaton without changing its semantics, we insert missing edges with label $x$ but explicitly forbid their usage by adding a conjunction with $\neg x$ to the corresponding local Boolean formula. Each inserted edge is connected to a trap state $t$ that contains a self-loop for every label of $A$.

Figure 2(a) illustrates this procedure for state $q_1$ of the extended annotated automaton $Q$ in Fig. 1(c). The set of the labels is $\{!t, !o, !c, ?b, ?r\}$. State $q_1$ has only three outgoing edges labeled with $!c$, $?b$ and $?r$. Therefore, we insert two new edges labeled with $!t$ and $!o$ connecting $q_1$ with the trap state $q_t$ and set the Boolean formula of state $q_1$ to $!c \wedge \neg !t \wedge \neg !o$.

In the second transformation, we *complete* the extended annotated automaton such that for each state $q$ and all labels $y$ the disjunction of the formulas of all states $q'$ with $q \xrightarrow{y} q'$ is equivalent to *true*. In Fig. 2(b), we illustrate the completion of state $q_1$ of the annotated automaton $Q$ in Fig. 1(c). State $q_1$ has only one $!c$-labeled edge leading to state $q_3$. Since the Boolean formula $\phi(q_3) = ?b \wedge ?r$ is not equivalent to *true* we add an edge labeled with $!c$ to a new state $q_{30}$. State $q_{30}$ has for every label an outgoing edge to the trap state $q_t$. The local formula of $q_{30}$ is set to $\neg(?b \wedge ?r)$. Now, the disjunction of the local formulas of all $?c$-successors is equivalent to *true*. The remaining successor states of state $q_1$ are treated likewise. To avoid a change in the semantics of $Q$ by inserting the new states $q_{30}$ and $q_{40}$ we explicitly forbid their usage by setting the global formula to $\chi = true \wedge \neg q_{30} \wedge \neg q_{40}$. That means, as soon as a service automaton $S$ covers one of these new states, the global formula is evaluated to *false*. Consequently, $S$ is correctly classified as a non represented service automaton.

After applying the two transformations to an extended annotated automaton $A$, every service automaton with the same interface as $A$ is a subautomaton of $A$. Only the global formula $\chi$ decides wether or not the service is represented by $A$.



(a) making total (b) making complete

Fig. 2: In (a), the procedure of making the automaton total is shown for state $q_1$ of $Q$ (from Fig. 1(c)). (b) illustrates the procedure of completion for the same state. The dashed parts are those that are inserted in the course of a transformation.

Consequently, the complement operation for a total and complete automaton turns out to be very simple: only the global formula $\chi$ has to be negated.

Figure 3(b) depicts the extended annotated automaton $\overline{Q}$ that represents the complement set of service automata represented by $Q$ in Fig. 1(c). Figure 3(a) shows a service automaton $T'$ which is represented by $\overline{Q}$. As both the button for iced tea and orange juice are pressed, it is not a customer of $Q$.

**Implementation.** Overall, Safira follows the procedure described above. As both transformations can be executed independently for every node, we store the nodes in a list, which is traversed. The totalization of the automaton can easily be done by checking the outgoing edges for every node. In contrast, the completion is more complicated, as boolean formulas have to be evaluated. Therefore, we integrated the open-source SAT solver Minisat[1] as a library into our tool Safira.

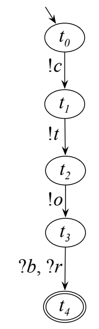
To use Minisat, every question concerning Boolean formulas must be converted into a satisfaction problem. Thus, to proceed with the completion, we negate the disjunction $f$ of the formulas of all $y$-successors for each node $q$ and ask Minisat, wether the formula $\neg f$ is satisfiable. If the answer is 'yes', then $f$ is not equivalent to *true* and an additional $y$-edge for $q$ is inserted.

Although, the satisfaction problem is NP-complete, the experimental results in the next section show that the complement can be computed efficiently. As there usually are only a small number of labels and the length of the formulas is rather small, the satisfaction problems we formulate for the complement generation are not challenging Minisat.

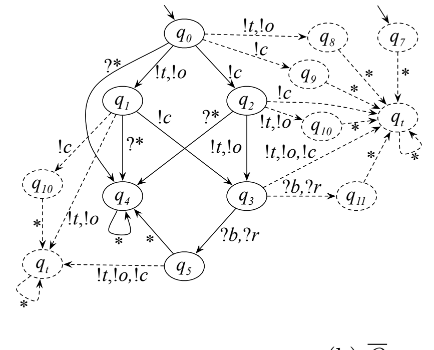
Our experimental results in Sect. 5 show that the complement of an extended annotated automaton can be computed. The applications mentioned in the first section of this paper require a further use of the resulting automaton. Thus, Safira also implements an optimization algorithm aiming at reducing the size of the result.

The main idea of this optimization is to merge added states with the same Boolean formula. This can be done, because all successors of each added state lead to the same state – the trap state $q_t$. To decide if a newly computed state can be merged with another state, we build a special decision tree during the completion operation. Each inner node of the tree contains an assignment over the labels of the automaton and has two outgoing edges labeled with 'yes' and 'no'. Each leaf represents a state, which was already added to the automaton by the completion operation. Figure 4 depicts the decision tree which was built during

[1] See the website of Minisat at `http://minisat.se`.



(a) $T'$



$$\phi(q_0) = !t \vee !o \vee !c$$
$$\phi(q_1) = !c \wedge \neg !t \wedge \neg !o$$
$$\phi(q_2) = (!t \vee !o) \wedge \neg !c$$
$$\phi(q_3) = ?b \wedge ?r \wedge \neg !c \wedge \neg !t \wedge \neg !o$$
$$\phi(q_4) = true$$
$$\phi(q_5) = final \wedge \neg !c \wedge \neg !t \wedge \neg !o$$
$$\phi(q_t) = true$$
$$\phi(q_7) = \neg !c \wedge \neg !t \wedge \neg !o$$
$$\phi(q_8) = \neg !c \vee !t \vee !o$$
$$\phi(q_9) = (\neg !t \wedge \neg !o) \vee !c$$
$$\phi(q_{10}) = \neg ?b \vee \neg ?r \vee !c \vee !t \wedge \vee !o$$
$$\phi(q_{11}) = \neg final \vee !c \vee !t \vee !o$$
$$\chi = q_t \vee q_7 \vee q_8 \vee q_9 \vee q_{10} \vee q_{11}$$

(b) $\overline{Q}$

Fig. 3: The extended annotated service automaton $\overline{Q}$ in (b) is the Complement of $Q$ in Fig. 1(c). For reducing the number of edge crossings, we depicted two copies of state $q_{10}$ and the trap state $q_t$. (a) depicts a service automaton $T'$, represented by $\overline{Q}$.

the complement generation for the vending machine $Q$. Each leaf represents one of the states $q_7, ..., q_{11}$ which was inserted into the automaton by the completion operation (cf. Figure 3(b)).

Consider a leaf $l$ representing state $p$ with its local Boolean formula $g$. The decision tree is constructed such that the assignment $\beta$ of each inner node $n$ along the path leading to $l$ holds the following condition: $\beta$ is a satisfying assignment of $g$ if the outgoing edge of $n$ leading to $l$ is labeled with 'yes' and is a non-satisfying assignment, otherwise. As an example, see Figure 4. The assignments $\beta_0$ and $\beta_1$ both do not satisfy the Boolean formula of state $q_7$.

At the beginning of the completion operation we start with an empty decision tree. When we insert the first new state $q_0$ to the automaton, we also add $q_0$ to the decision tree. Thus, at this moment the decision tree contains only a leaf which represents $q_0$. Then, we proceed as follows: Suppose, we want to connect an existing state of the automaton with a new state $q$ by edge $e$. Let $f$ be the local Boolean formula of $q$. To find out if there already exists a state which is annotated with the same Boolean formula, we traverse the decision tree. At each inner node $n$, we check if the assignment $\beta$ of $n$ satisfies $f$. If this is the case, we follow the outgoing edge labeled with 'yes'. Otherwise, we follow the outgoing edge labeled with 'no'. Arriving at a leaf representing a state $p$, we check if its formula $h$ is equivalent to $f$. If the answer is 'yes', no new state has to be inserted. Instead, the new edge $e$ is directed to the existing state $p$. Otherwise, there is an assignment $\alpha$ that satisfies $f$, but not $h$ or vice versa. In the decision tree, at the place of the leaf representing $q$, we insert a new inner node, containing $\alpha$. Its outgoing edges are then directed to two leafs representing $q$ and $p$, respectively.

In this manner, the decision tree is built successively during the completion operation. Note that for the totalization of the automaton, no additional nodes are necessary. Therefore the decision tree is not needed during this procedure.

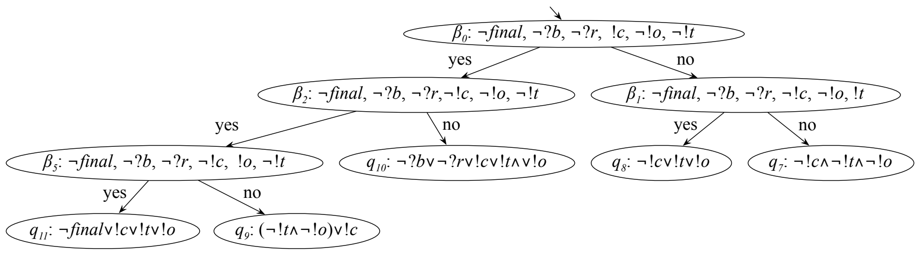

β0: ¬final, ¬?b, ¬?r, !c, ¬!o, ¬!t
yes
no
β2: ¬final, ¬?b, ¬?r,¬!c, ¬!o, ¬!t
β1: ¬final, ¬?b, ¬?r, ¬!c, ¬!o, !t
yes
no
yes
no
β5: ¬final, ¬?b, ¬?r, ¬!c, !o, ¬!t
q10: ¬?b∨¬?r∨!c∨!t∧∨!o
q8: ¬!c∨!t∨!o
q7: ¬!c∧¬!t∧¬!o
yes
no
q11: ¬final∨!c∨!t∨!o
q9: (¬!t∧¬!o)∨!c

Fig. 4: The decision tree of $\overline{Q}$.

**Complexity.** Due to our optimization, for each state in the given automaton at most one new state (with the negated formula) is added. Thus, in the worst case the size complexity for completion is linear in the number of the states. The time complexity for the completion depends mainly of the shape of the decision tree. Assuming the decision tree is a balanced tree, the automaton is completed in $\mathcal{O}(n \cdot log\, n \cdot l)$, whereas $n$ is the number of the states and $l$ the number of labels. For the transformation to a total automaton, the worst case complexity for both space and time is $\mathcal{O}(n \cdot l)$.

## 3.2 Intersection

**Theory.** The idea of implementing the intersection of two extended annotated automata is to construct the product automaton known from classical automata theory. A product automaton implements the idea that both constituents run synchronized – in every step executing transitions with the same label. The states in the product automaton are annotated with the conjunction of the local Boolean formulas of the constituents [3]. The global formulas are connected by $\wedge$.

**Implementation.** The algorithm for the intersection is well known from classical automata theory. It was just adapted to the specific characteristics of extended annotated automata.

**Complexity.** The size of the resulting automaton mainly depends on the degree of similarity of the given automata. The worst case complexity regarding both space and time of the intersection algorithm is in the product of the number of states of the two involved automata.

## 3.3 Union

**Theory.** Given the two operations of intersection and complement from the previous subsections, the implementation of union is trivial by using De Morgan's rule: $M \cup N = \overline{\overline{M} \cap \overline{N}}$.

**Implementation.** We already proofed that the product of two total and complete automata is again total and complete [2]. Thus, we do not need to transform the product automaton of $\overline{M} \cap \overline{N}$ to generate the complement. Instead, we only have to negate the global formula of the automaton representing $\overline{M} \cap \overline{N}$.

**Complexity.** The complexity of the union results by the complexity of the underlying operations complement and intersection.

## 4 Obtaining and using Safira

Safira is free software[2] and can be downloaded at `http://service-technology.org/safira`. It is written in C++ and uses the GNU build system to compile the binary. Thus, it is available for a several platforms, including Linux, Microsoft Windows (using cygwin) and Mac OS X.

Safira is a command-line tool implementing the following use cases. We assume that two extended annotated automata are given in files "coffeeVendor.og" and "juiceVendor.og".

– Complement. Call Safira with
```
safira -opdf -t --complement coffeeVendor.og
```
– Intersection. Call Safira with
```
safira -opdf -t --intersection coffeeVendor.og juiceVendor.og
```
– Union. Call Safira with
```
safira -opdf -t --union coffeeVendor.og juiceVendor.og
```

Option `-t` measures the time for computing the result and option `-o` triggers the output of Safira. In the examples above, a PDF file is generated which shows the graphical representation of the resulting extended annotated automaton. For a full description of the command line parameters, type `safira --help`.

## 5 Experimental results

In this section, we study how the algorithms for computing the complement automaton scale in practice. We do not examine the intersection, because the algorithm is of no issue and the size of the result mainly depends on the degree of similarity of the given automaton.

To generate the input automata for our experiments , we used the tool Wendy[3]. Wendy computes the operating guideline of a given service automaton. The examples are industrial service models and have been extracted from real BPEL processes using the tool BPEL2oWFN[4] (except the first one).

We executed the examples on a computer with a 1.83 GHz Intel Core 2 Duo processor and 2 GB of memory. The results are listed in Tab. 1. As expected, the number of states of the complement automaton computed by the optimization algorithm $A_2$ (usage of decision tree) is significantly smaller than the number of states of the automaton obtained by the basic algorithm $A_1$ (no usage of decision tree). The memory usage shows a similar result. There is one case (service "Car Analysis") in which the results concerning the memory usage diverge. This can be explained by the structure of the resulting decision tree. Compared with the decision trees generated for the other automata, this tree is not well balanced. Moreover, $A_2$ consumes more time than $A_1$. This can be explained by the additional data structure – the decision tree – which has to be built up and is traversed when a new state of the complement automaton is

[2] GNU Affero Public License Version 3, `http://gnu.org/licenses/agpl.html`.
[3] Available at `http://service-technology.org/wendy`.
[4] Available at `http://service-technology.org/bpel2owfn`.

Table 1: Experimental results showing the number of state of the complement automaton, the execution time and the memory usage for both the optimization algorithm $A_2$ and the basic algorithm $A_1$.

| service | automaton | states of complement | | time [sec] | | memory [MB] | |
|---|---|---|---|---|---|---|---|
| | | $A_1$ | $A_2$ | $A_1$ | $A_2$ | $A_1$ | $A_2$ |
| Vending Machine | 6 | 23 | 12 | 0.0 | 0.0 | 0.5 | 0.5 |
| Purchase Order | 169 | 887 | 338 | 0.1 | 0.1 | 1.3 | 0.8 |
| Car Analysis | 733 | 3,931 | 987 | 0.6 | 6.51 | 1.9 | 2.9 |
| SMTP | 3,307 | 19,995 | 3,460 | 2.9 | 29.4 | 10.9 | 10.6 |
| Quotation1 | 7,937 | 61,571 | 14,594 | 14.8 | 208.0 | 24.7 | 15.5 |
| Quotation2 | 11,265 | 88,323 | 22,539 | 19.2 | 286.1 | 132.8 | 36.1 |

calculated. The overall time, however, is not a crucial measure because in our application settings the complement automaton is usually calculated at a point in time in which the calculation time is not an issue. The number of states of the complement automaton is more critical. In most settings, the complement automaton is used as an input for complex, nested operations such as union and intersection, for instance.

## 6 Conclusion and Future Work

In this paper, we presented the tool Safira implementing the set operations complement, intersection, and union. With the help of experimental results, we demonstrated that the algorithms used for the calculation of the complement scale well. The implementation of the set operations is an important step towards their applications: substitutability, behavioral constraints, and organizing a service registry. For their realization, the decision problem membership $S \in M$ for a service $S$ and a set $M$ of services, and also the emptiness check $M = \emptyset$ still need to be implemented. For both, we already introduced the theory [2]. The complexity for the membership problem is linear in the size of $S$ so that we do not expect problems during the implementation. In contrast, we could proof that the emptiness check is NP-complete. Consequently, we have to find heuristics to decide this problem efficiently in practice.

## References

1. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. LNCS 4546, Springer (2007) 321–341
2. Kaschner, K., Wolf, K.: Set algebra for service behavior: Applications and constructions. In: BPM 2009. LNCS 5701, Springer (2009) 193–210
3. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007. LNCS 4714, Springer (2007) 271–287