Nevertrace Claims for Model Checking

Zhe Chen and Gilles Motet

LAAS-CNRS, INSA, Université de Toulouse
135 Avenue de Rangueil, 31077 Toulouse, France
{zchen, gilles.motet}@insa-toulouse.fr

Abstract. In this paper, we propose the `nevertrace` claim, which is a new construct for specifying the correctness properties that either finite or infinite *execution traces* (i.e., *sequences of transitions*) that should *never* occur. In semantics, it is neither similar to `never` claim and `trace` assertion, nor a simple combination of them. Furthermore, the theoretical foundation for checking `nevertrace` claims, namely the Asynchronous-Composition Büchi Automaton Control System (AC-BAC System), is proposed. The major contributions of the `nevertrace` claim include: a powerful construct for formalizing properties related to transitions and their labels, and a way for reducing the state space in design stage.

1 Introduction

The SPIN (Simple Promela INterpreter) model checker is an automated tool for verifying the correctness of asynchronous distributed software models [1,2,3]. System models and correctness properties to be verified are both described in Promela (Process Meta Language). This paper is based on SPIN Version 5.2.5, released on 17th April 2010.

Promela supports various constructs for formalizing different classes of properties. The most powerful constructs are the `never` claim, the `trace` and `notrace` assertion. The `never` claim specifies the properties on sequences of *states*, while the `trace` and `notrace` assertion specifies the properties on sequences of *transitions of simple channel operations*, i.e., simple send and receive operations on message channels, where a *transition* is a statement between two states. However, we observed that the existing constructs cannot specify the properties on *full sequences of transitions*, apart from the transitions of simple channel operations.

In this paper, we propose the `nevertrace` claim, which is a new claim construct for specifying correctness properties related to *all types of transitions* and their *labels*. A `nevertrace` claim specifies the properties that either finite or infinite *execution traces* (i.e., *sequences of transitions*) that should *never* occur. A `nevertrace` claim could be *nondeterministic*, and performed at *every single execution step* of the system.

Literally, it seems that the `nevertrace` claim combines the `never` claim and the `trace` assertion. However, we will show that, in semantics, it is neither similar to any of them, nor a simple combination of them.

The major contributions of this construct include:

First, the `nevertrace` claim provides a powerful construct for formalizing properties related to transitions and their labels. Furthermore, the `nevertrace` claim can be used to express the semantics of some existing constructs in Promela.

Second, the `nevertrace` claim provides a way for reducing the state space in design stage. We observed that variables are always used for two objectives: functional computation, or implicitly recording the execution trace for verification. The `nevertrace` claim can reduce the usage of variables for marking the execution trace. The decreased number of variables can reduce the state space.

The paper is organized as follows. In Section 2, the existing constructs in Promela are recalled to facilitate further discussion and comparison. In Section 3, the `nevertrace` claim is proposed and illustrated by example. The theoretical foundation for checking `nevertrace` claims, namely the Asynchronous-Composition Büchi Automaton Control System (AC-BAC System), is presented in Section 4. Then in Section 5 we show how to express some constructs in Promela using `nevertrace` claims. We discuss related work in Section 6 and conclude in Section 7.

To illustrate some constructs in Promela and our new construct in the sequel, we use a simple Promela model (see Listing 1.1) as an example. This model contains two channels (`c2s` and `s2c`), three processes (two clients and a server) and four types of messages. Client 1 can send `msg1` through the channel `c2s`, and receive `ack1` from the channel `s2c`, and repeat the procedure infinitely. Client 2 does the similar. There are nine labels in the model, e.g., `again` at Line 7, `c2srmsg` at Line 23. The variable `x` counts the number of messages in the channel `c2s`, thus is used for functional computation.

Listing 1.1. A Promela Model of Client and Server

```
1  mtype = {msg1, msg2, ack1, ack2};
2  chan c2s = [2] of {mtype};
3  chan s2c = [0] of {mtype};
4  int x = 0;
5
6  active proctype client1() {
7  again:
8  c2ssmsg1:   c2s!msg1;
9  x_inc:      x = x+1;
10             s2c?ack1;
11             goto again;
12 }
13
14 active proctype client2() {
15 again:
16 c2ssmsg2:   c2s!msg2;
17 x_inc:      x = x+1;
18             s2c?ack2;
19             goto again;
20 }
```

```
21
22 active proctype server() {
23 c2srmsg:  do
24           :: c2s?msg1;
25 x_dec1:      x = x-1;
26              s2c!ack1;
27           :: c2s?msg2;
28 x_dec2:      x = x-1;
29              s2c!ack2;
30           od;
31 }
```

2 Constructs for Formalizing Properties in SPIN

Promela supports the following constructs for formalizing correctness properties, of which numerous examples could be found in the monographs [3,4].

- Basic assertions. A *basic assertion* is of the form `assert(expression)`.
- End-state labels. Every label name that starts with the prefix `end` is an *end-state label*.
- Progress-state labels. Every label name that starts with the prefix `progress` is a *progress-state label*.
- Accept-state labels. Every label name that starts with the prefix `accept` is an *accept-state label*.
- Never claims. A `never` claim specifies either finite or infinite system behavior that should *never* occur.
- Trace assertions. A `trace` assertion specifies properties about sequences of simple send and receive operations on message channels.
- Notrace assertions. A `notrace` assertion specifies the opposite of a `trace` assertion, but uses the same syntax.

Note that `never` claims could be nondeterministic, whereas `trace` and `notrace` assertions must be deterministic. Furthermore, Promela also supports Linear Temporal Logic (LTL) formulas, which are converted into `never` claims for verification [5].

To facilitate the comparison in the sequel, we recall first the semantics of `notrace` assertions through a simple example.

The following `notrace` assertion specifies the property that there should *not* exist a sequence of send operations on the channel `c2s` that contains two consecutive `c2s!msg1`. Note that, for `notrace` assertions, an error is reported, if the assertion is matched completely. Note that only the send operations on the channel `c2s` are within the scope of this check, and other statements are ignored.

```
notrace {  /* containing two consecutive c2s!msg1 */
S0:
     if
```

```
       :: c2s!msg1 -> goto S1;
       :: c2s!msg2 -> goto S0;
     fi;
S1:
     if
       :: c2s!msg1;
       :: c2s!msg2 -> goto S0;
     fi;
/* S2 */
}
```

The model in Listing 1.1 violates this assertion, since sequences containing two consecutive `c2s!msg1` are feasible. These sequences cause the termination of the `notrace` assertion.

We observed that `notrace` assertions could be also constructed from LTL formulas. The procedure reuses the technique that translates LTL formulas into `never` claims [5]. Note that `never` claims specify Nondeterministic Finite Automata (NFA). Thus, an additional step of manually determinizing NFA [6] is needed, since a `notrace` assertion must be deterministic.

For this example, we can convert the LTL formula `<>(c2s!msg1 -> X c2s!msg1)` into an NFA in Fig. 1. Note that the condition statement `(1)` (or `true`) was replaced by all send operations on the channel `c2s` in the model. To obtain a deterministic `notrace` assertion, we have to manually determinize the NFA to a Deterministic Finite Automaton (DFA) and minimize the DFA. The result is shown in Fig. 2. Finally, we can write the above assertion according to the established DFA.

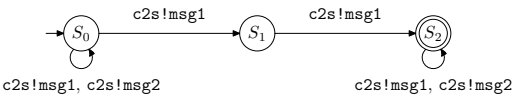

Fig. 1. The Nondeterministic Automaton of `notrace` Assertion

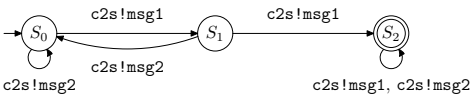

Fig. 2. The Deterministic Automaton of `notrace` Assertion

3 Nevertrace Claims

In this section, we will propose the `nevertrace` claim. A `nevertrace` claim may contain only control-flow constructs and *transition expressions*, while the major ingredient of transition expression is the *label expression*. Thus, we present three new constructs.

3.1 Label Expressions

In a program or model (e.g., Promela model), some statements have labels. A statement may have several labels. A *label* name in a model contains only the following characters: digits (`0` to `9`), letters (`a` to `z`, `A` to `Z`), and underscore (_). We assume that all the unlabeled statements have the empty string (denoted by ϵ) as its default label name.

A *label expression* is a regular expression for matching label names. The label expression reuses a subset of the characters of POSIX-extended regular expressions[1], and adds some new symbols. The special characters are listed in Table 1, where the additional symbols in the bottom part are not included in POSIX-extended regular expressions.

Note that all characters other than the special characters listed in Table 1, including digits, letters and underscore, match themselves. Also note that the interpretation of these symbols is over the restricted alphabet [0-9A-Za-z_].

Symbol	Meaning over the Alphabet [0-9A-Za-z_]
.	A *dot* matches any single character.
( )	*Parentheses* group a series of patterns into a new pattern.
[ ]	A *character class* matches any character within the brackets. If the first character is a *circumflex* [^], it matches any character except the ones within the brackets. A *dash* inside the brackets indicates a character range, e.g., [a-d] means [abcd], [^a-d] means [0-9A-Ze-z_].
{ }	If the braces contain one number, indicate the exact number of times the previous pattern can match. While two numbers indicate the minimum and maximum number of times.
*	A *Kleene star* matches zero or more copies of the previous pattern.
+	A *positive closure* matches one or more copies of the previous pattern.
?	A *question mark* matches zero or one copy of the previous pattern.
|	An *alternation* operator matches either the previous pattern or the following pattern.
	Additional Symbols
(^)	If the first character is a *circumflex* (^), it matches any string except the ones expressed by the expression within the parentheses.
#	A *hash mark* matches any string over the alphabet, i.e., [0-9A-Za-z_]*.

Table 1. Special Characters in Label Expressions

[1] POSIX-extended regular expressions are widely used in Unix applications.

For example, in Listing 1.1, the label expression `c2ssmsg#` matches the labels starting with `c2ssmsg`, i.e., `c2ssmsg1` and `c2ssmsg2`. The expression `(^c2ss#)` matches all labels in the model other than `c2ssmsg1` and `c2ssmsg2`. The empty string ϵ could be matched by `a{0}`, where `a` could be other letters or any digit.

Let us consider the complexity of deciding whether a label name is matched by a label expression. It is well known that a Deterministic Finite Automaton (DFA) can be effectively constructed from a regular expression [6]. In the label expression, most of the special characters are reused from regular expressions. It is easy to see, the additional characters also produce DFA's. For example, `(^)` means constructing the complementation of a regular language (and its DFA) [6]. Therefore, a DFA can be also effectively constructed from a label expression.

It is also well known that the membership problem for regular languages (accepted by DFA's) can be decided in linear time [6]. Therefore, the membership problem for label expressions can be also decided in linear time. This means, given a label name l of length n, whether l is matched by a label expression can be decided in linear time $O(n)$. This shows that label expressions are feasible in practice.

3.2 Transition Expressions

An *atomic transition expression* is of the form `procname[pid]$lblexp`, and may take three arguments. The first optional argument is the name of a previously declared `proctype procname`. The second optional argument is an expression enclosed in brackets, which provides the process identity number `pid` of an active process. The third required argument `lblexp` is a *label expression*, which matches a set of label names in the model. There must be a symbol $ between the second and the third arguments.

Given a transition and its labels, an atomic transition expression `procname [pid]$lblexp` matches the transition (i.e., returns *true*), if the transition belongs to the process `procname[pid]`, and *at least one of the labels* is matched by the label expression `lblexp`. We should notice that the first two arguments are only used to restrict the application domain of the label expression.

A *transition expression* contains one or more atomic transition expressions connected by propositional logic connectives. It can be defined in Backus-Naur Form as follows:

```
t ::= a | (!t) | (t && t) | (t || t) | (t -> t)
```

where `t` is transition expression, and `a` is atomic transition expression.

Given a transition and its labels, a transition expression matches the transition (i.e., returns *true*), if the propositional logic formula is evaluated to *true* according to the values of its atomic transition expressions. Note that the transition expression is side effect free. That is, it does not generate new system behavior, just like condition statements.

For example, in Listing 1.1, the (atomic) transition expression `client1[0] $c2ssmsg#` matches all transitions that have a label starting with `c2ssmsg` in

the process 0 of type `client1`, i.e., the statement with label `c2ssmsg1` at Line 8. The transition expression `(client2[1]$(c2s#)) && $again#` matches all transitions that have a label starting with `c2s` and a label starting with `again`, in the process 1 of type `client2`, i.e., the statement with two labels `again` and `c2ssmsg2` at Line 16.

In an atomic transition expression, the second arguments (together with the brackets) can be omitted, if there is only one active process of the type specified by the first argument, or the transition expression is imposed on all active processes of the type. The first and the second arguments (together with the brackets) can be both omitted, if the transition expression is imposed on all active processes. But note that the symbol $ cannot be omitted in any case.

For example, in Listing 1.1, the transition expression `client1[0]$c2ssmsg#` is equivalent to `client1$c2ssmsg#`. The transition expression `$c2ssmsg#` matches the transitions that have a label starting with `c2ssmsg` in all active processes, i.e., the statements at Lines 8 and 16.

The reader may find that the atomic transition expression is syntactically similar to the remote label reference in Promela, except the symbol $. Note that, at first, there are two superficial differences: (1) the first argument of the remote label reference cannot be omitted, (2) the third argument of the remote label reference should be an existing label name, rather than a label expression. Furthermore, we will show later that they have different semantics in their corresponding claims.

Let us consider the complexity of deciding whether a transition is matched by a transition expression.

For *atomic transition expressions*, we showed that the membership problem for label expressions (the third argument) can be decided in linear time $O(n)$. As mentioned, the first two arguments only check the owner of the label, so do not affect the complexity. Thus, given a transition with several labels of the total length n, whether it is matched by an atomic transition expression can be decided in linear time $O(n)$. That is, the membership problem for atomic transition expressions can be also decided in linear time $O(n)$.

Suppose a *transition expression* has i atomic transition expressions and j logic connectives, then the membership problem can be decided in $i \cdot O(n) + O(j)$ time. Since i, j are constants for a given transition expression, the membership problem can be decided in linear time $O(n)$. This shows that transition expressions are feasible in practice.

3.3 Nevertrace Claims

A `nevertrace` claim specifies the properties that either finite or infinite *execution traces* (i.e., *sequences of transitions*) that should *never* occur. A `nevertrace` claim could be *nondeterministic*, and performed at *every single execution step* of the system.

A `nevertrace` claim may contain only control-flow constructs and *transition expressions*. A `nevertrace` claim can contain end-state, progress-state and accept-state labels with the usual interpretation in `never` claims. Therefore, it

looks like a `never` claim, except the keyword `nevertrace` and allowing transition expressions instead of condition statements.

An example of `nevertrace` claim for the model in Listing 1.1 is as follows:

```
nevertrace {  /* ![]( $x_inc -> <> $x_dec#) */
T0_init:
    if
    :: (!$x_dec# && $x_inc) -> goto accept_S4
    :: $# -> goto T0_init
    fi;
accept_S4:
    if
    :: (!$x_dec#) -> goto accept_S4
    fi;
}
```

In the example, the claim specifies the property that increasing `x` always leads to decrease `x` later. In other words, if one of the transitions labeled `x_inc` is executed, then one of the transitions that have a label starting with `x_dec` will be executed in the future. By the way, if we replace `$x_inc` by `$x_i#nc#`, or replace `$x_dec#` by `server[2]$x_dec#`, for this model, the resulting claim is equivalent to the above one.

A `nevertrace` claim is performed as follows, starting from the initial system state. One transition expression of the claim process is executed each time after the system executed a transition. If the transition expression matches the *last executed transition*, then it is evaluated to *true*, and the claim moves to one of the next possible statements. If the claim gets stuck, then this means that the undesirable behavior cannot be matched. Therefore, no error is reported.

For a rendezvous communication, the system executes an atomic event in which two primitive transitions are actually executed at a time, one send operation and one receive operation. In this case, we assume that the send operation is executed before the receive operation in an atomic rendezvous event.

An error is reported, if the full behavior specified could be matched by any feasible execution. The violation can be caught as termination of the claim, or an acceptance cycle, just like `never` claims. Note that all the transitions of the model are within the scope of the check.

In the example, it is easy to see that there exists no violation, since the `nevertrace` claim cannot be matched completely.

Note that it is hard to express this property using existing constructs in Promela. For example, `trace` or `notrace` assertions are not capable of expressing this property, since they can only specify properties on simple channel operations. The three types of special labels do not have this power neither.

Fortunately, `never` claims can express this property by introducing new variables to implicitly record the information about execution traces. For instance, we introduce two boolean variables `a`, `b`. After each statement labeled `x_inc`, we add the statements "`a=1; a=0;`", which let `a` be 1 once. After each statement labeled `x_dec#`, we add the statements "`b=1; b=0;`", which let `b` be 1

once. Then the property can be expressed as LTL formula `[](a -> <>b)`. The negation of this formula can be converted into a `never` claim that specifies the required property.

However, please note that the additional variables quadruple the state space (different combinations of `a` and `b`), and make the program malformed and harder to read. In contrast, the `nevertrace` claim is more economic, since it takes full advantage of the transition information (e.g., control-flow states and their labels) that is already tracked as part of the state space in the verification mode of SPIN.

An interesting thing is that `nevertrace` claims could be also converted from LTL formulas. In the example, a `never` claim can be generated from the LTL formula `![]($x_inc -> <>$x_dec#)` by SPIN. Then we can obtain the `nevertrace` claim above by replacing the condition statement `(1)` or `true` by `$#` matching all transitions. This fact can facilitate the use of `nevertrace` claims in practice.

Finally, let us consider the syntax definition of `nevertrace` claims. There are various ways to modify the grammar of Promela to take into account the `nevertrace` claim. For example, we can add the following productions into the grammar of Promela.

```
unit :  nevertrace ;

nevertrace : NEVERTRACE body ;

expr    :  PNAME '[' expr ']' '$' expr_label
    | PNAME '$' expr_label
    | '$' expr_label
    ;
```

Note that `unit`, `body` and `expr` are existing nonterminals in the grammar of Promela, thus we only append the new productions. The productions for the nonterminal `expr_label` are omitted, since its syntax is clearly specified in Table 1.

4 Theoretical Foundation for Checking Nevertrace Claims

In this section, we propose the theory of *asynchronous-composition Büchi automaton control systems*. Then we will show the connection between the theory and the checking of `nevertrace` claims by example.

4.1 The Asynchronous Composition of Büchi Automata

At first, we recall the classic definition of Büchi automata [7,8].

Definition 1. *A* (nondeterministic) Büchi automaton *(simply automaton) is a tuple* $A = (Q, \Sigma, \delta, q_0, F)$*, where* Q *is a finite set of* states*,* Σ *is a finite* alphabet*,*

$\delta \subseteq Q \times \Sigma \times Q$ *is a set of* named transitions, $q_0 \in Q$ *is the* initial state, $F \subseteq Q$ *is a set of* accepting states. *For convenience, we denote the set of* transition names *also by* δ. □

Note that the concept of transition name is introduced. A transition in δ is of the form $p_k : (q, a, q')$, where p_k is the *name* of the transition. In the transition diagram, a transition $p_k : (q, a, q') \in \delta$ is denoted by an arc from q to q' labeled $p_k : a$.

Given a set of automata, they execute asynchronously, but may synchronize on rendezvous events. We assume that the send operation is executed before the receive operation in an atomic rendezvous event. Formally, we define their asynchronous composition as follows.

Let $N = \{n_1, ..., n_k\} \subseteq \mathbb{N}$ be a countable set with cardinality k, and for each $n_j \in N$, S_{n_j} be a set. We define the *Cartesian product* as:

$$\prod_{n_j \in N} S_{n_j} = \{(x_{n_1}, x_{n_2}, ..., x_{n_k}) \mid \forall j \in \{1, ..., k\}, x_{n_j} \in S_{n_j}\}$$

For each $j \in \{1, ..., k\}$, we denote the j-th component of the vector $q = (x_{n_1}, x_{n_2}, ..., x_{n_k})$ by the projection $q[j]$, i.e., $q[j] = x_{n_j}$.

Definition 2. *The* asynchronous composition $A = \prod_{n \in N} A_n$ *of a countable collection of Büchi automata* $\{A_n = (Q_n, \Sigma_n, \delta_n, q_n, F_n)\}_{n \in N}$ *is a Büchi automaton*

$$A = (\prod_{n \in N} Q_n, \Sigma, \delta, \prod_{n \in N} q_n, F)$$

where $\Sigma = \bigcup_{n \in N} \Sigma_n$,

$$\delta = \{p_k : (q, a, q') \mid \exists n_i \in N, a \in \Sigma_{n_i} \wedge p_k : (q[i], a, q'[i]) \in \delta_{n_i}, \\ \textit{and } \forall n_j \in N, n_j \neq n_i \rightarrow q[j] = q'[j]\}$$

$F = \{q \in \prod_{n \in N} Q_n \mid \exists n_i \in N, q[i] \in F_{n_i}\}$. □

We interpret the asynchronous composition as the expanded version which fully expands all possible values of the variables in the state space. The executability of a transition depend on its source state.

4.2 The Asynchronous-Composition BAC System

An Asynchronous-Composition Büchi Automaton Control System (AC-BAC System) consists of an asynchronous composition of Büchi automata and a Büchi controlling automaton. The controlling automaton controls all the transitions of the primitive components in the composition. Thus, the alphabet of the controlling automaton equals the set of transition names of the controlled automata.

Definition 3. *Given an asynchronous composition of a set of Büchi automata* $A = \prod_{n \in N} A_n$, *a* (Büchi) controlling automaton *over* A *is* $A_c = (Q_c, \Sigma_c, \delta_c, q_c, F_c)$ *with* $\Sigma_c = \bigcup_{n \in N} \delta_n$. *The global system is called an* Asynchronous-Composition Büchi Automaton Control System *(AC-BAC System).* □

The controlling automaton is used to specify the sequences of transitions that should *never* occur, since the ω-words accepted by a controlling automaton are sequences of transitions.

We compute the *meta-composition* of an asynchronous composition and a controlling automaton. In the *meta-composition*, a transition is allowed iff it is in the asynchronous composition and allowed by the controlling automaton. The name "*meta-composition*" denotes that the controlling automaton is at a higher level, since it treats the set of transitions rather than the alphabet. We formally define the *meta-composition* ($\vec{\cdot}$ operator) as follows.

Definition 4. *The* meta-composition *of an asynchronous composition* $A = (Q, \Sigma, \delta, q_0, F)$ *and a controlling automaton* $A_c = (Q_c, \Sigma_c, \delta_c, q_c, F_c)$ *is a Büchi automaton:*

$$A' = A \vec{\cdot} A_c = (Q \times Q_c, \Sigma, \delta', (q_0, q_c), (F \times Q_c) \cup (Q \times F_c)\ )$$

where for each $q_i, q_l \in Q$, $q_j, q_m \in Q_c$ *and* $a \in \Sigma$, *we have* $p_k : ((q_i, q_j), a, (q_l, q_m)) \in \delta'$ *iff* $p_k : (q_i, a, q_l) \in \delta$ *and* $(q_j, p_k, q_m) \in \delta_c$. □

If we let $\{A_n\}_{n \in N}$ contain only a single primitive automaton, i.e., $|N| = 1$, the definition will express the meta-composition over a single automaton. This means, the controlling automaton over a single automaton is only a special case of the above definition.

We observe that A has feasible sequences of transitions accepted by A_c (or A matches A_c), iff the language accepted by the meta-composition is not empty. This fact will be used for checking `nevertrace` claims.

Note that the emptiness problem for meta-compositions is decidable. It is well known that the emptiness problem for Büchi automata is decidable [8]. Since the meta-composition is a Büchi automaton, checking its emptiness is also decidable.

4.3 From Nevertrace Claims to AC-BAC Systems

We will show at first how to translate a model with a `nevertrace` claim into an AC-BAC system. Let us consider a simple example in Listing 1.2 consisting of a client and a server.

Listing 1.2. A Simplified Promela Model of Client and Server

```
1 mtype = {msg, ack};
2 chan c2s = [2] of {mtype};
3 chan s2c = [0] of {mtype};
4 int x = 0;
5
6 active proctype client() {
7 again:
8 c2ssmsg:    c2s!msg;
9             x = x+1;
```

```
10 s2crack:    s2c?ack;
11             goto again;
12 }
13
14 active proctype server() {
15 c2srmsg:  do
16            :: c2s?msg;
17               x = x-1;
18 s2csack:      s2c!ack;
19            od;
20 }
```

Figures 3 (1) and (2) show the automata A_1, A_2 describing the behavior of the client and the server, respectively. Each transition between two control-flow states is labeled by its transition name (may be related to line number and control-flow state etc.) and statement. For example, 8 is the transition name between the states 8 and 9.

The asynchronous composition $A_1 \cdot A_2$ is shown in Fig. 3(3). A system state consists of four elements: the control-flow states of A_1 and A_2, the value of `x` and the contents of the channel `c2s`. Note that the dashed transitions are not executable at their corresponding states, although themselves and omitted subsequent states are part of the composition. That is, the figure contains exactly the reachable states in the system. At the bottom of the figure, the transitions labeled 18 and 10 constitute a handshake.

Let us consider the following `nevertrace` claim:

```
nevertrace {  /* ![]( $c2s# -> <> $s2crack) */
T0_init:
    if
    :: (!$s2crack && $c2s#) -> goto accept_S4
    :: $# -> goto T0_init
    fi;
accept_S4:
    if
    :: (!$s2crack) -> goto accept_S4
    fi;
}
```

The claim specifies the property that any operation on channel `c2s` always leads to receive `ack` from the channel `s2c` later. In other words, if one of the transitions that have a label starting with `c2s` is executed, then one of the transitions labeled `s2crack` will be executed in the future.

Figure 3(4) shows the automaton specified by the `nevertrace` claim. Figure 3(5) shows the controlling automaton specified by the `nevertrace` automaton. For example, the transition expression `(!$s2crack && $c2s#)` matches the transitions 8 and 16.

The automata in Figures 3 (3) and (5) constitute an AC-BAC system, which is established from the model and the `nevertrace` claim. The next step is to check whether the asynchronous composition $A_1 \cdot A_2$ matches the claim A_c.

The meta-composition $(A_1 \cdot A_2) \vec{\cdot} A_c$ is shown in Fig. 3(6) (the state space starting from the state (9,15,q_0) is not drawn). The state of the controlling automaton is added into the system state. In the meta-composition, a transition is allowed, iff it is in $(A_1 \cdot A_2)$ and allowed by A_c. Note that the transition 10 is blocked, since it is not allowed by the state q_1 of A_c. It is easy to see there does not exist any acceptance cycle, thus the ω-language accepted by the meta-composition is empty. This means, no counterexample can be found, and the system satisfies the required correctness property.

To conclude, checking `nevertrace` claims is equivalent to the emptiness problem for meta-compositions. A `nevertrace` claim is violated if and only if the meta-composition is not empty. As we mentioned, the emptiness problem for meta-compositions is decidable. Therefore, checking `nevertrace` claims is feasible in practice. Furthermore, the emptiness of meta-compositions can be checked on-the-fly, using the technique for checking `never` claims.

5 On Expressing Some Constructs in SPIN

In this section, we will show how to express the semantics of some constructs in Promela using `nevertrace` claims, although the `nevertrace` claim does not aim at replacing them.

5.1 Expressing Notrace Assertions

There are various ways to convert a `notrace` assertion into a `nevertrace` claim. As an example, let us consider the `notrace` assertion in Section 2. We can construct an NFA in Fig. 4 which specifies the same property as the DFA in Fig. 2 of the `notrace` assertion for the model in Listing 1.1.

A `nevertrace` claim can be written according to the automaton.

```
nevertrace {  /* containing two consecutive c2s!msg1 */
S0:
     if
       :: $c2ssmsg1 -> goto S1;
       :: $c2ssmsg2 -> goto S0;
       :: (!$c2ss#) -> goto S0;
     fi;
S1:
     if
       :: $c2ssmsg1;
       :: $c2ssmsg2 -> goto S0;
       :: (!$c2ss#) -> goto S1;
     fi;
```

```
/* S2 */
}
```

It is easy to see, we used the following rules to convert the `notrace` assertion into a `nevertrace` claim.

First, in the system model, all send operations of message `msg` on a channel `ch` must have a label `chsmsg`, while all receive operations of message `msg` on the channel must have a label `chrmsg`. The labels of all statements other than channel operations should not start with the names of declared channels.

Second, in the `notrace` assertion, (1) replace `ch!msg` by `$chsmsg`, `ch?msg` by `$chrmsg`; (2) for each state, add new transition expressions to match the statements outside the scope of the `notrace` assertion. In the example, for each state, we add a transition from the state to itself with the transition expression `(!$c2ss#)`, since only send operations on the channel `c2s` are within the scope of the `notrace` assertion.

5.2 Expressing Remote Label Reference

The predefined function `procname[pid]@label` is a remote label reference. It returns a nonzero value only if the next statement that can be executed in the process `procname[pid]` is the statement with `label`.

It seems that `procname[pid]@label` can be also used as a transition expression, if replacing `@` by `$`. However, there is a little difference in semantics. The remote label reference is evaluated over the next statement of the process `procname[pid]`, but the transition expression is evaluated over the last executed transition, which does not necessarily belong to the process `procname[pid]`.

5.3 Expressing the Non-Progress Variable

The predefined non-progress variable `np_` holds the value *false*, if at least one running process is at a control-flow state with a progress-state label. It is used to detect the existence of non-progress cycles.

It seems that the variable `np_` is equivalent to our transition expression `!$progress#`. However, there is a little difference. The variable `np_` is evaluated over all the running processes, but the transition expression is evaluated over the last executed process.

5.4 Expressing Progress-State Labels

There are two types of progress cycles. A *weak* progress cycle is an infinite execution cycle that contains at least one of the progress-state labels, which denotes reaching some progress-state labels infinitely often. A *strong* progress cycle is a weak progress cycle with the requirement that each statement with a progress-state label in the cycle must be executed infinitely often.

Promela supports only the weak progress cycle, whereas our `nevertrace` claim can express the strong progress cycle.

As an example, let us consider the model in Listing 1.3 consisting of two processes `p1` and `p2`. Note that `p1` does not execute any statement, but waits at the label `progress` for ever.

Listing 1.3. A Promela Model of Two Processes

```
1  chan ch = [0] of {bool};
2
3  active proctype p1() {
4                bool x;
5    progress:   ch?x;
6  }
7
8  active proctype p2() {
9    bool x = 0;
10   do
11     :: x==0; x=1;
12     :: x==1; x=0;
13   od;
14 }
```

In the verification mode of SPIN, none (weak) non-progress cycle is found, both with or without fairness condition. All executions are weak progress cycles, since `p1` stays for ever at a progress-state label.

In contrast, we can find a strong non-progress cycle using the following `nevertrace` claim, which can be constructed from the LTL formula `<>[]!$progress#`. If we modify a `never` claim generated from the LTL formula by SPIN, remember to replace the condition statement `(1)` or `true` by `$#` matching all transitions.

```
nevertrace { /* strong non-progress cycle detector */
T0_init:
    if
    :: (!$progress#) -> goto accept_S4
    :: $# -> goto T0_init
    fi;
accept_S4:
    if
    :: (!$progress#) -> goto accept_S4
    fi;
}
```

Note that the evaluation of transition expressions is over the last executed transition, and all the executable transitions do not have a progress-state label. Therefore, strong non-progress cycles can be detected as counterexamples.

5.5 Expressing Accept-State Labels

There are two types of acceptance cycles. A *weak* acceptance cycle is an infinite execution cycle that contains at least one of the accept-state labels, which

denotes reaching some accept-state labels infinitely often. A *strong* acceptance cycle is a weak acceptance cycle with the requirement that each statement with an accept-state label in the cycle must be executed infinitely often.

Promela supports only the weak acceptance cycle, whereas our `nevertrace` claim can express the strong acceptance cycle.

As an example, let us replace the label `progress` by `accept` in the model of Listing 1.3. Note that `p1` does not execute any statement, but waits at the label `accept` for ever.

In the verification mode of SPIN, a (weak) acceptance cycle is found, both with or without fairness condition. All executions are weak acceptance cycles, since `p1` stays for ever at an accept-state label. Therefore, (weak) acceptance cycles can be detected as counterexamples.

In contrast, we cannot find any strong acceptance cycle using the following `nevertrace` claim, which can be constructed from the LTL formula `[]<>($accept#)`. If we modify a `never` claim generated from the LTL formula by SPIN, remember to replace the condition statement `(1)` or `true` by `$#` matching all transitions.

```
nevertrace  { /* strong acceptance cycle detector */
T0_init:
if
:: $accept# -> goto accept_S9
:: $# -> goto T0_init
fi;
accept_S9:
if
:: $# -> goto T0_init
fi;
}
```

Note that the evaluation of transition expressions is over the last executed transition, and all the executable transitions do not have an acceptance-state label. Therefore, there is no strong acceptance cycle.

6 Related Work

In the previous section, we mentioned some differences between `nevertrace` claims and some constructs in Promela. In this section, we would like to summarize the comparison with the most powerful two constructs in Promela, `never` claims and `notrace` assertions.

The major differences between `nevertrace` claims and `never` claims are obvious. They are used to specify properties on sequences of *transitions* (execution traces) and sequences of *states*, respectively. A `nevertrace` claim is performed after executing a transition, whereas a `never` claim is started from the initial system state. A `nevertrace` claim is evaluated over the last executed transition, whereas a `never` claim is evaluated over the current system state. Thanks to

their different focuses, they can be used together in a model checker to achieve stronger power.

`Nevertrace` claims and `notrace` (also `trace`) assertions are both about execution traces, their major differences are as follows.

1. They have different scopes of checking. `Nevertrace` claims consider all transitions, whereas only simple send/receive operations are within the scope of `notrace` assertions. Furthermore, only the channel names that are specified in a `notrace` assertion are considered to be within its scope. All other transitions are ignored.
2. The `notrace` assertion cannot contain random receive, sorted send, or channel poll operations. But these can be also tracked by a `nevertrace` claim.
3. The `notrace` assertion must be deterministic, whereas the `nevertrace` claim could be nondeterministic, just like the `never` claim.
4. The `notrace` assertion does not execute synchronously with the system, but executes only when events of interest occur. Whereas the `notrace` assertion executes synchronously with the system, just like the `never` claim.

7 Conclusion

In this paper, we proposed the `nevertrace` claim, which is a new construct for specifying the correctness properties that either finite or infinite *execution traces* (i.e., *sequences of transitions*) that should *never* occur. The Asynchronous-Composition Büchi Automaton Control System (AC-BAC System) provides the theoretical foundation for checking `nevertrace` claims.

References

1. Holzmann, G.J., Peled, D.: The state of SPIN. In Alur, R., Henzinger, T.A., eds.: Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996). Volume 1102 of Lecture Notes in Computer Science., Springer (1996) 385–389
2. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5) (1997) 279–295
3. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
4. Ben-Ari, M.: Principles of the SPIN Model Checker. Springer (2008)
5. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01). Volume 2102 of Lecture Notes in Computer Science., Springer (2001) 53–65
6. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading MA (1979)
7. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Stanford University Press (1960) 1–11
8. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics, Elsevier Science Publishers B.V. (1990) 133–191

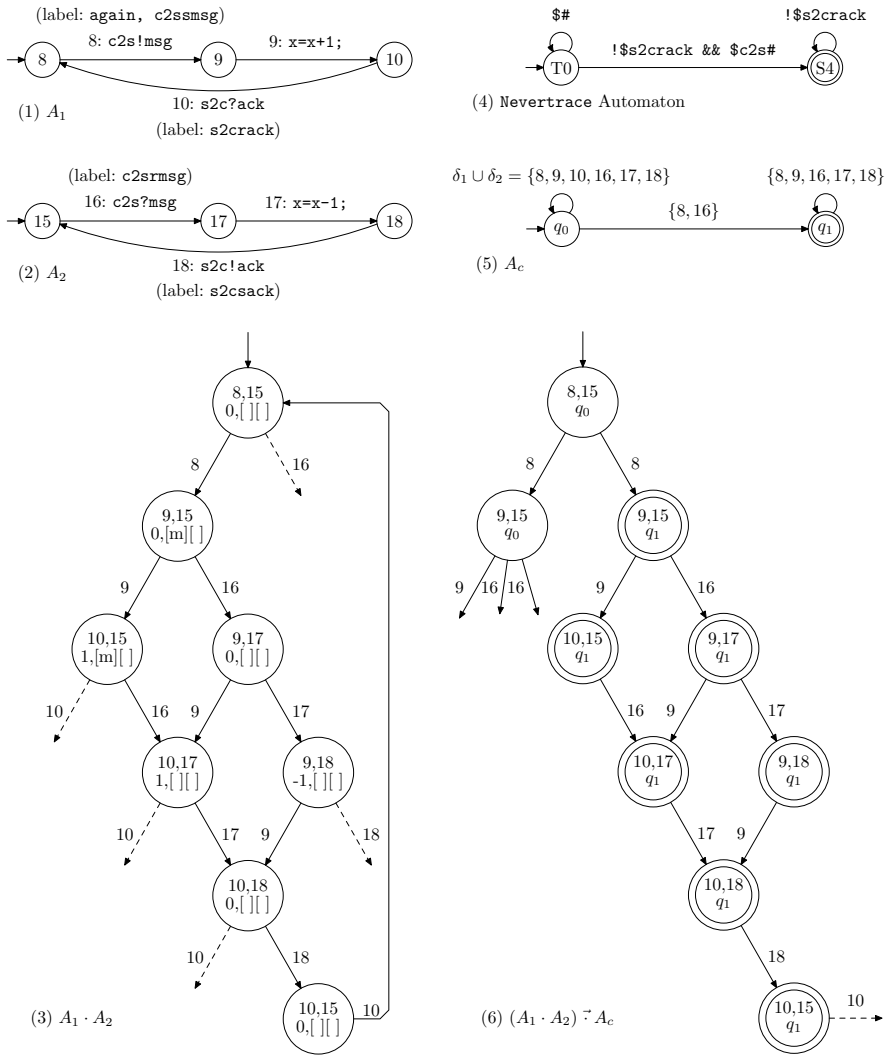

Fig. 3. An Example of Asynchronous-Composition BAC System

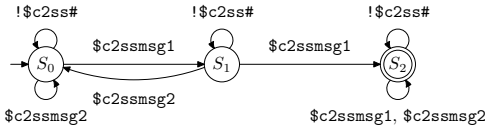

Fig. 4. The Nondeterministic Automaton of `nevertrace` Claim