Publication V

Jori Dubrovin. Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference (VMCAI 2010)*, pages 146–162, January 2010.

Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing

Jori Dubrovin*

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O.Box 5400, FI-02015 TKK, Finland
Jori.Dubrovin@tkk.fi

Abstract. This paper presents a new framework for checking bounded reachability properties of asynchronous systems by reducing the problem to satisfiability in difference logic. The analysis is bounded by fixing a finite set of potential events, each of which may occur at most once in any order. The events are specified using high-level Petri nets. The proposed logic encoding describes the space of possible causal links between events rather than possible sequences of states as in Bounded Model Checking. Independence between events is exploited intrinsically without partial order reductions, and the handling of data is symbolic. Experiments with a proof-of-concept implementation of the technique show that it has the potential to far exceed the performance of Bounded Model Checking.

1 Introduction

Design errors in concurrent hardware and software systems are notoriously difficult to find. This is due to the tremendous number of possible interleavings of events and combinations of data values. Symbolic model checking methods [7] attack the problem by expressing the actual and desired behavior of a system as formulas and using the tools of computational logic to search for a possible failure.

In this paper, we develop a new symbolic technique for verifying bounded reachability properties of asynchronous discrete-event systems. Instead of manipulating executions as sequences of states, we take an event-centered viewpoint. First, one fixes a collection of transitions, each of which describes one discrete step of execution. This collection is called an unwinding of the system. We only consider finite-length executions in which each transition of the unwinding occurs at most once, in whichever order. From the unwinding, we generate automatically a formula that is satisfiable if and only if a predefined condition, e.g. division by zero, can be reached within this bounded set of executions. For satisfiability checking, any SAT or SMT solver [18] can be used as long as it can handle the data constraints of transitions. If the reachability property holds within the bound, a witness execution can be extracted from an interpretation that satisfies the formula. Otherwise, longer executions can be covered by adding more transitions to the unwinding and generating a new formula. This technique will be called Bounded Event Tracing.

* Financial support from Hecse (Helsinki Graduate School in Computer Science and Engineering) and the Emil Aaltonen Foundation is gratefully acknowledged.

G. Barthe and M. Hermenegildo (Eds.): VMCAI 2010, LNCS 5944, pp. 146–162, 2010.

The approach is similar to Bounded Model Checking (BMC) [2]. Both methods can find bugs and report no false alarms, but they cannot be used as such to prove the absence of bugs in realistic systems. Unlike BMC, the new technique directly exploits the defining aspect of asynchronous systems: each transition accesses only a fraction of the global state of the system. Although the generation of optimal unwindings is not yet pursued in this work, Bounded Event Tracing is shown to be able to outperform BMC on several benchmarks.

In the next section, we will go through the central concepts with an extensive example. Section 3 defines unwindings as a class of high-level Petri nets [15] that allows concise modeling of concurrency and software features. The logic encoding is presented in Sect. 4, while Sect. 5 discusses the relationship to other approaches. In Sect. 6, we design one way to automatically generate unwindings for a class of state machine models and use these unwindings in an experimental comparison to BMC.

2 Bounded Event Tracing by Example

Figure 1a presents a system with three concurrent processes that run indefinitely. Suppose the reachability property in question is whether the system can ever print "equal". The execution in Fig. 1b shows that the property holds: after one cycle of process F and two cycles of G, both x and y have the value 9, and process H then runs the print statement. The circles represent the values of variables in states $M_1, M_2, \ldots$, and the rectangles f, g_1, g_2, and h represent the atomic execution of one cycle of process F, G, G, and H, respectively.

Figure 1c shows a related high-level Petri net. We can interpret Fig. 1b as a finite execution of the Petri net as follows. The *transition* (rectangle) named $init$ occurs first, producing a *token* in each of the *places* (circles) pj, px, and py, which correspond to the variables of the system. This leads to a state M_1, in which each place pj, px, and py contains one token that carries a value 3, 2, or 5, respectively. Transition f occurs next, consumes the token from place py and produces a new token with value 9. This results in a state M_2. Then, transition g_1 simultaneously consumes a token from each place pj and px, and uses their values to produce new tokens. Finally, the state M_5 is reached.

This is an example of a *one-off execution* of the Petri net. Generally, a one-off execution is a finite sequence that starts with a state in which no place contains a token. Then, a transition occurs, consuming exactly one token with each input arc (an arrow from a place to the transition) and producing exactly one token with each output arc (an arrow from the transition to a place) while fulfilling the data constraints. This leads to a new state and so on, as usual in Petri nets. The only distinctive requirement is that each transition occurs *at most once* in the sequence. The transitions that occur in a one-off execution are its *events*.

A Petri net whose set of one-off executions specifies a bounded portion of the behavior of a system is called an *unwinding* of the system. We assume that we are given an unwinding whose one-off executions map easily to finite-length executions of the original system. The unwinding of Fig. 1c has another one-off execution consisting of the sequence $init, g_2, h$ of events. This corresponds to process G running one cycle and then process H printing "equal". In total, this unwinding covers all executions of the

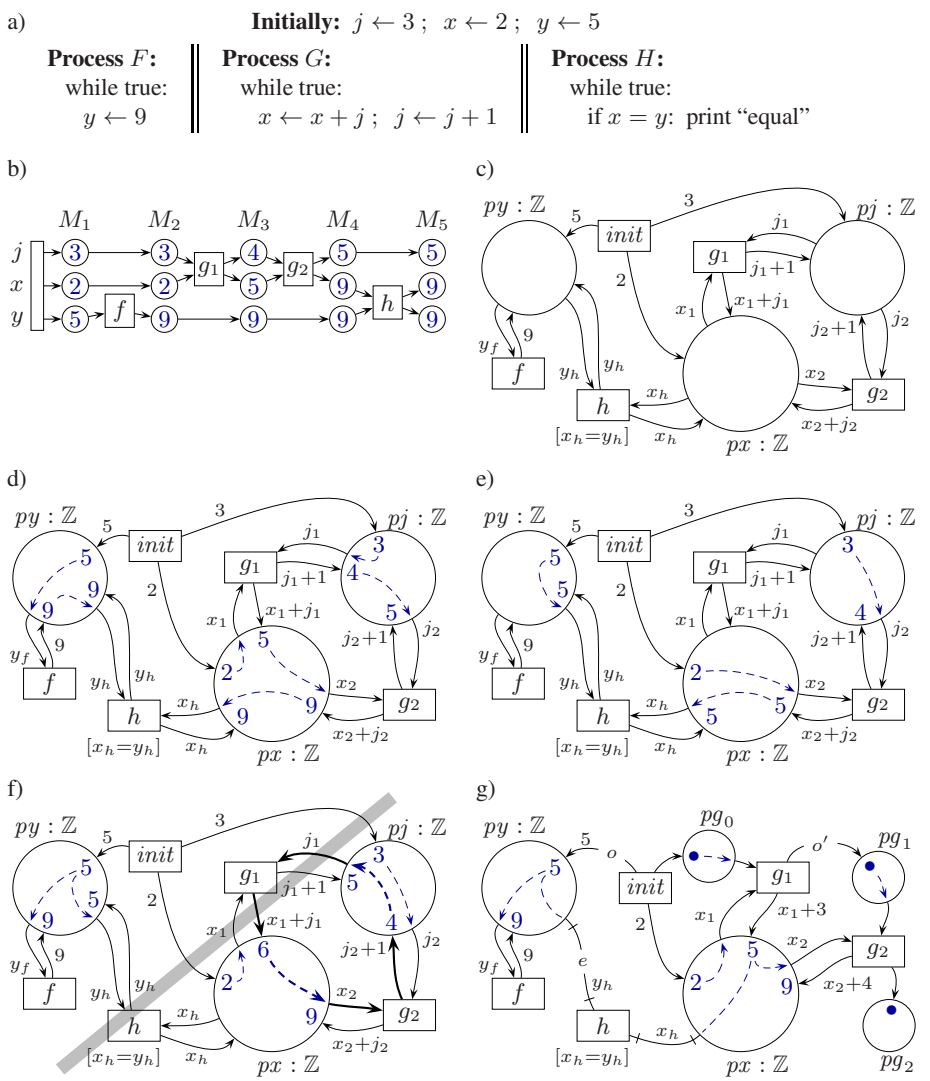

Fig. 1. An example system and illustrations of its behavior

system in which process F runs at most one cycle, process G at most two cycles, and H at most one cycle, in any possible order.

We observe that every token consumed during a one-off execution has been previously produced. Figure 1d illustrates this idea for the one-off execution of Fig. 1b. Transition g_1 consumes the token with value 2 produced by $init$, whereas the token with value 5 in place pj is not consumed at all. The numeric values and dashed arrows inside the big circles in Fig. 1d constitute an example of what we call a *token trace* of the unwinding. The token trace tells us some facts about the course of events. By following the arrows, we see that $init$ occurs before g_1, which occurs before g_2, but we

cannot infer whether f occurs before or after, say, g_2. A token trace generally fixes only a *partial order* of events. Figure 1e illustrates another token trace of the same unwinding. This time, transitions f and g_1 do not occur at all. We can check that this token trace describes the second one-off execution discussed above.

It turns out that by specifying a simple set of rules for constructing a token trace of a fixed unwinding, we can characterize the set of *all* one-off executions of the unwinding. In other words, an unwinding induces a set of one-off executions and a set of token traces, and there is a meaningful correspondence relation between the two sets. We can thus reduce the search for a one-off execution with a certain property to finding a corresponding token trace. Given an unwinding, its token traces are defined by the following rules.

1. A token trace consists of events, links (dashed arrows), and data values.
2. A subset of the transitions of the unwinding are chosen to be events.
3. Each output arc of each event is associated with a single token with a value.
4. Each input arc of each event is linked to an output arc of an event.
5. No two input arcs are linked to the same output arc.
6. The data constraints of all events are fulfilled by the values of tokens.
7. The links impose a partial order on the events.

Figure 1f contains a third attempt at a token trace of the same unwinding. However, there are several problems. First, transitions f and h are consuming the same token at place py. This breaks rule 5—an input arc denotes a destructive read operation. Second, transition h poses as an event although it gets no input from place px, breaking rule 4. Third, there is an illegal cycle, illustrated in thick arrows, that breaks rule 7: event g_1 produces a token with value 6, then g_2 consumes it and produces a token with value 4, which in turn is consumed by g_1. No chronological ordering of the occurrences agrees with the picture. Any of these three mistakes suffices to tell that Fig. 1f does not represent a valid token trace.

A model checking procedure. The discussion above suggests the following procedure for checking reachability properties of an asynchronous system. Generate an unwinding such that one-off executions of the unwinding map to finite executions of the system, and the property corresponds to the occurrence of a designated transition $t^\diamond$. Generate automatically a formula that encodes the rules for a token trace of the unwinding and add the constraint that $t^\diamond$ is an event. Feed the formula to an off-the-shelf satisfiability solver. If the formula is satisfiable, convert the satisfying interpretation to a token trace and further to an execution that witnesses the property. If the formula is unsatisfiable, expand the unwinding to cover more executions of the system, and start over.

Assembling unwindings. Figure 1c demonstrates a rudimentary way of obtaining unwindings, with a place for each variable and a transition or several identical transitions for each atomic action that the system can perform. However, we expect to gain better performance by further exploiting the versatility of Petri nets. In general, one can set up arcs in arbitrary configurations, and the number of tokens in a place needs not be fixed. With the multitude of possible design choices, it is generally not obvious how to find the best way to generate unwindings for a given class of systems.

Figure 1g shows another unwinding that covers the same set of executions as the previous one. The labels o, o', and e do not contribute to the semantics—they only name some arcs for later reference. A token in place pg_0, pg_1, or pg_2 denotes the fact that process G has executed 0, 1, or 2 cycles, respectively. The token carries a meaningless value denoted by $\bullet$. This solution breaks the symmetry of transitions g_1 and g_2, and has allowed us to inline the fixed values $j_1 = 3$ and $j_2 = 4$ in g_1 and g_2 and to eliminate the place pj. In Sect. 6, we will use similar ideas in an automated unwinding scheme.

Another change in Fig. 1g is that transition h is incident to two *test arcs* (lines with cross bars close to each end). A test arc represents a non-destructive read operation. It is like an input arc but does not consume the token, and it is usually behaviorally equivalent to a pair of input and output arcs. The use of test arcs is optional, but they may result in a more efficient encoding. The following rules need to be added for token traces. Each test arc is linked to an output arc, and multiple test arcs plus at most one input arc can be linked to the same output arc. The partial order must be such that a transition that tests a token occurs after the transition that produces the token. A third transition can consume the token, but it must occur after the testing transition. The token trace of Fig. 1g imposes a partial order that obeys these rules. In particular, because of the links within place px, transition h occurs after g_1 and before g_2.

3 Semantics of Unwindings

We will use the following notations for formalizing unwindings and token traces. For a function $f : X \to Y$, sets $A \subseteq X$, $B \subseteq Y$, and an element $y \in Y$, we adopt the usual notation $f(A) := \{f(x) \mid x \in A\}$, $f^{-1}(B) := \{x \in X \mid f(x) \in B\}$, and $f^{-1}(y) := f^{-1}(\{y\})$. We will use *types*, *variables*, and *expressions* to model data manipulation in systems. Each type is identified with the set of elements of the type; in particular, the Boolean type is $\mathbb{B} = \{\mathrm{false}, \mathrm{true}\}$. Every variable v and expression ϕ has a type $\mathit{type}(v)$ or $\mathit{type}(\phi)$. The set of variables in an expression or a set of expressions ϕ is denoted by $\mathit{vars}(\phi)$. A *binding* of a set V of variables maps each variable $v \in V$ to a value $d \in \mathit{type}(v)$. If ϕ is an expression and b is a binding of (a superset of) $\mathit{vars}(\phi)$, the *value of ϕ in b*, denoted by ϕ^b, is obtained by substituting $b(v)$ for each occurrence of a variable $v \in \mathit{vars}(\phi)$ in the expression and evaluating the result. We will not fix a concrete language for expressions—the choice of a proper language depends on the problem domain and on the capabilities of the satisfiability solver used.

A *multiset* M over a set U is a function $U \to \mathbb{N}$, interpreted as a collection that contains $M(u)$ indistinguishable copies of each element $u \in U$. A multiset M is *finite* iff the sum $\sum_{u \in U} M(u)$ is finite. When the base set U is clear from the context, we will identify an ordinary set $A \subseteq U$ with the multiset χ_A over U, defined as $\chi_A(u) = 1$ if $u \in A$ and $\chi_A(u) = 0$ otherwise. If M_1 and M_2 are multisets over U, then M_1 is a *subset* of M_2, denoted $M_1 \leq M_2$, iff $M_1(u) \leq M_2(u)$ for all $u \in U$. A multiset M *contains* an element $u \in U$, denoted $u \in M$, iff $M(u) \geq 1$. We will use $M_1 + M_2$ and $M_2 - M_1$ with their usual meanings (as functions) to denote multiset union and multiset difference, respectively. The latter is defined only if $M_1 \leq M_2$.

A binary relation $\prec$ over a set X is a *strict partial order* iff it is irreflexive, asymmetric, and transitive, that is, iff for all $x, y, z \in X$ (i) $x \prec y$ implies not $y \prec x$ and (ii) $x \prec y$ and $y \prec z$ together imply $x \prec z$.

3.1 Colored Contextual Unweighted Petri Nets

Colored Petri Nets [15] are a powerful language for the design and analysis of distributed systems. In this work however, we use Petri nets with restricted semantics to specify a bounded portion of the behavior of a system. Our variant is called *Colored Contextual Unweighted Petri Nets*, or "nets" for short. The word *contextual* means that nets can contain test arcs [5], allowing compact modeling of non-destructive read operations. By *unweighted* we mean that each arc is associated with a single token instead of a multiset of tokens as in Colored Petri Nets. This restriction is crucial for the encoding, but does not seriously weaken the formalism. Places can still contain multisets of tokens, and multiple arcs can be placed in parallel to move several tokens at the same time.

Definition 1. *A* net *is a tuple* $N = \langle \Sigma,\ P,\ T,\ A_{in},\ A_{test},\ A_{out},\ place,\ trans,\ colors,\ guard,\ expr \rangle$*, where*

1. Σ *is a set of non-empty* types *(sometimes called color sets),*
2. P *is a set of* places*,*
3. T *is a set of* transitions*,*
4. A_{in} *is a set of* input arcs*,*
5. A_{test} *is a set of* test arcs*,*
6. A_{out} *is a set of* output arcs*,*
7. P*,* T*,* A_{in}*,* A_{test}*, and* A_{out} *are all pairwise disjoint,*
8. $place$ *is a* place incidence function $A_{in} \cup A_{test} \cup A_{out} \to P$*,*
9. $trans$ *is a* transition incidence function $A_{in} \cup A_{test} \cup A_{out} \to T$*,*
10. *the set* $trans^{-1}(t)$ *is finite for all* $t \in T$*,*
11. $colors$ *is a* color function $P \to \Sigma$*,*
12. $guard$ *is a* guard function *over* T *such that for all* $t \in T$*,* $guard(t)$ *is an expression with* $type(guard(t)) = \mathbb{B}$ *and* $type(vars(guard(t))) \subseteq \Sigma$*,*
13. $expr$ *is an* arc expression function *over* $A_{in} \cup A_{test} \cup A_{out}$ *such that for all arcs* a*,* $expr(a)$ *is an expression with* $type(expr(a)) = colors(place(a))$ *and* $type(vars(expr(a))) \subseteq \Sigma$*,*

A net is *finite* iff P and T are finite sets. For a transition or a set of transitions t and a place or a set of places p, we use the shorthand notations

$$\begin{aligned} in(t) &:= A_{in} \cap trans^{-1}(t)\ , & in(p) &:= A_{in} \cap place^{-1}(p)\ , \\ test(t) &:= A_{test} \cap trans^{-1}(t)\ , & test(p) &:= A_{test} \cap place^{-1}(p)\ , \\ out(t) &:= A_{out} \cap trans^{-1}(t)\ , & out(p) &:= A_{out} \cap place^{-1}(p)\ , \\ vars(t) &:= vars(guard(t)) \cup \bigcup_{a \in trans^{-1}(t)} vars(expr(a))\ . \end{aligned}$$

In the net of Fig. 1g, we have $place(o) = py$, $trans(o) = init$, $test(py) = \{e\}$, $out(pg_1) = \{o'\}$, $place(in(g_2)) = \{px, pg_1\}$, $colors(py) = \mathbb{Z}$, $expr(e) = y_h$, $guard(h) = (x_h{=}y_h)$, $vars(h) = \{x_h, y_h\}$, and $vars(init) = \emptyset$. We omit vacuously true guards, so $guard(f) = \mathrm{true}$ implicitly. Also, $colors(pg_1)$ is implicitly the type $\{\bullet\}$ with only one meaningless value, and $expr(o')$ is the constant expression $\bullet$.

A *token element* is a pair $\langle p, d\rangle$, where $p \in P$ is a place and $d \in colors(p)$ is a value. A *marking* M is a finite multiset over the set of token elements. Markings represent states of the system. The interpretation is that if $M(\langle p, d\rangle) = n$, then place p contains n tokens of value d in state M.

A *binding element* is a pair $\langle t, b\rangle$, where $t \in T$ is a transition and b is a binding of $vars(t)$. The shorthand $consumed_{\langle t,b\rangle} := \sum_{c \in in(t)} \left\{ \langle place(c), expr(c)^b \rangle \right\}$ will mean the multiset of token elements consumed by a binding element, while $produced_{\langle t,b\rangle} := \sum_{o \in out(t)} \left\{ \langle place(o), expr(o)^b \rangle \right\}$ means the multiset of produced token elements. A binding element $\langle t, b\rangle$ is *enabled* in a marking M iff the following conditions hold.

1. $consumed_{\langle t,b\rangle} \leq M$,
2. $\langle place(e), expr(e)^b \rangle \in \left(M - consumed_{\langle t,b\rangle}\right)$ for all $e \in test(t)$, and
3. $guard(t)^b = \text{true}$.

The binding element can *occur* in the marking iff it is enabled in the marking, leading to a new marking $M' = M - consumed_{\langle t,b\rangle} + produced_{\langle t,b\rangle}$. We denote by $M\ [t, b\rangle\ M'$ the fact that the binding element is enabled in M and leads from M to M' if it occurs. A *finite occurrence sequence* of a net is a finite sequence $M_0\ [t_1, b_1\rangle\ M_1\ \cdots\ [t_k, b_k\rangle\ M_k$ such that $k \geq 0$ and $M_{i-1}\ [t_i, b_i\rangle\ M_i$ holds for each $1 \leq i \leq k$.

3.2 Unwindings and One-Off Executions

We define an *unwinding* to be any net $N = \langle \Sigma, P, T, \ldots, expr\rangle$ that fulfills the two constraints below.

1. Transitions do not share variables: when $t \in T$ and $u \in T$ are distinct, $vars(t) \cap vars(u) = \emptyset$. We can always achieve this by renaming variables if necessary, as done in Fig. 1c by using subscripts.
2. Every place is incident to an output arc: $out(p) \neq \emptyset$ for all $p \in P$. This is not a crucial restriction either: places with no incident output arcs are useless in unwindings and can be eliminated.

These constraints are just technicalities—the true restriction is that the transitions of an unwinding are treated as *potential events*: each of them occurs once or not at all. Thus, we define a *one-off execution* of an unwinding as a finite occurrence sequence $M_0\ [t_1, b_1\rangle\ M_1\ \cdots\ [t_k, b_k\rangle\ M_k$ such that $M_0 = \emptyset$ and $t_i \neq t_j$ for all $1 \leq i < j \leq k$. The set $\{t_1, \ldots, t_k\}$ is the *event set* of the one-off execution. A transition $t \in T$ is *one-off reachable* iff it is an event in some one-off execution. For example, the unwinding of Fig. 1c has a one-off execution $M_0\ [init, b_{init}\rangle\ M_1\ [f, b_f\rangle\ M_2$, where $M_2 = \{\langle pj, 3\rangle, \langle px, 2\rangle, \langle py, 9\rangle\}$, the binding b_{init} is empty, and ${y_f}^{b_f} = 5$. The initial marking M_0 is fixed to be empty, but we work around this by specifying the starting conditions with a transition $init$ that necessarily occurs once in the beginning of any non-trivial one-off execution.

3.3 Token Traces

Let us formalize the rules presented in Sect. 2 for a token trace.

Definition 2. *A* token trace *of an unwinding $N = \langle \Sigma, P, T, \ldots, expr \rangle$ is a tuple $R = \langle E, src, b \rangle$, where*

1. *$E \subseteq T$ is a finite set of* events*,*
2. *src is a* source function *$in(E) \cup test(E) \to out(E)$ such that*
   (a) $place(a) = place(src(a))$ for all arcs $a \in in(E) \cup test(E)$ and
   (b) $src(c_1) \neq src(c_2)$ for all input arcs $c_1, c_2 \in in(E)$ such that $c_1 \neq c_2$,
3. *b is a binding of $vars(E)$, called the* total binding*, such that $expr(a)^b = expr(src(a))^b$ for all arcs $a \in in(E) \cup test(E)$,*
4. *$guard(t)^b = \mathrm{true}$ for all events $t \in E$,*
5. *there exists a strict partial order $\prec$ over the set E such that*
   (a) $trans(src(a)) \prec trans(a)$ for all arcs $a \in in(E) \cup test(E)$ and
   (b) $trans(e) \prec trans(c)$ for all test arcs $e \in test(E)$ and input arcs $c \in in(E)$ such that $src(e) = src(c)$.

Relating to Sect. 2, the source function forms the links between the arcs, while the total binding takes care of the data constraints. According to item 3, the arc expression at each end of a link must evaluate to the same value, i.e. the value of the token. As $vars(E)$ is a *disjoint* union of the variables of each event, b can bind the variables of each event independently. Item 5 above says that the events can be ordered in such a way that each token is produced before any event consumes or tests it, and a token is not tested during or after its consumption. Any strict partial order over (a superset of) E that fulfills item 5 will be called a *chronological partial order* of the token trace.

Figure 1g portrays a token trace where $E = T$, ${y_f}^b = {x_h}^b = 5$, $src(e) = o$, $src(in(E)) \cap out(g_2) = \emptyset$, and necessarily $init \prec g_1 \prec h \prec g_2$. One of $f \prec g_2$ and $g_2 \prec f$ can be true, or both can be false, but not both true.

From a one-off execution $M_0\ [t_1, b_1\rangle\ M_1\ \cdots\ [t_k, b_k\rangle\ M_k$, we can construct a token trace by conjoining $b_1, \ldots, b_k$ to a total binding and tracing each consumed or tested token to its source. The interleaving $t_1 \prec t_2 \prec \cdots \prec t_k$ then gives a chronological partial order. Conversely, we can take a token trace and linearize its chronological partial order to obtain a one-off execution. These constructions constitute the proof of the following theorem. See the report [8] for details.

Theorem 1. *Given an unwinding N and a finite subset E of transitions, there is a one-off execution of N with event set E if and only if there is a token trace of N with event set E.*

4 Encoding Token Traces

Let $N = \langle \Sigma, P, T, \ldots, expr \rangle$ be a finite unwinding. We are interested in whether a transition $t^\diamond \in T$ is one-off reachable, or equivalently, whether there is a token trace of N whose event set contains $t^\diamond$. In this section, we will construct a formula that is satisfiable if and only if such a token trace exists.

A formula ϕ is satisfiable iff there is an interpretation I such that ϕ^I is true. In this context, an interpretation is a binding of the symbols in the formula. In propositional satisfiability (SAT), the formula only contains propositional (Boolean) symbols

and Boolean connectives. Extensions known as SMT [18] also allow non-Boolean constraints. For example, an interpretation I satisfies the formula $\bigwedge_{j\in J}(\mathsf{X}_j < \mathsf{Y}_j)$, where the X_j and Y_j are symbols of real type, if and only if ${\mathsf{X}_j}^I$ is less than ${\mathsf{Y}_j}^I$ for all $j \in J$.

The formula will be built using the following set of symbols:

- for each $t \in T$, a propositional symbol Occur_t ("transition t occurs"),
- for each $t \in T$, a symbol Time_t of type $\mathbb{R}$ ("when transition t occurs"),
- for each pair $o \in A_{out}, a \in A_{in} \cup A_{test}$ such that $place(o) = place(a)$, a propositional symbol $\mathsf{Link}_{o,a}$ ("arc a is linked to arc o"), and
- for each $v \in vars(T)$, a symbol Val_v of type $type(v)$ ("the value of v").

We get an interpretation from a token trace $\langle E, src, b\rangle$ by setting ${\mathsf{Occur}_t}^I$ to true iff $t \in E$, setting ${\mathsf{Link}_{o,a}}^I$ to true iff $o = src(a)$, letting ${\mathsf{Val}_v}^I := v^b$, and assigning the values ${\mathsf{Time}_t}^I$ according to some chronological partial order $\prec$. Because the symbols Time_t are used for ordering and not arithmetic, we could as well type them as e.g. integers instead of reals. The detailed constructions from a token trace to a satisfying interpretation and vice versa are in the report version [8].

The formula ϵ below (denoted by $\epsilon_\emptyset$ in the report [8]) encodes the rules for a token trace in terms of the introduced symbols. Checking the existence of a token trace containing the event $t^\diamond$ then reduces to checking the satisfiability of the formula $\epsilon \wedge \mathsf{Occur}_{t^\diamond}$.

$$\epsilon := \bigwedge_{t\in T} \gamma_t \;\wedge \bigwedge_{a\in A_{in}\cup A_{test}} \Big(\beta_a \wedge \bigwedge_{o\in out(place(a))} \psi_{o,a}\Big) \wedge \bigwedge_{p\in P} \delta_p \;. \tag{1}$$

The subformulas γ_t and β_a encode items 4 and 2a of Definition 2. For a guard or arc expression ϕ, we use the special notation ϕ^{vals} to denote the substitution of each variable $v \in vars(T)$ with the symbol Val_v.

$$\gamma_t := \mathsf{Occur}_t \rightarrow guard(t)^{vals} \;,$$
$$\beta_a := \mathsf{Occur}_{trans(a)} \rightarrow \bigvee_{o\in out(place(a))} \mathsf{Link}_{o,a} \;.$$

The subformula $\psi_{o,a}$ places constraints on linking arc a to output arc o, namely that $trans(o)$ must be an event, and items 5a and 3 of Definition 2 must hold.

$$\begin{aligned}\psi_{o,a} := {} & \big(\mathsf{Link}_{o,a} \rightarrow \mathsf{Occur}_{trans(o)}\big) \wedge \\ & \big(\mathsf{Link}_{o,a} \rightarrow (\mathsf{Time}_{trans(o)} < \mathsf{Time}_{trans(a)})\big) \wedge \\ & \big(\mathsf{Link}_{o,a} \rightarrow (expr(o)^{vals} = expr(a)^{vals})\big) \;.\end{aligned}$$

The constraints in δ_p are required to make sure that tokens consumed from a place p are indeed removed. We encode items 2b and 5b of Definition 2 as

$$\begin{aligned}\delta_p := {} & \bigwedge_{o\in out(p)} AtMostOne\left(\left\{\mathsf{Link}_{o,c} \;\middle|\; c \in in(p)\right\}\right) \wedge \\ & \bigwedge_{o\in out(p)} \bigwedge_{e\in test(p)} \bigwedge_{c\in in(p)} \big(\mathsf{Link}_{o,e} \wedge \mathsf{Link}_{o,c} \rightarrow (\mathsf{Time}_{trans(e)} < \mathsf{Time}_{trans(c)})\big) \;,\end{aligned}$$

where $AtMostOne\,(\Phi)$ denotes a formula that is true iff exactly zero or one formulas in the finite set Φ are true. This can be expressed in size linear in $|\Phi|$.

4.1 Properties of the Encoding

The principal motivation for formula (1) is that it can be used for model checking reachability properties.

Theorem 2. *Let $N = \langle \Sigma, P, T, \ldots, expr \rangle$ be a finite unwinding and let $t^\diamond \in T$ be a transition. Then, $t^\diamond$ is one-off reachable if and only if the formula $\epsilon \wedge \mathsf{Occur}_{t^\diamond}$ is satisfiable.*

The proof [8], which is based on Theorem 1, is constructive and can be used to extract witness executions.

Concerning compactness, the formula ϵ contains *one* instance of each guard and arc expression of the unwinding, so there is no duplication involved here. The rest of the encoding adds a term $O(|out(p)|\,(1 + |in(p)|)(1 + |test(p)|))$ to the size for each place p. The encoding is thus *locally* cubic in the number of arcs incident to a place, or quadratic if there are no test arcs. We could generally avoid the cubic formulation by replacing every test arc with a behaviorally equivalent pair of input/output arcs. Such a transformation is always sound, except when it is possible that some transition accesses a single token with two different test arcs—a presumably rare construct. However, there are two reasons for not dropping test arcs out of the formalism. First, if the arcs incident to place p are mostly test arcs (the number of test arcs is at least of the order $|out(p)|\,|in(p)|$), then the quadratic encoding size obtained by eliminating test arcs can be actually larger than the original cubic size. Second, input/output arc pairs can introduce unnecessary orderings of successive non-destructive read operations. Consider duplicating transition h in Fig. 1g. In a token trace, several copies of h can have their test arcs linked to the same output arcs without imposing an ordering of the copies. If input/output arcs are used instead as in Fig. 1c, any token trace necessarily fixes an ordering of the copies of h because successive copies have to be linked to each other. Thus, a single token trace represents a smaller set of interleavings if test arcs have been eliminated. Further experiments are needed to determine whether the smaller encoding size compensates for the potentially larger search space in satisfiability solving.

Apart from the inner parts of guards and arc expressions, our encodings are examples of *difference logic* formulas. General difference logic allows inequalities of the form $var_i < var_j + constant$, but here the constant term is always zero. Such inequalities offer us a very compact way to rule out all illegal cycles of the form $t_1 \prec t_2 \prec \cdots \prec t_n \prec t_1$. Many SMT solvers support difference logic natively, and often the solver implementation is indeed based on illegal cycle detection [13]. Another possibility is to encode the inequalities in propositional logic [21] and use a SAT solver. As the constant term is always zero in our formulas, the size increment using the encoding [21] is $O(|T|^3)$ instead of exponential as in the worst case. The report [8] shows how to further reduce the size by exploiting the absence of inequalities under negations.

5 Comparison to Related Work

A straightforward way to apply Bounded Model Checking [2] to an asynchronous system is to unroll its interleaving transition relation k times to cover all executions of k steps [16]. Consider a system that performs one of n possible atomic actions in each

step. The BMC view of executions corresponds to Fig. 1b. The long horizontal lines represent the realizations of frame conditions, which are parts of the formula that say when a variable must maintain its value. Because of unrolling, the BMC formula describes kn potential events, and only k of them are scheduled to occur. Furthermore, the notion of fixed time points means that insignificant reorderings of independent events, e.g. changing the order of Fig. 1b into $g_1 \rightarrow g_2 \rightarrow f \rightarrow h$, result in completely different interpretations of the SAT formula, potentially encumbering the solver.

In contrast, the encoding of token traces contains no frame conditions for conveying data over time steps, and no time points between independent transitions. Instead, the inputs and outputs of transitions are directly linked to each other. The selection of links is nondeterministic, which incurs some encoding overhead, and there are the potentially costly constraints for ordering the transitions. Using kn potential events, we can cover executions up to length kn instead of k, but this depends on the unwinding.

There have been several proposals for making BMC better suited to asynchronous systems. Using alternative execution semantics [16,9], several independent actions can occur in a single step of BMC, allowing longer executions to be analyzed without considerably increasing the size of the encoding. In [22], partial order reductions are implemented on top of BMC by adding a constraint that each pair of independent actions can occur at consecutive time steps only in one predefined order. An opposite approach [14] is to start BMC with some particular interleaving and then allow more behavior by iteratively removing constraints. As Bounded Event Tracing is inherently a partial order method, there is no need for retrofitted reductions.

Ganai and Gupta present a concurrent BMC technique [12] based on a similar kind of intuition as this paper. Individual BMC unrolling is applied to each thread of a multithreaded program, and all globally visible operations are potentially linked pairwise, with constraints that prevent cyclic dependencies. Lockset analysis is proposed for reducing the number of potential links. In the encodings of single threads, various BMC techniques are needed to avoid blowup. Bounded Event Tracing uses places to localize the communication between concurrent components, but [12] does not support this. Instead, operations in different threads can be linked even if there is no causal relation between them, and every thread has a local copy of all global variables. A similar, globally quadratic encoding would result from an unwinding where global communication goes through a single place that holds a vector of all global variables, with incident input arcs for accessing the vector and output arcs for restoring the possibly modified vector.

The CBMC approach [6] unwinds (up to a bound) the loops of a sequential C program, converts it to static single assignment form, and encodes the constraints on the resulting set of variables. A version for threaded programs [20] is based on bounding also the number of context switches. Each global read operation is conditioned on the number of context switches that have occurred so far, with the help of explicit symbols in the encoding for representing the value of each global variable x after i context switches. This value is in turn conditioned on the location where x is assigned the last time before the ith context switch. The encoding is geared towards the possibility of finding a witness with a low number of context switches. As in [12], a context switch involves copying all global variables to another thread. In contrast, the read operations

in Bounded Event Tracing are conditioned directly on where the latest write operation occurred, with no intermediate encoding symbols that keep the data values between writing and reading.

CheckFence [4] is also based on CBMC-like unwinding of individual C threads and additional constraints for modeling the communication between threads. Although CheckFence is designed to find bugs specifically under relaxed memory models, an encoding of the ordinary sequential memory model is used as a baseline. Unlike in [12,20], context switches are not made explicit in the encoding. Instead, there are symbols encoding the potential causal relations between individual read and write operations, much like the potential links in Bounded Event Tracing. A global memory order plays the same role as the chronological partial order in this paper. The proposed encoding (details in [3]) is cubic in size and is in many ways similar to what we would obtain by consistently using test arcs for read operations and input/output arcs for write operations as in Fig. 1g. The possibility of a quadratic-size encoding or the decoupling of producing and consuming values are not discussed in [4,3].

A completely different symbolic technique for concurrent systems is based on unfoldings [11], which are partial-order representations of state spaces as (infinite) low-level Petri nets of a fixed form. Model checking is performed by taking a suitable finite prefix of an unfolding and encoding its behavior and the desired property in SAT. As unfoldings are acyclic, the encoding is simple. Although an unfolding represents interleavings implicitly, every possible control path and every nondeterministic choice of data is explicitly present, and in practice, the generation of the unfolding prefix is the most expensive part. We could obtain unwindings directly from unfoldings, but this would mean to abandon symbolic data and arbitrary connections between places and transitions.

6 Unwindings of State Machine Models

As a proof of concept, we will sketch a simple mechanical unwinding scheme for a class of state machine models and use it in an experimental comparison to Bounded Model Checking. Our input is a subset of the DVE modeling language, which is used e.g. by the model checking benchmark set Beem [19].

A DVE system consists of fixed sets of communication channels and processes, and the behavior of a process is defined by control locations connected with edges (Fig. 2a). An action of a system is either (i) the simultaneous firing of two edges in different processes such that one edge is labeled with $ch!$ and the other with $ch?$, where ch is the name of a channel, or (ii) the firing of a τ-edge, i.e. one not labeled with a channel. Edges can additionally be labeled with guard expressions (in square brackets) and assignments to local or global variables. The treatment of other important system features, such as arrays and buffered channels, is left for future work.

The first step is to obtain a new *unwound system* that contains cycle-free copies of the original processes. For each process, we perform a depth-first search from the initial location to identify a set of *retreating edges* [1], i.e. those that complete a control flow cycle (e.g. all edges leaving location `wa` in Fig. 2a). For each location s, the corresponding unwound process has the distinct locations $s_0, s_1, \ldots, s_L$ until some loop

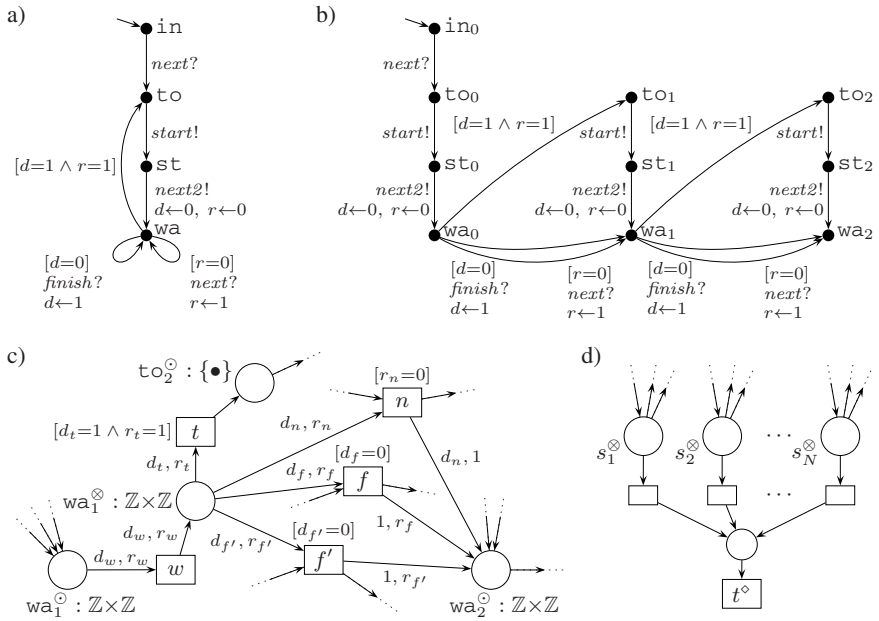

Fig. 2. Process `scheduler` and its unwinding

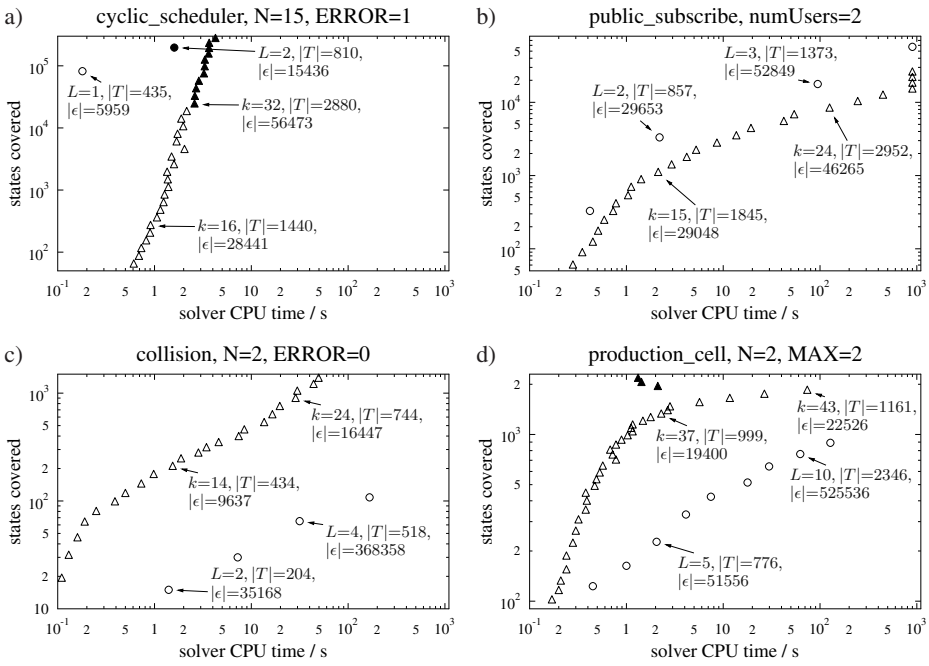

Fig. 3. Test runs of BMC (△) and Bounded Event Tracing (○) on four benchmarks

bound L. For each edge $s \to s'$, the unwound process has the edges $s_i \to s'_i$ for all i, or $s_i \to s'_{i+1}$ in the case of a retreating edge to guarantee acyclicity. Figure 2b illustrates the expansion. With these design choices, the unwound system behaves like the original one but the number of cycles executed in each process is bounded.

A Petri net unwinding (Fig. 2c) is then constructed from the unwound system by defining a pair of places $s^{\odot}, s^{\otimes}$ for each location s of each unwound process, and a transition for every possible action. The places contain the values of all local variables that are live (see [1]) in the corresponding location. For example, a token with value $(1, 0)$ in place $\mathtt{wa}_1^{\odot}$ or $\mathtt{wa}_1^{\otimes}$ in Fig. 2c means that the location $\mathtt{wa}_1$ is active with $d = 1$ and $r = 0$. A location s is always entered through $s^{\odot}$ and exited through $s^{\otimes}$, and we add a single trivial transition in the middle (transition w in Fig. 2c). This construct makes the encoding smaller by eliminating the quadratic number of potential links between the entering and exiting arcs. Transition t in the figure corresponds to the τ-edge from $\mathtt{wa}_1$ to $\mathtt{to}_2$. The edge labeled with $\mathit{finish}?$ is modeled with several transitions (f and f' in Fig. 2c), one corresponding to each $\mathit{finish}!$-labeled edge elsewhere in the unwound system. These transitions thus also connect to places that belong to the other processes. In the resulting unwinding, two transitions associated with the same process either have a fixed mutual ordering, or they exclude one another. Global variables would be modeled as in Fig. 1g, using a single place for each variable. Assuming that the reachability property—like in many Beem benchmarks—is whether any location in some set $\{s_1, \ldots, s_N\}$ can become active, we add new places and transitions as in Fig. 2d and check whether $t^{\diamond}$ is a one-off reachable transition.

6.1 Experimental Evaluation

Bounded Event Tracing with the above unwinding scheme was applied to some of the Beem benchmarks [19] that fit in the described subset, possibly after minor modifications such as replacing arrays with multiple scalar variables or adding a reachability property. The same properties were also checked using Bounded Model Checking with a transition relation formula that follows the structure of the interleaving encoding in [9]. In both approaches, Yices 1.0.22 64-bit (`http://yices.csl.sri.com/`) was used for solving satisfiability modulo bit vectors and difference logic, running on one core of an Intel Xeon 5130 processor. The results for four benchmarks that exhibit typical behavior are plotted in Fig. 3. Each triangular marker corresponds to a BMC instance with bound k. Each circle marks a Bounded Event Tracing instance with loop bound L, using the same value of L for all processes for simplicity. Filled markers mean satisfiable cases, i.e. the discovery of a witness execution. The horizontal axes denote the (non-cumulative) median CPU time used by the solver over 11 runs. The range of fluctuation in CPU times was generally small compared to the difference between the methods; the exceptions are specified below. Some of the instances timed out at the limit of 900 seconds. The vertical axes show the number of states of the original system reachable within each unwinding or BMC bound. The states were counted by running an explicit-state model checker on an instrumented system. Selected instances are annotated with the bound k or L, the number of encoded potential events $|T|$, which in

the case of BMC is k times the number of different actions the system can perform, and the circuit size $|\epsilon|$ of the formula given to the solver.

In Fig. 3a, an unwinding with loop bound 2 is sufficient for finding a witness of more than 30 steps, while using on average less CPU time (ranging from 1.0 to 4.9 seconds) and covering a larger number of states than the corresponding BMC instance. Figure 3b shows a benchmark where Bounded Event Tracing covers states faster than BMC, and the relative speed-up increases with the bound. In Figs. 3c and 3d, BMC is the faster method. In these cases, the number of transitions in the unwindings is much higher than the number of states reached, which indicates that the used unwinding scheme can result in the inclusion of many unnecessary transitions, mainly due to the design choices of a fixed system-wide loop bound L and quadratic-size modeling of channel synchronization. Furthermore, many of the resulting large number of transitions are connected to a common place that models a global variable, causing unwieldy growth in the formula size. Possibly because of this, there were also four individual Bounded Event Tracing runs of the production_cell benchmark that exceeded the median CPU time by a factor of more than 20.

The technical report [8] presents another set of experiments, in which Bounded Event Tracing with an alternative encoding is shown to outperform BMC on a family of models with very simple control flow but heavy dependence on a global variable.

7 Conclusions and Future Work

Bounded Event Tracing offers a new, well-defined framework for symbolically checking reachability properties of asynchronous systems. The analysis is bounded by a finite unwinding that fixes a collection of potential events that may occur but leaves the order of occurrences open. Unwindings are formalized as high-level Petri nets because the semantics of Petri nets rises naturally from the underlying concepts. The reachability problem is translated to a fragment of difference logic. The hard work is done by a SAT or SMT solver.

The technique incorporates ideas from Bounded Model Checking and unfoldings. Like in BMC, data handling is symbolic, but we avoid many pitfalls of BMC caused by viewing an execution of an asynchronous system as a sequence synchronized by fixed time steps. Like unfolding methods, Bounded Event Tracing has partial order reductions built in, but without the advance cost of explicit branching at every choice point.

Using a simple automated unwinding scheme, Bounded Event Tracing already performs better than interleaving BMC on a number of benchmark systems, but evident bottlenecks in the unwindings remain. In particular, the undirected expansion of unwindings easily becomes impractical when processes are tightly coupled with global variables. Interesting future research topics include better guidance of the expansion of unwindings e.g. using reachability information from smaller unwindings, integrating the expansion with incremental SAT solving [10], modeling interprocess communication more compactly, exploiting nested loops when unwinding control flow, modeling collections such as arrays or message queues using a place that contains a multiset of index-value pairs, and incorporating abstraction techniques [17] in some form to better cope with software features. The conjecture is that the construction of unwindings

allows for much greater flexibility than, say, adjusting the bound or the transition relation formula in BMC, and that we can gain significant improvements in speed by using a sophisticated unwinding scheme.

Acknowledgements. The author gives many thanks to Tommi Junttila for discussions and inspiration.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers. Principles, Techniques, & Tools, 2nd edn. Addison-Wesley, Reading (2007)
2. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded model checking of concurrent data types on relaxed memory models: a case study. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
4. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI 2007, pp. 12–21. ACM, New York (2007)
5. Christensen, S., Hansen, N.D.: Coloured Petri Nets extended with place capacities, test arcs and inhibitor arcs. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 186–205. Springer, Heidelberg (1993)
6. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
7. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
8. Dubrovin, J.: Checking bounded reachability in asynchronous systems by symbolic event tracing. Tech. Rep. TKK-ICS-R14, Helsinki University of Technology, Department of Information and Computer Science (2009)
9. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 96–112. Springer, Heidelberg (2008)
10. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4) (2003)
11. Esparza, J., Heljanko, K.: Unfoldings — A Partial-Order Approach to Model Checking. Springer, Heidelberg (2008)
12. Ganai, M.K., Gupta, A.: Efficient modeling of concurrent systems in BMC. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 114–133. Springer, Heidelberg (2008)
13. Ganai, M.K., Talupur, M., Gupta, A.: SDSAT: Tight integration of small domain encoding and lazy approaches in solving difference logic. Journal on Satisfiability, Boolean Modeling and Computation 3(1-2), 91–114 (2007)
14. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL 2005, pp. 122–131. ACM, New York (2005)
15. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods, and Practical Use, vol. 1. Springer, Heidelberg (1997)

16. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. Int. Journal on Software Tools for Technology Transfer 7(2), 89–101 (2005)
17. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
18. de Moura, L.M., Dutertre, B., Shankar, N.: A tutorial on satisfiability modulo theories. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 20–36. Springer, Heidelberg (2007)
19. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
20. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
21. Strichman, O., Seshia, S.A., Bryant, R.E.: Deciding separation formulas with SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 209–222. Springer, Heidelberg (2002)
22. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)