A Usable Reachability Analyser

Victor Khomenko[⋆]

School of Computing Science, Newcastle University,
Newcastle upon Tyne, United Kingdom
E-mail: `Victor.Khomenko@ncl.ac.uk`

Abstract. Reachability analysis consists in checking if a state satisfying some property is reachable. In this paper a solution to the problem of generating formulae expressing reachability properties for concrete models is suggested. The traditional methods either require the user to input the formula manually, which can be very tedious and error-prone, or automatically generate formulae for some fixed set of common properties, which does not allow one to check custom properties. The proposed approach allows the user to write a concise abstract specification of the property in a specially developed language Reach, which is then automatically expanded into a formula for a concrete model. The usefulness of this method is demonstrated on several case studies.

Keywords: Reachability analysis, model checking, Petri nets, STG.

1 Introduction

A reachability property is a property of some system that can be formulated as follows:

Check if there is a reachable state s satisfying some predicate $R(s)$.

Typically R specifies some undesirable states which should never be reached in a correct system, and the result of checking such a property is either a trace leading to a state satisfying this predicate or a message that no reachable state satisfies it. Reachability properties play a crucial role in formal verification; in particular, deadlock freeness, mutual exclusion and assertions are examples of reachability properties.

In this paper we assume that the system is specified as a safe (i.e. 1-bounded) Petri net, but the main ideas are still applicable to other formalisms, see Sec. 5. Hence one can assume that R is a Boolean formula built upon the elementary predicates $p_1, \ldots, p_n$ that correspond to the places of the Petri net. For example, the predicate

$$R \overset{\text{df}}{=} (p_1 \wedge p_2) \vee (p_1 \wedge p_3) \vee (p_2 \wedge p_3) \tag{1}$$

would specify a violation of the mutual exclusion of places p_1, p_2 and p_3.

Traditionally, the predicate R is either provided by the user or is generated automatically by the tool for some fixed set of common properties, like deadlocks or mutual exclusion. Unfortunately, both these approaches are of limited use in practice. Consider for example the Petri net modelling two dining philosophers shown in Fig. 1. The specification of the deadlock condition for it is as follows:

$$R \overset{\text{df}}{=} \overline{p_1} \wedge (\overline{p_2} \vee \overline{p_7}) \wedge (\overline{p_3} \vee \overline{p_8}) \wedge (\overline{p_4} \vee \overline{p_5}) \wedge \overline{p_6} \wedge \overline{p_9} \wedge (\overline{p_7} \vee \overline{p_{10}}) \wedge (\overline{p_8} \vee \overline{p_9}) \wedge (\overline{p_{12}} \vee \overline{p_{13}}) \wedge \overline{p_{14}} \tag{2}$$

From this rather small example one can see that manual specification of even the simplest properties for more realistic nets can easily get extremely tedious and error-prone.

The other alternative is to generate such a property automatically, which most of reachability analysers can do for deadlocks. However, this has a disadvantage: only a fixed set of

[⋆] V. Khomenko is a Royal Academy of Engineering/Epsrc Post-Doctoral Research Fellow. This research was supported by the Royal Academy of Engineering/Epsrc post-doctoral research fellowship EP/C53400X/1 (Davac).

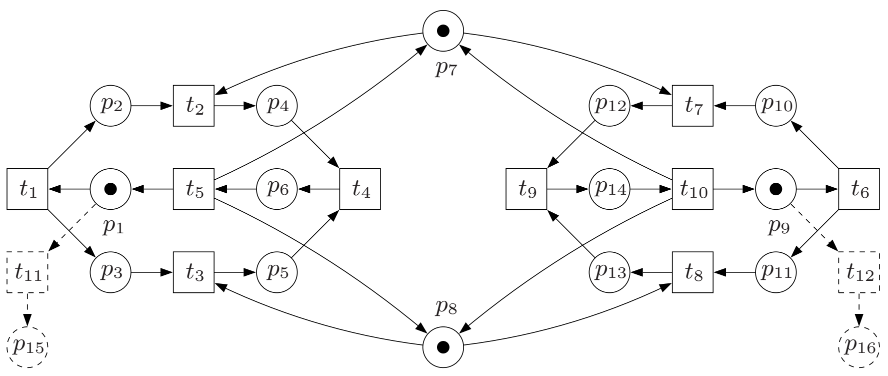


Fig. 1. A Petri net modelling two dining philosophers. The elements shown in dashed lines model a proper termination of the system.

properties can be implemented in this way, and if the property the user wants to check is not in this set, the tool becomes either useless, or the user has to resort to manually specifying the property. This is especially disappointing when the user's property is just a minor variation of some standard one. For example, suppose the transitions and places shown in dashed lines in Fig. 1, are added to the net in order to model the proper termination of the philosophers. The state when both p_{15} and p_{16} are marked is considered a proper terminal state, and should be distinguished from deadlocks. One can see that this is just a minor variation of the deadlock property, in particular it would be enough to add the clause $(\overline{p_{15}} + \overline{p_{16}})$ to the automatically generated deadlock formula; however, this is usually impossible, since the formula generation is hardwired into the tool.[1]

In practice, users are often forced to implement generators for their custom properties. While simple in theory, such generators still require a considerable implementation effort, in particular they have to have data structures and methods for representing and accessing Petri nets, a parser for reading in a net from the file and, in all but simple cases, data structures and routines for Boolean expressions manipulation. Obviously, few users would be prepared (or even able) to undertake such an effort.

In this paper we describe an idea which solves the aforementioned problem. It allows the user to specify complex custom properties of large nets with little effort. In fact, in most cases it is no harder for the user than mathematically defining the property.

2 Property specification language

In this section we outline the main idea of how to cope with the problem described above. Namely, we design a language Reach for specifying reachability properties. This approach has the following advantages:

– custom properties can be easily and concisely specified;
– the user does not have to modify the model in any way, in particular the model does not have to be translated into an input language of some model checker;[2]

[1] In this particular example one can use a trick of adding yet another transition to the net, which takes the tokens from p_{15} and p_{16} and immediately puts them back. This makes the state corresponding to the proper termination non-deadlocked, and so one can use the usual deadlock checking. However, such tricks do not always exist, and in general it is a bad idea to make the user modify the model or invent tricks.

[2] In many model checkers, e.g. in SPIN [Hol04], *both the model and the property* are specified in the same language (cf. the `never` clause in Promela— the input language of SPIN). Hence, one would have to translate a model into Promela to verify it. In contrast, the output of our tool is simply a Boolean formula, which is essentially independent on the original format of the Petri net.

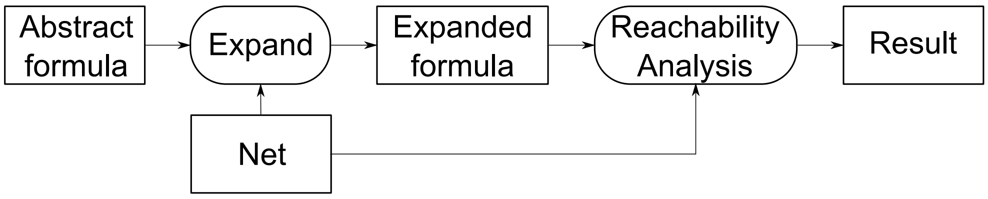


Fig. 2. Reachability analysis flow.

– almost any reachability analyser can be used as the back-end.

For example, the deadlock property can be mathematically defined as follows:

$$\bigwedge_{t \in T} \bigvee_{p \in {}^\bullet t} \overline{p},$$

where T is the set of transitions of the Petri net. The tool would take as an input the following Reach description of this property:

```
forall t in TRANSITIONS {
    exists p in pre t { ~$p }
}
```

The other input of the tool is the Petri net, and the tool is able to automatically *expand* this abstract property specification into a concrete Boolean formula for this particular net, e.g. the deadlock formula (2) is generated for the net in Fig. 1. Note that the `forall` and `exists` operators expand into a conjunction and disjunction, respectively, `TRANSITIONS` is substituted by the set of transitions of the net, the operator `pre` computes the preset ${}^\bullet t$ of a transition, `~` is the Boolean negation, and the `$` operator refers to the status of the place (i.e. whether it has a token) in the marking to be reached. In fact, the above property can be re-written as

```
forall t in TRANSITIONS { ~@t }
```

where the operator `@` refers to the enabledness status of a transition in the marking to be reached.[3]

This property specification can be easily modified to take into account the proper termination:

```
forall t in TRANSITIONS { ~@t } & (~$P"p15" | ~$P"p16")
```

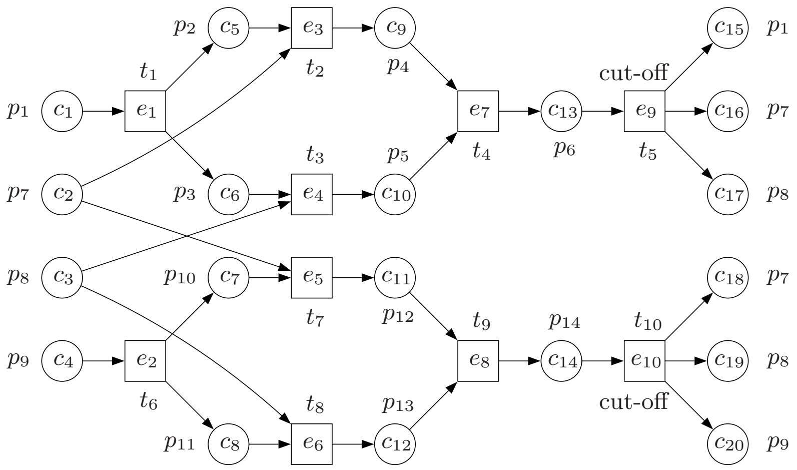
where `&` and `|` denote the Boolean conjunction and disjunction, respectively, and the operator `P` indicates that the following string should be interpreted as a place name.[4] The Reach language contains a few more operators and constructs for accessing the net and iterating through it, some of them are demonstrated in Sec. 4.

The proposed reachability analysis flow is illustrated in Fig. 2. The main novelty is the *expansion* stage that precedes the analysis. Given a net and a Reach property, the tool automatically *expands* this abstract specification into a concrete Boolean expression, which is then optimised and fed to a reachability analyser. The developed MPSat tool implementing the described idea uses an efficient technique based on unfolding prefixes and SAT as the back-end reachability engine. However, many other reachability analysers would fit into the described flow.

3 Net unfoldings

A *finite and complete unfolding prefix* of a bounded Petri net Υ is a finite acyclic net which implicitly represents all the reachable states of Υ together with transitions enabled at those

[3] Note that `@t` is simply 'syntax sugar' for `forall p in pre t { $p }`.
[4] There are also the corresponding operators `T` for transitions and `S` for STG signals, see Sec. 4.



p_2 c_5 e_3 c_9
t_1 t_2 p_4
p_1 c_1 e_1 e_7 c_{13} e_9 c_{15} p_1
cut-off
c_{16} p_7
t_3 p_5 t_4 p_6 t_5
p_7 c_2 p_3 c_6 e_4 c_{10} c_{17} p_8
p_8 c_3 p_{10} c_7 e_5 c_{11} c_{18} p_7
t_7 p_{12} t_9 p_{14} t_{10}
p_9 c_4 e_2 e_8 c_{14} e_{10} c_{19} p_8
t_6 t_8 p_{13} cut-off
p_{11} c_8 e_6 c_{12} c_{20} p_9

Fig. 3. A finite and complete prefix of the unfolding of the Petri net in Fig. 1.

states. Intuitively, it can be obtained through *unfolding* Υ, by successive firing of transitions, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. For example, a finite and complete prefix of the Petri net in Fig. 1 is shown in Fig. 3. Due to its structural properties (such as acyclicity), the reachable states of Υ can be represented using *configurations* of its unfolding. A configuration C is a causally closed set of events (being causally closed means that if $e \in C$ and f is a causal predecessor of e then $f \in C$) without *choices* (i.e. for all distinct events $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \emptyset$). For example, in the prefix shown in Fig. 3, $\{e_1, e_3, e_4\}$ is a configuration, whereas $\{e_1, e_3, e_7\}$ and $\{e_1, e_2, e_3, e_5\}$ are not (the former does not include e_4, which is a predecessor of e_7, while the latter contains a choice between e_3 and e_5). Intuitively, a configuration is a partially ordered execution, i.e. an execution where the order of firing of some of its events (viz. concurrent ones) is not important; e.g. the configuration $C = \{e_1, e_3, e_4\}$ corresponds to two totally ordered executions, $e_1e_3e_4$ and $e_1e_4e_3$, reaching the same marking $\mathit{Cut}(C) = \{c_4, c_9, c_{10}\}$, called a *cut*. This cut corresponds to the marking $\mathit{Mark}(C) = \{p_4, p_5, p_9\}$ of the original net. Since a configuration can correspond to multiple executions, it is often much more efficient in model checking to explore configurations rather than executions. We will denote by $[e]$ the *local* configuration of an event e, i.e. the smallest (w.r.t. $\subseteq$) configuration containing e (it is comprised of e and its causal predecessors).

The unfolding is infinite whenever the original Υ has an infinite run; however, if Υ is bounded and hence has only finitely many reachable states, the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. Intuitively, an event e can be declared cut-off if the already built part of the prefix contains a configuration C^e (called the *corresponding* configuration of e) such that $\mathit{Mark}(C^e) = \mathit{Mark}([e])$ and C^e is smaller than $[e]$ w.r.t. some well-founded partial order on the configurations of the unfolding, called an *adequate order* [ERV02].

Efficient algorithms exist for building such prefixes [ERV02,Kho03], which ensure that the number of non-cut-off events in a complete prefix never exceeds the number of reachable states of the original Petri net. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional interleaving ‘diamonds’ as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. Also, one can observe that if the example in Fig. 3 is scaled up (by increasing the number of philosophers), the size of the prefix is linear in the number of philosophers, even though the number of reachable states grows

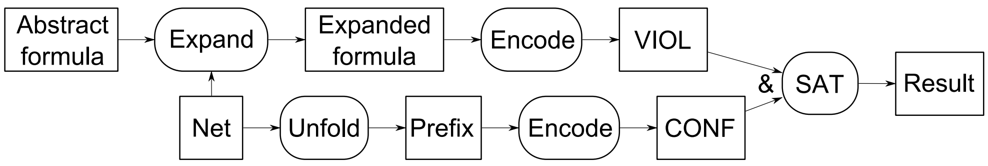


Fig. 4. Reachability analysis flow based on unfolding prefixes.

exponentially. Thus, unfolding prefixes significantly alleviate the state space explosion in many practical cases.

A fundamental property of a finite and complete prefix is that each reachable marking of Υ is a final marking of some configuration C (without cut-offs) of the prefix, and, conversely, the final marking of each configuration C of the prefix is a reachable marking of Υ. Thus a reachability property of Υ can be restated as a reachability property of the prefix (with the corresponding formula obtained automatically from the original one), and then efficiently checked.

Most of 'interesting' problems for safe Petri nets are PSpace-complete [Esp98], but the same problems for prefixes are often in NP or even P. (Though the size of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small, as explained above.) As was already mentioned, a reachability property of Υ can easily be reformulated for a prefix, and then translated into some canonical problem, e.g. Boolean satisfiability (SAT). Then an off-the-shelf solver can be used for efficiently solving it. Such a combination 'unfolder & solver' turns out to be quite powerful in practice [KKY04].

3.1 Unfolding-Based Model Checking

The developed tool MPSat uses the following approach to model checking. First, a finite and complete prefix of the Petri net unfolding is built. Then the expanded reachability formula R is translated into the corresponding formulae $\mathcal{R}$ for the prefix using the method detailed below. Finally, the resulting formulae is combined with the constraint $\mathcal{CONF}$ expressing that a set of events is a configuration of the prefix that contains no cut-off events, and the resulting formula $\mathcal{CONF} \wedge \mathcal{R}$ is fed to a SAT solver. If the formula is not satisfiable then the marking satisfying the original reachability property is not reachable in the original Petri net. Otherwise, the variable assignment returned by the SAT solver allows one to retrieve a configuration of the prefix, which can be translated into an execution of the original net leading to a marking that satisfies the reachability property. This flow is depicted in Fig. 4; one can see that it is a refinement of the flow in Fig. 2.

$\mathcal{CONF} \wedge \mathcal{R}$ has for each non-cut-off event e of the prefix a variable conf_e. For every satisfying assignment A, the set of events $C \overset{\text{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration such that $Mark(C)$ satisfies R. The role of the property-independent *configuration constraint* $\mathcal{CONF}$ is to ensure that C is a configuration of the prefix (not just an arbitrary set of events). $\mathcal{CONF}$ can be defined as the conjunction of the formulae[5]

$$\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in {}^{\bullet\bullet}e} (\overline{\mathsf{conf}_e} \vee \mathsf{conf}_f) \quad \text{and} \quad \bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in Ch_e} (\overline{\mathsf{conf}_e} \vee \overline{\mathsf{conf}_f}),$$

where E is the set of events of the prefix, $E_{cut} \subseteq E$ is the set of cut-offs, and $Ch_e \overset{\text{df}}{=} \{(({}^{\bullet}e)^{\bullet} \setminus \{e\}) \setminus E_{cut}\}$ is the set of non-cut-off events which are in the direct choice relation with e. The former formula is basically a set of implications ensuring that if $e \in C$ then each its immediate

[5] The size of this constraint can be quadratic in the size of the prefix. A linear SAT encoding of $\mathcal{CONF}$ is possible (by introducing auxiliary variables), but it is more complicated and not discussed here, even though it was implemented in MPSat.

predecessor f is also in C, i.e. C is causally closed. The latter one ensures that C contains no choices. $\mathcal{CONF}$ is given in *conjunctive normal form (CNF)* as required by most SAT solvers.

The role of the property-dependent *violation constraint* $\mathcal{R}$ is to express the property violation condition for a configuration C. If a configuration C satisfying this constraint is found then the property does not hold, and C can be translated into a violation trace. We now show how to build the $\mathcal{R}$ constraint given a Boolean reachability expression R (see also [Hel99,Kho03,KKY04,MR97]). We introduce for each non-post-cut-off condition c a Boolean variable cut_c, conveying that c belongs to $\mathit{Cut}(C)$. These variables are related to conf_* (i.e. the indexed conf-variables) by the following constraints:

$$\mathsf{cut}_c \iff \left(\bigwedge_{e\in c^\bullet\setminus E_{cut}} \overline{\mathsf{conf}_e}\right) \wedge \begin{cases} \mathit{conf}_e & \text{if } {}^\bullet c = \{e\} \\ 1 & \text{if } {}^\bullet c = \emptyset. \end{cases}$$

Intuitively, a condition c belongs to $\mathit{Cut}(C)$ iff it is either an initial condition or it has been produced by some (non-cut-off) event in C, and it has not been consumed by any event of C. Furthermore, we introduce for each place p of the Petri net a Boolean variable mark_p, conveying that p belongs to $\mathit{Mark}(C)$. These variables are related to cut_* as follows:

$$\mathsf{mark}_p \iff \bigvee_{\substack{c\in B\setminus E_{cut}^\bullet \\ h(c)=p}} \mathsf{cut}_c,$$

where B is the set of conditions of the prefix. Intuitively, a place belongs to $\mathit{Mark}(C)$ iff some (non-post-cut-off) condition labelled by this place belongs to $\mathit{Cut}(C)$. Now $\mathcal{R}$ can be built simply by rewriting the constraint R using the variables mark_*. For example, the mutual exclusion constraint (1) can be re-written as

$$\mathcal{R} \overset{\mathrm{df}}{=} \Big((\mathsf{mark}_{p_1} \wedge \mathsf{mark}_{p_2}) \vee (\mathsf{mark}_{p_1} \wedge \mathsf{mark}_{p_3}) \vee (\mathsf{mark}_{p_2} \wedge \mathsf{mark}_{p_3})\Big) \wedge \mathcal{DEF},$$

where $\mathcal{DEF}$ is the conjunction of the constraints defining the variables mark_p for $p \in \{p_1, p_2, p_3\}$ and cut_c for all non-post-cut-off conditions c labelled by p_1, p_2 or p_3. Note that the length of $\mathcal{R}$ is linear in the size of the branching process plus the length of R.

3.2 Prefix-based formula generation

In fact, an advanced user can specify the reachability property directly on the prefix, which often can result in a smaller SAT instance. For example, the deadlock property can be specified as

$$\bigwedge_{e\in E} \Big(\bigvee_{f\in{}^{\bullet\bullet}e} \overline{\mathsf{conf}_f} \vee \bigvee_{f\in({}^\bullet e)^\bullet\setminus E_{cut}} \mathsf{conf}_f \Big),$$

which intuitively means that C cannot be extended by any event e as either some predecessor f of e is not in C or some non-cut-off event g which is in direct choice relation with e is already in C, and hence C is a deadlocked configuration. Note that in the prefix the cut-off events serve as the border separating the fake deadlocks introduced by the truncation of the unfolding. Since the $\mathcal{CONF}$ constraint ensures that C contains no cut-off events, it is guaranteed that C can be extended by a (possibly cut-off) event e iff $\mathit{Mark}(C)$ is not a deadlock. The corresponding Reach specification is as follows:

```
forall e in EVENTS {
    let pre_e = pre e {
        exists f in pre pre_e { ~$f } |
        exists g in post pre_e s.t. ~is_cutoff g { $g }
    }
}
```

where `EVENTS` refers to the set of events of the prefix, the operator `$` applied to an event refers to its status, i.e. whether the event is in C or not, the `is_cutoff` predicate evaluates to true iff its parameter is a cut-off event, the `let` operator introduces a name for a common subexpression, and the `s.t.` clause in the `exist` operator allows one to restrict its variable's domain.[6] Alternatively, the above specification can be re-written as

```
forall e in EVENTS { ~@e }
```

where the operator `@` applied to an event refers to its enabledness status, i.e. it is true iff C can be extended by this event.

These REACH specifications can be automatically translated into the $\mathcal{R}$ formula, combined with $\mathcal{CONF}$ and fed to a SAT solver. Their advantage is that the mark_* and cut_* variables, as well as the corresponding defining constraints, are not generated.

4 Case studies

In this section we present a number of case studies. They come mostly from the domain of asynchronous circuits (ACs), which, intuitively, are circuits without clocks. ACs have been getting more and more attention in the last few years, as they often have lower power consumption and electro-magnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations. Though the listed advantages look rather attractive in the view of the current and anticipated microelectronics design challenges, correct and efficient ACs are notoriously difficult to design (there are a few published asynchronous designs which subsequently turned out to be incorrect).

We focus on an important subclass of ACs, called *speed-independent (SI)* circuits; this model follows the classical Muller's approach [MB59] and regards each gate as an atomic evaluator of a Boolean function, with a delay element associated with its output. In the SI framework this delay is unbounded, i.e. the circuit must work correctly regardless of its gates' delays, and the wires are assumed to have negligible delays (or, alternatively, wire forks are assumed to be isochronic — in such a case the circuit is often referred to as *quasi-delay-insensitive (QDI)* [Mar90]; for the purposes of this paper, these two models are indistinguishable).

We now briefly explain Signal Transition Graphs (STGs) [Chu87,CKK$^+$02,RY85] — a Petri net based formalism which is widely used for specifying asynchronous circuits. STGs are a particular type of labelled Petri nets. They associate a set of Boolean variables, referred to as *signals*, with a Petri net to represent the state of the actual digital signals (i.e. wires) within a circuit. The Petri net's transitions are then labelled to represent changes in the state of these signals; a transition label has the form either $a+$ to indicate a signal a goes from 0 to 1, or $a-$ to indicate the signal goes from 1 to 0. In general, several transitions can have the same label, e.g. $a+$; in such a case, these transitions are named $a+$, $a+/1$, $a+/2$, etc. Thus, the underlying Petri net specifies the causal relationship between signal changes and is intended to capture the behaviour of a circuit. Clearly, for an STG to correctly represent a circuit one has to ensure that the labels $a+$ and $a-$ are correctly alternated between for each signal.

An STG can be represented graphically simply as a labelled Petri net. However, a shorthand notation is often used, in which transitions are simply represented by their labels, and places with one incoming and one outgoing arc are contracted. Moreover, we will use a single grey line without arrowheads to represent *read arcs*, i.e. pairs of arcs (p, t) and (t, p) with the same end nodes and opposite directions. Such arcs are used to test for the presence of a token in a place without consuming it.

[6] Note that `exists x in X s.t. Y { Z }` is 'syntax sugar' for `exists x in X { Y & Z }`, and, similarly, `forall x in X s.t. Y { Z }` can be replaced by `forall x in X { Y -> Z }`, where the `->` operator denotes Boolean implication.

– Each signal (i.e. wire) s is represented by two places, p_s^1 and p_s^0, indicating whether the corresponding voltage is high or low, respectively. Exactly one of these places is marked at any time.
– Since we do not have any information about the environment's behaviour at this stage, it is taken to be the most general (i.e. it can always change the value of any input). This is modelled for each input signal s by adding transitions $s+$ (consuming a token from p_s^0 and depositing a token to p_s^1) and $s-$ (consuming a token from p_s^1 and depositing a token to p_s^0).
– For each local signal s the circuit computes the next-state value $[s]$ of s using a logic gate or a latch, which can be described by a Boolean equation $[s] = E_s$. (In the case of a latch s occurs in E_s.) For each term (i.e. prime implicant) m_i in the minimised disjunctive normal form (DNF) of $E_s|_{s=0}$ (where $E_s|_{s=b}$ denotes the Boolean expression resulting from substituting s by $b \in \{0,1\}$ in E_s), we add a transition $s+/i$ which switches s on. We add an arc from place p_s^0 to $s+/i$ and an arc from $s+/i$ to place p_s^1. For each s' (resp. $\overline{s'}$) occurring in m_i, we connect $s+/i$ to the place $p_{s'}^1$ (resp. $\overline{p_{s'}^0}$) by a read arc. We use a similar process to define the transitions $s-/i$ which reset s based on $\overline{E_s|_{s=1}}$.

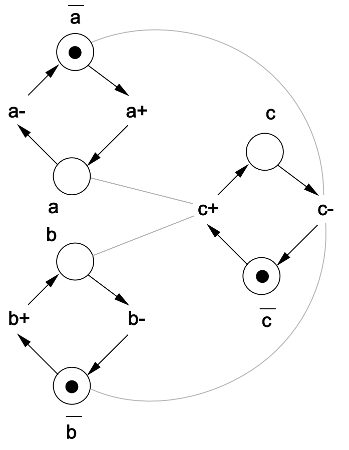
Fig. 5. The circuit-STG construction.

The signals of an STG are partitioned into *input*, *output* and *internal* signals; the output and internal signals are collectively referred to as *local* signals. The inputs are controlled by the environment of the STG, and the outputs are controlled by the system itself and are observable by the environment. Internal signals represent some auxiliary entities needed to produce outputs; like outputs, they are controlled by the system, but are not observable by the environment.

The behaviour of an STG is based on its underlying Petri net's behaviour, in particular the concepts of enabling and firing of transitions are the same. Intuitively, an STG represents a contract between the system and its environment, and is interpreted in the following way. If an input signal transition is enabled, then the environment is allowed (but is not obliged) to send this input, and vice versa, the environment is not allowed to send inputs which are not enabled. If a local transition is enabled, then the system is obliged eventually to produce this signal (or it is eventually disabled by another transition, in which case the *output-persistency* (discussed later) is violated), and vice versa, it is not allowed to produce outputs which are not enabled. That is, an STG specifies the behaviour of a system in the sense that the system must provide *all and only* the specified outputs, and that it must allow *at least* the specified inputs (in fact, it could optionally allow more inputs, which means that it could work in a more demanding environment).

STGs can be used for different purposes. One way of using them is to model and verify a gate-level circuit. Any digital circuit can be converted into an STG using a variant of the well-known translation based on complementary places [Rei85], see Fig. 5. Fig. 6(left) illustrates this construction for the C-element circuit. Then, one has to separately model the environment's behaviour as an STG,[7] as illustrated in Fig. 6(right), and then the parallel composition [VW02] of these two STGs (which is also an STG) is computed. (Roughly speaking, parallel composition simply puts two STGs side-by-side and fuses pairs of identically labelled transitions.) This combined STG can then be used to verify the properties of the circuit, e.g. violations of SI can be automatically detected by reachability analysis. Alternatively, an STG can be provided by the designer as a specification of a circuit. In such a case a number of properties have to be checked to ensure its implementability as an SI circuit.

[7] It is important to note that one cannot speak about the speed-independence of a circuit per se, as the gate-level description carries no information about the behaviour of the environment. In particular, a circuit can be SI in one environment, and non-SI in another.



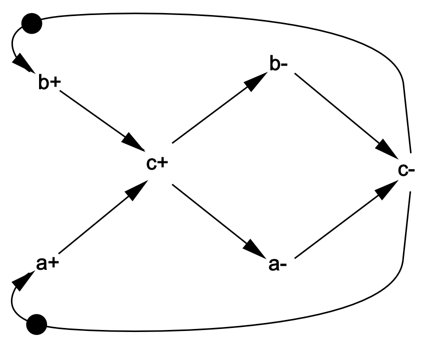


Fig. 6. The circuit-STG for the C-element circuit $[c] = ab + c(a + b)$ (left), and the STG describing the behaviour of its environment (right).

We first explain how to check the basic properties required for an STG to be implementable as an SI circuit, and then show how several custom properties of a flat arbiter circuit proposed in [MKY09] have been verified.

4.1 Consistency

One of the basic well-formedness properties of STGs is the *consistency,* requiring that in each possible execution, the transitions representing the rising and falling edges of each signal must be correctly alternated between, always starting from the same edge (either rising or falling). This ensures that the values of all the signals are always binary. Consistency is guaranteed to hold for STGs obtained from circuits, but for manually constructed STG specifications it has to be formally verified.

Consistency can be checked using the following REACH specification:

```
exists s in SIGNALS {
    let Ts = tran s {
        $s & exists t in Ts s.t. is_plus t { @t }
        |
        ~$s & exists t in Ts s.t. is_minus t { @t }
    }
}
```

Here the operator `tran` computes the set of all transitions labelled by a signal, the predicates `is_plus` and `is_minus` check that the transition is labelled by a rising and falling, respectively, signal edge, and the operator `$` applied to a signal refers to its status, i.e. whether the signal is high or not.[8] Note that if the STG is not consistent and the value of a signal s is not binary at some reachable state, the `$` operator will simply return that value modulo 2.[9] However, since all the signal values are binary at the initial state, one can show that if consistency is violated

[8] Similarly, an operator `@` applied to a signal refers to its enabledness status, i.e. it is true iff some transition labelled by this signal is enabled. One can see that `@s` is just 'syntax sugar' for `exists t in tran s { @t }`.

[9] The status of a signal s is computed from a configuration C by counting modulo 2 the number of events labelled by s, i.e. `$s` is a 'syntax sugar' for `is_init s ^ xorsum e in ev s { $e }`, where `is_init` is the operator returning the initial value of the signal, `^` is the binary 'exclusive or' operator, `xorsum` is the iterated 'exclusive or' operator, and `ev` returns the set of events labelled by a signal.

at some state with some signal having a non-binary value, it is also violated at some earlier state where all signals had binary values, and so the Reach specification above is correct. In what follows, we assume that all the STGs are consistent.

4.2 Output persistency

The *output persistency (OP)* is a correctness condition requiring that if some local signal becomes enabled, it cannot be disabled by firing some other transition, i.e. there should be no choices involving local transitions. The rationale for this is that once a signal becomes enabled, its voltage starts, e.g. to rise from 0 to 1. If the signal is disabled during this process, the voltage is suddenly pulled down, resulting in a glitch. This glitch can be interpreted in different ways by the logic gates listening to this signal, depending on whether the voltage has crossed the threshold between 0 and 1 or not. Hence the behaviour of the circuit becomes non-deterministic and non-digital.

Visually, if OP is violated then there are two transitions with different labels in the STG with at least one of them marked by a local signal, which share some pre-places and can be enabled simultaneously (unless both transitions are connected to these shared pre-places by read arcs).

Note that a choice involving only inputs is not a violation of OP, and simply models a choice in the environment. Since this choice does not have to be implemented by the system, SI circuits can be synthesised for such STGs (provided that all the other conditions necessary for SI are met).

A Reach specification for OP is as follows:

```
exists t1 in TRANSITIONS s.t. sig(t1) in LOCAL {
  @t1 &
  exists t2 in TRANSITIONS s.t. sig(t2)!=sig(t1) &
                                 |pre(t1)*(pre(t2)\post(t2))|!=0 {
    @t2 &
    forall t3 in tran(sig(t1))\{t1} s.t. |pre(t3)*(pre(t2)\post(t2))|=0 {
      exists p in pre(t3)\post(t2) { ~$p }
    }
  }
}
```

Here the operator `sig` returns the signal of a transition, the operators `*`, \ and `|...|` denote the set intersection, difference and cardinality, respectively, the operator `tran` returns a set of transitions labelled by a signal, and $\{\ldots\}$ (in $\{$`t1`$\}$) is a set constructor.

Intuitively, we are looking for a state enabling some transition t_1 labelled by a local signal (lines 1 and 2), which can be disabled by some transition t_2 labelled by a different signal (lines 3–5). The disabling condition is that t_2 is enabled and ${}^\bullet t_1 \cap ({}^\bullet t_2 \setminus t_2^\bullet)$, i.e. t_2 consumes (not just reads!) some token from ${}^\bullet t_1$. Lines 6 and 7 specify that after t_2 fires, no other transition with the same label as t_1 is enabled, i.e. the signal of t_1 has been disabled.

Note that in some cases (e.g. if the net has no structural choices) the expanded formula simplifies to Boolean 0, and no reachability analysis is actually needed. In such cases MPSat does not call a SAT solver and simply prints the result.

4.3 Complete State Coding (CSC) and Universal State Coding (USC)

If the STG has two reachable states in which the values of all the signals coincide, but the sets of enabled local signals are different, then these two states are said to be in *Complete State Coding (CSC)* conflict. The STG satisfies the *CSC property* if no two of its reachable states are in CSC conflict.

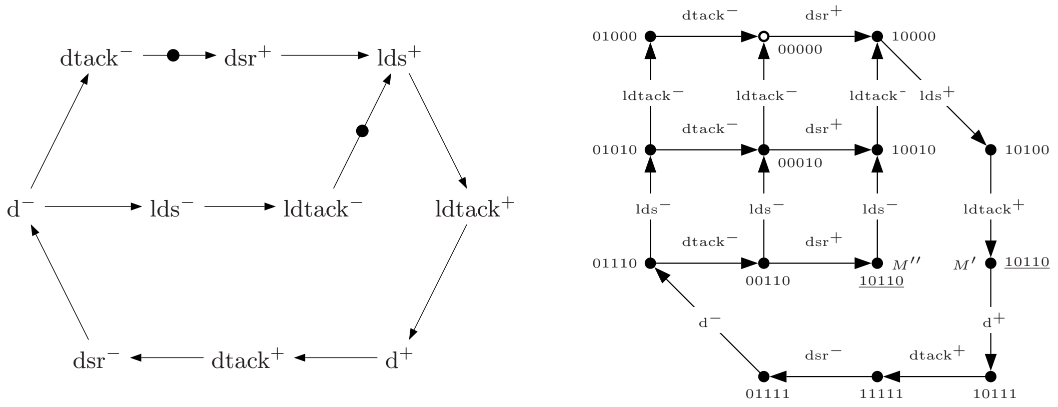


Fig. 7. An STG modelling a simplified VME bus controller (left) and its state graph with a CSC conflict between two states (right). The order of signals in the binary codes is: *dsr*, *dtack*, *lds*, *ldtack*, *d*.

An STG not satisfying the CSC property cannot be directly implemented as an SI circuit. Intuitively, during its execution the circuit can 'see' only the values of its signals, but not the marking of the STG. Hence, if two semantically different reachable states with the same values of all the signals exist, the system cannot distinguish between them, and so cannot know what to do next. An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark, see, e.g. [CKK$^+$02]) is shown in Fig. 7(left). The picture on the right of this figure illustrates a CSC conflict between two different states, M' and M'', that have the same values of all signals, but M' enables only d and M'' enables only *lds*. This means that, e.g. the gate producing *lds* does not have sufficient information: the states M' and M'' are indistinguishable from its point of view, but the next-state value of *lds* should be 1 according to the state M' and 0 according to the state M''.[10]

Note that in fact the CSC property is not a reachability property of the STG as defined in the beginning of this paper, as one has to look for *two* reachable states that are in a certain relationship. A possible way around this is to put two copies of the STG side by side, and formulate the CSC property in the original STG as a reachability property of this joint STG. However, the approach implemented in MPSAT is to generalise the reachability properties. A *generalised reachability property* is a property that can be formulated as follows:

Check if there are reachable states $s_1, \ldots, s_k$ *satisfying some predicate* $R(s_1, \ldots, s_k)$.

It is now easy to see that the CSC property can be formulated as a generalised reachability property for $k = 2$:

```
forall s in SIGNALS { $s <-> $$s } & exists s in LOCAL { @s^@@s }
```

Note that the operator `<->` denotes Boolean equivalence,[11] and operators `$$` and `@@` are analogous to `$` and `@`, but refer to the second state.[12]

To check generalised reachability properties using unfoldings, one can generalise the approach describing in Sec. 3.1 by generating k sets of variables conf_*, cut_* and mark_*, as well as

[10] In practice, to resolve a CSC conflict, new *internal* signals helping to distinguish between the conflicting states are inserted into the STG in such a way that its 'external' behaviour does not change. Intuitively, insertion of a signal introduces additional memory into the circuit, helping it to trace the current state. For example, the CSC conflict shown in Fig. 7 can be resolved with the help of an additional internal signal *csc*, as shown in Fig. 8.

[11] `a <-> b` is syntax sugar for `~(a^b)`.

[12] The operators `$$$` and `@@@` refer to the third state, etc. However, the author is not aware of any practical properties that would require $k > 2$.

k copies of the $\mathcal{CONF}$ and $\mathcal{DEF}$ constraints, one for each of the k sets of variables. (The $\mathcal{R}$ formula is not replicated, but it now can refer to variables from several sets.)

Historically, the Universal State Coding (USC) property has been used as a sufficient condition for CSC. Two different reachable states are said to be in a *USC conflict* if the values of all the signals in these states coincide. The STG satisfies the *USC property* if no two of its reachable states are in USC conflict. One can see that the USC property indeed implies the CSC property (but not vice versa). It can be checked using the following Reach specification:

```
forall s in SIGNALS { $s <-> $$s } & exists p in PLACES { $p^$$p }
```

4.4 Normalcy

The property of *normalcy* [SBG+01] is a necessary condition for an STG to be implementable using only logic gates without input inversions (‘bubbles’). This in turn guarantees that the circuit is SI without the necessity to neglect the delays of input inverters.

To define normalcy formally, we need the following auxiliary definitions. For each reachable state M of the STG, we denote by $Code(M)$ the Boolean vector comprised of the values of all signals at this state, and by $Code_s(M)$ we denote the component of $Code(M)$ corresponding to a signal s (note that $Code_s(M)$ can be computed by the Reach expression `$s`). Furthermore, by $Out_s(M)$ we denote the enabledness status of a signal s at this state (note that $Out_s(M)$ can be computed by the Reach expression `@s`). Now the *next-state function* Nxt_s *for a local signal* s can be defined as

$$Nxt_s(M) \overset{\text{df}}{=} Code_s(M) \oplus Out_s(M),$$

where $\oplus$ is the ‘exclusive or’ operation. The CSC condition introduced above ensures that the value of this function is determined without ambiguity by the encoding of each reachable state, i.e. $Nxt_s(M)$ is a function of $Code(M)$ rather than of M: $Nxt_s(M) = F_s(Code(M))$ for some Boolean function F_s. (F_s will eventually be implemented as a logic gate).

An STG satisfies the *positive normalcy* (or *p-normalcy*) condition w.r.t. a local signal s if for every pair of reachable states M' and M'', $Code(M') \leq Code(M'')$ implies $Nxt_s(M') \leq Nxt_s(M'')$. Similarly, it satisfies the *negative normalcy* (or *n-normalcy*) condition w.r.t. a local signal s if for every pair of reachable states M' and M'', $Code(M') \leq Code(M'')$ implies $Nxt_s(M') \geq Nxt_s(M'')$. Finally, an STG is *normal* if w.r.t. each local signal it is either p-normal or n-normal. (It turns out that normalcy implies CSC [SBG+01].)

An example of normalcy violation is illustrated in Fig. 8. This STG is implementable — the gates for all the local signals are determined by the following functions:

$$F_{lds} = d \vee csc \qquad F_{dtack} = d \qquad F_d = ldtack \wedge csc \qquad F_{csc} = dsr \wedge (\overline{ldtack} \vee csc)$$

Nonetheless, normalcy is violated for signal csc. Indeed, because $Code(M) < Code(M'')$ and $Nxt_{csc}(M) > Nxt_{csc}(M'')$, csc cannot be p-normal; on the other hand, $Code(M') < Code(M)$ and $Nxt_{csc}(M') < Nxt_{csc}(M)$, so csc cannot be n-normal. This is reflected in the implementation of csc, which is positive w.r.t. dsr and negative w.r.t. $ldtack$ (note that the corresponding gate has an input inverter).

The following Reach specification allows one to verify the normalcy property:

```
exists s in LOCAL {
    let pos = exists e in ev s, f in trig e { is_plus f <-> is_plus e },
        neg = exists e in ev s, f in trig e { is_plus f ^ is_plus e } {
            pos & neg | pos & s' & ~s'' | neg & ~s' & s''
    }
}
&
forall ss in SIGNALS { $ss -> $$ss }
```

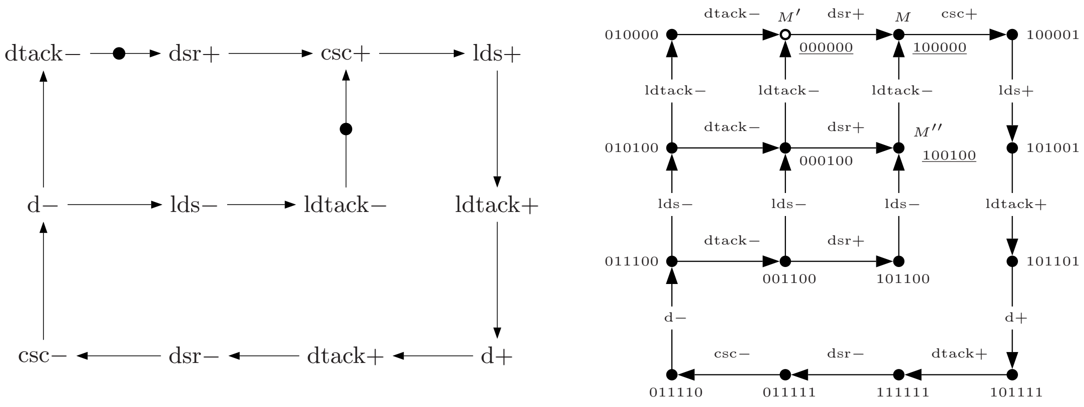


Fig. 8. An STG (left) and its state graph (right). The STG satisfies the CSC property but has a normalcy violation for signal *csc*, as witnesses by the states M, M' and M''. The order of signals in the binary codes is: *dsr*, *dtack*, *lds*, *ldtack*, *d*, *csc*.

Here, the `ev` operator computes the set of events labelled by a particular signal, the `trig` operator computes the set of *triggers* of an event e, i.e., informally, the set of events whose firing can enable e,[13] and the postfix operator `'` computes the next-state function of a signal.[14]

Intuitively, in lines 2 and 3 we make hypotheses about the normalcy type of s based on its triggers. Indeed, if an event e labelled by s has a trigger labelled by a signal transition with the same edge ('+' or '−') then the signal cannot be n-normal, and so a hypothesis that it is p-normal is made. Similarly, if e has a trigger labelled by a signal transition with the opposite edge then the signal cannot be p-normal, and so a hypothesis that it is n-normal is made. The first part of the expression in Line 4 checks whether contradictory hypotheses have been made about the normalcy type of s (in such a case the normalcy is trivially violated), and the remaining two parts of this expression check the violation of the normalcy condition for s depending on what hypothesis about its type (p- or n-normal) has been made. Line 8 simply specifies that $Code(M') \leq Code(M'')$, as required in the definition of normalcy.

In practice, if normalcy is violated then this violation in most cases is due to contradictory hypotheses made for some signal (e.g. this is the case for the STG in Fig. 8). That is, the expression in lines 1–6 is simplified to 1 during the expansion phase, and so only the expression in line 8 remains; though one can see that the resulting SAT instance will be trivially satisfiable, it would be advantageous to simplify the whole specification to 1 in the case of contradictory hypotheses, so that the call to SAT solver is completely avoided. This can be accomplished by the following Reach specification:

```
exists s in LOCAL {
    let pos = exists e in ev s, f in trig e { is_plus f <-> is_plus e },
        neg = exists e in ev s, f in trig e { is_plus f ^ is_plus e } {
            pos & neg
    }
}
|
exists s in LOCAL {
    let pos = exists e in ev s, f in trig e { is_plus f <-> is_plus e },
        neg = exists e in ev s, f in trig e { is_plus f ^ is_plus e } {
            // the pos and neg cannot hold together -
            // checked by the trivial case above
```

[13] The triggers of e are exactly the causally maximal events in $^{\bullet\bullet}e$.

[14] Note that `s'` is 'syntax sugar' for `$s^@s`.

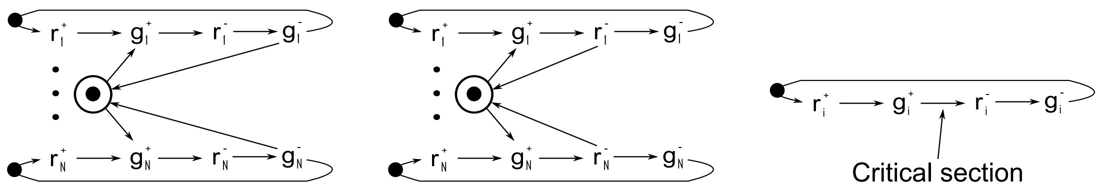


Fig. 9. A specification of an N-way arbiter: the traditional (left) and early (middle) protocols, together with a model of a client (right).

```
            pos & s’ & ~s’’ | neg & ~s’ & s’’
    }
}
&
forall ss in SIGNALS { $ss -> $$ss }
```

Intuitively, the first part checks if one can make contradictory hypotheses for some signal; if this is the case, then this part evaluates to 1 during expansion, causing the whole expression to evaluate to 1. Otherwise this part evaluates to 0 and does not affect the SAT instance.[15]

4.5 Model checking of flat arbiters

Arbiters [Kin07] are basic blocks guarding access to shared resources, and as such, they play a very important role in circuit design. The top-level specification of an N-way arbiter is shown in Fig. 9(left). Suppose there are N clients using a shared resource in a mutually exclusive way (Fig. 9(right) shows an STG modelling the behaviour of each client). Before accessing the resource, i^{th} client sends a request to the arbiter (by raising signal r_i). Such requests can be sent concurrently by different clients. In response, the arbiter issues a grant (by raising signal g_i). At most one of $g_1, \ldots, g_N$ can be high at any time, no matter how many concurrent requests have been received by the arbiter. Upon receipt of the grant, a client can safely use the resource, with the guarantee that no interference is possible from other clients. Having finished using the resource, the client lowers its request r_i, and in response the arbiter lowers g_i. At this point the arbiter can issue a grant to another client.

An alternative *early* protocol is shown in Fig. 9(middle); the difference here is that once i^{th} client lowers the request r_i, the arbiter is allowed to immediately issue a grant g_j $(j \neq i)$ to another client, in parallel with lowering the grant g_i. Hence, g_i and g_j can be simultaneously high, but this is harmless since i^{th} client has already declared (by lowering r_i) that it had finished using the shared resource, and, according to this early protocol, it will not send another request (i.e. raise r_i again) until the arbiter lowers g_i.

N-way arbiters are usually constructed using basic 2-way *mutual exclusion (ME) elements.* (It is a well-known fact that one cannot construct even a 2-way arbiter using only digital logic gates [Var90].) When the two requests arrive almost simultaneously, the ME element, like Buridan’s ass, has to make an arbitrary choice between them. It enters a *metastable* state, in which it can stay indefinitely. Though the time for resolving metastability is exponentially distributed, and so most arbitrations are fast, the fact that there is no upper bound on this time means that systems that require the arbitration decision within bounded time occasionally fail. Hence, reducing the latency is one of the key objectives in arbiter design.

One of traditional ways of designing N-way arbiters is to combine the basic ME elements in a balanced tree-like fashion. This design is simple and results in a small circuit; however, its disadvantage is that several $(\log N)$ arbitrations happen sequentially. This can significantly

[15] Generally, any REACH subexpression that does not contain occurrences of the ‘status’ operators `$`, `@` and `’` (either directly or indirectly, via a name defined by the `let` operator) will evaluate to a constant during the expansion stage.

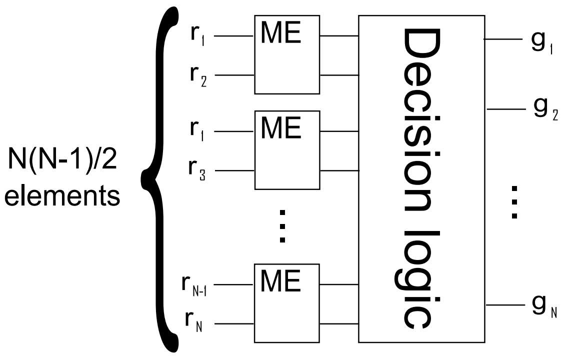


Fig. 10. The top-level view of the flat arbiter.

increase the latency of the arbiter, especially in balanced systems where the requests usually arrive almost simultaneously, and so several sequential ME elements, one after another, can spend long time in their metastable states.

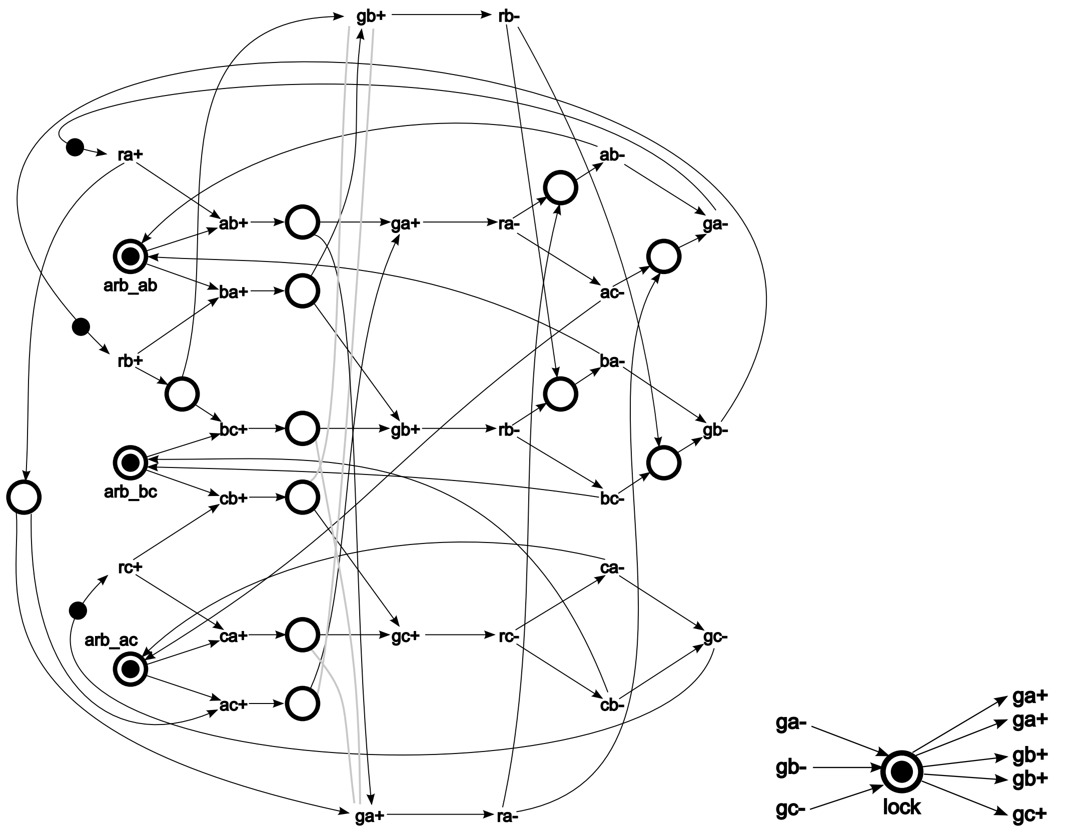
In [MKY09] an alternative way of constructing N-way arbiters was proposed. The main idea was to perform concurrent arbitrations between all pairs of requests, and then make the decision on what grant to issue based on their outcomes, see Fig. 10. Crucially, all the ME elements in such an arbiter work in parallel (hence the name 'flat arbiter'), and the subsequent decision logic has bounded latency.

In [MKY09] a 3-way flat arbiter implementing the early protocol was designed as an STG shown in Fig. 11. It can be converted into an STG of the traditional arbiter by augmenting it with a new initially marked place *lock*, the arcs from it to all the transitions labelled $ga+$, $gb+$ and $gc+$, and the arcs from all the transitions labelled $ga-$, $gb-$ and $gc-$ to it, as illustrated in the bottom-right corner of the picture (note that there are two transitions labelled $ga+$, and two transitions labelled $gb+$ in the STG). Intuitively, this new place does not allow to issue a grant until the previous grant has been lowered. In spite of being in the preset of several output transitions, *lock* does not cause violations of output persistency, as the transitions corresponding to the rising edges of the grant signals cannot be enabled simultaneously (i.e. this is a *controlled choice*).

Both the original and augmented STGs can be automatically synthesised as SI circuits. However, building such STGs for larger N and then synthesising them is impractical [MKY09], so a generic way of constructing N-way flat arbiters was also developed in [MKY09]. For $N = 3$ this generic approach results in the flat arbiter circuit implementing the early protocol, which is shown in Fig. 12. (This circuit is different from the one obtained by synthesising the STG in Fig. 11.) As explained in the beginning of this section, a circuit can be converted into an STG using the circuit-STG construction in Fig. 5 and composing it with the STG modelling the environment, which in this case is simply three copies of the client STG shown in Fig. 9(right).

Deadlock checking of a flat arbiter The correct STGs modelling flat arbiters must obviously be deadlock free. Unfortunately, the standard deadlock checking does not quite work, as the definition of the deadlock in an arbiter is slightly different.[16] Indeed, in the situation when only some requests have arrived, the arbiter is obliged to eventually issue a grant, *even when the remaining requests never arrive.* Hence, if some state (except the initial one) does not enable any transitions besides the rising request transitions ($ra+$, $rb+$ and $rc+$ in Fig. 11) then it is classified as a deadlock. Another way of putting it is that in the STG the rising

[16] Like in the example with the proper termination described in the introduction, deadlock freeness of an arbiter is yet another minor variation of a standard property, rendering the standard deadlock checking engines virtually useless.



inputs: $ra, rb, rc, ab, ba, bc, cb, ac, ca$; **outputs:** ga, gb, gc

Fig. 11. STG specification of the decision logic of an early 3-way flat arbiter, and a way of inserting a *lock* place into it to implement the traditional protocol.

request transitions are not *weakly fair,* i.e. they may remain enabled forever, without firing. To summarise, the differences from the standard deadlock checking are:

- the rising request transitions are not weakly fair, i.e. any state (except the initial one) enabling only such transitions is a deadlock;
- though the initial state enables only the rising request transitions, it is not a deadlock (i.e. this state has to be treated in a special way).

A Reach specification of this property is as follows:

```
let requests = {T"ra+", T"rb+", T"rc+"} {
    forall t in TRANSITIONS\requests { ~@t }
}
&
exists p in PLACES { $p ^ is_init p }
```

Intutitively, the `let` operator defines the set of transitions which are not weakly fair,[17] the subexpression starting with the `forall` ketword is similar to the standard deadlock specification, except that the transitions that are not weakly fair are not required to be disabled, and the `exists` operator eliminates the initial state from consideration by requiring that the marking of some place is different from its initial marking (the latter is returned by the `is_init` operator).

The `let` statement in the above specification can be made more general by using a regular expression to define the set of all requests:

[17] Recall that the `T` operator interpretes the following string as a transition name.

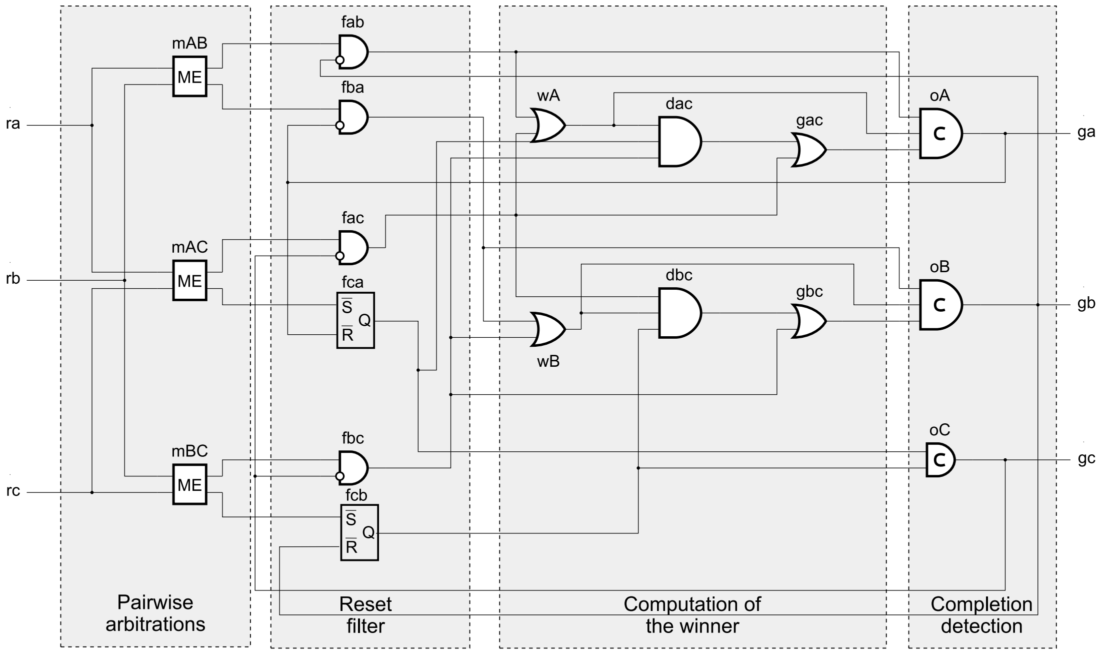


Fig. 12. A decomposed implementation of a 3-way flat arbiter.

```
let requests = TT "r[a-z]\\++\\(/[0-9]\\+\\)\\?" ...
```

Here the TT operator[18] computes the set of all transitions whose name matches the regular expression given by the following string.[19] Intuitively, this regular expression matches the strings starting with 'r', followed by a non-empty sequence of letters, followed by a '+', optionally followed by a '/' with a number appended.[20] Note that using a regular expression allows one to use the same Reach specification to check the deadlock freeness of an N-way flat arbiter for any N, provided that the names of rising request transitions (and only such names) match this pattern.

Mutual exclusion checking of a flat arbiter Another important property of flat arbiters is the mutual exclusion of the grants. This is similar to the standard mutual exclusion property, except that it is formulated for signals rather than places. Though such a property would not be directly checkable using a standard mutual exclusion checker, there is a standard construction allowing one to add for each signal of the STG a place which is marked iff the corresponding signal is high. Hence, a standard mutual exclusion checker can be used if the user is willing to modify the original STG. However, it is much easier to use the following Reach specification:

```
let a = $S"ga", b = $S"gb", c = $S"gc" { a & b | b & c | a & c }
```

Note that if the number of entities that have to be checked for mutual exclusion is large, such a specification can become quite long, as its size is quadratic in the number of entities. Hence Reach provides the threshold operator for writing such specifications concisely:

[18] The Reach language has also the analogous operators PP and SS, computing the set of places and signals, respectively, whose names match a given regular expression.

[19] Currently the basic POSIX regular expressions are supported. Note that POSIX requires all the regular operators and brackets for grouping to be escaped with a backslash; moreover, since the usual C escape sequences are applied when parsing the string, double backslashes are needed in this specification.

[20] Recall that the rising transitions for a signal a are labelled $a+$, $a+/1$, $a+/2$, etc.

```
threshold[2]($S"ga", $S"gb", $S"gc")
```

Intuitively, the `threshold` operator evaluates to 1 iff the number of inputs evaluating to 1 is not smaller than the threshold (given in `[...]`). Similarly to the deadlock specification for an arbiter given above, one can use a regular expression to specify the set of signals:

```
let grants = SS "g[a-z]\\+" {
    threshold[2] g in grants { $g }
}
```

Note that the iterative form of the `threshold` operator is used here.

The above specifications can be used to check the mutual exclusion property of arbiters implementing the traditional protocol (i.e. the STG in Fig. 11 must be augmented with the *lock* place). However, arbiters implementing the early protocol violate these properties; in fact, increasing the value of the threshold to 3 in the above specifications allows one to establish that the STG in Fig. 11 and the circuit in Fig. 12 have reachable states in which all three grants are high (which is correct according to Fig. 9(middle)).

Hence the mutual exclusion specification for arbiters implementing the early protocol has to be changed as follows. Instead of requiring that there is at most one client whose grant is high, one should require that there is at most one client for which both the request and the grant are high. Again, this is a minor variation of the standard mutual exclusion property which is not easy to check with standard tools. However, it is easy to capture with the following REACH specification:

```
let a = $S"ra" & $S"ga", b = $S"rb" & $S"gb", c = $S"rc" & $S"gc" {
    a & b | b & c | a & c
}
```

Similarly to the above specifications for the traditional protocol, this specification can be shortened using the `threshold` operator:

```
threshold[2]($S"ra" & $S"ga", $S"rb" & $S"gb", $S"rc" & $S"gc")
```

Furthermore, a regular expression can be used to specify the set of requests (together with the iterative form of the `threshold` operator):

```
let req = SS "r[a-z]\\+" {
    threshold[2] r in req {
        $r & $S("g" + (name r)[1..])
    }
}
```

Here, the `name` operator returns the name of the entity (place, transition or signal) it is applied to, the `+` operator concatenates strings, and the `[`m`..`n`]` operator extracts a substring from a string, from m^{th} to n^{th} character, inclusive. Optionally, one of the indices in this operator can be dropped, e.g. in this example the `[1..]` operator returns the original string without the head character.[21] Intuitively, in this example we assume that the names of the request and grant signals of a client differ only in the head character (it is 'r' for requests and 'g' for grants). Hence, the name of a grant signal can obtained from the name of the corresponding request signal by dropping the head character and then pre-pending the result with 'g'.

5 Conclusions and future work

Existing reachability analysers either require the user to input the formula manually, which can be very tedious and error-prone, or automatically generate formulae for some fixed set of

[21] The numbering of characters in the string starts from 0.

common properties, which rules out custom properties. In this paper a solution to the problem of generating formulae expressing custom reachability properties is suggested. The proposed approach allows the user to write a concise abstract specification of the property in a specially developed language Reach, which is then automatically expanded into a formula for a concrete model. The usefulness of this method is demonstrated on several case studies.

The presented idea can be extended to other formalisms in a straightforward way. For example, to extend the Reach language to general Petri nets it is enough to change the semantics of the 'status' operator `$`: it should return a non-negative integer (the number of tokens in a place) rather than a Boolean value.[22] Similarly, to extend Reach to coloured Petri nets, the operator `$` should return a multiset of token colours, which are elements of the algebra of colours (whose operators should also be added to Reach), and the operator `@` should take into account the transition guards. Generally, most formalisms that have an explicit notion of state can be adopted without much difficulty.

An orthogonal way of extending Reach is to use a different class of properties. As the semantics of Reach is simply a Boolean expression, one can easily add various modalities to the language, such as LTL or CTL temporal modalities. For example, the following specification defines the property that whenever a client sends a request to an arbiter, it is eventually granted:[23]

```
let req = SS "r[a-z]\\+" {
    forall r in req {
        []($r -> <>$S("g" + (name r)[1..]))
    }
}
```

Here the `[]` and `<>` operators correspond to the LTL modalities $\Box$ and $\Diamond$, respectively.

On the practical side, the efficiency of Reach expander can be improved by enhancing it with the capability to identify and share common subterms, i.e. the underlying data structure would become a DAG rather than a tree. (Such common subterms are often created when expanding the iterative operators such as `forall` and `exists`.) This change would be invisible to the user, while increasing the efficiency and resulting in smaller SAT instances. At the user level, Reach already provides a possibility to share common subexpression by means of the `let` operator. However, this is quite rudimentary; one can add more powerful constructs, such as recursive definitions and graph rewriting rules.

Other potential extensions include a more elaborated way of interfacing the SAT solver, e.g. the ability to look up the returned satisfying assignment and formulate new reachability queries depending on it. (Contemporary SAT solvers can be used in the incremental mode, when the SAT instance is repeatedly modified by adding and/or removing a small number of clauses, and the knowledge gathered while solving the previous instance can be partially re-used for the new instance, improving the performance.) In particular, this would allow finding all reachable states satisfying the property, or, alternatively, finding a solution optimising the user-provided cost function.[24]

References

[Chu87] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, Lab. for Comp. Sci., MIT, 1987.

[22] Of course, one has to take care of decidability, as some reachability properties of general Petri nets are undecidable. However, this is a separate issue from the property generation.

[23] This property is violated unless suitable fairness constraints on the ME elements are added; they can also be expressed in LTL.

[24] MPSat already has a capability to compute all reachable states satisfying the property, as well as finding a shortest path to a state satisfying the property. However, currently there is no way for the user to specify a custom cost function.

[CKK$^+$02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[ERV02] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *FMSD*, 20(3):285–310, 2002.

[Esp98] J. Esparza. Decidability and complexity of Petri net problems — an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer-Verlag, 1998.

[Hel99] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fund. Inf.*, 37(3):247–268, 1999.

[Hol04] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[Kho03] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, Newcastle University, 2003.

[Kin07] D.J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley & Sons Ltd., 2007.

[KKY04] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. *Fund. Inf.*, 62(2):1–21, 2004.

[Mar90] A.J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proc. 6th MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[MB59] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proc. Int. Symp. of the Theory of Switching*, pages 204–243, 1959.

[MKY09] A. Mokhov, V. Khomenko, and A. Yakovlev. Flat arbiters. In *Proc. ACSD'09*. IEEE Comp. Soc. Press, 2009. submitted paper.

[MR97] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 352–363. Springer-Verlag, 1997.

[Rei85] W. Reisig. *Petri nets: an introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer-Verlag, 1985.

[RY85] L. Rosenblum and A. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proc. Int. Workshop on Timed Petri Nets*, pages 199–206. IEEE Comp. Soc. Press, 1985.

[SBG$^+$01] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov, and A. Smirnov. Towards synthesis of monotonic asynchronous circuits from signal transition graphs. In *Proc. ACSD'01*, pages 179–188. IEEE Comp. Soc. Press, 2001.

[Var90] V.I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, 1990. Translated from Russian, published by Nauka, Moscow, 1986.

[VW02] W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *LNCS*, pages 152–190. Springer-Verlag, 2002.