A Usable Reachability Analyser

Victor Khomenko*

***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. Its usefulness is demonstrated on several case studies.**

***Index Terms*—Reachability analysis, model checking, Petri nets, STG.**

I. Introduction

A *reachability property* of a system can be formulated as follows:

Check if there is a reachable state s
satisfying a given 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 Sect. V. Hence R is a Boolean formula built upon the elementary predicates $p_1, \ldots, p_n$ that correspond to the places of the Petri net (a predicate p_i is true iff there is a token in the i^{th} place of the Petri net at the state to be reached).

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 elements shown in dashed lines should not be considered for the moment). The specification of the deadlock condition for it is as follows:

$$\begin{aligned} R \overset{\text{df}}{=} \overline{p_1} \wedge (\overline{p_2} \vee \overline{p_7}) \wedge (\overline{p_3} \vee \overline{p_8}) \wedge (\overline{p_4} \vee \overline{p_5}) \wedge \overline{p_6} \wedge \\ \overline{p_9} \wedge (\overline{p_7} \vee \overline{p_{10}}) \wedge (\overline{p_8} \vee \overline{p_{11}}) \wedge (\overline{p_{12}} \vee \overline{p_{13}}) \wedge \overline{p_{14}} \end{aligned} \tag{1}$$

Intuitively, this formula contains one clause per transition, which is true iff this transition cannot fire because some place in its preset contains no token. 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.

*V. Khomenko is a Royal Academy of Engineering/EPSRC Post-Doctoral Research Fellow. He is affiliated with School of Computing Science, Newcastle University, UK. E-mail: `Victor.Khomenko@ncl.ac.uk`.

This research was supported by the Royal Academy of Engineering/EPSRC post-doctoral research fellowship EP/C53400X/1 (DAVAC).

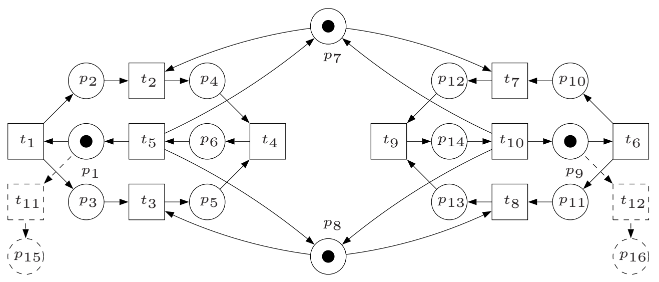


Fig. 1. A Petri net modelling two dining philosophers. Here p_1 and p_9 are the 'thinking' states of the philosophers, p_7 and p_8 are the forks, t_2, t_3, t_7 and t_8 model taking the forks, t_4 and t_9 model eating, and t_5 and t_{10} model returning the forks. The elements shown in dashed lines model a proper termination of the system.

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 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 (i.e. they are full up and leave the table). 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 $(\overline{p_{15}} \vee \overline{p_{16}})$ 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 require a considerable implementation effort, in particular they need data structures and methods for representing and accessing Petri nets, a parser for reading 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.

The full version of this paper can be found in the technical report [1].

[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 cannot always be applied, and in general it is a bad idea to force the user to modify the model or invent tricks.

II. Property specification language

In this section we outline the main idea of how to cope with the problem described in the previous section. Namely, we design a language REACH for specifying reachability properties.[2] 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;
- almost any general-purpose reachability analyser (e.g. MCSMODELS [3] or PROD [4]) 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 (1) is generated for the net in Fig. 1. Note that the `forall` and `exists` operators are expanded 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 Sect. III.

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

[2]The idea is somewhat similar to the one behind the *Property Specification Language* (PSL) [2], which allows the user to augment HDL code with assertions that can be checked using simulation or formal verification. However, PSL is not well suited for non-textual specifications like Petri nets.

[3]Note that `@t` is '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 Sect. III.

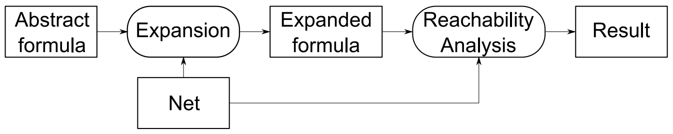


Fig. 2. Reachability analysis flow.

to a reachability analyser. The developed tool MPSAT 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.

III. Case studies

In this section we present a number of case studies. They come mostly from the domain of asynchronous circuits (ACs),[5] 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 approach of Muller [5] 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 gate delays (the wires are assumed to have negligible delays).

Signal Transition Graphs (STGs) [6] are a Petri net based formalism which is widely used for specifying asynchronous circuits. STGs 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.

An STG can be represented graphically as a labelled Petri net. However, a short-hand notation is often used, in which transitions are simply represented by their labels, and places with one incoming and one outgoing arc are contracted (if such a place contained a token, it is drawn directly on the resulting arc). Moreover, single grey lines without arrowheads will be used 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.

[5]ACs are one of the author's primary research fields. In fact, the experience gathered when implementing tools for analysis and synthesis of ACs led to the development of REACH.

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 circuit itself and are observable by the environment. Internal signals represent some auxiliary entities needed to produce outputs; like outputs, they are controlled by the circuit, but are not observable by the environment.

We now explain how to check the basic properties required for an STG to be directly implementable as an SI circuit. Some further case studies can be found in the technical report [1].

A. Consistency

One of the basic well-formedness properties of STGs is *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. The following REACH specification can be used for checking this property:

```
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 `let` defines a new name `Ts` to share a common subexpression `tran s` (note that `Ts` occurs twice in the body of the `let` operator), the operator `tran` computes the set of all transitions labelled by a signal, the `s.t.` ('such that') clause in an `exist` operator allows one to restrict its index's domain,[6] the predicates `is_plus` and `is_minus` check that the transition is labelled by a rising and falling, respectively, signal edge, and the operator `$` applied to a signal refers to its status, i.e. whether the signal is high or not.[7] Note that if the STG is not consistent and the value of a signal s is not binary at some reachable state, the `$` operator will simply return that value modulo 2, see [1] for more detail. However, since all the signal values are binary at the initial state, one can show that if consistency is violated 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.

B. 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 signals. 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' this signal, depending on whether the voltage has crossed the threshold between 0 and 1 or not, i.e. the behaviour of the circuit becomes non-deterministic and non-digital.

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 circuit, SI circuits can be synthesised for such STGs (provided that all the other conditions necessary for SI are met).

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). 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 `{...}` (in `{t1}`) is a set constructor.

Intuitively, we are looking for a state enabling some transition t_1 labelled by a local signal (lines 1–3), which can be disabled by some transition t_2 labelled by a different signal (lines 4–7). 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 ${}^\bullet t_1$. The remaining lines 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.

C. Complete State Coding (CSC)

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. An STG violating the CSC property cannot be directly implemented as an SI circuit: 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, this introduces additional memory into the circuit, helping it to trace the current state).

[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.

[7]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 }`.

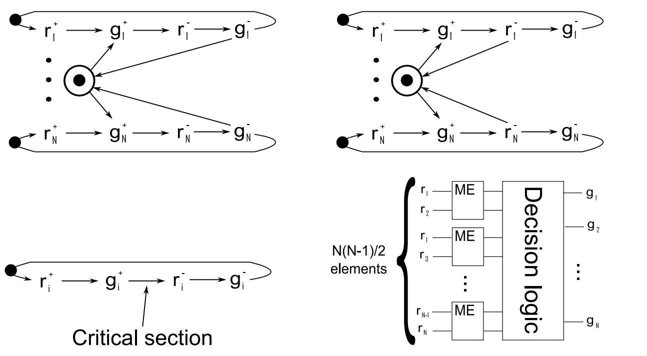


Fig. 3. A specification of an N-way arbiter: the traditional (top-left) and early (top-right) protocols, together with a model of a client (bottom-left); and the top-level view of the flat arbiter (bottom-right).

Note that in fact the CSC property is not a reachability property of an STG as defined at 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 an STG side by side, and reformulate the CSC property of the original STG as a reachability property of this joint STG. However, the approach implemented in our tool 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 a given 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 }
```

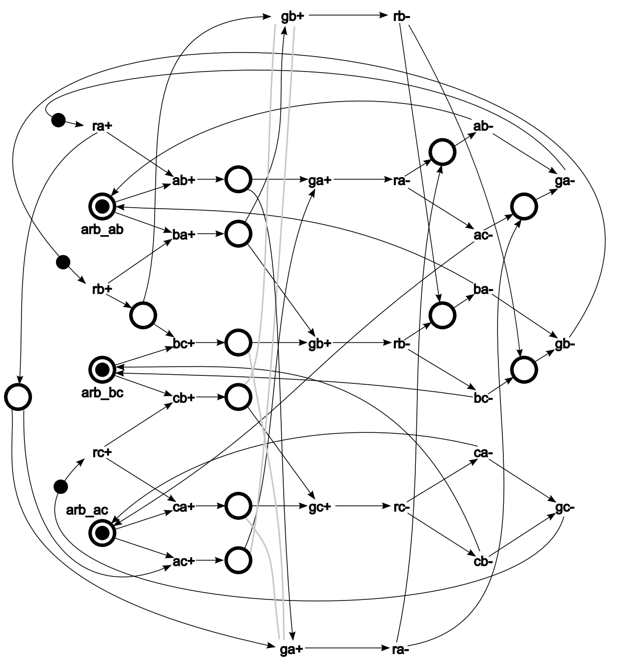
Here the operators `<->` and `^` denote Boolean equivalence[8] and 'exclusive or', respectively, and operators `$$` and `@@` are analogous to `$` and `@`, but refer to the second state.[9]

IV. Model checking of flat arbiters

Arbiters [7] 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. 3(top-left), and Fig. 3(bottom-left) shows an STG modelling the behaviour of each client. An alternative *early* protocol is shown in Fig. 3(top-right); 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.* When the two requests arrive almost simultaneously, an ME element can be slow due to the need to resolve the arising *metastability*.

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 increase the latency of the arbiter, especially in balanced circuits where the requests usually arrive almost simultaneously, and so several sequential ME elements, one after another, can spend long time in their metastable states.[10]



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

Fig. 4. An STG for the decision logic of an early 3-way flat arbiter.

In [9] 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. 3(bottom-right). 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 [9] a 3-way flat arbiter implementing the early protocol was manually designed as an STG shown in Fig. 4.

A. Deadlock checking of a flat arbiter

Correct STGs modelling flat arbiters must obviously be deadlock free. Unfortunately, the standard deadlock checking does not quite work, as the definition of a deadlock in an arbiter is slightly different.[11] 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 requests ($ra+$, $rb+$ and $rc+$ in Fig. 4) then it is classified as a deadlock. Another way of putting it is that in the STG the rising request transitions are not *weakly fair,* i.e. they may remain enabled forever, without firing. A REACH specification of this property is as follows:

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

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

[10]Some sophisticated tree arbiter designs address this problem by early propagation of requests, see [8].

[11]Like in the example with 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.

```
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,[12] the subexpression starting with the `forall` keyword 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:

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

Here the `TT` operator[13] computes the set of all transitions whose name matches the regular expression given by the following string.[14] 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.[15] 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.

B. Mutual exclusion checking of a flat arbiter

Another important property of flat arbiters is the mutual exclusion of the grants. This differs from the standard mutual exclusion of places, since the property is formulated for signals rather than places. Another difference is that for arbiters implementing the early protocol (which is the case for the STG in Fig. 4) there are reachable states where several grants are high (which is correct according to Fig. 3(top-right)). Hence, 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 straightforward to check with standard tools. However, it is easy to capture in REACH:

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

Here, the `threshold` operator evaluates to 1 iff the number of its 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 (with the iterative form of the `threshold` operator):

```
  threshold[2] r in SS "r[a-z]\\+" {
    $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 (the numbering of characters in a string starts from 0). 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 be obtained from the name of the corresponding request signal by dropping the head character and then pre-pending the result with 'g'.

V. 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 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 (i.e. unbounded) 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.[16] Generally, most formalisms that have an explicit notion of state can be adopted.

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.

REFERENCES

[1] V. Khomenko, "A usable reachability analyser," School of Comp. Sci., Newcastle Univ., Tech. Rep. CS-TR-1140, 2009.

[2] C. Schlenoff, M. Gruninger, F. Tissot *et al.*, "The Process Specification Language PSL Overview and Version 1.0 Specification," 1999.

[3] "MCSMODELS tool home page." [Online]. Available: http://www.tcs.hut.fi/~kepa/tools/mcsmodels

[4] "PROD tool home page." [Online]. Available: http://www.tcs.hut.fi/Software/prod

[5] D. Muller and W. Bartky, "A theory of asynchronous circuits," in *Proc. Int. Symp. of the Theory of Switching*, 1959, pp. 204–243.

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

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

[8] M. Josephs and J. Yantchev, "CMOS design of the tree arbiter element," *IEEE Trans. on VLSI*, vol. 4, no. 4, pp. 472–476, 1996.

[9] A. Mokhov, V. Khomenko, and A. Yakovlev, "Flat arbiters," in *Proc. ACSD'09*. IEEE Comp. Soc. Press, 2009, pp. 99–108.

[12]Recall that the `T` operator interpretes its operand as a transition name.

[13]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.

[14]Currently the basic POSIX regular expressions are supported. Note that POSIX requires all the regular operators and brackets for grouping to be escaped with '\'; moreover, since the usual C escape sequences are applied when parsing the string, '\\' are used.

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

[16]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.