A Liveness Checking Algorithm that Counts

Koen Claessen
Chalmers University of Technology
koen@chalmers.se

Niklas Sörensson†
Mentor Graphics Corporation
niklas_sorensson@mentor.com

***Abstract*—We present a simple but novel algorithm for checking liveness properties of finite-state systems, called k-LIVENESS, which is based on counting and bounding the number of times a fairness constraint can become true. Our implementation of the algorithm is completely SAT-based, works fairly well in practice, and is competitive in performance with alternative methods. In addition, we present a pre-processing technique which can automatically derive extra fairness constraints for any given liveness problem. These constraints can be used to potentially boost the performace of any liveness algorithm. The experimental results show that the extra constraints are particularly beneficial in combination with our k-LIVENESS algorithm.**

I. INTRODUCTION

LTL properties for model checking are traditionally partitioned into two sets: *safety* and *liveness* properties. Roughly, safety properties are properties for which all possible counter examples are finite traces. Liveness properties can have infinite counter examples that are impossible to make finite.

Safety properties are more commonly used in practice, easier to understand, and theoretically easier to check than liveness properties. However, liveness properties still play an important role on many verification projects. The SAT Revolution in model checking at the end of the 1990s [8] gave us new ways of battling the "blow-up" problems associated with typical BDD model checking algorithms, and sparked off a long sequence of new SAT-based safety checkers [11], [4], [10], [5]. On the liveness side, no such explosion of new algorithms took place. (Interesting to mention in this context is that the original paper on Bounded Model Checking [2] treats safety properties and liveness properties equally.)

Related Work The first practical complete SAT-based liveness checking approach was made possible by the *liveness to safety* (LTS) translation by Biere et al. [1]. LTS translates any liveness checking problem into a safety checking problem, after which any safety checker can be used, including SAT-based checkers. The second complete method for liveness, by Bradley et al., called FAIR [6], was only published last year. It consists of a dedicated liveness algorithm, based on a symbolic exploration of the state space using a SAT-solver, looking for strongly connected components. FAIR performed rather well in the Hardware Model Checking Competition 2011 [3], proving more liveness properties than any other participant in the competition.

This paper presents a new model checking algorithm for full LTL, called k-LIVENESS, that is amenable for a SAT-based implementation. Like LTS, k-LIVENESS translates liveness checking into safety checking, but it generates an (infinite) sequence of safety problems rather than just one safety problem. If one of the safety problems in the sequence can be shown to hold, then the original liveness property holds as well. In principle, any safety checker can be used to solve the problems in the sequence of problems, but we implemented and use a SAT-based incremental safety checker especially for this purpose.

As we shall see, k-LIVENESS is a surprisingly simple algorithm, much simpler than FAIR, and arguably also simpler than LTS, but it performs nonetheless quite well when compared to these other algorithms. A drawback of our chosen approach is that, although it is complete for proving as well as disproving LTL properties, it is not suitable in practice yet for finding counter examples. Therefore, there is a need to combine it with a dedicated counter example finder, for example one based on Bounded Model Checking [2].

The second contribution of the paper is a preprocessing step that automatically adds extra fairness constraints to a given liveness problem. The addition of these constraints is sound; the validity of properties of the original circuit is not changed by these extra constraints. The potential benefit is that, depending on the liveness checking algorithm used, these extra fairness constraints make many liveness problems much easier. Our experimental results show that k-LIVENESS in particular, but also LTS benefits very much from these extra constraints.

II. PRELIMINARIES

A trace t is a function from time point and signal name to Boolean value:

$$t : \mathbf{N} \times \text{Signal} \to \mathbf{B}$$

We will use LTL formulas containing operators $\Box, \Diamond, \mathsf{next}, \vee, \wedge, \rightarrow, \neg, =$ having their standard meaning. We write $S \vdash \phi$ when the system S satisfies the property ϕ, and we write $S, \psi \vdash \phi$ when the system S makes ϕ true under the assumption ψ.

When performing model checking, we assume that the LTL property ϕ at hand has already been translated into a *liveness signal* q, such that

$$S \vdash \phi \quad \Leftrightarrow \quad S \vdash \Diamond\Box q \tag{1}$$

In other words, in order to deal with full LTL, we only need to consider proving LTL properties of the form $\Diamond\Box q$. (This is

†The bulk of this work was carried out at Chalmers University.

978-0-9835678-2-0/12/$31.00 ©2012 FMCAD Inc.

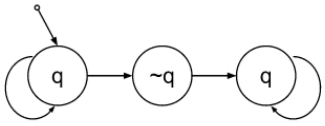

Fig. 1. Counter-example showing that the simple algorithm does not work.

standard [12]; here, q is the negation of the acceptance signal of the Buchi automaton for $\neg\phi$. The above right-hand side is sometimes expressed using the *fairness signal* $\neg q$ as follows:

$$S, \Box\Diamond\neg q \vdash \mathbf{false}$$

but we prefer the expression used in (1) above.)

In some contexts, a property ϕ may be translated into several liveness signals, in which case we need to prove their disjunction:

$$S \vdash \phi \quad \Leftrightarrow \quad S \vdash \bigvee_i \Diamond\Box q_i \tag{2}$$

In this case, we first combine all liveness signals q_i into one liveness signal q, and then proceed with the liveness signal q. This combination can be done in many (standard) ways; the simplest one is to introduce one auxiliary register for each liveness signal q_i that keeps track of whether that signal has been 0 yet. If all q_i have been 0 at least once, q also becomes 0 and we reset all auxiliary registers. This construction introduces n extra registers and $O(n)$ extra gates for combining n liveness signals.

III. K-LIVENESS

In this section, we present our basic algorithm for checking liveness, called k-LIVENESS.

A. A simple algorithm for $\Diamond\Box q$ (that does not work)

Consider proving the eventuality property

$$S \vdash \Diamond q$$

for a boolean signal q. For finite state systems, we may prove this by searching for a natural number k such that

$$S \vdash \bigvee_{i\in 0\ldots k} \mathsf{next}^i\ q$$

Indeed, if $\Diamond q$ holds, we can always find such a k. The gain is that, for a given k, the above proof obligation is a safety property, which can be checked by a safety checker. A simple checking algorithm thus tries $k = 0, 1, 2\ldots$ until $\bigvee_{i\in 0\ldots k} \mathsf{next}^i\ q$ holds.

One might be tempted to try a similar idea for checking the more general safety property

$$S \vdash \Diamond\Box q$$

Alas, there are finite state systems for which the above holds, but for which there is no k such that

$$S \vdash \mathsf{next}^k\ \Box q$$

An example of such a system and property is shown in Fig. 1. Clearly, $\Diamond\Box q$ holds for all traces accepted by the system, but for every k there is a trace such that q becomes false after k steps. So, this method is sound but not complete.

B. A simple, correct algorithm for checking $\Diamond\Box q$

Instead of counting (and bounding) the number of clock cycles until the signal q must become true forever, we can instead count (and bound) the number of times k the signal q can be false. If we can find such a bound k, then q has to eventually become true forever. Moreover, if q is a valid liveness signal for a finite state system, we can always find such a k. In the system in Fig. 1, the bound is 1; q can only become false once in every trace.

What we want is expressed more formally by the following lemma.

Lemma 1: Given a finite-state system S for which we have $S \vdash \Diamond\Box q$. Then, there exists a k such that for any trace t of S, there are at most k different points in time i for which $t(i, q) = 0$.

Proof. Assume the opposite: for any k there is a trace t where q becomes false at least k times. Now, pick any k larger than the number of states in S; there must be a trace t in which q becomes false at least k times. Consequently, there must be two different time points i and j for which $t(i) = t(j)$ and $t(i, q) = t(j, q) = 0$, since not all states where q is false can be unique. We can now construct a looping trace t' (obtained from t by repeating the states between i and j) for which q is false infinitely often, which contradicts our original assumption that $S \vdash \Diamond\Box q$. □

The experimental observation we make is that, in practice, k is often very small (see Fig. 9 in Sect. V), which suggests that finding k might be a practical method for checking liveness properties.

Our algorithm works as follows. We start by setting $k := 0$. Now, we try to show that q can only become false at most k times (this is a safety property). If we succeed, we are done; the property holds. If we fail, we increase k by 1 and try again.

Because of the above lemma, this algorithm is complete for valid properties. However, it does not terminate for properties that are not valid. Theoretically, if we keep finding counter examples for growing k, at some point there must be a trace which contains a repeated state at the appropriate place, thus forming a valid counter example for the original liveness signal. However, this is unlikely to work well in practice. In order to get a complete algorithm also for false properties in practice, a dedicated counter example finding method is used in parallel or in lock-step with k-LIVENESS.

C. Implementation

In our implementation, instead of repeatedly calling a safety checker every time we change k, we use an incremental safety checker. The incremental safety checker proves or disproves a given safety property, after which we can add some more logic and registers to the circuit, and continue with a new safety property. The incremental checker keeps its internal state

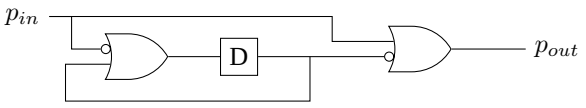

Fig. 2. Absorbing one 0 from a liveness signal (initial state for D is 0)

between calls, so it can reuse information about reachabilty it discovered in earlier runs in later runs too.

Making an existing safety checker incremental is more or less difficult, depending on the underlying algorithm that the checker is based on. The safety checker we used was our own implementation of Bradley's SAT-based safety checking algorithm as implemented in IC3 [5]. Bradley's safety algorithm maintains a finite sequence of sets of clauses, each set belonging to a concrete point in time. It turns out that the algorithmic invariants between these sets are completely independent of the property one is currently proving. So, if the current property has been proven or disproven, we can keep all sets of clauses for the next run of the checker, even though we might add some new logic and change to a new property. The algorithm also maintains a "current depth" counter which does not need to be reset when a new property is checked. Thus, Bradley's safety algorithm is a very nice fit for the liveness checker we are building.

The liveness algorithm starts with $k = 0$, and so the safety property p we have to consider is actually the liveness signal q. So, we start by trying to prove that q can never become false. If this is disproved, we want to increase k and run again. We implement increasing k by 1 by attaching the *absorbing* circuitry shown in Fig. 2 to the safety property p we have just disproved. If p is fed as its input p_{in}, a new safety signal p_{out} is created that behaves just like the previous signal p_{in}, except that it absorbs the first 0 that is produced by its input and turns it into a 1. So, adding the absorbing circuit in the figure and checking its output as the new safety property has the same effect as increasing k by 1. If we disprove the new safety property p_{out}, we attach yet another copy of the circuit to it, and so on, until we have attached enough copies of the absorbing circuit to smother all possible 0's (or we go on forever).

Making Bradley's algorithm incremental amounted to adding only about 30 lines of C++ code to our original implementation[1]. The effect of using an incremental checker is evaluated in Sect. V.

What is presented in this section leaves us with a basic liveness checking algorithm that performs reasonably well (see Sect. V for more details), but there are some bottlenecks, especially when k needs to be large. The next section presents a pre-processing step that greatly boosts the performance of the basic algorithm, and has the potential of improving other liveness checking algorithms as well.

[1]In this way, we ended up with a model checker that can check multiple safety and liveness properties simultaneously, something we have not seen before. We have however not experimentally evaluated the advantages and disadvantages of actually using the model checker as such.

IV. Automatic constraint extraction

Suppose we are checking the following proof obligation:

$$S \vdash \phi \tag{3}$$

Here, ϕ can be a safety property as well as a liveness property. Automatic constraint extraction may construct a new formula ψ which may be used as a constraint (an assumption), thus:

$$S, \psi \vdash \phi \tag{4}$$

The constraint extraction is correct if and only if the proof obligations 3 and 4 are equivalent. The hope is that a model checker may benefit from making use of the constraints ψ.

For safety checking, this idea has been proposed before, e.g. in [7]. As far as we know, we are the first to explore this in the context of liveness checking.

The ideas described here are very much inspired by the algorithm that finds so-called *arenas* in [6]. The idea behind arenas is to divide the state space of the system up into partitions, such that any trace of the system will eventually end up and stay in one such partition only. Arenas are an intricate part of the liveness checking algorithm in [6]; we decoupled the idea from the algorithm, generalized and improved the idea somewhat, and repackaged it as a pre-processing technique for liveness algorithms in general.

A. Stabilizing constraints

The kind of constraints we are going to extract are of the form

$$\Diamond\Box s$$

such that

$$S \vdash \Diamond\Box q \quad \text{iff.} \quad S, \Diamond\Box s \vdash \Diamond\Box q$$

We call constraints of the form $\Diamond\Box s$ a *stabilizing constraint*.

As observed in [6], it turns out that many liveness problems, for example liveness problems involving counters, admit such stabilizing constraints. The first observation we make is directly inspired by [6]: If we find a signal x that is *monotonic*, in other words such that:

$$S \vdash \Box(x \rightarrow \mathsf{next}\ x)$$

then it is safe to add $\Diamond\Box(x = \mathsf{next}\ x)$ as a stabilizing constraint. The reason is that for any trace t of S, x must either be 0 all the time, or become 1 at some point and then stay 1 forever. In both cases, we have that eventually, x will keep its value.

Next, we generalize this observation for signals x that are *eventually monotonic*.

Lemma 2: Given a system S and a signal x. If we have

$$S \vdash \Diamond\Box(x \rightarrow \mathsf{next}\ x)$$

then we may use $\Diamond\Box(x = \mathsf{next}\ x)$ as a (stabilizing) constraint.

B. Making use of the property

We can also make use of the original liveness signal q we want to prove. Assume that we are solving the liveness problem:

$$S \vdash \Diamond\Box q$$

and that we have found a stabilizing constraint $\Diamond\Box(x \rightarrow \mathsf{next}\ x)$. However, suppose we also find out that:

$$S \vdash \Diamond\Box(x \rightarrow q) \tag{5}$$

then it is safe to assume, not only that x will eventually stabilize to some value (0 or 1), but also the much stronger, that x will stabilize to 0! In other words, we can add $\Diamond\Box\neg x$ as a stabilizing constraint and check:

$$S, \Diamond\Box\neg x \vdash \Diamond\Box q$$

instead. Why? Because there are two cases: either x stabilizes to 0 or to 1. If x stabilizes to 1, we already know (because of (5)) that q stabilizes to 1 also, and we have shown the original property. The only interesting case left is when x stabilizes to 0, which is the one we add.

Similarly, if we find out instead of (5) that

$$S \vdash \Diamond\Box(\neg x \rightarrow q)$$

we can add $\Diamond\Box x$ as a stabilizing constraint.

C. Multiple stabilizing constraints

Stabilizing constraints may be used to help find other stabilizing constraints. So, if we have found the stabilizing constraint $\Diamond\Box(x = \mathsf{next}\ x)$ by showing:

$$S \vdash \Diamond\Box(x \rightarrow \mathsf{next}\ x)$$

then we may use it when considering another candidate y:

$$S, \Diamond\Box(x = \mathsf{next}\ x) \vdash \Diamond\Box(y \rightarrow \mathsf{next}\ y)$$

In general, we may have found a *set* of stabilizing constraints that we can use to derive new stabilizing constraints, which in turn can give rise to even more stabilizing constraints.

D. Approximating stability checking

In general, when looking for stabilizing constraints as described above, we are asking questions of the following shape:

$$S, \bigwedge_i \Diamond\Box a_i \vdash \Diamond\Box b$$

Here, the a_i are the stabilizing constraints we have already found, and b is a proof obligation that may give rise to a new stabilizing constraint. In order to *decide* questions like the above, we would need a liveness checker, which would defeat the purpose of using this as a pre-processing step to a liveness checker!

Instead, we *approximate* the answer to this question by using a SAT-solver which only talks about two consecutive states of S. We assume we are at a state in the trace where all stability constraints a_i have already become true, and then ask if the desired stability constraint b is now also true. Two states is enough since every a_i and also b contains at most one next operator. We add the assumptions a_i to the first state, and then ask the SAT-solver if b also holds. If the answer from the SAT solver is yes, we know the constraint holds. It is a crude approximation, but very fast and quite effective in practice.

E. Algorithm

In the overall constraint extraction algorithm, we work with a set of circuit points P and a set of found stabilizing constraints M. Initially, the set of found stabilizing constraints M is empty, and the set P consists of all internal points in the circuit (and their negations).

The derivation algorithm works as follows.

For all points x from the set P, we try to prove (using our overapproximation):

$$S, M \vdash \Diamond\Box(x \rightarrow \mathsf{next}\ x)$$

If this succeeds, we add the stabilizing constraint $\Diamond\Box(x = \mathsf{next}\ x)$ to M, and remove x from P. We then also try to prove (also using the overapproximation):

$$S, M \vdash \Diamond\Box(x \rightarrow q)$$

If this succeeds, we add $\Diamond\Box\neg x$ to M. If not, we try:

$$S, M \vdash \Diamond\Box(\neg x \rightarrow q)$$

If this succeeds, we add $\Diamond\Box x$ to M.

If we discover any new stability constraints, we go through all points x in P again in a new round. If no new stability constraints have been found, we terminate with M.

Note that we actually have a choice of what set of points P we start with. In our experimental results (see Sect. V), we have compared starting with all internal points (and their negations) of the circuit, as well as just having the registers (and their negations), which may be cheaper[2].

F. Making use of stabilizing constraints

Once we find the set M, we can add each of these as constraints, if the model checker can handle such constraints. However, our model checker only handles a single liveness signal q, so here we describe how we can deal with this.

The first step is to turn constraints in M of the form $\Diamond\Box(x = \mathsf{next}\ x)$ into a stabilizing constraint with just a Boolean signal c: $\Diamond\Box c$. This is cheap and easy if x is the output of a register (or its negation), because then points representing x and $\mathsf{next}\ x$ already exist, and we just create one extra XOR-gate. But if x is an internal point, we may have to introduce extra logic or perhaps even a register to represent c. We might not be willing to pay this price, in which case we can just throw away the constraint $\Diamond\Box(x = \mathsf{next}\ x)$.

So, here we have another choice of parameter to the algorithm: Do we keep constraints of the form $\Diamond\Box(x = \mathsf{next}\ x)$ even if x is not a register? We have also compared this choice in our experimental results. Note that it may actually

[2]Bradley et al. restrict themselves to registers in their arena discovery method [6].

be beneficial to start with P being all internal points, even if in the end we keep only the constraints on registers. This is because finding constraints on internal points may help in finding more constraints on registers.

Once all constraints in M are of the form $\Diamond\Box c_i$ for Boolean signals c_i, we can turn these into one big constraint $\Diamond\Box(\bigwedge_i c_i)$. This is because $\Diamond\Box$ distributes over $\wedge$. So, we are now checking:

$$S, \Diamond\Box(\bigwedge_i c_i) \vdash \Diamond\Box q$$

which is equivalent to

$$S, \Diamond\Box(\bigwedge_i c_i) \vdash \Diamond\Box((\bigwedge_i c_i) \rightarrow q)$$

for which it is enough to check

$$S \vdash \Diamond\Box((\bigwedge_i c_i) \rightarrow q)$$

since $\Diamond\Box(\bigwedge_i c_i)$ is a correct extracted constraint.

As we can see, $\bigwedge_i c_i \rightarrow q$ is a much weaker liveness signal than q, and therefore it can become false a lot less often, reducing (in many cases significantly) the value of k needed for k-LIVENESS.

V. EXPERIMENTAL RESULTS

In this section, we present an experimental evalation of implementations of the three algorithms discussed in this paper: k-LIVENESS, LTS, and FAIR. The implementations of k-LIVENESS and LTS were made by ourselves, and are based on the same safety checker. The implementation of FAIR we used was made by the original authors. Our implementation of LTS seems to be slightly better than the one used in [6], which explains the differences in evaluations in this paper and [6].

We have run a number of variants of these algorithms on a public set of liveness benchmarks, obtained from the Hardware Model Checking Competition [3]. From that set, we discarded a few problems that were not solvable by any algorithm; a total of 52 problems were solvable by at least one of the tested checkers.

All experiments were run on a cluster of quad-core Intel Xeon E5620 CPUs clocked at 2.4 GHz. The detailed results of most experiments reported here are presented later in the table in Fig. 9. In the table, a dash (—) represents a time-out. The table also contains the values of the k's that were needed for the various versions of k-LIVENESS, including the k's that were reached in case of a time-out.

To get a baseline, we start our evaluation by comparing the *basic* versions of each of the three algorithms, where no fairness constraint extraction is performed. To this end, we made a change to the source code of FAIR, and switched off the arena finding part of their algorithm, which roughly corresponds to our fairness constraint extraction. This arguably crippled (!) version of FAIR is called fair-snd0 in the tables.

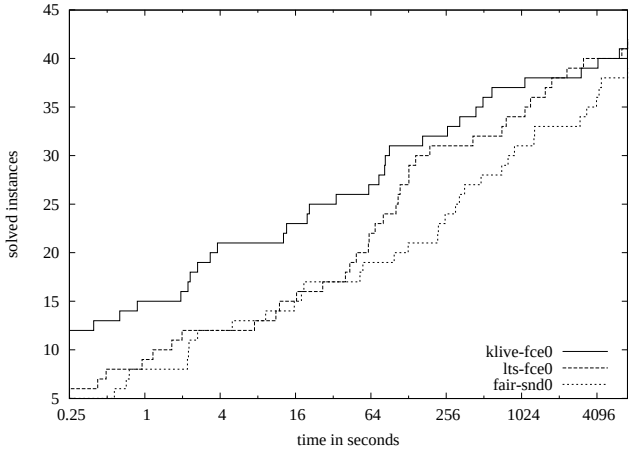

Fig. 3. Comparison of core algorithms

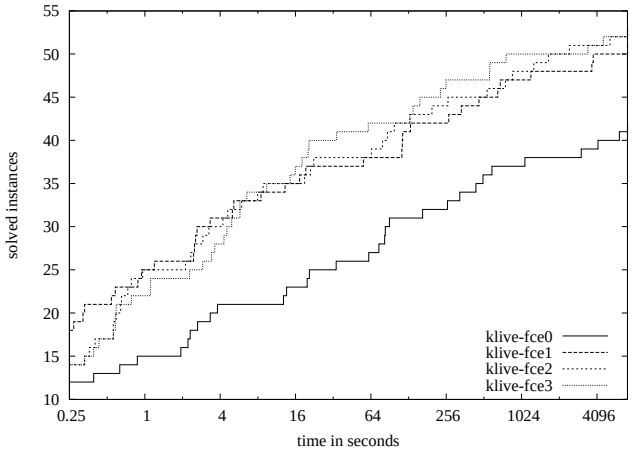

Fig. 4. Effect of stabilizing constraints on k-LIVENESS

Fig. 3 presents this comparison in the form of a cactus plot[3], where we display time outs vs. number of solved problems. We can see here that all three algorithms perform comparably (k-LIVENESS and LTS solve the same amount of problems for the maximum time out). k-LIVENESS performs slightly better than the others at time-outs around a minute or so.

The second comparison we make is about the effect of the various versions of the fairness constraint extraction on the algorithms k-LIVENESS and LTS. We chose not to include FAIR in this comparison, because the original uncrippled version of FAIR already has a similar technique built-in. The results are displayed as cactus plots in Fig. 4 and Fig. 5. Here, fce0 means no constraint extraction, fce1 means constraint extraction only for registers, fce2 means constraint extraction for all points in the circuit, but only added for registers, and fce3 means full constraint extraction and addition for all

[3]Our cactus plots might be considered slightly non-standard; the time axis is at the bottom (where time axes should be) and is logarithmic (since when comparing running times it is the factor, not the difference, which is important).

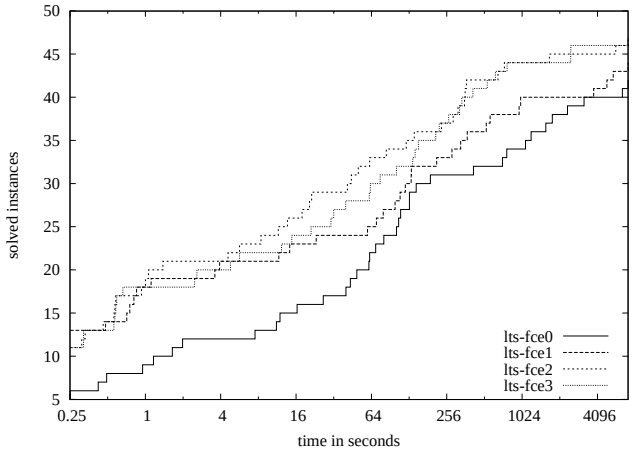

Fig. 5. Effect of stabilizing constraints on LTS

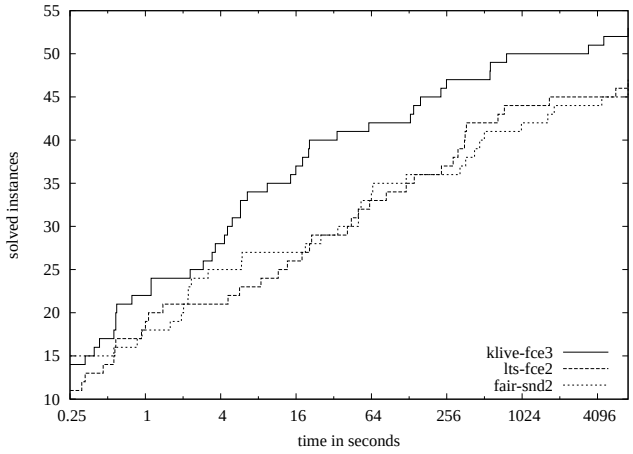

Fig. 6. Comparison of algorithms with best stabilizing constraints

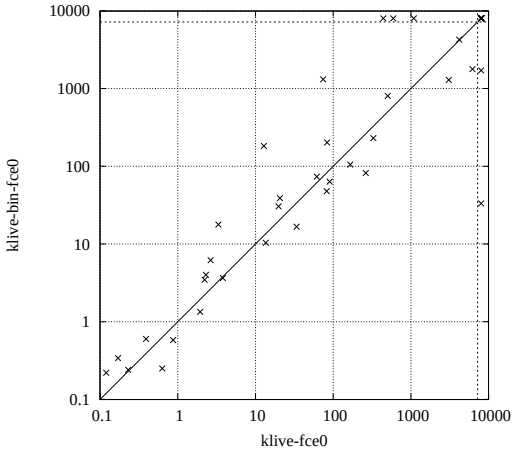

Fig. 7. klive using a binary counter

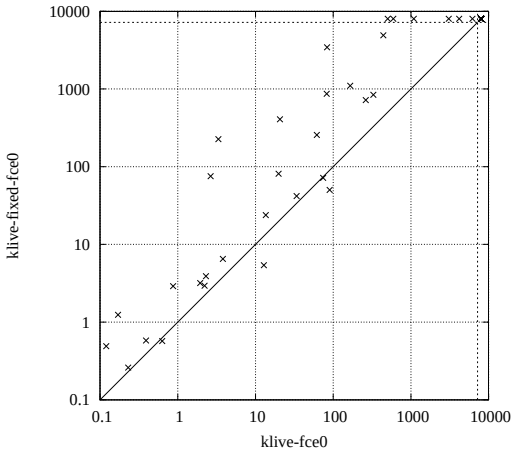

Fig. 8. Non-incremental klive with perfect guessing

circuit points. The conclusion we draw from these graphs is that fce2 and fce3 work best, but it is hard to decide which of those is best based on the set of benchmarks we have.

From Table 9 we can see that when the analysis has a big improvement on the total running time, it is mainly due to a significant reduction in k.

The times reported here is the total time of first running the analysis and then the model checking algorithm. We have not reported detailed results on the running times of the analysis alone. However, in the vast majority of the benchmarks the running time of the analysis is negligible (less than 1 second). Only in a couple of benchmarks did the analysis take a significant amount of time relative to the total time. This indicates that it is possible to find classes of circuits where the analysis as implemented today will not scale up.

The third comparison we want to make is between the best versions of the three algorithms. For k-LIVENESS and LTS, we (rather arbitrarily) chose fce3 and fce2, respectively. For FAIR, we run the original unmodified algorithm. The cactus plots for this comparison are displayed in Fig. 6. We can see that with the fairness constraint extraction on, k-LIVENESS outperforms the other two, who in turn perform remarkably similar.

Finally, we would like to experimentally answer two questions that naturally arise in connection with the k-LIVENESS algorithm. The first question is: For large k, it seems problematic to use an approach that adds circuitry linear in k. What happens when we use a binary counter instead? It turned out it was quite simple to change our algorithm to double k at every incremental step by using a binary counter. The comparison with the original, linearly growing algorithm is displayed in Fig. 7. We can see that some problems indeed are solved a bit faster, but many more problems are solved quite a bit slower. That difference is clearest for fce0, which is why we chose to show that version in the figure. The conclusion is that it might be beneficial to use a binary counter for problems that need a large k, but it is a bad idea to pick this as the default method.

The second question is: Suppose we had a way to (almost) perfectly guess the right k on beforehand. Could we base

name	klive-fce0		klive-fce2		klive-fce3		lts-fce0	lts-fce2	lts-fce3	fair-snd0	fair-snd2
	time	k	time	k	time	k	time	time	time	time	time
arbi0s08p03	442	9	1676	7	567	7	—	359	324	—	—
cuabq2f	0	3	0	2	0	2	62	355	75	5	3
cuabq2mf	0	3	0	2	0	2	2	5	5	1	0
cuabq4f	2	5	2	5	4	5	6474	5739	—	485	121
cuabq4mf	0	5	1	5	1	5	189	44	223	2	2
cuabq8f	74	9	98	9	158	9	—	—	—	1285	—
cuabq8mf	13	9	19	9	16	9	—	—	—	219	66
cucnt10	—	625	0	0	0	0	40	0	0	710	0
cucnt128	—	622	6	0	6	0	—	6	6	—	1
cucnt12	—	614	0	0	0	0	713	0	0	3393	0
cucnt32	—	533	0	0	0	0	—	0	0	—	0
cucnt3	0	0	0	0	0	0	0	0	0	0	0
cufq1	3069	9	2464	8	34	4	—	—	2506	—	—
cugbak	83	32	87	32	131	32	1207	738	2504	127	50
cugcd	3	16	1	5	2	5	1	1	12	2	2
cujc128	—	1960	21	0	20	0	—	21	21	—	2
cujc12	—	1928	0	0	0	0	61	0	0	52	0
cujc32	—	2061	1	0	1	0	—	1	1	—	0
culock	21	83	4	31	7	31	11	12	40	9	6
cunim1	593	60	0	0	0	0	43	0	0	18	0
cunim2	1088	60	3	0	6	0	418	18	265	—	1011
cuom1	—	386	1	0	1	0	—	1	1	—	—
cuom2	—	517	8	0	20	0	1785	50	32	—	64
cuom3	—	866	1	0	1	0	1090	1	1	—	2
cusarb16	0	15	0	0	0	0	0	0	0	2	0
cusarb32	3	31	0	0	0	0	8	0	0	19	0
cutarb16	503	159	131	79	139	76	146	368	209	221	25
cutarb32	—	261	5207	191	4599	188	—	—	—	—	362
cutarb4	1	23	1	11	1	8	1	1	3	1	0
cutarb8	20	63	9	31	9	28	12	14	15	16	2
cutf1	166	8	197	8	14	4	1583	315	777	898	327
cutf3	0	3	1	0	1	0	26	1	2	3	1
cutq1	4176	9	866	7	18	3	2360	—	63	4061	4433
lmcs06abp4p1	2	3	2	3	4	3	3199	20	—	326	34
lmcs06abp4p2	2	4	3	4	3	4	771	1691	626	304	511
lmcs06abp4p4	0	1	1	1	1	1	2	8	30	251	6
lmcs06bc57sp1	14	4	22	3	3	2	80	41	414	—	53
lmcs06bc57sp2	34	2	2	0	5	0	110	232	539	98	50
lmcs06bc57sp3	4	1	5	0	5	0	16	656	137	55	19
lmcs06brp0	0	1	0	0	0	0	0	0	0	—	2
lmcs06brp2	1	2	0	0	0	0	106	1	1	798	—
lmcs06counter0	0	7	0	6	0	5	0	0	0	0	0
lmcs06dme3p2	6193	2	1283	2	3467	2	—	—	—	—	—
lmcs06mutex0	0	2	0	0	0	0	0	0	0	0	0
lmcs06prodcell2	90	19	79	18	61	17	49	62	338	358	469
lmcs06prodcell3	82	80	65	25	230	25	69	288	102	4260	1849
lmcs06prodcell4	62	60	264	24	255	24	102	84	62	1300	426
lmcs06prodcell5	328	109	760	106	769	105	129	121	152	4442	—
lmcs06prodcell6	262	109	543	106	571	105	129	141	143	2999	1643
lmcs06ring0	0	4	0	4	0	4	0	0	0	0	0
lmcs06short0	0	1	0	0	0	0	0	0	0	0	0
lmcs06srg5p0	0	6	0	5	0	5	0	0	1	1	1

Fig. 9. Detailed experimental results.

an algorithm on that? We ran an experiment where we first computed the right k for each benchmark, and then ran the algorithm again, jumping to that k immediately. The results are displayed as a scatterplot in Fig. 8. We can see here that solving the problem for the correct, fixed k directly is much slower than the incremental approach. This is surprising because the incremental approach actually proves more; it also proves that none of the other k are large enough. One explanation is that the state space exploration we force the algorithm to go through when considering lesser k actually helps in finding the proof for the right k. This might also partially explain why the binary counter approach is worse than the linear approach, because it jumps over many k at once as well. However, these explanations are merely speculations and more investigation is needed to fully understand these results.

VI. Discussion and Conclusions

We have presented a new, simple liveness algorithm, called k-Liveness, based on finding a limit k on the number of times a fairness signal can become true, that compares favorably against existing liveness algorithms. Finding the right limit k is done by using an incremental safety checker. Our experiments show that the incrementality of the approach is crucial for its efficiency.

Moreover, we developed a preprocessing technique that is heavily inspired by the Fair algorithm, that can boost the performance of liveness algorithms in general. The main differences between the pre-processing presented here and the arena analysis as part of the Fair algorithm are: (1) our approach works on all points in the circuit rather than just the registers, (2) our approach makes use of the liveness signal in a different (stronger) way, (3) our approach generates

extra constraints separately from the main model checking algorithm.

We evaluated the preprocessing technique positively in particular for k-LIVENESS and LTS. Our experiments show that it is important for the preprocessor to take all internal points of the circuit into account, not only the registers. k-LIVENESS plus fairness constraint extraction performs best in our experiments. It seems that the Achilles heel of k-LIVENESS, namely when the needed k is growing too large, is nicely covered by the fairness constraint extraction, which works very well for counter-like sub-circuits.

The resulting algorithm is arguably much simpler than FAIR, even when taking the preprocessing analysis step into account. Moreover, FAIR is non-deterministic by design (and by necessity), which our algorithm is not.

A drawback of our approach is that it does not seem practical to extract counter examples. To make a model checker that is complete in practice even for false properties, the method needs to be combined with a dedicated counter example finder, for example based on Bounded Model Checking [2]. Our tool Tip, which implemented k-LIVENESS (without preprocessing) in lock-step with a simple BMC method actually won the overall liveness track of the Hardware Model Checking Competition in 2011 [3], showing that this is not a problem in practice. It is future work to investigate how to practically extract possible counter examples from failed safety checks.

For more future work, we intend to investigate the effect that alternative choices have on the efficiency of the algorithm. For example, the circuit in Fig. 2 that is added at each incremental step can be implemented in many different ways, for example by using a shift register. We do not know how changing this circuit affects the performance.

Moreover, we want to find out what effect our preprocessor has on other liveness algorithms, for example BDD-based algorithms.

A more open question is wether it is possible to make an efficient SAT-based model checker for CTL. Ideas from [9] seem promising in this regard.

ACKNOWLEDGMENTS

We would like to thank Johan Mårtensson and Nir Piterman for initial discussions and insights on this work. The anonymous referees' comments were also highly appreciated. Finally, many thanks to the Colorado team (Aaron Bradley, Fabio Somenzi, and Zyad Hassan) for helping us to run their code.

REFERENCES

[1] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *Proc. of Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of Conference on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[3] Armin Biere and Keijo Heljanko. Hardware model checking competition, 2011. Associated with FMCAD'11. http://fmv.jku.at/hwmcc11/.

[4] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[5] Aaron Bradley. SAT-based model checking without unrolling. In *Proc. of Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Lecture Notes in Computer Science. Springer Verlag, 2011.

[6] Aaron Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2011.

[7] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *Proc. of Conference on Design Automation and Test in Europe (DATE)*, 2009.

[8] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. Sat-solving in practice. In *Proc. of Workshop on Discrete Event Systems (WODES)*. IEEE, May 2008.

[9] Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In *Proc. of International Conference on Computer-Aided Verification (CAV)*, 2011.

[10] Ken McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. of Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[11] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[12] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. of Symposium on Foundations of Computer Sciece (FOCS)*, 1983.