Distributed Computation of Fixed Points on Dependency Graphs

Andreas Engelbredt Dalsgaard, Søren Enevoldsen,
Kim Guldstrand Larsen, and Jiří Srba

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.
{andrease,senevoldsen,kgl,srba}@cs.aau.dk

Abstract. Dependency graph is an abstract mathematical structure for representing complex causal dependencies among its vertices. Several equivalence and model checking questions, boolean equation systems and other problems can be reduced to fixed-point computations on dependency graphs. We develop a novel distributed algorithm for computing such fixed points, prove its correctness and provide an efficient, open-source implementation of the algorithm. The algorithm works in an on-the-fly manner, eliminating the need to generate a priori the entire dependency graph. We evaluate the applicability of our approach by a number of experiments that verify weak simulation/bisimulation equivalences between CCS processes and we compare the performance with the well-known CWB tool. Even though the fixed-point computation, being a P-complete problem, is difficult to parallelize in theory, we achieve significant speed-ups in the performance as demonstrated on a Linux cluster with several hundreds of cores.

1 Introduction

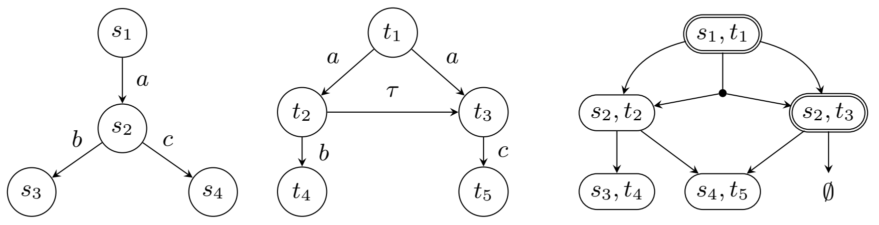
Formal verification techniques are increasingly applied in industrial development of software and hardware systems, both to ensure safe and reliable behaviour of the final system, and to reduce cost and time by finding bugs at early development stages. In particular industrial take-up has been boosted by the maturing of computer aided verification, where development of a variety of techniques helps in applying verification to critical parts of systems. Heuristics for SAT solving, abstraction, decomposition, symbolic execution, partial order reduction, and other techniques are used to speed up the verification of systems with various characteristics. Still, the problem of automatic verification is hard, and some difficult cases occur frequently in practical experience. For this reason, we aim in this paper at exploiting the computational power of parallel and distributed machine architectures to further enlarge the scope of automated verification.

Automated verification methods contain a large variety of model-checking and equivalence/preorder-checking algorithms. In the former, a system model is (dis-)proved correct with respect to a logical property expressed in a suitable temporal logic. In the latter, the system model is compared with an abstract model of the system with respect to a suitable behavioural equivalence or preorder, e.g. trace-equivalence, weak or strong bisimulation equivalence. Aiming at providing parallel and distributed support to (essentially) all of these problems, we design a distributed algorithm based on the notion of *dependency graphs* [1,2]. In particular, dependency graphs have proven a useful and universal formalism for representing several verification problems, offering efficient analysis through linear-time (local and global) algorithms [2] for fixed-point computation of the corresponding dependency graph. The challenge we undertake here is to provide a distributed algorithm for this fixed-point computation. The fact that dependency graphs allow for representation of bisimulation equivalences between system models suggests that we should not expect our distributed algorithm to exhibit linear speed-up in all cases as bisimulation equivalence is known to be P-complete [3]. Our experiments though still document significant speed-ups that together with the on-the-fly nature of our algorithm (where we possibly avoid the construction of the entire dependency graph in situations where it is not necessary) allow us to outperform the tool CWB [4] for equivalence/model checking of processes described in the CCS process algebra [5].

Related Work. Most closely related to our work are those of [6,7,8] offering parallel algorithms for model-checking systems with respect to the alternation-free modal μ-calculus. The approach in [6] is based on games and tree decomposition but the tool prototype mentioned in the paper is not available anymore. The work in [8] reduces μ-calculus formulae into alternation free Boolean equation systems. Finally [7] uses a global symbolic BDD-based distributed algorithm for modal μ-calculus but does not mention any implementation. We share the on-the-fly technique with some of these works but our framework is more universal in the sense that we deal with the general dependency graphs where the problems above are reducible to. There also exist several mature tools with modern designs like FDR3 [9], CADP [10], SPIN [11] and mCRL2 [12], some of them offering also distributed and/or on-the-fly algorithms. The input language of the tools is however often strictly defined and extensions to these languages as well as the range of verification methods require nontrivial changes in the implementation. The advantage of our approach is that we first reduce a wide range of problems into dependency graphs and then use our optimized distributed implementation on these generic graphs. Finally, we have recently introduced CAAL [13] as a tool for teaching CCS and verification techniques. The tool CAAL, running in a browser and implemented in TypeScript (a typed superset of JavaScript), is also based on dependency graphs but offers only the sequential version of the local algorithm by Liu and Smolka [2]. Here we provide an optimized C++ implementation of the distributed algorithm thus laying the foundation for offering CAAL verification tasks as a cloud service.

2 Definitions

A *labelled transition system* (LTS) is a triple $(S, A, \rightarrow)$ where S is a set of states, A is a set of actions that includes the silent action τ, and $\rightarrow \subseteq S \times A \times S$ is the



(a) Labelled transition system

(b) Corresponding DG

Fig. 1: Dependency graph for weak bisimulation

transition relation. Instead of $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$. We also write $s \overset{a}{\Rightarrow} t$ if either $a = \tau$ and $s \xrightarrow{\tau}^* t$, or if $a \neq \tau$ and $s \xrightarrow{\tau}^* s' \xrightarrow{a} t' \xrightarrow{\tau}^* t$ for some $s', t' \in S$.

A binary relation $R \subseteq S \times S$ over the set of states of an LTS is *weak simulation* if whenever $(s, t) \in R$ and $s \xrightarrow{a} s'$ for some $a \in A$ then also $t \overset{a}{\Rightarrow} t'$ such that $(s', t') \in R$. If both R and $R^{-1} = \{(t, s) \mid (s, t) \in R\}$ are weak simulations then R is a *weak bisimulation.*

We say that s is *weakly simulated* by t and write $s \ll t$ (resp. s and t are *weakly bisimilar* and write $s \approx t$) if there is a weak simulation (resp. weak bisimulation) relation R such that $(s, t) \in R$.

Consider the LTS in Figure 1a (even though it consists of two disconnected parts, it can still be considered as a single LTS). It is easy to see that s_1 weakly simulates t_1 and vice versa. For example the weak simulation relation $R = \{(s_1, t_1), (s_2, t_2), (s_3, t_4), (s_4, t_5)\}$ shows that s_1 is weakly simulated by t_1. However, s_1 and t_1 are not weakly bisimilar. Indeed, if s_1 and t_1 were weakly bisimilar, the transition $t_1 \xrightarrow{a} t_3$ can only be matched by $s_1 \xrightarrow{a} s_2$ but s_2 has a transition under the label b whereas t_3 does not offer such a transition.

2.1 Dependency Graphs

A dependency graph [2] is a general structure that expresses dependencies among the vertices of the graph and by this allows us to solve a large variety of complex computational problems by means of fixed-point computations.

Definition 1 (Dependency Graph).
A dependency graph *is a pair (V, E) where V is a set of vertices and $E \subseteq V \times 2^V$ is a set of hyperedges. For a hyperedge $(v, T) \in E$, the vertex $v \in V$ is called the* source *vertex and $T \subseteq V$ is the* target *set.*

Let $G = (V, E)$ be a fixed dependency graph. An *assignment* on G is a function $A : V \to \{0, 1\}$. Let $\mathcal{A}$ be the set of all assignments on G. A *fixed-point assignment* is an assignment A that for all $(v, T) \in E$ satisfies the following condition: if $A(v') = 1$ for all $v' \in T$ then $A(v) = 1$.

Figure 2 shows an example of a dependency graph. The hyperedge $(a, \emptyset)$ with the empty target set is depicted by the arrow from a to the symbol $\emptyset$. The figure

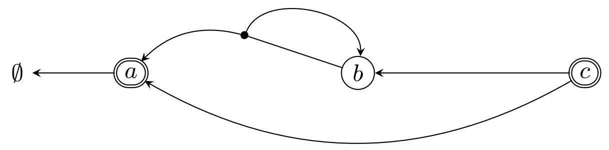


Fig. 2: Dependency graph $G = (\{a, b, c\}, \{(a, \emptyset), (b, \{a, b\}), (c, \{b\}), (c, \{a\})\})$

also denotes a particular assignment A such that vertices with a single circle have the value 0 and vertices with a double circle have the value 1, in order words $A(a) = A(c) = 1$ and $A(b) = 0$. It can be easily verified that the assignment A is a fixed-point assignment.

We are interested in the minimum fixed-point assignment. Let $A_1, A_2 \in \mathcal{A}$ be assignments. We write $A_1 \sqsubseteq A_2$ if $A_1(v) \leq A_2(v)$ for all $v \in V$, where we assume that $0 \leq 1$. Clearly $(A, \sqsubseteq)$ is a complete lattice. Let us also define a function $F : \mathcal{A} \to \mathcal{A}$ such that $F(A)(v) = 1$ if there is a hyperedge $(v, T) \in E$ such that $A(v') = 1$ for all $v' \in T$, otherwise $F(A)(v) = A(v)$. Observe that an assignment A is a fixed-point assignment iff $F(A) = A$, and that the function F is monotonic w.r.t. $\sqsubseteq$. By Knaster-Tarski theorem [14] there exists a unique minimum fixed-point assignment, denoted by A_{min}. The assignment A_{min} on a finite dependency graph can be computed by a repeated application of the function F on the assignment A_0 where $A_0(v) = 0$ for all $v \in V$, and we are guaranteed that there is a number m such that $F^m(A_0) = F^{m+1}(A_0) = A_{min}$.

Consider again our example from Figure 2 and assume that each assignment A is represented by the vector $(A(a), A(b), A(c))$. We can see that $A_0 = (0, 0, 0)$, $F(A_0) = (1, 0, 0)$ and $F^2(A_0) = (1, 0, 1) = F^3(A_0)$. Hence the depicted assignment $(1, 0, 1)$ is the minimum fixed-point assignment.

2.2 Applications of Dependency Graphs

Many verification problems can be encoded as fixed-point computations on dependency graphs. We shall demonstrate this on the cases of weak simulation and bisimulation, however other equivalences and preorders from the linear/branching-time spectrum [15] can also be encoded as dependency graphs [16] as well as model checking problems e.g. for the CTL logic [17], reachability problems for timed games [18] and the general framework of Boolean equation systems [2], just to mention a few applications of dependency graphs.

Let $T = (S, A, \to\ )$ be an LTS. We define a dependency graph $G_{\approx}(T) = (V, E)$ such that $V = \{(s, t) \mid s, t \in S\}$ and the hyperedges are given by

$$E = \{\big((s,t), \{(s',t') \mid t \overset{a}{\Rightarrow} t'\}\big) \mid s \xrightarrow{a} s'\} \cup \{\big((s,t), \{(s',t') \mid s \overset{a}{\Rightarrow} s'\}\big) \mid t \xrightarrow{a} t'\} .$$

The general construction is depicted in Figure 3 and its application to the LTS from Figure 1a, listing only the pairs of states reachable from (s_1, t_1), is shown in Figure 1b. Observe that the size of the produced dependecy graph is polynomial with respect to the size of the input LTS.

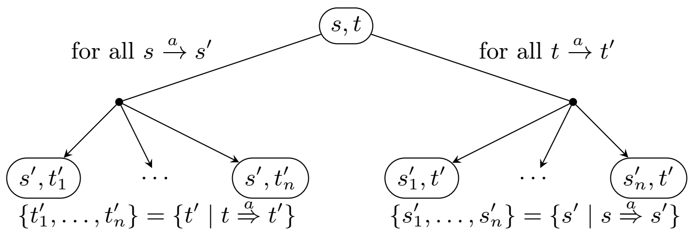

s, t
for all $s \xrightarrow{a} s'$
for all $t \xrightarrow{a} t'$
s', t'_1
$\cdots$
s', t'_n
s'_1, t'
$\cdots$
s'_n, t'
$\{t'_1, \ldots, t'_n\} = \{t' \mid t \overset{a}{\Rightarrow} t'\}$
$\{s'_1, \ldots, s'_n\} = \{s' \mid s \overset{a}{\Rightarrow} s'\}$

Fig. 3: Bisimulation reduction to dependency graph

Proposition 1. *Let $T = (S, A, \rightarrow)$ be an LTS and $s, t \in S$. We have $s \approx t$ if and only if $A_{min}((s,t)) = 0$ in the dependency graph $G_{\approx}(T)$.*

Proof (Sketch). "$\Rightarrow$": Let R be a weak bisimulation such that $(s,t) \in R$. The assignment A defined as $A((s',t')) = 0$ iff $(s',t') \in R$ can be shown to be a fixed-point assignment. Then clearly $A_{min} \sqsubseteq A$ and because $A((s,t)) = 0$ then also $A_{min}((s,t,)) = 0$. "$\Leftarrow$": Let $A_{min}((s,t)) = 0$. We construct a binary relation $R = \{(s',t') \mid A_{min}((s',t')) = 0\}$. Surely $(s,t) \in R$ and we invite the reader to verify that R is a weak bisimulation. ☐

In our example in Figure 1b we can see that $A_{min}((s_1, t_1)) = 1$ and hence $s_1 \not\approx t_1$. The construction of the dependency graph for weak bisimulation can be adapted to work also for the weak simulation preorder by removing the hyperedges that originate by transitions performed by the right hand-side process.

We know that computing A_{min} for a given dependency graph can be done in linear time [19]. By the facts that deciding bisimulation on finite LTS is P-hard [3] and the polynomial time reduction described above, we can conclude that determining the value of a vertex in the minimum fixed-point assignment of a given dependency graph is a P-complete problem.

Proposition 2. *The problem whether $A_{min}(v) = 1$ for a given dependency graph and a given vertex v is P-complete.*

3 Distributed Fixed-Point Algorithm

We shall now describe our distributed algorithm for computing minimum fixed-points on dependency graphs. Let $G = (V, E)$ be a dependency graph. For the purpose of the on-the-fly computation, we represent G by the function

$$\textsc{Successors}(v) = \{(v, T) \mid (v, T) \in E\}$$

that returns for each vertex $v \in V$ the set of outgoing hyperedges from v.

We assume a fixed number of n workers. Let i, $1 \leq i \leq n$, denote a worker with id i. Each worker i uses the following local data structures.

– A local assignment function $A^i : V \rightharpoonup \{0, 1\}$, which is a partial function mapping each vertex to the values *undefined*, 0 or 1.

– A local dependency function $D^i : V \to 2^E$ returning the current set of dependent edges for each vertex.
– A local waiting set $W^i \subseteq E$ containing edges that are waiting for processing.
– A local request function $R^i : V \to 2^{\{1,\dots,n\}}$ where the worker i remembers who requested the value for a given vertex.
– A local input message set $M^i \subseteq \{$*"value of v needed by worker j"* $\mid v \in V,\ 1 \le j \le n\} \cup \{$*"$v$ has value* 1*"* $\mid v \in V\}$. For syntactic convenience, we assume that a worker i can add a message m to M^j of another worker j simply by executing the assignment $M^j \leftarrow M^j \cup \{m\}$.

We moreover assume some standard function TerminationDetection, computed distributively, that returns true if there are no messages in transit and all waiting sets of all workers are empty, in other words if $\bigcup_{1 \le i \le n} M^i \cup W^i = \emptyset$. Finally, we assume a global partitioning function $\delta : V \to \{1, \dots, n\}$ that partitions vertices to workers.

The distributed algorithm for computing the minimal fixed-point assignment for a given vertex v_s is presented in Algorithm 1. First, all n workers are initialized and the worker that owns the vertex v_s updates its local assignment to 0 and adds the successor edges to its local waiting set. Then the workers start processing the edges on the waiting sets and the messages in their input message sets until they terminate either by one worker announcing that $A_{min}(v_s) = 1$ at line 18 or all waiting edges and messages have been processed and then the workers together claim that $A_{min}(v_s) = 0$ at line 13.

Lemma 1 (Termination). *Algorithm 1 terminates.*

Proof. First observe that for each vertex v and each local assignment A^i the value of $A^i(v)$ is first undefined. Then when v is discovered either as a target vertex of some hyperedge on the waiting set (line 23) or when the value of v gets requested by another worker (line 37), the value $A^i(v)$ changes to 0. Finally the value of $A^i(v)$ can be upgraded to the value 1 either by the presence of a hyperedge where all target vertices already have the value 1 (line 17) or by receiving a message from another worker (line 33). The point is that for every v, each of the assignments $A^i(v) \leftarrow 0$ and $A^i(v) \leftarrow 1$ is executed at most once during any execution of the algorithm. This can be easily noticed by the inspection of the conditions on the if-commands guarding these assignments.

Next we notice that new hyperedges are added to the waiting set W^i only when an assignment of some vertex v gets upgraded from *undefined* to 0, or from 0 to 1. As argued above, this can happen only finitely many times, hence only finitely many hyperedges can be added to each W^i. Similarly, new messages to the message sets can be added only at lines 19, 28 and 40. At line 19 a finite number of messages of the form *"v has value* 1*"* is added only when a value of $A^i(v)$ was upgraded to 1. This can happen only finitely many times. At line 28 the message *"value of v' needed by worker i"* is added only when a value of a vertex was upgraded from *undefined* to 0, hence this can happen only finitely many times. Finally, at line 40 a message is added only when we received the message *"value of v needed by worker i"* but this message was sent only finitely many

Algorithm 1: Distributed Algorithm for Worker i, $1 \leq i \leq n$

Input: A dependency graph $G = (V, E)$ represented by the function SUCCESSORS, a vertex $v_s \in V$ and a vertex partitioning function $\delta : V \to \{1, \ldots, n\}$ where n is the number of workers.

Output: The minimum fixed-point assignment $A_{min}(v_s)$ for the vertex v_s.

```
1  $A^i(v) \leftarrow$ *undefined* for all $v \in V$          ▷ implemented via hashing
2  $W^i \leftarrow \emptyset$; $D^i \leftarrow \emptyset$; $M^i \leftarrow \emptyset$; $R^i \leftarrow \emptyset$
3  **if** $\delta(v_s) = i$ **then**                          ▷ initialize the computation
4      $A^i(v_s) \leftarrow 0$; $W^i \leftarrow$ SUCCESSORS($v_s$)
5  **repeat**
6      **while** $W^i \neq \emptyset$ *or* $M^i \neq \emptyset$ **do**
7          Let $x \in W^i \cup M^i$                        ▷ process message or hyperedge
8          **if** $x \in W^i$ **then**
9              $W^i \leftarrow W^i \setminus \{x\}$; PROCESSHYPEREDGE($x$)
10         **else**
11             $M^i \leftarrow M^i \setminus \{x\}$; PROCESSMESSAGE($x$)
12 **until** TERMINATIONDETECTION
13 output "$A_{min}(v_s) = 0$"

14 **Procedure** PROCESSHYPEREDGE($(v, T)$) **is**
15     **if** $A^i(v) \neq 1$ **then**
16         **if** $\forall v' \in T : A^i(v') = 1$ **then**
17             $A^i(v) \leftarrow 1$
18             **if** $v = v_s$ **then** output "$A_{min}(v_s) = 1$" ; terminate all workers
19             **for** *all* $j \in R^i(v)$ **do** $M^j \leftarrow M^j \cup \{$"*$v$ has value 1*"$\}$
20             $R^i(v) \leftarrow \emptyset$
21             $W^i \leftarrow W^i \cup D^i(v)$
22         **else if** $\exists v' \in T : A^i(v')$ *is undefined* **then**
23             $A^i(v') \leftarrow 0$
24             $D^i(v') \leftarrow D^i(v') \cup \{(v, T)\}$
25             **if** $\delta(v') = i$ **then**                ▷ is value of $v'$ my responsibility?
26                 $W^i \leftarrow W^i \cup$ SUCCESSORS($v'$)
27             **else**                                     ▷ send request for value of $v'$
28                 $M^{\delta(v')} \leftarrow M^{\delta(v')} \cup \{$"*value of $v'$ needed by worker $i$*"$\}$
29         **else if** $\exists v' \in T : A^i(v') = 0$ **then**
30             $D^i(v') \leftarrow D^i(v') \cup \{(v, T)\}$

31 **Procedure** PROCESSMESSAGE($m$) **is**
32     **if** $m =$ "*$v$ has value 1*" *and* $A^i(v) \neq 1$ **then**
33         $A^i(v) \leftarrow 1$
34         $W^i \leftarrow W^i \cup D^i(v)$
35     **else if** $m =$ "*value of $v$ needed by worker $j$*" **then**
36         **if** $A^i(v)$ *is undefined* **then**
37             $A^i(v) \leftarrow 0$
38             $W^i \leftarrow W^i \cup$ SUCCESSORS($v$)
39         **if** $A^i(v) = 1$ **then**
40             $M^j \leftarrow M^j \cup \{$"$v$ has value 1"$\}$          ▷ we already know it is 1
41         **else**
42             $R^i(v) \leftarrow R^i(v) \cup \{j\}$          ▷ remember that $j$ needs value of $v$
```

times. All together, only finitely many elements can be added to the waiting and message sets and as the main while-loop repeatedly removes elements from those sets, eventually they must become empty and the algorithm terminates at line 13 (unless it terminated earlier at line 18). □

We can now observe that if a vertex is assigned the value 1 for any worker, then the value of the vertex in the minimal fixed-point assignment is also 1.

Lemma 2 (Soundness). *At any moment of the execution of Algorithm 1 and for all i, $1 \leq i \leq n$, and all $v \in V$ it holds that*

a) if $A^i(v) = 1$ then $A_{min}(v) = 1$, and
b) if "v has value 1" $\in M^i$ then $A_{min}(v) = 1$.

Proof. The invariant holds initially as $A^i(v)$ is undefined for all i and all v and all input message sets are empty.

Let us assume that both condition a) and b) hold and that we assign the value 1 to $A^i(v)$ for some worker i and a vertex v. This can only happen at lines 17 and 33. In the first assignment at line 17 we know that there is a hyperedge (v, T) such that all vertices $v' \in T$ satisfy that $A^i(v') = 1$. However, this by our invariant part a) implies that $A_{min}(v') = 1$ and then necessarily also $A_{min}(v) = 1$ by the definition of fixed-point assignment. Hence the invariant for the case a) is preserved. Similarly, if $A^i(v)$ gets the value 1 at line 33, this can only happen if *"v has value 1"* $\in M^i$ and by the invariant part b) this implies that $A_{min}(v) = 1$ and hence the invariant for the condition a) is established .

Similarly, let us assume that conditions a) and b) hold and that a message *"v has value 1"* gets inserted into M^j by some worker i. This can only happen at lines 19 and 40. In both situations it is guaranteed that $A^i(v) = 1$ and hence by the invariant part a) we know that $A_{min}(v) = 1$, implying that adding these messages to M^j is safe. □

The next lemma establishes an important invariant of the algorithm.

Lemma 3. *For any vertex $v \in V$, whenever during the execution of Algorithm 1 the worker $\delta(v)$ is at line 6 then the following invariant holds: either*

a) $A^{\delta(v)}(v) = 1$, or
b) $A^{\delta(v)}(v)$ is undefined, or
c) $A^{\delta(v)}(v) = 0$ and for all $(v, T) \in E$ either
  i) $(v, T) \in W^{\delta(v)}$, or
  ii) there is $v' \in T$ such that $A^{\delta(v)}(v') = 0$, and $(v, T) \in D^{\delta(v)}(v')$.

Proof. Initially, the invariant is satisfied as $A^{\delta(v)}(v)$ is undefined and the invariant, more specifically the subcase i), clearly holds also when $v = v_s$ and the worker $\delta(v_s)$ performed the assignments at line 4.

Assume now that the invariant holds. Clearly, if it is by case a) where $A^{\delta(v)}(v) = 1$ then the value of v will remain 1 until the end of the execution.

If the invariant holds by case b) then it is possible that the value of $A^{\delta(v)}(v)$ changes from *undefined* to 0. This can happen either at lines 23 or 37. If the assignment took place at line 23 (note that here $v = v'$) then clearly line 26 will be executed too and all successor edges of v will be inserted into the waiting set and hence the invariant subcase i) will hold once the execution of the procedure is finished. Similarly, if the assignment took place at line 37 then all successors of v are at the next line 38 immediately added to the waiting set, hence again satisfying the invariant subcase i).

Consider now the case c). The invariant can be challenged by either removing the hyperedge (v, T) from $W^{\delta(v)}$ hence invalidating the subcase i) or by upgrading the value of the vertex v' in case ii) such that $A^{\delta(v)}(v') = 1$. In the first case where the subcase i) gets invalidated we can notice that this can happen only at line 9 after which the removed hyperedge (v, T) is processed. There are two possible scenarios now. Either all vertices from T have the value 1 and then the value of $A^{\delta(v)}(v)$ also gets upgraded to 1 at line 17 hence satisfying the invariant a), or there is a vertex $v' \in T$ such that $A^{\delta(v)}(v') = 0$ and then the hyperedge (v, T) is added at line 30 to the dependency set $D^{\delta(v)}(v')$ satisfying the subcase (ii) of the invariant. In the second subcase, we assume that the vertex v' satisfying the subcase ii) gets upgraded to the value 1. This can happen at line 17 or line 33. In both cases the dependency set $D^{\delta(v)}(v')$ (that by our invariant assumption contains the hyperedge (v, T)) is added to the waiting set (lines 21 and 34) implying that the invariant subpart i) holds. □

The following lemmas shows that after the termination, the value 0 for a vertex v in a local assignment of some worker implies the same value also in the assignment of the worker that owns the vertex v. This is an important fact for showing completeness of our algorithm.

Lemma 4. *Once all workers in Algorithm 1 terminate at line 13 then for all vertices $v \in V$ and all workers i holds that if $A^i(v) = 0$ then $A^{\delta(v)}(v) = 0$.*

Proof. Observe that the assignment of 0 to $A^i(v)$ where $i \neq \delta(v)$ can happen only at line 23 (the assignment at line 37 is performed only if $i = \delta(v)$ as the message *"value of v is needed by worker i"* is sent only to the owner of the vertex v). After the assignment at line 23 done by worker i, the message requesting the value of the vertex is sent to its owner at line 28. Clearly, before the workers terminate, this message must be read by the owner and the value of the vertex is either set to 0 at line 37, or if the value is already known to be 1 the worker i is informed about this via the message *"v has value* 1*"* at line 40 and this message will be necessarily read by the worker i before the termination and the value $A^i(v)$ will be updated to 1. Otherwise we remember the worker's id requesting the assignment value at line 42. Should the owner upgrade the value of v to 1 at some moment, all workers that requested its value will be informed about this by a message sent at line 19 and before the termination these workers must read these messages and update the local values for v to 1. It is hence impossible for the algorithm to terminate while the owner of v set its value to 1 and some other worker still has only the value 0 for the vertex v. □

Lemma 5 (Completeness). *If all workers in Algorithm 1 terminate at line 13 then for all vertices $v \in V$ the fact $A^{\delta(v)}(v) = 0$ implies that $A_{min}(v) = 0$.*

Proof. Note that after the termination we have $W^i = M^i = \emptyset$ for all i. Assume now that $A^{\delta(v)}(v) = 0$. Then by Lemma 3 and the fact that $W^{\delta(v)} = \emptyset$ we can conclude that for all $(v, T) \in E$ there exists $v' \in T$ such that $A^{\delta(v)}(v') = 0$. By Lemma 4 this means that also $A^{\delta(v')}(v') = 0$. Let us now define an assignment A such that $A(v) = A^{\delta(v)}(v)$. By the arguments above, A is a fixed-point assignment. As A_{min} is the minimum fixed-point assignment, we have $A_{min} \sqsubseteq A$ and because $A(v) = 0$ we can conclude that $A_{min}(v) = 0$. $\square$

Theorem 1 (Correctness). *Algorithm 1 terminates and outputs either*

- *"$A_{min}(v_s) = 1$" implying that $A_{min}(v_s) = 1$, or*
- *"$A_{min}(v_s) = 0$" implying that $A_{min}(v_s) = 0$.*

Proof. Termination is proved in Lemma 1. The algorithm can terminate either at line 18 provided that $A^i(v_s) = 1$ but then by Lemma 2 clearly $A_{min}(v_s) = 1$. Otherwise the algorithm terminates when all workers reach line 13. This can only happen when $A^{\delta(v_s)}(v_s) = 0$ and by Lemma 5 we get $A_{min}(v_s) = 0$. $\square$

Note that the algorithm is proved correct without imposing any specific order by which messages and hyperedges are selected from the sets W^i and M^i or what target vertices are selected in the expressions like $\exists v' \in T$. In the next section we discuss some of the choices we have made in our implementation.

4 Implementation and Evaluation

The distributed algorithm described in the previous section is implemented as an MPI-program in C++, enabling the workers to cooperate not only on a single machine but also across multiple machines. The MPI-program requires a successor generator to explore the dependency graph, a partitioning function and a (de)serialisation function for the vertices (we use LZ4 compression on the generated hyperedges before they leave the successor generator). For our experiments, these functions were implemented for the case of weak bisimulation/simulation on CCS processes but they can be easily replaced with other custom implementations to support other equivalence and model checking problems, without the need of modifying the distributed engine itself.

In our implementation we use hash tables to store the assignments (A^i) and the dependent edges (D^i). The algorithm does not constrain specific structures on W^i or M^i. For the waiting list (W^i) two deques are used, one for the forward propagation (outgoing hyperedges of newly discovered vertexes) and one for the backwards propagation (hyperedges that were inserted due to dependencies). Then the graph can be explored depth-first, or breadth-first, or a probabilistic combination of those, independently for both the forward and backwards propagation. Our experiments showed that it is preferable to prioritize processing of messages rather than hyperedges to free up buffers used by the senders. The distributed termination detection is determined using [20].

The implementation is open-source and available at `http://code.launchpad.net/pardg/` in the branch `dfpdg-paper` that includes also all experimental data.

The distributed engine is currently being integrated within the CAAL [13] user interface.

4.1 Evaluation

We evaluate the performance of our implementation on the traditional leader election protocol [21] where we scale the number of processes and on the alternating bit protocol (ABP) [22] where we scale the size of communication buffers. We ask the question whether the specification and implementation (both described as CCS processes) are weakly bisimilar. For both cases we consider a variant where the weak bisimulation holds and where it does not hold (by injecting an error). Finally, we also ask about the schedulability of 180 different task graphs from the well known benchmark database [23] on two processors within a given deadline. Whenever applicable, the performance is compared with the tool Concurrency WorkBench (CWB) [4] version 7.1 using 1 core (there is no parallel/distributed version of CWB). CWB implements the best performing global algorithms for bisimulation checking on CCS processes.

All experiments are performed on a Linux cluster, composed of compute nodes with 1 TB of DDR3 1600mhz memory, four AMD Opteron 6376 processors (in total 64 cores@2,3Ghz with speedstep disabled) and interconnected using Intel True Scale InfiniBand (40 Gb/s) for low latency communication. All nodes run an identical image of Ubuntu 14.04 and MPICH 3.2 was used for MPI communication. We use the depth first search order for the forward search strategy and the breadth first search order for the backwards search strategy.

The results for the leader election and ABP are presented in Tables 1 and 2, respectively. For each entry in the tables, four runs were performed and the mean run time and the relative sample standard deviation are reported. We also report on how many microseconds were used (in parallel) per explored vertex of the dependency graph. This measure gives an idea of the speedup achieved when more cores are available. We note that for small instances this time can be very high due to the initialization of the distributed algorithm and memory allocations for dynamic data structures.

We observe that in the positive cases where the entire dependency graph must be explored, we achieve (with 256 cores) speedups 32 and 52 for leader election with 9 and 10 processes, respectively. For ABP with buffer sizes 3 and 4 the speedups are 102 and 98, respectively. However we do see a relative high standard deviation for 8-32 cores if the run time is short. This is because the scheduler is not configured to ensure locality among NUMA nodes. Compared to the performance of CWB, we observe that on the smallest instances we need up to 64 cores in leader election and 16 cores in ABP to match the run time of CWB. However, on the next instance the run time of CWB is matched already by 8 and 2 cores, respectively. This demonstrates that the performance of our distributed algorithm considerably improves with the increasing problem size.

In the negative cases, it is often enough to explore only a smaller portion of the dependency graph in order to provide a conclusive answer and here the on-the-fly nature of our distributed algorithm shows a real advantage compared

Leader election where implementation and specification are weakly bisimilar												
	9 processes			10 processes			11 processes			12 processes		
cores	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv
CWB	**8.21**	0.2	N/A	**328**	0.5	N/A	**-**	-	N/A	**-**	-	N/A
1	**187**	0.6	6399	**1957**	1.0	17921	**-**	-	-	**-**	-	-
2	**102**	0.7	482	**1020**	0.6	9338	**-**	-	-	**-**	-	-
4	**55.7**	1.0	907	**553**	1.1	5065	**-**	-	-	**-**	-	-
8	**38.6**	31.0	322	**304**	6.3	2783	**2885.7**	1.1	7013	**-**	-	-
16	**28.5**	17.6	975	**208**	5.9	1903	**2098.6**	1.1	5100	**-**	-	-
32	**16.8**	14.3	574	**120**	6.9	1099	**1172.6**	0.5	2850	**-**	-	-
64	**9.7**	3.0	332	**81**	3.5	738	**723.9**	1.7	1759	**-**	-	-
128	**7.0**	1.7	241	**53**	6.3	489	**407.4**	2.9	990	**3464**	1.3	2221
256	**5.8**	1.9	200	**38**	2.8	345	**276.8**	1.4	673	**2115**	1.0	1356

Leader election where implementation and specification are not weakly bisimilar												
	8 processes			9 processes			10 processes			11 processes		
CWB	**4.1**	0.4	N/A	**33.7**	1.3	N/A	**3765.0**	0.9	N/A	**-**	-	N/A
1	**1.5**	5.5	349.8	**13.1**	7.9	521.6	**122.3**	7.0	736.0	**1110**	0.1	920
2	**1.1**	12.7	258.2	**5.0**	10.0	908.6	**7.8**	39.8	178011	**236**	58.8	959
4	**2.1**	79.1	157.1	**8.5**	24.8	74.5	**303.4**	47.7	97.4	**2148**	*	82
8	**4.5**	46.0	25.9	**37.6**	151.9	37.0	**516.6**	164.1	52.7	**2764**	8.2	104
16	**3.6**	97.1	21.1	**31.8**	103.2	55.1	**83.3**	31.7	69.7	**1078**	7.5	342
32	**1.7**	30.9	4.7	**10.7**	67.7	19.0	**49.4**	12.7	28.5	**1072**	15.4	107
64	**0.9**	2.2	3.6	**5.2**	5.8	7.9	**75.0**	5.0	9.9	**1231**	26.1	19
128	**0.8**	13.0	3.5	**6.4**	10.3	2.7	**28.5**	13.0	8.3	**812**	32.7	7
256	**1.2**	13.4	9.4	**5.6**	6.7	1.5	**22.6**	6.9	1.5	**243**	23.8	6

Table 1: Time is reported in seconds, RSD is the relative sample standard deviation in percentage and µs/tv is the time spend per vertex in micro seconds. The star in RSD column means that only one run finished within the given timeout.

to the global algorithms implemented in CWB. For on-the-fly exploration the search order is very important and we can note that increasing the number of cores does not necessarily imply that we can compute the fixed-point value for the root faster. Even though the algorithm scales still very well and with more cores explores a substantially larger part of the dependency graph, it may (by the combined search strategy of the workers) explore large parts of the graph that are not needed for finding the answer. For example in leader election for 10 processes, two cores produced a very successful search strategy that needed only 7.8 seconds to find the answer, however, increasing the number of cores led the search in a wrong direction.

Finally, results for checking the simulation preorder on the task graph benchmark can be seen in Table 3. As this is a large number of experiments requiring nontrivial time to run, we tested the scaling only up to 64 cores. We queried whether all the tasks in the task graph (or rather their initial prefixes) can be completed within 25 time units. Out of the 180 task graphs, 61 of them are solvable in one hour (and 34 of them are schedulable while 27 are not schedulable).

ABP where implementation and specification are weakly bisimilar									
	buffer size 3			buffer size 4			buffer size 5		
cores	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv
CWB	**9.7**	0.6	N/A	**1610.3**	1.3	N/A	**-**	-	N/A
1	**81.3**	0.5	113.6	**2409.5**	0.3	161.4	**-**	-	-
2	**42.0**	0.7	58.7	**1268.5**	3.8	85.0	**-**	-	-
4	**22.4**	2.1	31.3	**650.3**	1.2	43.6	**-**	-	-
8	**13.8**	11.6	19.3	**332.0**	1.9	22.2	**-**	-	-
16	**10.2**	13.6	14.3	**239.1**	6.2	16.0	**-**	-	-
32	**5.9**	14.4	8.2	**127.0**	3.9	8.5	**3314.7**	1.0	10.8
64	**3.4**	1.2	4.7	**78.8**	2.5	5.3	**1970.5**	0.4	6.4
128	**2.1**	3.7	3.0	**42.4**	0.8	2.8	**1020.3**	1.2	3.3
256	**1.8**	23.1	2.5	**24.7**	2.7	1.7	**551.2**	0.6	1.8

ABP where implementation and specification are not weakly bisimilar									
	buffer size 4			buffer size 5			buffer size 6		
CWB	**8.3**	0.9	N/A	**170.2**	0.5	N/A	**-**	-	N/A
1	**5.0**	0.4	15365.9	**3.4**	0.3	109113	**4.1**	0.4	584643
2	**15.0**	1.2	56.9	**1.3**	14.8	179286	**4.1**	2.8	590714
4	**7.8**	4.4	37.8	**168.3**	0.5	95.9	**3125.1**	0.8	202.2
8	**6.4**	25.6	65.0	**98.1**	17.0	297.3	**1602.2**	1.0	669.4
16	**4.4**	20.0	45.5	**66.1**	13.2	108.7	**1128.2**	1.1	15391.5
32	**2.2**	3.5	694.9	**35.8**	1.6	1792.8	**649.6**	9.9	7481.1
64	**1.3**	7.3	367.4	**21.8**	1.5	1006.6	**370.9**	0.4	3869.9
128	**0.8**	3.7	289.2	**14.4**	1.4	755.7	**197.5**	1.2	2482.5
256	**0.5**	3.9	127.6	**7.9**	2.1	436.1	**107.7**	1.1	1305.1

Table 2: Time is reported in seconds, RSD is the relative sample standard deviation in percentage and µs/tv is the time spend per vertex in micro seconds.

As CWB does not support simulation preorder, the weaker trace inclusion property is used but CWB cannot solve any of the task graphs within one hour. We achieve an average 25 times speedup using 64 cores, both for the positive and negative cases, showing a very satisfactory performance on this large collection of experiments.

5 Conclusion

We presented a distributed algorithm for computing fixed points on dependency graphs and showed on weak bisimulation/simulation checking between CCS processes that, even though the problem is in general P-hard, we can in many cases obtain reasonable speed-ups as we increase the number of cores. Our algorithm works on-the-fly and hence for the cases where only a small portion of the dependency graphs needs to be explored to provide the answer, we perform significantly better than the global algorithms implemented in the CWB tool. Compared to CWB we also scale better with the increasing instance sizes, even for the cases

Weak Simulation Preorder on Task Graphs						
	Total		Positive		Negative	
cores	solved	AAT	solved	AAT	solved	AAT
1	35	19660	16	7818	19	11841
2	39	10278	18	4085	21	6192
4	43	5301	21	2095	22	3205
8	49	2996	26	1201	23	1794
16	51	2240	28	858	23	1381
32	57	1271	33	493	24	777
64	61	798	34	310	27	487

Table 3: Number of solved task graphs within 1 hour for all, positive and negative instances. The accumulated average time (AAT) is projected on 9 task graphs that 1 core is able to solve between 20 minutes and 1 hour.

where the whole dependency graph must be explored. The advantage of our approach based on dependency graphs is that we provide a general distributed algorithm and its efficient implementation that can be directly applied also to other problems like e.g. model checking—most importantly without the need of designing and coding specific single-purpose distributed algorithms for the different applications. In our future work we plan to look into finding better parallel search strategies that will allow for early termination in the cases where the fixed-point value of the root is 1 and also terminating the parallel search of the graph once we know that the exploration is not needed any more.

Acknowledgments The present work was supported by the Danish e-Infrastructure Cooperation by co-funding acquisitions of the MCC Linux cluster at Aalborg University and received funding from the Sino-Danish Basic Research Center IDEA4CPS funded by the Danish National Research Foundation and the National Science Foundation, China, the Innovation Fund Denmark center DiCyPS, as well as the ERC Advanced Grant LASSO. The fourth author is partially affiliated with FI MU in Brno.

References

1. Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 5–19. Springer, 1998.
2. X. Liu and S.A. Smolka. Simple linear-time algorithms for minimal fixed points. In *ICALP'98*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
3. José L. Balcázar, Joaquim Gabarró, and Miklos Santha. Deciding bisimilarity is p-complete. *Formal Asp. Comput.*, 4(6A):638–648, 1992.
4. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.
5. R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

6. Benedikt Bollig, Martin Leucker, and Michael Weber. *Proceedings of SPIN'02*, chapter Local Parallel Model Checking for the Alternation-Free μ-Calculus, pages 128–147. Springer, 2002.
7. Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. *Formal Methods in System Design*, 26(2):197–219, 2005.
8. Christophe Joubert and Radu Mateescu. Distributed on-the-fly model checking and test case generation. In Antti Valmari, editor, *Proceedings of SPIN'06*, volume 3925 of *LNCS*, pages 126–145. Springer, 2006.
9. T. Gibson-Robinson, Ph. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3—A modern refinement checker for CSP. In *TACAS'14*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.
10. H. Garavel, F. Lang, R. Mateescu, and W.Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.
11. Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
12. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.
13. J.R. Andersen, N. Andersen, S. Enevoldsen, M.M. Hansen, K.G. Larsen, S.R. Olesen, J. Srba, and J.K. Wortmann. CAAL: Concurrency workbench, Aalborg edition. In *Proceedings of the 12th International Colloquium on Theoretical Aspec ts of Computing (ICTAC'15)*, volume 9399 of *LNCS*, pages 573–582. Springer, 2015.
14. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5(2), 1955.
15. R. J. van Glabbeek. *The linear time - branching time spectrum*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
16. J.R. Andersen, M.M. Hansen, and N. Andersen. CAAL 2.0: Equivalences, preorders and games for CCS and TCCS. Master's thesis, Aalborg University, 2015.
17. J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Local model checking of weighted CTL with upper-bound constraints. In *Proceedings of SPIN'13*, volume 7976 of *LNCS*, pages 178–195. Springer-Verlag, 2013.
18. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of CONCUR'05*, volume 3653, pages 66–80. Springer, 2005.
19. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 53–66, London, UK, UK, 1998. Springer-Verlag.
20. E. W. Dijkstra. Shmuel Safra's version of termination detection. *EWD Manuscript 998*, 1987.
21. Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.
22. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.
23. Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

A Leader election (3 nodes)

Listing 1.1 shows the leader election with 3 nodes. As shown, `Ring` $\approx$ `Spec`, but if `P2_3` is replaced by `P2_3_bad` then `Ring` $\not\approx$ `Spec`. When `P2` knows that there exists `P3` with higher rank, the next time it receives a message with that rank, it 'steals' the election.

```
**** Ring-based Leader Election Protocol
* This example has processes P1 to P3, each process have several states
* denoted by process name underscore largest rank received. * E.g. P1_2
* is P1 in a state where it has received a * message with rank 2.
* Messages are on the form 'mreceiver rrank'
P1 = 'm3r1.P1 + m1r1.leader.0 + m1r2.P1_2 + m1r3.P1_3;
P1_2 = 'm3r2.P1_2 + m1r1.leader.0 + m1r2.P1_2 + m1r3.P1_3;
P1_3 = 'm3r3.P1_3 + m1r1.leader.0 + m1r2.P1_3 + m1r3.P1_3;

P2 = 'm1r2.P2 + m2r1.P2 + m2r2.leader.0 + m2r3.P2_3;

P2_3 = 'm1r3.P2_3 + m2r2.leader.0 + m2r1.P2_3 + m2r3.P2_3;
P2_3_bad = 'm1r3.P2_3 + m2r2.leader.0 + m2r1.P2_3 + m2r3.leader.0;

P3 = 'm2r3.P3 + m3r1.P3 + m3r2.P3 + m3r3.leader.0;

Ring = (P1 | P2 | P3)
        \ {m1r1, m1r2, m1r3, m2r1, m2r2, m2r3, m3r1, m3r2, m3r3};

Spec = leader.0;
```

Listing 1.1: Leader election with 3 nodes

B ABP (buffer size 3)

Listing 1.2 shows the alternating bit protocol with lossy medium. The medium has a buffer with 3 cells, and the difference between `Buffer3L` and `Buffer3LBad` is that one of the buffer cells in `Buffer3LBad` flips the bit.

```
*** Sender
agent Send_0 = accept.Send_out_0;
agent Send_1 = accept.Send_out_1;
agent Send_out_0 = 'send_0.Send_wait_0;
agent Send_out_1 = 'send_1.Send_wait_1;
agent Send_wait_0 = Send_out_0 + dack_0.Send_1 + dack_1.Send_wait_0;
agent Send_wait_1 = Send_out_1 + dack_1.Send_0 + dack_0.Send_wait_1;

***  Receiver
agent Receive_0 = dsend_0.Receive_ack_0 + dsend_1.'ack_1.Receive_0;
agent Receive_1 = dsend_1.Receive_ack_1 + dsend_0.'ack_0.Receive_1;
agent Receive_ack_0 = 'deliver.'ack_0.Receive_1;
agent Receive_ack_1 = 'deliver.'ack_1.Receive_0;
```

```
*** Lossy channel
agent Buffer1L = in_0.('out_0.Buffer1L + tau.Buffer1L) +
                 in_1.('out_1.Buffer1L + tau.Buffer1L);
agent Buffer2L = ((Buffer1L [shift_0/out_0, shift_1/out_1]) |
        (Buffer1L [shift_0/in_0, shift_1/in_1])) \ {shift_0, shift_1};
agent Buffer3L = ((Buffer1L [shift_0/out_0, shift_1/out_1]) |
        (Buffer2L [shift_0/in_0, shift_1/in_1])) \ {shift_0, shift_1};
agent Buffer2LBad = ((Buffer1L [shift_1/out_0, shift_0/out_1]) |
        (Buffer1L [shift_0/in_0, shift_1/in_1])) \ {shift_0, shift_1};
agent Buffer3LBad = ((Buffer1L [shift_0/out_0, shift_1/out_1]) |
        (Buffer2LBad [shift_0/in_0, shift_1/in_1])) \ {shift_0, shift_1};

set Internals = { send_0, send_1, ack_0, ack_1,
                  dsend_0, dsend_1, dack_0, dack_1 };

agent ABPl_3_good = (Send_0 | Receive_0 |
    (Buffer3L [send_0/in_0, dsend_0/out_0, send_1/in_1, dsend_1/out_1]) |
    (Buffer3L [ack_0/in_0, dack_0/out_0, ack_1/in_1, dack_1/out_1]))
    \ Internals;
agent ABPl_3_bad = (Send_0 | Receive_0 |
    (Buffer3LBad [send_0/in_0, dsend_0/out_0, send_1/in_1, dsend_1/out_1]) |
    (Buffer3L [ack_0/in_0, dack_0/out_0, ack_1/in_1, dack_1/out_1]))
    \ Internals;

agent SPEC = accept.'deliver.SPEC;
```

Listing 1.2: Alternating bit protocol with buffer size 3

C Task graph (4 tasks, 12 ticks)

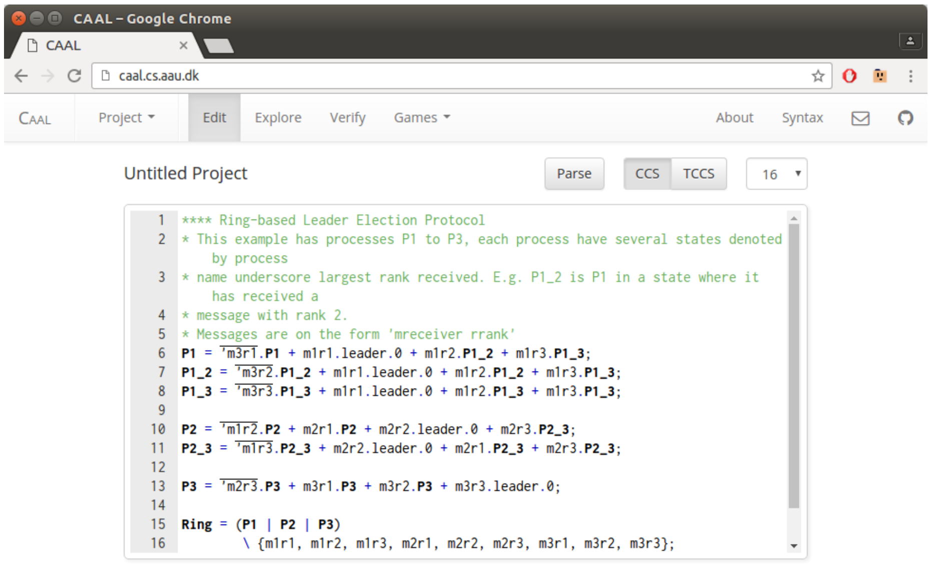
Listing 1.3 shows a task graph with 4 tasks. The specification asks, using weak simulation, is it possible to complete all task in 12 ticks (`t`).

```
agent T0 = T0D;
agent T0D = done0.T0D + 't0d.T0D;

agent T1 = t0d.(e1.e1.e1.e1.e1.e1.e1.T1D + e2.e2.e2.e2.e2.e2.e2.T1D);
agent T1D = done1.T1D + 't1d.T1D;

agent T2 = t0d.(e1.e1.e1.e1.e1.T2D + e2.e2.e2.e2.e2.T2D);
agent T2D = done2.T2D + 't2d.T2D;

agent T3 = t0d.(e1.e1.e1.e1.e1.e1.e1.T3D + e2.e2.e2.e2.e2.e2.e2.T3D);
agent T3D = done3.T3D + 't3d.T3D;

agent T4 = t0d.(e1.e1.e1.e1.e1.e1.T4D + e2.e2.e2.e2.e2.e2.T4D);
agent T4D = done4.T4D + 't4d.T4D;
```

```
* Tasks
agent Tasks = (T0 | T1 | T2 | T3 | T4) \ {t0d, t1d, t2d, t3d, t4d};

* Processors
agent Processors = (P1 | P2) \ {tick};

agent P1 = tick.t.tick.('e1.P1 + P1);
agent P2 = 'tick.'tick.('e2.P2 + P2);

* System definitions
agent System = (Processors | Tasks) \ {e1, e2};

agent Spec = t.t.t.t.t.t.t.t.t.t.t.t.(done0.0 + done1.0 +
                                      done2.0 + done3.0 + done4.0);
```

Listing 1.3: Task graph with 4 tasks

D CAAL Screenshot



```
**** Ring-based Leader Election Protocol
* This example has processes P1 to P3, each process have several states denoted
    by process
* name underscore largest rank received. E.g. P1_2 is P1 in a state where it
    has received a
* message with rank 2.
* Messages are on the form 'mreceiver rrank'
P1 = 'm3r1.P1 + m1r1.leader.0 + m1r2.P1_2 + m1r3.P1_3;
P1_2 = 'm3r2.P1_2 + m1r1.leader.0 + m1r2.P1_2 + m1r3.P1_3;
P1_3 = 'm3r3.P1_3 + m1r1.leader.0 + m1r2.P1_3 + m1r3.P1_3;

P2 = 'm1r2.P2 + m2r1.P2 + m2r2.leader.0 + m2r3.P2_3;
P2_3 = 'm1r3.P2_3 + m2r2.leader.0 + m2r1.P2_3 + m2r3.P2_3;

P3 = 'm2r3.P3 + m3r1.P3 + m3r2.P3 + m3r3.leader.0;

Ring = (P1 | P2 | P3)
        \ {m1r1, m1r2, m1r3, m2r1, m2r2, m2r3, m3r1, m3r2, m3r3};
```

Fig. 4: Screenshot of the CAAL user interface.