On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency

Kim Völlinger(✉) and Samira Akili

Humboldt University of Berlin, Berlin, Germany
voellinger@hu-berlin.de

Abstract. A major problem in software engineering is assuring the correctness of a distributed system. A *certifying distributed algorithm* (CDA) computes for its input-output pair (i, o) an additional *witness* w – a formal argument for the correctness of (i, o). Each CDA features a *witness predicate* such that if the witness predicate holds for a triple (i, o, w), the input-output pair (i, o) is correct. An accompanying *checker* algorithm decides the witness predicate. Consequently, a user of a CDA does not have to trust the CDA but its checker algorithm. Usually, a checker is simpler and its verification is feasible. To sum up, the idea of a CDA is to adapt the underlying algorithm of a program at design-time such that it verifies its own output at runtime. While certifying *sequential* algorithms are well-established, there are open questions on how to apply certification to *distributed algorithms*. In this paper, we discuss *distributed checking* of a *distributed witness*; one challenge is that all parts of a distributed witness have to be *consistent* with each other. Furthermore, we present a method for *formal instance verification* (i.e. obtaining a *machine-checked* proof that a particular input-output pair is correct), and implement the method in a framework for the theorem prover Coq.

Keywords: Certification · Distributed algorithm
Formal instance verification

1 Introduction

A major problem in software engineering is assuring the *correctness* of distributed systems. A distributed system consist of computing components that can communicate with each other. An algorithm that is designed to run on a distributed system is called a distributed algorithm. The correctness of a distributed algorithm usually relies on subtle arguments in hand-written proofs. Consequently, these proofs can easily be flawed. While complete formal verification is often too costly, testing is not sufficient if the system is of critical importance. Runtime verification tries to bridge this gap by being less costly than complete verification while still using mathematical reasoning.

C. Baier and L. Caires (Eds.): FORTE 2018, LNCS 10854, pp. 161–180, 2018.
https://doi.org/10.1007/978-3-319-92612-4_9

We investigate certifying distributed algorithms. A *certifying distributed algorithm* (CDA) computes for its input-output pair (i, o) additionally a *witness* w – a formal argument for the correctness of the input-output pair (i, o). Each CDA features a *witness predicate* such that if the witness predicate holds for a triple (i, o, w), the input-output pair (i, o) is correct. A "correct" CDA always computes a witness such that the witness predicate holds. However, the idea is that a user of a CDA does not have to trust the algorithm. That is why, an accompanying *checker* algorithm decides the witness predicate at runtime. The user of a CDA has to trust neither the implementation nor the algorithm nor the execution. However, the user has to trust the checker to be sure that if the checker accepts on (i, o, w), the particular input-output pair (i, o) is correct. Usually, a checker is simple and its verification feasible. By combining a CDA with program verification (e.g. verifying the checker), we gain formal instance correctness (i.e. a *machine-checked proof* that a particular input-output pair is correct). To sum up, the idea of a CDA is to adapt the underlying algorithm of a program at design-time such that it verifies its input-output pair at runtime. Hence, using a CDA is a formal method and a runtime verification technique.

While certifying *sequential* algorithms are well-established [19], there are open questions on how to apply certification to *distributed algorithms* [29]. In particular, there are various ways of applying the concept of certification to distributed algorithms. For instance, one question is whether to verify the input-output pair of a component or the distributed input-output pair of the system. Another question is whether the witness is checked by a distributed or sequential checker.

In this paper, we introduce a class of CDAs which features *distributed* checking of a *distributed* witness that verifies the correctness of a distributed input-output pair. Particularly, we discuss the challenge that all parts of a distributed witness have to be *consistent* with each other (Sect. 2). Moreover, we present a method for formal instance verification where we integrate the notion of consistency. We implement the method in a framework for the theorem prover Coq such that a verified distributed checker can be deployed on a real distributed system (Sect. 3). Our Coq formalization is on GitHub[1]. Moreover, we discuss related work (Sect. 4), as well as our contributions and future work (Sect. 5).

2 Certifying Terminating Distributed Algorithms

A distributed algorithm is designed to run on a distributed system, e.g. a network. We assume networks that are *asynchronous*, *static* and *id-based*. We model the topology of a network as a connected undirected graph $G = (V, E)$ with $V = \{1, 2, ..., n\}$: a vertex represents a component and an edge a channel. A *distributed algorithm* consists of an algorithm for each component such that all components together solve one problem (e.g. leader election or coloring) [17,25]. Components communicate with each other by sending messages via the channels.

[1] https://github.com/voellinger/verified-certifying-distributed-algorithms/tree/master/Framework.

A distributed algorithm can be either designed to terminate or to run continuously (e.g. a communication protocol). In this paper, we focus on *terminating* distributed algorithms. Thus, we deal with verifying a *distributed* input-output pair. In contrast, for a non-terminating algorithm, we would verify a behavior during the execution.

The rest of this Section is organized as follows. We start by defining the interface of a CDA. (Sect. 2.1). Moreover, we give a small example of a CDA to illustrate our formalization (Sect. 2.2). Subsequently, we define a witness predicate (Sect. 2.3) and a consistent witness (Sect. 2.4). For distributed checking of the witness predicate, we discuss how to decide a set of predicates for each component (Sect. 2.5). Finally, we define a class of CDAs (Sect. 2.6) and present the accompanying distributed checker of such a CDA (Sect. 2.7).

2.1 Interface of a CDA

The input of a distributed algorithm is *distributed* over the network in the way that each component gets a part of it. A *terminating* distributed algorithm computes an output in the way that each component computes a part of it. We call the algorithm of a component a *sub-algorithm* of the distributed algorithm, and a component's part of the (distributed) input/output its *sub-input*/*sub-output*. As usual when considering distributed algorithms, we abstract from distributing the input and collecting the sub-output.

Analogously to the computation of the output, a CDA additionally computes a distributed witness. We then call the algorithm of a component a *certifying* sub-algorithm of the CDA, and a component's part of the witness its *sub-witness*. We distinguish between a witness and a *potential* witness. While a witness is a proper correctness argument, a potential witness is an artifact computed by an untrusted algorithm. We formally define a witness in Sect. 2.3.

For our formalization, we assume that an input assigns values to variables, and analogously, for an output and potential witness. A variable gets assigned exactly one value for a sub-input. An input is composed of all sub-inputs, and thus, in contrast, the same variable may get assigned multiple values. That is why, we distinguish two types of assignments for our formalization. For sets A and B, a function $f : A \to B$ is an *assignment* of A in B. A relation $r \subseteq A \times B$ is a *weak assignment* of A in B. We denote the set of all assignments of A in B as $[A]$ and the set of all weak assignments of A in B as $[\![A]\!]$ (assuming B from the context).

Let I, O and W be finite sets of variables for the input, the output and the potential witness, respectively. For readability, we use different sets even though they do not have to be disjoint. We assume subsets $I_v \subseteq I$, $O_v \subseteq O$ and $W_v \subseteq W$ of variables for each component $v \in V$ such that $I = \cup_{v \in V} I_v$, $O = \cup_{v \in V} O_v$ and $W = \cup_{v \in V} W_v$. Let Val be a set of values. For each $v \in V$, let the sets of assignments $[I_v]$, $[O_v]$ and $[W_v]$ in Val be the sets of sub-inputs, sub-outputs, and sub-witnesses. Let the sets of weak assignments $[\![I]\!]$, $[\![O]\!]$ and $[\![W]\!]$ in Val be the sets of inputs, outputs and potential witnesses. The following holds for an input: if we have a sub-input $i_v \in [I_v]$ for each $v \in V$, then the weak assignment

$i = \cup_{v \in V} i_v$ is the according input. The same holds each for an output and a potential witness.

In the sequel, we fix

– the graph G as the network topology,
– the set Val as a domain,
– the sets of weak assignments $[\![I]\!]$, $[\![O]\!]$ and $[\![W]\!]$ in Val as inputs, outputs and potential witnesses,
– and the sets $[I_v]$, $[O_v]$ and $[W_v]$ in Val for each $v \in V$ as sub-inputs, sub-outputs, and sub-witnesses of v.

Moreover, we assume the minimal sub-input of a component is its own ID and the IDs of its neighbors in the network graph. Hence, the minimal input is the network itself.

2.2 Example: Witness for a Bipartite Network

As an example, consider distributed bipartite testing [5] where the components decide together whether the underlying network graph is bipartite (i.e. its vertices can be divided into two partitions such that each edge has a vertex in each partition). The input is the network itself presented by the sub-input of each component: the component's ID and the IDs of its neighbors in the network. In the case of a bipartite network, the sub-output of each component is 'true'. While in the case of a non-bipartite network, some components have the sub-output 'false' and the other components 'true'. In either case, the output is composed of those sub-outputs.

We consider a certifying variant of distributed bipartite testing. It follows from the definition of bipartiteness that a bipartition of the network's components is a witness for a network being bipartite. The witness is distributed in the way that each component has a bipartition of its neighborhood as a sub-witness. For the more sophisticated witness of a non-bipartite network, see [28].

For a better understanding of the formalization, consider the concrete network shown in Fig. 1 where e.g. the sub-input $\mathrm{i}_3 \in [\mathrm{I}_3]$ of component 3 assigns the value $\{6\} \in \mathrm{P(V)}$ to the variable $\mathrm{nbrs}_3 \in \mathrm{I}_3$. In the remainder of this Section, we refer to this example to illustrate concepts.

2.3 Witness Predicate

For the problem to be solve by a terminating distributed algorithm, we assume a specification given as a precondition $\phi \subseteq [\![I]\!]$ and a postcondition $\psi \subseteq [\![I]\!] \times [\![O]\!]$. In the following, we fix the specification over input-output pairs as

$$\forall i \in [\![I]\!], o \in [\![O]\!] : \psi(i, o) \vee \neg\phi(i)$$

We define a witness predicate over inputs, outputs and potential witnesses for the ϕ-ψ specification, and define the notion of a witness:

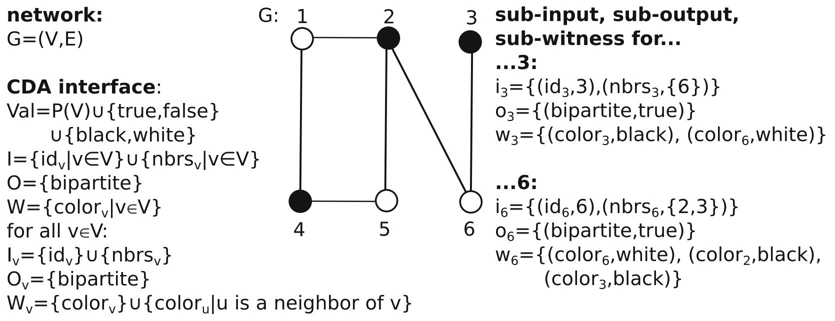
network:
G=(V,E)
CDA interface:
Val=P(V)∪{true,false} ∪{black,white}
I={id_v|v∈V}∪{nbrs_v|v∈V}
O={bipartite}
W={color_v|v∈V}
for all v∈V:
I_v={id_v}∪{nbrs_v}
O_v={bipartite}
W_v={color_v}∪{color_u|u is a neighbor of v}
G:
1
2
3
4
5
6
sub-input, sub-output, sub-witness for...
...3:
i_3={(id_3,3),(nbrs_3,{6})}
o_3={(bipartite,true)}
w_3={(color_3,black), (color_6,white)}
...6:
i_6={(id_6,6),(nbrs_6,{2,3})}
o_6={(bipartite,true)}
w_6={(color_6,white), (color_2,black), (color_3,black)}

Fig. 1. Example of a bipartite network with the CDA interface and the sub-input, sub-output and sub-witness of components 3 and 6. P(V) denotes the power set of V.

Definition 1 (witness predicate, witness, complete).

(i) *A predicate* $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *with the* witness property

$$\forall i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!] : \Gamma(i, o, w) \longrightarrow (\psi(i, o) \vee \neg\phi(i))$$

is a witness predicate *for a ϕ-ψ specification.*

(ii) *If* $(i, o, w) \in \Gamma$*, w is a* witness *for the correctness of* (i, o).

(iii) Γ *is a* complete *witness predicate if additionally holds*

$$\forall i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!] : \Gamma(i, o, w) \longleftarrow (\psi(i, o) \vee \neg\phi(i))$$

Note that an algorithm computes a *potential* witness w since it may be that $(i, o, w) \notin \Gamma$. However, if clear from context, we simply say witness from now on.

The witness predicate of the bipartite example states that the witness is a bipartition in the network. Its witness property follows by the definition of bipartiteness. Since the witness predicate holds with a biimplication, it is complete.

2.4 Consistency of a Distributed Witness

In the bipartite example, a sub-witness contains the colors of the neighbours – a bipartition of the neighborhood. Note that the sub-witnesses of neighbors have some common variables. In the example shown in Fig. 1, the components 3 and 6 have the variable $color_3$ in common. Consequently, in order to form a bipartition in the network, the common variables have to be consistent in their assignment.

In the general case, all sub-witnesses have to be consistent with each other in order to form a proper argument for the correctness of an input-output pair.

Definition 2 (consistent). *Let $w \in [\![W]\!]$ be a witness.*

(i) *For $u, v \in V$: sub-witnesses $w_u \subseteq w$ and $w_v \subseteq w$ are* consistent *if and only if for all $a \in W_u \cap W_v$ holds $w_u(a) = w_v(a)$.*

(ii) *w is* consistent *if and only if $w \in [W]$.*

In the example of bipartite testing (Fig. 1), the sub-witnesses of components 3 and 6 have common variables: $W_3 \cap W_6 = \{color_3, color_6\}$. Since $w_3(color_3) = black = w_6(color_3)$ and $w_3(color_6) = white = w_6(color_6)$, the sub-witnesses w_3 and w_6 are consistent.

A witness is trivially consistent if for all $u, v \in V$ pairwise holds $W_u \cap W_v = \emptyset$. However, having a trivially consistent witness is often only possible by having a trivial distribution of the witness, since the witness is basically centralized, i.e. $W_v = W$ for one $v \in V$ and holds $W_u = \emptyset$ for all other $u \in V$. For instance, in the bipartite example, one component $v \in V$ has to have the whole bipartition of the network and network topology as a sub-witness then. Assume there is one other component u that has a part of the bipartition and the topology as its sub-witness. Then the two bipartitions presented in w_v and w_u have to be related to each other. Otherwise, the two bipartitions together may not form a bipartition. Hence, $W_v \cap W_u \neq \emptyset$ – a contradiction to the witness being trivially consistent. As a consequence, there are usually some components $u, v \in V$ with common variables in their sub-witnesses, i.e. $W_u \cap W_v \neq \emptyset$.

Lemma 1. *A witness is consistent if and only if all of its sub-witnesses are pairwise consistent.*

Proof. Let $w \in [W]$ be a consistent witness. Then for all $a \in W$ there is a unique value $w(a)$. Thus, for all $u, v \in V$ with $w_u, w_v \in w$ holds $w_u(a) = w(a) = w_v(a)$ if $a \in W_u \cap W_v$. Consequently, all sub-witnesses are pairwise consistent.

For the other direction, assume all sub-witnesses of $w \in [\![W]\!]$ are pairwise consistent. For all $a \in W$, there is a at least one component, w.l.o.g. $v \in V$, with $a \in W_v$ since $W = \cup_{v \in V} W_v$. For every component $u \in V$ with $a \in W_u$ holds $w_u(a) = w_v(a)$. Hence, $w \in [W]$ is consistent.

The need for consistency arises because the witness is *distributed*. Hence, certifying *sequential* algorithms do not have to deal with consistency (c.f. [19]). As a consequence, checking becomes more challenging for certifying *distributed* algorithms. To avoid checking consistency of all sub-witnesses pairwise, we restrict ourselves to a connected witness. We define a connected witness over all a-components:

Definition 3 (a-component). *If $a \in W_v$ for a component $v \in V$, then v is an a-component.*

Definition 4 (connected). *A witness $w \in [\![W]\!]$ is* connected *if for all $a \in W$, the sub-graph induced by the the a-components is connected.*

In the example shown in Fig. 1, the witness is connected. For instance, the components 2, 3 and 6 are the $color_6$-components and they induce a connected sub-graph.

As an example for a witness that is not connected, assume a bipartite network where components belonging to the same partition solve one task together. Moreover, assume a part of this task is agreeing on some choice with one consent (i.e. a consensus problem [17]). In order to verify that all components of one partition agree on their choice, the sub-witness of a component consists of its own choice and of the choices of the components in 1-hop-distance – components that share a neighbor are in 1-hop-distance. For example in Fig. 1, component 3 is in 1-hop-distance of component 2. The components in 1-hop-distance always belong to the same partition. The witness predicate is satisfied if each component agrees on its choice with the components in a 1-hop-distance. The witness is not connected since only components of the same partition share variables in their sub-witnesses, and therefore do not induce a connected subgraph.

Lemma 2. *Let $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ be a predicate. For every triple $(i, o, w) \in \Gamma$ where w is not connected, there is a triple $(i, o, w) \in \Gamma$ where w' is connected.*

Proof. Since $w \in [\![W]\!]$ is not connected, there are components $u, v \in V$ with $a \in W_u \cap W_v$ such that there is no path $p = (u, x_1, x_2, ..., x_m, v)$ between u and v with all components x_l on the path having $a \in W_{x_l}$ for $l = 1, 2, .., m$.

We construct a connected witness w' from w. We add for each such outlined pair of components u, v on one path between u and v the missing variables $a \in W_u \cap W_v$. W.l.o.g. let this path be $p = (u, x_1, x_2, ..., x_m, v)$. For each component x_l for $l = 1, 2, ..., m$ is $W'_{x_l} := W_{x_l} \cup \{a\}$. It follows that w' is connected.

To ensure $(i, o, w') \in \Gamma$, we construct the sub-witnesses w'_{x_l} by adding the assignments of u (or analogously v): $w'_{x_l} := w_{x_l} \cup \{(a, w_u(a)) | a \in W'_{x_l} \setminus W_{x_l}\}$ for all $l = 1, 2, ..., m$. Since w and w' are the union of the sub-witnesses, it holds $w = w'$, and therefore $(i, o, w') \in \Gamma$.

For a connected witness, it is sufficient to check the consistency in each neighborhood.

Definition 5 (consistent neighborhood). *Let $w \in [\![W]\!]$ be a witness. $v \in V$ has a* consistent *neighborhood if and only if for all neighbors u of v holds the sub-witnesses $w_v \subseteq w$ and $w_u \subseteq w$ are consistent.*

Theorem 1. *Let $w \in [\![W]\!]$ be a connected witness. w is consistent if and only if the neighborhood is consistent for all $v \in V$.*

Proof. If w is consistent, then it follows from Lemma 1 that all sub-witnesses of w are pairwise consistent. Thus, for each $v \in V$ the neighborhood is consistent.

For the other direction, let $u, v \in V$ with $a \in W_u \cap W_v$. From the definition of a connected witness follows, there exists a path between the a-components u and v over a-components. Since on this path all neighboring components are consistent, it follows by transitivity that u and v are consistent. Thus, the witness w is consistent.

For some CDAs, a sub-witness of a component v holds variables of the sub-output of a component u, c.f. [28–30]. Revisit the example where the components of one partition in a bipartite network solve a consensus problem. The sub-output of a component is its own choice. Part of the sub-witness of a component is the choices of the components in 1-hop-distance. Hence, the sub-witness of a component consists partly of sub-outputs of other components. For the shared variables, the sub-outputs and sub-witnesses have to be consistent in their assignments. Since we do not want to check the consistency between sub-witnesses and sub-outputs or sub-inputs, we define a complete witness:

Definition 6 (complete). *A witness $w \in [\![W]\!]$ is* complete *if for all $u, v \in V$ and all $a \in W_u$ holds if $a \in I_v \cup O_v$, then $a \in W_v$.*

Note that if for all $v \in V$ holds $i_v \subseteq w_v$ and $o_v \subseteq w_v$, then the witness is complete.

2.5 Distributable Witness Predicate

In Sect. 2.7, we present a distributed checker that decides the witness predicate. However, the Definition 1 of the witness predicate is defined over the input, output and potential witness of a CDA and does not take into account sub-inputs, sub-outputs and sub-witnesses of the components. For distributed checking of the witness predicate, we define predicates that are decided for each component over the sub-input, sub-output and sub-witness, and are then combined to decide the witness predicate (c.f. [28]). A witness predicate is distributable in a network if some predicates hold for all components while others hold for at least one:

Definition 7 (distributable, completely).

(i) Let $i \in [\![I]\!]$ be an input and its sub-inputs $i_v \in [I]$ for $v \in V$ such that $i = \cup_{v \in V} i_v$, let $o \in [\![O]\!]$ be an output and its sub-outputs $o_v \in [O]$ for $v \in V$ such that $o = \cup_{v \in V} o_v$, and let $w \in [\![W]\!]$ be a potential witness and its sub-witnesses $w_v \in [W]$ for $v \in V$ such that $w = \cup_{v \in V} w_v$. A predicate $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ is distributable *if one of the following holds:*

1. *Γ is* universally *distributable with a predicate $\gamma \subseteq [I] \times [O] \times [W]$ if:*
   $(\forall i_v \in [I_v], o_v \in [O_v], w_v \in [W_v] : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$.
2. *Γ is* existentially *distributable with a predicate $\gamma \subseteq [I] \times [O] \times [W]$ if:*
   $(\exists i_v \in [I_v], o_v \in [O_v], w_v \in [W_v] : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$.
3. *There exist distributable predicates Γ_1, Γ_2 such that*
   $(\Gamma_1(i, o, w) \wedge \Gamma_2(i, o, w)) \longrightarrow \Gamma(i, o, w)$.
4. *There exist distributable predicates Γ_1, Γ_2 such that*
   $(\Gamma_1(i, o, w) \vee \Gamma_2(i, o, w)) \longrightarrow \Gamma(i, o, w)$.

(ii) If the implications of 1-4 are also biimplications, then Γ is completely *distributable.*

The predicates Γ_1 and Γ_2 "divide" the witness predicate in universally or existentially distributable predicates that are linked together by a conjunction or disjunction. We call the predicates Γ_1 and Γ_2 the *distribution-predicates* of Γ, and a predicate γ a *sub-predicate* of a universally or existentially distributable predicate.

Revisit the example of bipartite testing (Sect. 2.2), the witness predicate holds if the witness is a bipartition of the network. This witness predicate is universally distributable with a distribution-predicate that is satisfied if there is a bipartition of the neighborhood for all components, and a sub-predicate stating that the sub-witness of component is a bipartition of the neighborhood. For an example of a not simply universally distributable witness predicate, see [28].

Note that not every predicate is distributable since we allow only conjunction and disjunction of distributable predicates (see rules 3 and 4). As a consequence, we cannot form a nesting of quantifiers for instance. However, the chosen restrictions enable us to decide the sub-predicates γ for each component independently, and to evaluate distribution-predicates in the whole network by using a spanning tree (c.f. Sect. 2.7). A more complex structure than a spanning tree would be needed to evaluate nested quantification in the network.

2.6 A Class of Certifying Distributed Algorithms

We define a class of certifying distributed algorithm that terminate and verify their distributed input-output pair at runtime by a distributed witness such that the distributable witness predicate is decided by a distributed checker:

Definition 8 (Certifying Distributed Algorithm). *A* certifying distributed algorithm *solving a problem specified by a ϕ-ψ specification computes for each input $i \in [\![I]\!]$ an output $o \in [\![O]\!]$, and a witness $w \in [\![W]\!]$ in the way that each component $v \in V$ computes for a sub-input $i_v \in [I]$, a sub-output $o_v \in [O]$ and a sub-witness $w_v \in [W]$ such that $i = \cup_{v \in V} i_v$, $o = \cup_{v \in V} o_v$ and $w = \cup_{v \in V} w_v$. Let $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ be a complete witness predicate for a ϕ-ψ specification. The following holds:*

(i) $(i, o, w) \in \Gamma$,
(ii) *Γ is completely distributable,*
(iii) *w is consistent,*
(iv) *w is complete with $i_v \subseteq w_v$ and $o_v \subseteq w_v$ for all $v \in V$, and*
(v) *w is connected.*

From (i) follows the correctness of the input-output pair (i, o). With (ii), we enable distributed checking of Γ. Usually, there are some components $u, v \in V$ with common variables in their sub-witnesses, i.e. $W_u \cap W_v \neq \emptyset$. Hence, the distributed witness has to be consistent as stated in (iii). By having a complete and connected witness as stated in (iv) and (v), we enable distributed checking of the consistency of the witness. Note that a connected witness is no restriction on the kind of possible correctness arguments following from Lemma 2.

Remark 1. For every distributed algorithm solving a problem specified by ϕ and ψ, there is a certifying variant belonging to the outlined class. A terminating distributed algorithm can always compute a witness for a correct input-output pair, e.g. the history of computation and communication for each component. The witness predicate then is satisfied if the computation and communication is in accordance with the algorithm.

However, proving the witness property then becomes complete verification of the distributed algorithm. Hence, a challenge is to find a "good" witness (c.f. [19] for certifying *sequential* algorithms). Finding a witness is a creative task just like developing an algorithm. However, design patterns such as using characterizing theorems or a spanning tree help.

There are two perspectives on a CDA: the one of the developer and the one of the user. The developer proves the correctness of his/her algorithm. By the definition of a CDA, the developer has for instance to prove that the algorithm computes a witness *for all* input-output pairs. For the user, however, it is enough to be convinced that his/her *particular* input-output pair is correct. To this end, the user has to understand the witness property of the witness predicate and to understand that the witness predicate is distributable. The user does not have to understand that the witness predicate is complete or that it is completely distributable. If the witness predicate is satisfied, the particular input-output pair is correct; if not, the output or the witness is not correct. Consequently, using a CDA comes at the expense of incomplete correctness.

Since for a satisfied witness predicate, the user still has to trust in the witness property, we discuss machine-checked proofs for a reduced trust base in Sect. 3.

2.7 Distributed Checker of a Distributed Witness

Let Γ be a distributable witness predicate with distribution-predicates $\Gamma_1, \Gamma_2, .., \Gamma_k$ and according sub-predicates γ_j, $j = 1, 2, .., k$. For distributed checking of Γ, each component has a *sub-checker* that checks the completeness of its sub-witness, the consistency of the sub-witnesses in the neighborhood, decides the sub-predicates for its component, and plays its part in checking the connectivity of the witness, and in evaluating the witness predicate. We assume a sub-checker gets a trusted copy of the sub-input (c.f. [19]). After termination is detected (e.g. as in [25]), a sub-checker receives the sub-output and sub-witness of its component, and starts checking. We assume a spanning tree as a communication structure in the network. This spanning tree is either reused or computed as discussed in Sect. 3.

Completeness. For each $v \in V$, let the predicate $comp_v$ denote whether w_v is complete: $i_v \subseteq w_v$ and $o_v \subseteq w_v$. The sub-checker of v decides $comp_v(i_v, o_v, w_v)$.

Connectivity. For each variable $a \in W$ in each connected subgraph ofpg a-components, the components select the a-component with the smallest ID as a leader: First, each component v suggests itself as a leader for all its variables $a \in W_v$ to its neighbors. If a component receives a message containing a suggestion of a smaller leader for one of its variables, it updates the leader and forwards the message to all neighbors. After detection of termination, each component v holds a list associating the according leader ID with each variable: $((a_1, v_1), (a_2, v_2), ..., (a_m, v_m))$ with $a_j \in W_v$, $v_j \in V$ and $j = 1, 2, ..., m$. Note that a component v does not forward a message if it receives a suggestion for a leader of a variable $a \notin W_v$. Thus, if there are two different leaders for the same variable a in the network, then the subgraph of a-components is unconnected and thereby the witness is not connected. Deciding whether there are multiple leaders for one for one variable can be done by using a spanning tree. Since we use a spanning tree as well for deciding the witness predicate, we describe this step as part of the evaluation.

Consistency. For each $v \in V$, let the predicate $cons_v \subset [W_v] \times [W_{u1}] \times [W_{u2}] \times ... \times [W_{ul}]$ denote whether the neighborhood of v is consistent with neighbors $u1, u2, ..., ul \in V$. We assume the sub-checkers of neighbors can communicate with each other. It follows from Theorem 1 that the consistency of a connected witness can be decided by a distributed algorithm where a component only once exchanges messages with its neighbors. Each sub-checker sends the sub-witness of its component to the neighboring sub-checkers. Subsequently, a sub-checker of each component v compares the sub-witness w_v with each of the received sub-witnesses: If for all $a \in W_v \cap W_{ui}$, $w_v(a) = w_{ui}(a)$, then $cons_v(w_v, w_{u1}, w_{u2}, ...w_{ul})$ holds.

Sub-Predicates. Each sub-checker of a component $v \in V$ decides each sub-predicate γ_1, γ_2,..., γ_k for the triple (i_v, o_v, w_v). Finally, the sub-checker holds a k-tuple containing the according evaluated sub-predicates.

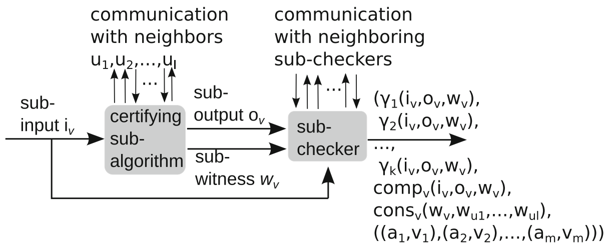
communication with neighbors u1,u2,...,ul
communication with neighboring sub-checkers
sub-input iv
certifying sub-algorithm
sub-output ov
sub-witness wv
sub-checker
(γ1(iv,ov,wv), γ2(iv,ov,wv), ..., γk(iv,ov,wv), compv(iv,ov,wv), consv(wv,wu1,...,wul), ((a1,v1),(a2,v2),...,(am,vm)))

Fig. 2. A certifying sub-algorithm of $v \in V$ and its sub-checker.

Evaluation. Figure 2 shows a component with its sub-checker: Each sub-checker of a component v with neighbors $u1, u2, ..., ul \in V$ holds a $k+3$-tuple consisting of k evaluated predicates, the evaluated predicates $comp_v$ and $cons_v$, and the list of associated leaders for each $a \in w_v$. To evaluate the witness predicate, the sub-checkers combine their tuples by using the rooted spanning tree: Starting by the leaves, each sub-checker gets the tuple of each child and combines it with its own tuple: if the j-th sub-predicate is universally distributable, then the j-th position of both tuples is combined by logical conjunction; otherwise the j-th sub-predicate is existentially distributable and logical disjunction is used instead. Let the predicate $Comp$ denote whether each sub-witness is complete and the predicate $Cons$ denote whether a witness w is consistent in the network; hence, both predicates are treated as universally distributable. For the connectivity, each component compares the chosen leaders of itself and its children. If a variable has multiple leaders, the component sends 'false' to its parent otherwise a list with the so far chosen leaders. If a component receives false from a child, it just sends 'false' to its parent. Finally, the root creates the tuple $(\Gamma_1(i,o,w), \Gamma_2(i,o,w), .., \Gamma_k(i,o,w), Comp(w), Cons(w), Con(w))$ where the predicate Con is fulfilled if there are no multiple leaders for a variable – hence the witness is connected. The root evaluates the witness predicate by combining the distribution-predicates accordingly. The evaluation terminates when root receives a message from all its children; if the witness is complete, connected and consistent, and the witness predicate satisfied, the root accepts. All sub-checkers together build a distributed checker of Γ. From the definition of a CDA and the outlined distributed checker follows: if the distributed checker accepts on a triple (i, o, w), then $(i, o) \in \psi$ or $(i) \notin \phi$.

3 Framework: Formal Instance Verification

We present a method for *formal instance verification* for CDAs (c.f. [29]). While formal verification establishes the correctness *for every* input-output pair at *design-time*, formal *instance* verification establishes the correctness *for a particular* input-output pair at *runtime*. In analogy to formal verification, formal instance verification requires a *machine-checked* proof. Hence, we have *formal instance correctness* for a particular input-output pair if there is a *machine-checked* proof for the correctness of this pair. While formal verification is often too costly, formal instance verification is often feasible but at the expense of not being complete.

To achieve formal instance correctness, we combine CDAs with theorem proving and program verification. We give an overview of the proof obligations to solve (Sect. 3.1). We implement the method in a framework for the proof assistant Coq (Sect. 3.2).

3.1 Proof Obligations for Formal Instance Verification

Using a CDA comes with a trust base: for example we have to trust that the witness predicate has the witness property or that the distributed checker algorithm is correct. According proofs have to be provided by the developer of the CDA but usually only exist on paper. Even if a distributed checker algorithm is correct on paper, the implemented distributed checker program could still be flawed. Assume a CDA with a witness predicate Γ, we have to solve the following proof obligations (PO) to obtain formal instance correctness:

PO I The implemented termination detection is correct.
PO II Witness predicate Γ has the following properties:
  (i) Γ has the witness property (c.f. Sect. 2.3)
  (ii) Γ is distributable (c.f. Sect. 2.5).
PO III The Theorem 1 for distributed checking of consistency (c.f. Sect. 2.3).
PO IV The implemented distributed checker is correct (c.f. Sect. 2.7):
  (i) Each sub-checker checks if its sub-witness is complete.

(ii) Each sub-checker takes part in checking if the witness is connected.
(iii) Each sub-checker checks the consistency sub-witnesses in the neighborhood.
(iv) Each sub-checker decides the sub-predicates for its component.
(v) Each sub-checker takes part in evaluation of Γ.

By solving these proof obligations, it follows: If the distributed checker accepts on an input, output and witness, we have a machine-checked proof that the particular input-output pair is correct. Note that the computation of the output is not mentioned in the proof obligations; the CDA is treated as a black box.

According to the concept of certifying algorithms the verification of the checker should be easier than verifying the actual algorithm. We note that for our class of CDA the checker has to perform five tasks making it seemingly complex. Note that, except for PO IV(iv), each task only needs to be verified once for the outlined class of CDA. As a consequence, the verification effort for each certifying algorithm is the same in the distributed setting as in the sequential setting.

3.2 Overview of the Framework

We use the proof assistant Coq [14] for theorem proving and program verification. Coq provides a higher-order logic, a programming language, and some proof automations. Even though Coq's programming language is not turing-complete (since every program halts), Coq implements a mechanism to extract programs to functional programming languages like OCaml. To model a network in Coq, we use the graph library Graph Basics [8] for the topology, and the framework Verdi [31] for the communication. By using Verdi, we extract a distributed checker that can be deployed on a real network.

The framework is illustrated in Fig. 3. The network model and the CDA model are fundamental for all proof obligations. The network model consists of a formalization of the network's topology and communication. The CDA model consists of the CDA Interface – a formalization of the sub-input/output/witness and witness-predicate of a particular CDA – and a verified termination detection algorithm. We use theorem proving to show for the witness predicate Γ that it has the witness property and that it is distributable (PO II) as well as for the proof of Theorem 1 (PO III). We use program verification for the termination detection algorithm (PO I) as well as for the distributed checker (PO IV). Some proof obligations have to be proven for each CDA (indicated by an arrow), others have to be proven only once for the outlined class of CDAs. In this paper, we focus on the latter ones. Note that computation of a spanning tree is an implicit part of termination detection (PO I) and evaluation PO IV(v). Hence, it makes sense to verify the computation once and then to reuse the spanning tree. Verified Coq programs can be extracted to verified OCaml programs.

We formalized the network model and CDA interface, and solved the proof obligations that deal with the consistency of the witness (PO III and PO IV(iii)). We formalized the notion of consistency and solved PO III (proof of Theorem 1)

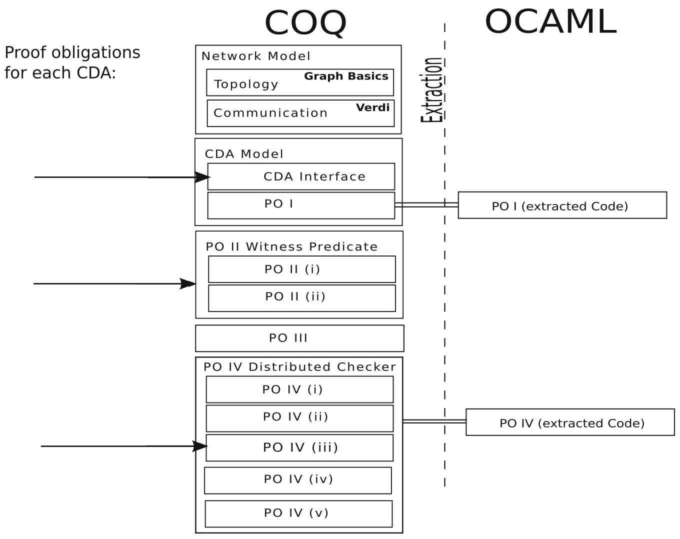

Fig. 3. A Coq framework for formal instance verification using CDAs.

in Coq. The formalization follows the definitions and the proof in Sect. 2.3 tightly. We forgo giving details in this paper. In the remainder of this section, we explain the network model and CDA interface (Sect. 3.3). We discuss the verification of the distributed consistency check (Sect. 3.4) and describe its extraction such that it runs on a real network (Sect. 3.5). PO IV(i), PO IV(ii), PO IV(v) follow the same approach as the distributed consistency check and are work-in-progress.

3.3 Network Model and CDA Interface

Topology. Since GraphBasics offers a connected graph, the representation of a network is straightforward. We assume that a component and its checker are two *logical components* which are co-located on one *physical component*. A vertex of the connected graph `Component` represents the physical component.

Communication. We model the communication between a component and its sub-checker, and between sub-checkers. To implement the communication of the distributed checker, we specify the following definitions given by Verdi: The type of a sub-checker (`Name`), the set of sub-checkers (`Nodes`), the state each sub-checker maintains (`Data`) and a function to initialize this state (`initData`). Verdi distinguishes between internal (`Input` and `Output`) and external messages (`Msg`): While internal messages are exchanged between logical components running on the same physical component, external messages are exchanged across the network. We use internal messages for the communication between a component and its sub-checker, and external messages for the communication

```
 1  Inductive Name := Checker : Component -> Name.
 2  Definition Nodes : list Name := (map Checker (CV_list v a g)).
 3  Inductive Msg := Checkermessage : Certificate -> Msg.
 4  Inductive Input : Type :=
 5    | Checkerknowledge -> Input
 6    | Checkerinput -> Input .
 7
 8  Record Data := mkData{
 9   myCertificate : Certificate;
10   variables: list A;
11   nbrslist : list Component;
12   consistent : bool ;
13   initialized : bool;
14   (...)
15    }.
16
17  Definition initData (n: Name) := mkData
18   [] [] (neighbors g (name_component n))  false false (...).
19
20  Definition InputHandler (me : Name) (c : Input) (state: Data) :=
21   match me,c  with
22   | Checker x, Checkerknowledge => if (initialized == false) then
23                  variables := Checkerknowledge.(variables)
24                  (...)
25   | Checker x, Checkerinput =>     if (initialized == false) then
26                                     myCertificate := Checkerinput.(certificate)
27                                     sendToAllNeighbors (mycertificate)
28                                     (...)
29
30  Definition NetHandler (me : Name) (src: Name) (m : Msg) (state: Data) :=
31  match m with
32   | Checkermessage certificate => if  (initialized == true) then
33                        nbrslist := remove src state.(nbrslist)
34                        consistent := state.(consistent) &&
35                        Consistency_Nbr (certificate)
```

Fig. 4. Outline of the implementation of the consistency check of a sub-checker.

between sub-checkers. For the behaviour of a sub-checker, we implement the functions `InputHandler` and `NetHandler`. The `InputHandler` runs if a sub-checker receives an internal message and the (`NetHandler`) runs if a sub-checker receives an external message. For our network model, we assume reliable communication.

Combining Verdi and Graph Basics. Verdi does not offer to specify the topology of a network. However, to reason about properties such as consistency in a neighborhood, we have to specify the underlying topology of a network. That is why, we combine Verdi with Graph Basics. To this end, we instantiate the set of `Nodes` in the network with the vertices of the topology graph.

CDA Interface. We abstract from the actual computation of a CDA. However, as a sub-checker needs to process sub-input, sub-output and sub-witness, we have to formalize them. The CDA interface consists of a formalization of the sub-input, sub-witness and sub-output as well as the structure of the witness predicate (i.e. if the distribution-predicates of the witness predicate are universally or existentially distributable). The latter is used by a sub-checker to perform the distributed evaluation of the witness predicate.

Initialization of a Sub-Checker. A sub-checker needs knowledge about its neighborhood; we implement the `initData` function (Fig. 4 l. 19) such that each sub-checker is initialized with the IDs of its neighbors. Furthermore a sub-checker is initialized with the CDA Interface. We divide the CDA interface into two parts: The first part is independent from the actual computation of the CDA and contains the minimal sub-input and structure of the witness predicate. To this end, we define the internal message `Checkerknowledge`. The second part contains additional sub-input, the sub-output and sub-witness. To this end, we define the internal message `Checkerinput`. We define the `InputHandler` of a sub-checker such that it initializes the sub-checker's state with the values obtained from `Checkerknowledge` and `Checkerinput` (Fig. 4 l. 20–28).

3.4 Checking Consistency in the Neighborhood

To check the consistency of a witness, each sub-checker checks the consistency in its neighborhood (Theorem 1). In our implementation the state of each sub-checker contains a list of its neighbors (`nbrslist`). We use `nbrslist` to keep track of the messages received from the neighbors. Additionally, the state contains the boolean `initialized` which indicates if the sub-checker is initialized as described in the previous section, and the boolean `consistent` which indicates if the sub-witness of the component is consistent with all sub-witnesses received so far (Fig. 4 l. 10–17). When a sub-checker receives a sub-witness from a neighbor, it removes the neighbor from its `nbrslist`. As a result, if `nbrslist` is empty, a sub-checker received a sub-witness from each of its neighboring sub-checkers. Subsequently, a sub-checker calls the function `Consistency_Nbr` which takes two sub-witnesses as an input and returns true if they are consistent. If `Consistency_Nbr` returns true, the checker sets `consistent` to true. After being set to false once, the value of `consistent` cannot become true again. If the consistency check fails for at least one neighborhood, the witness is inconsistent.

Verification of the Consistency Check. For the verification of the consistency check, we show that if the consistency check succeeds, the neighborhood of each sub-checker is consistent. After initialization, if a sub-checker s received and processed a sub-witness from each neighboring sub-checker and `consistent` is true, consistency in the neighborhood of s holds:

Theorem 2 (in Coq). $\forall$ *s, initialized(s)* $\wedge$ *nbrslist(s) = empty* $\wedge$ *consistent(s)* $\longrightarrow$ *Neighborhood Consistency (s)*

We prove this theorem in the following steps using Coq. First, we show that *for all reachable network states* that the following lemmas hold for each sub-checker s:

Lemma 3 (in Coq). *All components in* `nbrslist(s)` *are neighbors of s.*

Lemma 4 (in Coq). *From* `initialized(s)` *follows that, if* `nbrslist(s)` *is empty, a message was received from each neighbor of s.*

Lemma 5 (in Coq). *From* `initialized(s)` *follows that, if* `consistent(s)` *is true, the witness is consistent with each witness received so far.*

We prove the Lemmas 1–3 by *inductive state invariants* [31]. A property is an inductive state invariant if it holds in the initial state (defined by the `initState` function – Fig. 4 l. 19) and each state reachable by processing a message. Note that Lemma 2 and 3 rely on the value of `initialized` which has nothing to do with Verdi's `initState` function but with the initialization of our network model described in the previous section. As a next step, we verify the function `Consistency_Nbr` by proving that it returns true for two sub-witnesses if and only if the sub-witnesses are consistent. Finally, we show that the Lemmas 1–3 and the correctness of the function `Consistency_Nbr` together imply the correctness of Theorem 2.

3.5 Extraction of a Distributed Checker

To run our distributed checker on a real network we extract it to Ocaml and link it with the Verdi Shim – a small library which e.g. provides network primitives. In order to extract our distributed checker we have to provide a specific topology and instantiate the types of the CDA interface accordingly.

The trusted computing base of a distributed checker consists of the following: Coq's proof checker and extraction mechanism – both proven on paper, the Ocaml compiler – widely used, Verdi's Shim and the underlying operating system.

4 Related Work

Literature offers numerous certifying algorithms [1–4, 6, 9–13, 15, 18–20, 22–24, 27]. A theory of certifying algorithms and further reading is given in [19]. A formal instance verification method is discussed in [26]. All this work is on *sequential* algorithms. To the best of our knowledge, there is little research on certifying *distributed* algorithms. A certifying variant of a routing algorithm was presented in [30], a discussion on how to distribute a witness predicate over a network in [28], and a method for formal instance verification in [29]. Moreover, CDAs share similarities to self-stabilizing algorithms [7], proof labeling schemes [16], and decentralized runtime verification [21].

In this paper, we built up on previous work [28, 29]. We integrated the idea of a consistent witness, and focused on distributed checking of consistency and the witness predicate in contrast to [28]. Moreover, we discussed proof obligations that have to be proven only once for the outlined class of CDAs while in [29] one particular case study is discussed. Moreover, we integrated Verdi for verification of a *distributed* program. As a consequence, the verified distributed checker runs on a real network in contrast to [29].

5 Discussion

We considered CDAs which verify an input-output pair at runtime. There are many open questions on how to apply the concept of certification to distributed algorithms. We focused on the *distributed checking* of a *distributed witness*. In order to form a valid correctness argument a distributed witness has to be *consistent*. By a restriction to *connected* witnesses, consistency can be checked in the neighborhoods (Theorem 1). We presented a method for formal instance verification, and implemented this method in a framework for Coq. Moreover, we discussed a verified implementation of the consistency check as an example of a task of the distributed checker. We showed how to deploy the verified distributed checker on a real network.

In the discussed framework, some proof obligations require manual work for each CDA (Sect. 3). For the proof obligations PO II(i) and PO II(ii) we have to find a proof. Automatic theorem provers can help to partly automate this undecidable task. However, using different tools creates an overhead: We have to formalize a proof obligation for different tools, and to show that the different formalizations are equivalent. Moreover, the tools add up on the trust base (c.f [26]). For the proof obligation PO IV(iv) we have to verify the correctness of the checkers task to decide the sub-predicates. By restricting to simple sub-predicates, i.e. sub-predicates that can be expressed as a propositional logic formula, we could use a verified program that gets a sub-predicate and generates a decision procedure correct by construction. By implementing and verifying such a program in Coq, we could easily integrate it to the presented framework.

We focused on terminating distributed algorithms. However, some distributed algorithms are intended to run continuously such as communication protocols. On a synchronous network, each round could additionally consist of a checking phase. By restricting to a universally distributable witness predicate, a sub-checker can raise an alarm if a sub-predicate does not hold. If not restricting to universally distributable witness predicates, the overhead of the evaluation of the witness predicate could be reduced by evaluating each k rounds. As a consequence, a bug would be discovered with a possible delay.

We focused on networks. However, for shared memory systems, the consistency of a distributed witness could be guaranteed by sharing the according variables between neighbors. The witness still has to be connected however. An alternative is to have sub-checkers that act like an interface of its component. That way, a sub-checker could check whether all messages sent are consistent with the internal state of its component, that its component does not corrupt messages when forwarding them, and that its component reads out a message properly. By that, the computed witness would be consistent. However, an overhead would be created during the computation.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: Verification of certifying computations. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 67–82. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_7
2. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. J. Autom. Reason. **52**(3), 241–273 (2014)

3. Arkoudas, K., Rinard, M.C.: Deductive runtime certification. Electron. Notes Theor. Comput. Sci. **113**, 45–63 (2005)
4. Bruce, D., Hoàng, C.T., Sawada, J.: A certifying algorithm for 3-colorability of P^5-free graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 594–604. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_61
5. Censor-Hillel, K., Fischer, E., Schwartzman, G., Vasudev, Y.: Fast distributed algorithms for testing graph properties. In: Gavoille, C., Ilcinkas, D. (eds.) DISC 2016. LNCS, vol. 9888, pp. 43–56. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53426-7_4
6. Corneil, D.G., Dalton, B., Habib, M.: LDFS-based certifying algorithm for the minimum path cover problem on cocomparability graphs. SIAM J. Comput. **42**(3), 792–807 (2013)
7. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
8. Duprat, J.: A Coq toolkit for graph theory. Rapport de recherche **15** (2001)
9. Finkler, U., Mehlhorn, K.: Checking priority queues. In: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999, pp. 901–902. Society for Industrial and Applied Mathematics, Philadelphia (1999)
10. Glesner, S.: Program checking with certificates: separating correctness-critical code. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 758–777. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_41
11. Heggernes, P., Kratsch, D.: Linear-time certifying recognition algorithms and forbidden induced subgraphs. Nord. J. Comput. **14**(1), 87–108 (2007)
12. Hell, P., Huang, J.: Certifying LexBFS recognition algorithms for proper interval graphs and proper interval bigraphs. SIAM J. Disc. Math. **18**, 554–570 (2004)
13. Hung, R.W., Chang, M.S.: An efficient certifying algorithm for the Hamiltonian cycle problem on circular-arc graphs. Theoret. Comput. Sci. **412**(39), 5351–5373 (2011)
14. INRIA: The Coq Proof Assistant. http://coq.inria.fr/
15. Kaplan, H., Nussbaum, Y.: Certifying algorithms for recognizing proper circular-arc graphs and unit circular-arc graphs. Disc. Appl. Math. **157**(15), 3216–3230 (2009)
16. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distrib. Comput. **22**(4), 215–233 (2010)
17. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
18. McConnell, R.M.: A certifying algorithm for the consecutive-ones property. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, pp. 768–777. Society for Industrial and Applied Mathematics, Philadelphia (2004)
19. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**, 119–161 (2011)
20. Mehlhorn, K., Näher, S.: From algorithms to working programs: on the use of program checking in LEDA. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 84–93. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055759
21. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, pp. 494–503. IEEE Computer Society, Washington, DC (2015)

22. Necula, G.C., Lee, P.: The Design and implementation of a certifying compiler. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, pp. 333–344. ACM, New York (1998)
23. Nikolopoulos, S.D., Palios, L.: An O(nm)-time certifying algorithm for recognizing HHD-free graphs. Theor. Comput. Sci. **452**, 117–131 (2012)
24. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through autocorres and simpl. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 46–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_4
25. Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38123-2
26. Rizkallah, C.: Verification of program computations. Ph.D. thesis (2015)
27. Schmidt, J.M.: Construction sequences and certifying 3-connectivity. Algorithmica **62**, 192–208 (2012)
28. Völlinger, K.: Verifying the output of a distributed algorithm using certification. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 424–430. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_29
29. Völlinger, K., Akili, S.: Verifying a class of certifying distributed programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 373–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_27
30. Völlinger, K., Reisig, W.: Certification of distributed algorithms solving problems with optimal substructure. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 190–195. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_14
31. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: ACM SIGPLAN Notices, vol. 50, pp. 357–368. ACM (2015)