Cardinality Abstraction for Declarative Networking Applications

Juan A. Navarro Pérez, Andrey Rybalchenko, and Atul Singh

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. Declarative Networking is a recent, viable approach to make distributed programming easier, which is becoming increasingly popular in systems and networking community. It offers the programmer a declarative, rule-based language, called P2, for writing distributed applications in an abstract, yet expressive way. This approach, however, imposes new challenges on analysis and verification methods when they are applied to P2 programs. Reasoning about P2 computations is beyond the scope of existing tools since it requires handling of program states defined in terms of collections of relations, which store the application data, together with multisets of tuples, which represent communication events in-flight. In this paper, we propose a cardinality abstraction technique that can be used to analyze and verify P2 programs. It keeps track of the size of relations (together with projections thereof) and multisets defining P2 states, and provides an appropriate treatment of declarative operations, e.g., indexing, unification, variable binding, and negation. Our cardinality abstraction-based verifier successfully proves critical safety properties of a P2 implementation of the Byzantine fault tolerance protocol Zyzzyva, which is a representative and complex declarative networking application.

1 Introduction

Declarative networking is a recent approach to the programming of distributed applications [29]. This approach allows the programmer to focus on a high-level description of the distributed system from the point of view of the global system, while the underlying runtime environment is responsible for the automatic distribution of computation and communication among nodes participating in the system. Declarative networking is increasingly employed by the distributed systems and networking community. It has been successfully applied to implement several network protocols, including sensor networks, Byzantine fault tolerance, and distributed hash tables, see [11,25,30,39], and is a subject of study from the compilation, debugging, and protocol design perspectives, see [14, 39, 40]. Complementing these lines of research, this work proposes a reasoning technique for declarative networking.

An implementation of the declarative networking approach is given by P2, a rule-based programming language [15]. The language is derived from distributed Datalog. Formally, P2 programs describe parameterized systems whose nodes

execute the same set of rules [20]. Application data of P2 programs is stored at network nodes in form of relations (i.e., sets of named tuples organized in tables) that satisfy programmer-defined projection constraints. These constraints require that the projection of each table on a specified subset of its columns does not contain duplicate elements. The P2 runtime environment enforces these constraints by pruning existing table rows whenever addition of new tuples leads to a violation. Nodes communicate by exchanging events that are represented by named tuples. P2 treats events in-flight as a collection of multisets. An event received by a node triggers evaluation of the program rules at the node. For each rule, this corresponds to computing all solutions of the relational query represented by the rule body with respect to the stored relations. Each one of these solutions determines an action to be executed that, according to the rule head, can either modify the relations stored at the node or send a new event.

Declarative networking imposes new challenges on existing analysis and verification methods that cannot be directly applied to P2 programs. First, reasoning about P2 computations requires dealing with relations and multisets. In contrast, existing verification methods represent the program state as a single tuple—a valuation of program variables in scope. Second, declarative statements used in P2 carry out complex operations on relations, e.g., joins and projections on tables, as well as addition and deletion of tuples. Such artifacts are beyond the scope of the existing verification tools.

In this paper, we present a *cardinality abstraction* technique for the analysis and verification of declarative networking applications. Cardinality abstraction aims at discovering quantitative information about data-storing relations and multisets of events such as those manipulated during the execution of P2 programs. This information is expressed in terms of *cardinality measures* that count the number of stored facts as well as the size of projections of the corresponding tables on their columns. Cardinality abstraction yields an over-approximation of the one-step transition relation represented by a P2 program, thus providing a basis for its further analysis and verification.

Cardinality abstraction keeps track of cardinality measures under the application of declarative operations during program execution. We represent the effect of these operations using a set of equations over cardinality measures. Given a P2 program, the corresponding set of equations is derived automatically. The equations are precise, i.e., for every satisfying valuation of cardinality measures there is a corresponding pair of states in the program's transition relation. Cardinality abstraction naturally handles data values, stored in tables and delivered by events, to the following extent. First, it takes into account the binding of variables during event matching, which triggers rule evaluation for the respective values, and the propagation of values through the rule. Second, the cardinality abstraction is sensitive to a priori fixed data values, possibly symbolic, by a syntactic program transformation that partitions tables according to the appearance of such data values.

The analysis and verification of declarative networking applications is then performed by applying abstract interpretation [17] to the cardinality abstraction

```
data(token/1,    keys(1)).
data(neighbor/2, keys(1)).
event(release/1).
event(pass/1).

r1  del  token(X)  :-  release(X), token(X).
r2  snd  pass(Y)   :-  release(X), token(X), neighbor(X, Y).
r3  add  token(Y)  :-  pass(Y).
```

Fig. 1. Program TOKEN implementing a token passing protocol in P2.

of a P2 program. Since the abstracting equations provide a relational abstraction of the one-step transition relation given by a P2 program, cardinality abstraction can be used to verify temporal safety and liveness properties. The computation of the abstract semantics can be carried out using existing tools and techniques, e.g., ASTRÉE, BLAST, INTERPROC, SLAM, and TERMINATOR [4, 5, 6, 16, 27].

We implemented the cardinality analysis-based tool CARDAN for the verification of safety properties of P2 programs[1]. CARDAN uses the ARMC model checker for the computation of the abstract semantics.[2] We applied CARDAN to automatically prove crucial safety properties of a P2 implementation of the Zyzzywa protocol for Byzantine fault tolerance [39], which is a representative, complex declarative networking application.

In summary, our contribution is a cardinality abstraction technique that enables analysis and verification of declarative networking applications. It relies on a quantitative abstraction of complex program states consisting of relations and multisets and manipulated using declarative statements. Our experimental evaluation using the CARDAN tool indicates that cardinality abstraction can be successfully applied for the automatic verification of declarative networking applications.

2 Example

In this section, we briefly describe P2 and illustrate cardinality abstraction on a simple program, TOKEN in Figure 1, that implements a token passing protocol.

States The first four lines in the figure define the structure of TOKEN states. Each node maintains two tables `token` and `neighbor` that keep track of the token ownership and the network connectivity between nodes, respectively. Communication events in the multisets `release` and `pass` initiate the token transfer and convey the act of passing between the nodes. Distribution in P2 is achieved by keeping the address of a tuple, where it should be stored or sent to, as the value in its first argument. The `keys` declarations in Figure 1 require that the projection of `neighbor` on its first column does not have duplicate elements, i.e., each node has at most one neighbor. In Figure 2 we show an example state s_0

[1] Tool and examples available at: `http://www.mpi-sws.org/~jnavarro/tools`

[2] The choice is due to implementation convenience.

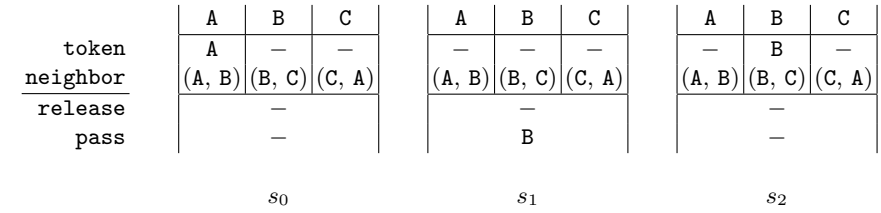

Fig. 2. Sequence of TOKEN states s_0, s_1, and s_2 obtained by applying rules `r1`, `r2`, and `r3` on the state s_0. Tables `token` and `neighbor` define the stored data, `release` and `pass` refer to the events in-flight. "–" denotes an empty table.

of TOKEN, in a network with three nodes `A`, `B` and `C`, under an assumption that there are no events in-flight. We use the symbol "–" to denote an empty table.

Rules Program statements in P2 are represented by rules consisting of a head and a body separated by the symbol "`:-`". The rule head specifies the name of the rule, its action and the tuple on which the action is applied. TOKEN uses rules `r1` and `r3` to delete and add tuples from the table `token`, as specified by the keywords `del` and `add`. The rule `r2` sends events in the form of `pass` tuples.

The body of each rule provides values for the variables appearing in the rule head. The evaluation is triggered by events arriving at a node. Assume that an event `release` is periodically broadcasted to all nodes from some external source whose nature is not important for this example, i.e., the nodes `A`, `B`, and `C` receive the tuples `release(A)`, `release(B)`, and `release(C)`, respectively. Then, the runtime environment of each node triggers evaluation of rules whose first conjunct in the body matches with the event. Triggering consumes the corresponding event, i.e., the multiset of events in-flight becomes empty. At node `A` the rules `r1` and `r2` are triggered by `release(A)`, but the rule `r3` is not. The same rules are triggered at nodes `B` and `C`. For each triggered rule the set of all solutions to the rule body is computed. This step is similar to the bottom-up evaluation procedure of Datalog. Given the state s_0, the evaluation of `r1` at `A` produces an action `del token(A)` by setting `X` = `A`, and for `r2` we obtain `snd pass(B)` due to the presence of `neighbor(A, B)` which sets `Y` = `B`. Only a single `snd` action can be produced by TOKEN at `A` because of the projection constraint on `neighbor`. In fact, if TOKEN had the declaration `data(neighbor/2, keys(1,2))` then each node could have multiple neighbors, and executing `r2` would result in the event `pass` to be delivered at each of them. At nodes `B` and `C` no actions are produced since the rule evaluation fails due to the lack of `token` tuples.

The execution of rules produces actions to manipulate data and send events. After executing `del token(A)` we obtain a state s_1 shown in Figure 2. The runtime environment sends the event `pass(B)` to the node `B`. Upon arrival, this event triggers the execution of `r3` at the node `B`, which consumes the event and adds the tuple `token(B)` to its tables. The resulting state s_2 is shown in Figure 2.

Property If correct, the TOKEN program should satisfy the property of mutual exclusion when executed on a network with an arbitrary number of nodes. The

property states that, at any given moment in time, at most one node can hold the token; under the assumption that the initial state of TOKEN already satisfies this condition and does not have any events in-flight.

This property relies on the invariant that the number of `token` and `pass` tuples together does not exceed one. This invariant is maintained in TOKEN through an interplay between the rules and the projection constraint on `neighbor`. Whenever the token leaves a node then only one of its neighbors will receive a `pass` tuple and will obtain the token.

Cardinality abstraction Checking the validity of the mutual exclusion property for TOKEN requires reasoning about its set of reachable states. Instead of dealing with program states in full detail, we only consider their quantitative properties by applying cardinality abstraction.

We use cardinality measures to keep track of the size of tables and their projections on subsets of columns. For example, the measure $\texttt{\#neighbor}_{\texttt{1,2}}$, where the subscript "`1,2`" refers to its first and second column, represents the number of tuples in the table; whereas $\texttt{\#neighbor}_{\texttt{1}}$ represents the size of the table projected on its first column. The measures `#pass` refers to the number of `pass` events in-flight. This measure does not refer to any projection, since P2 treats events in-flight as multisets and multiset projection does not affect cardinality.

Cardinality abstraction over-approximates the semantics of rule execution in form of a binary relation over cardinality measures. For example, the execution of rules `r1` and `r2` triggered by an event `release` produces the following modification of the measures $\texttt{\#token}_1$ and `#pass`, expressed in terms of a cardinality operator $|\cdot|$ and primed notation to denote measures after rule execution:

$$\begin{aligned} \texttt{\#token}_1' &= \texttt{\#token}_1 - |\{\texttt{X} \mid \texttt{release(X)} \wedge \texttt{token(X)}\}| \,, \\ \texttt{\#pass}' &= \texttt{\#pass} + |\{\texttt{Y} \mid \texttt{release(X)} \wedge \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| \,. \end{aligned}$$

The cardinality expressions in the above equations are further constrained by applying algebraic properties of relational queries, the semantics of projection constraints, and the definition of measures, which are marked by (a), (b), and (c) respectively.

$$\begin{aligned} |\{\texttt{X} \mid \texttt{release(X)} \wedge \texttt{token(X)}\}| &\leq |\{\texttt{X} \mid \texttt{release(X)}\}| && (a) \\ |\{\texttt{Y} \mid \texttt{release(X)} \wedge \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| &\leq |\{\texttt{Y} \mid \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| && (a) \\ |\{\texttt{Y} \mid \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| &\leq |\{\texttt{X} \mid \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| && (b) \\ |\{\texttt{X} \mid \texttt{token(X)} \wedge \texttt{neighbor(X,Y)}\}| &\leq |\{\texttt{X} \mid \texttt{token(X)}\}| && (a) \\ |\{\texttt{X} \mid \texttt{token(X)}\}| &= \texttt{\#token}_1 && (c) \end{aligned}$$

Additionally, we use the fact that only one event is consumed at a time, i.e.,

$$|\{\texttt{X} \mid \texttt{release(X)}\}| \leq 1 \,,$$

and that measures are always non-negative.

Cardinality abstraction-based verification To verify TOKEN, we compute the set of reachable valuations of cardinality measures and show that it implies the assertion $\texttt{\#token}_1 \leq 1$, stating that at most one node holds the token.

We apply a standard algorithm for assertion checking and obtain the following invariant that implies the property

$$\texttt{\#token}_1 \leq 1 \wedge \texttt{\#pass} + \texttt{\#token}_1 \leq 1 \ .$$

3 Preliminaries

In this section we briefly describe P2 programs [29] following the presentation in [34], which provides a better basis from the program analysis perspective.

Programs A P2 program $P_2 = \langle \mathcal{L}, \mathcal{D}, \mathcal{K}, \mathcal{R}, S_0 \rangle$ is defined by a set of predicate symbols $\mathcal{L}$, a data domain $\mathcal{D}$, a function $\mathcal{K}$ defining projection constraints, a set of rules $\mathcal{R}$, and an initial state S_0. For the rest of the exposition we assume that variables are elements from a set of variables $\mathcal{V}$. We write *vars*(W) to denote the set of variables occurring in an arbitrary expression W.

Given a predicate symbol $p \in \mathcal{L}$, with a positive arity n, a *predicate* is a term $p(u_1, \ldots, u_n)$, where each argument u_i is either a variable from $\mathcal{V}$, or a data element from $\mathcal{D}$. Variable-free predicates are called *tuples*. The first position in a predicate has a special role in P2 and is called *address*. The set of predicate symbols is partitioned into data and event symbols, which induces a partitioning of corresponding predicates and tuples.

The function $\mathcal{K}$ assigns to each data predicate symbol p of arity n a subset of its positions that includes the first one, i.e., $1 \in \mathcal{K}(p)$ and $\mathcal{K}(p) \subseteq \{1, \ldots, n\}$. Given a data tuple $P = p(v_1, \ldots, v_n)$, the projection operator $P{\downarrow_{\mathcal{K}}}$ computes a sub-tuple obtained from P by removing all of its arguments whose positions do not appear in the set $\mathcal{K}(p)$. For example, given $P = p(a, b, c, d)$ and $\mathcal{K}(p) = \{1, 3\}$, we obtain $P{\downarrow_{\mathcal{K}}} = p(a, c)$.

P2 uses *rules* of the form

$$r \ a \ H \ \texttt{:-} \ T, \ B$$

that consist of a head and a body separated by the symbol "`:-`". The head specifies the rule name r, determines the action kind $a \in \{\texttt{add}, \texttt{del}, \texttt{snd}\}$, and the action predicate H. For rules with the action kind `snd`, P2 requires H to be an event predicate, otherwise it must be a data predicate. The body consists of an event predicate T, called *trigger*, and a sequence of data predicates $B = B_1, \ldots, B_n$, called *query*. Each variable in the head must also appear in the body. We assume that all predicates in the body have the same address, i.e., they share the variable in the first position, and for each event name there is one rule in $\mathcal{R}$ with the equally named trigger predicate.[3]

Computations The *state* of a P2 program $\langle \mathcal{M}, \mathcal{E} \rangle$ consists of a *data store* $\mathcal{M}$ and an *event queue* $\mathcal{E}$. The store $\mathcal{M}$ is a set of data tuples that satisfies the projection constraints given by $\mathcal{K}$. The queue $\mathcal{E}$ is a multiset of event tuples.

[3] These assumptions are not proper restrictions and can be removed at the expense of a more elaborate exposition of the proposed technique.

```
    procedure EVALUATE
    input
      P_2 = ⟨L, D, K, R, S_0⟩: P2 program
    vars
      ⟨M, E⟩: program state
      E: selected event tuple
      Δ: derived tuples
    begin
 1:   ⟨M, E⟩ := S_0
 2:   while E ≠ ∅ do
 3:     E := take from E
 4:     find r a H :- T, B ∈ R such that T unifies with E
 5:     Δ := {Hσ | σ: V → D and E = Tσ and M ⊨ Bσ}
 6:     case a of
 7:       snd: E := E \ {E} ∪ Δ
 8:       add:
 9:         Δ^K := {D↓_K | D ∈ Δ}
10:         M := {D | D ∈ M and D↓_K ∉ Δ^K} ∪ Δ
11:       del: M := M \ Δ
12:     end case
13:   done
    end.
```

Fig. 3. Operational semantics for P2.

Figure 3 shows the procedure EVALUATE that defines the operational semantics of P2 programs. One iteration of the main loop, lines 2–12, defines the binary *transition relation* on program states, as represented by the P2 program. The state is given by the valuation of variables $\langle \mathcal{M}, \mathcal{E} \rangle$ in EVALUATE. Each transition starts selecting and removing an event tuple E from the event queue $\mathcal{E}$. Then, we select the rule with the matching trigger and compute all solutions to its query. The resulting (multi)set Δ of tuples is further processed according to the action kind a. If the rule is of kind `add`, and to guarantee the satisfaction of projection constraints, conflicting tuples are deleted from $\mathcal{M}$ before adding the new ones.

4 Cardinality abstraction

This section presents the cardinality abstraction technique.

Cardinality measures *Cardinality measures* are expressions of the form $\#F_X$ and $\#E$, where F is a conjunction of predicates, X is a set of variables, and E is an event predicate. Sometimes we use an arbitrary expression W instead of X. In this case, we assume $X = \mathit{vars}(W)$. Given a substitution function $\sigma\colon \mathcal{V} \to \mathcal{V} \cup \mathcal{D}$ and a set of variables X, we write $\sigma{\downarrow_X}$ to denote the restriction of σ wrt. X, i.e., the function such that $x\sigma{\downarrow_X} = x\sigma$ if $x \in X$, and $x\sigma{\downarrow_X} = x$ if $x \notin X$. Note that $\sigma{\downarrow_\emptyset}$ is the identity function.

Given a program state $\mathcal{S} = \langle \mathcal{M}, \mathcal{E} \rangle$, the cardinality measures $\#F_X$ wrt. $\mathcal{S}$ counts the number of valuations for X that are determined by the solutions of

F wrt. $\mathcal{M}$. Similarly, $\#E$ counts the number of events in $\mathcal{E}$ that unify with the event predicate E. Formally, we have

$$[\![\#F_X]\!]_{\mathcal{S}} = \ |\{\sigma{\downarrow_X} \mid \sigma\colon \mathcal{V} \to \mathcal{D}, \mathcal{M} \models F\sigma\}| \ ,$$
$$[\![\#E]\!]_{\mathcal{S}} = \ \textstyle\sum \{\mathcal{E}(E\sigma) \mid \sigma\colon \mathit{vars}(E) \to \mathcal{D}\} \ .$$

For an expression Φ over cardinality measures, we write $[\![\Phi]\!]_{\mathcal{S}}$ to denote the expression where cardinality measures are replaced by their evaluation wrt. $\mathcal{S}$.

The measure $\#F_\emptyset$ evaluates to one if the query F has at least one solution since all solutions are equal to the identity. If F does not have any solutions then $\#F_\emptyset$ evaluates to zero. We assume that variables in X that do not appear in F can be assigned arbitrary values from $\mathcal{D}$. Formally, in case X contains a set of variables X^- that do not occur in F, say $X = X^+ \uplus X^-$, we obtain

$$[\![\#F_X]\!]_{\mathcal{S}} = [\![\#F_{X^+}]\!]_{\mathcal{S}} \times |\mathcal{D}|^{|X^-|} \ .$$

Example 1 (Measure evaluation). We consider a case where $\mathcal{L} = \{p/2, q/2\}$, $\mathcal{D} = \{a, b, c, d, e, f\}$, $\mathcal{K}(p) = \{1, 2\}$, and $\mathcal{K}(q) = \{1\}$. Let $\mathcal{S} = \langle \mathcal{M}, \mathcal{E} \rangle$ be a state such that $\mathcal{M}$ is given by the tables p and q below and $\mathcal{E} = \emptyset$. We present examples of cardinality measures and their evaluation wrt. $\mathcal{S}$ below. □

p	q	F	X	$[\![\#F_X]\!]_{\mathcal{S}}$	F	X	$[\![\#F_X]\!]_{\mathcal{S}}$
a, b	a, b	$p(x, y)$	$\{x, y\}$	5	$p(x, y) \wedge q(y, z)$	$\{x, y, z\}$	4
a, c	b, c	$p(x, y)$	$\{x\}$	4	$p(x, y) \wedge q(y, z)$	$\{y, z\}$	2
b, b	d, a	$p(x, y)$	$\{y\}$	3	$p(x, y) \wedge q(y, z)$	$\emptyset$	1
c, a	e, a	$p(x, y)$	$\{y, z\}$	18	$p(x, y) \wedge q(y, x)$	$\emptyset$	0
d, b	f, a	$p(a, x)$	$\{x\}$	2	$p(x, y) \wedge p(y, x)$	$\{x, y\}$	3

4.1 Computing the cardinality abstraction

Figure 4 shows the CARDAN algorithm that together with a function STRUCTCLOSURE computes the cardinality abstraction of P2 programs. The algorithm generates for each event, and corresponding triggered rule, a constraint that describes the state transition in terms of cardinality measures. Each constraint represents a relation between measures evaluated before and after the rule is executed. The primed cardinality measures $\#F'_X$ and $\#E'$ represent the next state of the program, and the evaluation $[\![\Phi]\!]_{\mathcal{S},\mathcal{S}'}$ of an expression Φ is obtained by evaluating current and next state measures on the states $\mathcal{S}$ and $\mathcal{S}'$, respectively.

These constraints take into account how the measures on all tables (and their projections), as well as on all events, are updated according to the kind of rule executed (`add`, `del`, or `snd`) and the consumption of the selected event. A *structural closure*, computed by STRUCTCLOSURE, provides a set of constraints satisfied by the measures produced by the main algorithm. It constrains the values of measures for complex queries in terms of measures for their components. The execution of STRUCTCLOSURE terminates since at each iteration only cardinality measures with fewer predicates or fewer variables are introduced. Constraints

procedure CARDAN
input
$P_2 = \langle \mathcal{L}, \mathcal{D}, \mathcal{K}, \mathcal{R}, S_0 \rangle$: a P2 program
vars
Ψ, Φ: constraints over cardinality measures
begin
1: $\Psi := \bot$
2: **for each** event $S = s(w_1, \ldots, w_m)$ **do**
3: $\Phi := \top$
4: **let** r a $H :- T, B \in \mathcal{R}$ **such that** T and S unify
5: **for each** data predicate $P = p(v_1, \ldots, v_n)$ and set $V \subseteq vars(P)$ **do**
6: **if** there is a substitution σ **such that** $H = P\sigma$ **then**
7: **if** $a = \texttt{add}$ **then**
8: $\Phi := \Phi \wedge (\#P'_V = \#P_V + \#B_{V\sigma} - \#(B \wedge (P\sigma{\downarrow_V}))_{V\sigma})$
9: **else if** $a = \texttt{del}$ **then**
10: **if** $V = \{v_1, \ldots, v_n\}$ **then**
11: $\Phi := \Phi \wedge (\#P'_V = \#P_V - \#(B \wedge H)_{V\sigma})$
12: **else**
13: $\Phi := \Phi \wedge (\#P_V - \#(B \wedge H)_{V\sigma} \leq \#P'_V \leq \#P_V)$
14: **else**
15: $\Phi := \Phi \wedge (\#P'_V = \#P_V)$
16: **for each** event predicate $E = e(v_1, \ldots, v_n)$ **do**
17: $\Delta := 0$
18: **if** E and H unify **then**
19: $\Delta := \#B_H$
20: **if** E and T unify **then**
21: $\Delta := \Delta - 1$
22: $\Phi := \Phi \wedge (\#S' = \#S + \Delta)$
23: $\Psi := \Psi \vee$ STRUCTCLOSURE(Φ)
24: **end**
25: **return** Ψ
end.

function STRUCTCLOSURE
input
Φ: constraints over cardinality measures
begin
1: **do**
2: $\Phi := \Phi \wedge \bigwedge \{0 \leq \#F_\emptyset \leq 1 \mid \#F_X$ occurs in $\Phi\}$
3: $\Phi := \Phi \wedge \bigwedge \{\#F_X \leq \#F_{X \cup Y} \leq \#F_X \times \#F_Y \mid \#F_{X \cup Y}$ occurs in $\Phi\}$
4: $\Phi := \Phi \wedge \bigwedge \{\#(F \wedge G)_X \leq \#F_X \mid \#(F \wedge G)_X$ occurs in $\Phi\}$
5: $\Phi := \Phi \wedge \bigwedge \{\#P_X = \#P_{P{\downarrow_\mathcal{K}}} \mid \#P_X$ occurs in Φ and $vars(P{\downarrow_\mathcal{K}}) \subseteq X \subseteq vars(P)\}$
6: **for each** data predicate $P = p(v_1, \ldots, v_n)$ and set $V \subseteq vars(P)$ **do**
7: $\Phi := \Phi \wedge \bigwedge \{P\sigma_X \leq \#P_V \mid \#P\sigma_X$ occurs in Φ, $|V| = |X|$, and $X = V\sigma\}$
8: **while** Φ is updated
9: **return** Φ
end.

Fig. 4. CARDAN algorithm for computing cardinality abstraction.

computed by CARDAN are in the worst case exponentially larger than the input program. Section 4.2 presents optimizations that address this explosion in order to achieve a practical analysis.

Example 2 (Measure update). We show the constraints that CARDAN creates for an event $s(w_1)$ that triggers a rule

$$r \texttt{ add } p(x,y) :- s(x),\ B(x,y,z)$$

where $B(x,y,z)$ is some query over variables x, y, and z. For $P = p(v_1,v_2)$ and $V = \{v_1\}$, line 8 of CARDAN creates the constraint

$$\#p(v_1,v_2)'_{\{v_1\}} = \#p(v_1,v_2)_{\{v_1\}} + \#B(x,y,z)_{\{x\}} - \#(B(x,y,z) \wedge p(x,v_2))_{\{x\}} \ .$$

In this case, the substitution σ that unifies $p(v_1,v_2)$ with the head of the rule is given by $\sigma = \{v_1 \to x, v_2 \to y\}$ and its restriction is $\sigma{\downarrow_{\{v_1\}}} = \{v_1 \to x\}$.

This expression describes the change in the number of values in the p table—after executing the rule and projecting wrt. its first column—by adding first the number of solutions of the query projected on the variable x, and then subtracting the number of values that were already in the table. This is the role of the last term, which asks for those values of x that appear both as a solution to the query and as the first component in some tuple currently in the table. The second component of such tuple is free to contain any arbitrary value.

For other predicates, line 15 creates constraints encoding the frame conditions, e.g., for predicate $q(v_1,v_2)$ we obtain $\#q(v_1,v_2)'_V = \#q(v_1,v_2)_V$. □

Example 3 (Structural closure). We illustrate the STRUCTCLOSURE function on expressions from Example 1. The set of computed constraints includes

$$0 \leq \#(p(x,y) \wedge q(y,x))_{\emptyset} \leq 1 \ , \tag{1}$$

$$\#p(x,y)_{\{x\}} \leq \#p(x,y)_{\{x,y\}} \leq \#p(x,y)_{\{x\}} \times \#p(x,y)_{\{y\}} \ , \tag{2}$$

$$\#(p(x,y) \wedge (y,z))_{\{y,z\}} \leq \#p(x,y)_{\{y,z\}} \ , \tag{3}$$

$$\#q(x,y)_{\{x,y\}} = \#q(x,y)_{\{x\}} \ , \tag{4}$$

$$\#p(a,x)_{\{x\}} \leq \#p(v_1,v_2)_{\{v_2\}} \ . \tag{5}$$

These constraints correspond to algebraic properties satisfied by cardinality measures (1-3), relations imposed by projection constraints (4), and relations between arbitrary single-predicate queries and table projections (5). One can check that all (1-5) are valid for the state $\mathcal{S}$ presented earlier in Example 1. □

Correctness of Cardan The constraints generated by STRUCTCLOSURE are valid for all possible states, as formalized in the following theorem.

Theorem 1. *Given an arithmetic constraint Φ over cardinality measures and a state $\mathcal{S}$, the constraint $[\![\Phi]\!]_{\mathcal{S}}$ holds if and only if $[\![\text{STRUCTCLOSURE}(\Phi)]\!]_{\mathcal{S}}$ does.*

Although we omit a proof for brevity, it is not hard to verify that all the constraints generated by CARDAN are valid. Moreover, as a direct consequence from the previous result, soundness of the approach follows.

Theorem 2. *Given a P2 program P_2, and a pair of states $\mathcal{S}$, $\mathcal{S}'$ related by the transition relation given by* EVALUATE*, the constraint* $[\![\text{CARDAN}(P_2)]\!]_{\mathcal{S},\mathcal{S}'}$ *holds.*

By Theorem 1, the STRUCTCLOSURE function gives a sound and relatively complete abstraction (modulo data values) of relations in terms of cardinality measures. Moreover, Skolem symbols can be used to refine the abstraction by taking into account particular data values and, in that sense, obtain completeness for the overall approach, see Section 5.

4.2 Extensions and optimizations

The presentation in the previous sections was simplified with the assumption that, for each selected event, only one rule is executed. Programs in P2, including our TOKEN example from Section 2, often require the simultaneous execution of several rules in a single step. We automatically find sets of rules that have to be evaluated together (also propagating bindings of variables across rules), and prune combinations of rules that will never be evaluated concurrently. The algorithm presented earlier in Figure 4 can then to be modified to generate individual transitions not for each rule, but for each one of those groups of rules that perform atomic updates.

The implementation of many applications in P2, in particular the Zyzzyva protocol discussed in the next section, rely on rules that can also, as part of their evaluation, count the number of solutions of arbitrary queries on the store. To accommodate for this feature, the syntax of rules has to be extended to allow the use of *counting* operators on their bodies. However, since these counting operators can be expressed in terms of our cardinality measures, they don't impose any new challenges in the theory or implementation of the approach.

Finally, as an optimization to reduce the number of variables in queries, and therefore the number of constraints generated by the STRUCTCLOSURE function, we implement a *symbolic constant propagation* technique. This procedure simplifies rules by computing a set of variables that will be bound to at most one value, and replacing those variables with symbolic constants. This set of variables is initialized with those appearing in the trigger (since they have to exactly match the selected event), and then expanded to include more variables by propagating these symbolic constants through the projection constraints.

5 Experience

In this section we describe our experiences applying cardinality abstraction for the verification of the P2 implementation of the Zyzzyva protocol [39]. Zyzzyva is a state-of-the-art Byzantine fault tolerance (BFT) protocol designed to improve the reliability of client-server applications such as web services. In Zyzzyva, the

service state and request processing logic is replicated on $3F+1$ computer hosts, where F is a non-negative integer. Zyzzyva guarantees correct and consistent execution of the service even when at most F replica hosts can fail *arbitrarily*, e.g., due to software bugs or malicious behavior caused by a computer virus.

To be correct, Zyzzyva must assign a distinct sequence number to each client request. This safety property is amenable to cardinality analysis since, by counting the number of messages that are sent between replicas and clients, it is possible to identify how many requests have been assigned to a given sequence number. Specifically, the safety property is violated if the table `done`, which collects the responses accepted by clients, contains two tuples with a different client or request values but the same sequence number. Our approach can show that the safety property is valid under the assumption that at most F hosts are faulty among the totaol of $3F+1$ hosts. The BFT guarantees are not assumed, but rather derived from the basic assumption.

Since cardinality abstraction does not handle the data values stored in tuples (i.e., values of particular requests or sequence numbers), we represent this information by partitioning the original program tables with respect to a finite set of values of interest. For example, a table `reply(Replica, Request, SeqNo)` is partitioned to contain a table `reply_a_s(Replica)` whose elements correspond to tuples of the form `reply(Replica, a, s)`.

We distinguish the behavior of *correct* and *faulty* replicas. To model Byzantine failures, we simulated the *worst possible* faulty behavior for replicas that, for this property, corresponds to sending (without the corresponding checks) confirmation messages agreeing to two conflicting request assignments for the same sequence number. Correct replicas behave according to the protocol implementation. We apply Cardan on the resulting P2 program, which computes cardinality abstraction of the program. The model checking backend Armc analyzes the resulting transition relation over cardinality measures and proves the property in five seconds.

In the on-going work, we consider further examples of BFT protocols [10,28, 42], as well as other distributed applications.

6 Related work

Verification of distributed applications is a classical research topic. Recent efforts have been focused on the synthesis of quantified invariants [1, 3, 35] and counting abstraction [36] for parameterized, bounded-data systems. These techniques, however, are not directly applicable to P2 programs due to the complex program state and declarative representation of transition relations. Our approach, although closely related to counting abstraction, differs in that we count the number of solutions of complex relational queries, rather than the number of processes in one of finitely many possible states [36]. On the other hand, the network topology in P2 programs is abstracted away by a table of reachable neighbors. This eliminates the need to perform a network reachability analysis—one of the common difficulties of distributed protocol verification.

Program analysis has been extensively studied for Datalog and other forms of logic programming. Comprehensive abstract interpretation frameworks for Prolog exist, including [9, 18]. These frameworks are supported by state-of-the-art implementations, e.g., PLAI/CiaoPP [24] and Analyser [21]. These tools perform size analysis, cost analysis, determinacy analysis, non-failure analysis, termination analysis, and resource analysis. The cardinality analysis for Prolog, see e.g. [8], approximates number of solutions to a goal, but it does not handle indexing and bottom-up evaluation semantics, which we found crucial for declarative networking applications written in P2.

Existing approaches to the analysis of networking applications, see e.g. MaceMC [26] and CMC [32,33], focus on finding defects using symbolic executions techniques. While in theory they can be applied exhaustively to prove absence of defects, it is extremely difficult to achieve this in practice. In the context of declarative networking, early steps have been given by clarifying the semantics of P2 programs [34], and designing translations of program properties into formulas suitable for use in interactive theorem provers [41]. Our analysis complements these techniques by supporting automated proof discovery.

Abstraction of sets and relations for imperative programs focuses on dynamically allocated heap used to store graph structures of a particular shape, e.g., shape analysis [7, 38] and separation logic [37, 43]. [23] refines these approaches with information about the size of the allocated heap fragments. In contrast, declarative networking uses relations as a general purpose-data store without particular shape invariants and, unlike heap models, has to deal with database operations that manipulate tables. The result of cardinality abstraction can be analysed by existing tools and techniques for computing abstract semantics, including numerical abstract domains, e.g. [2, 19, 31], automatic abstraction refinement, invariant generation, and predicate abstraction [12, 13, 22].

References

1. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 2008.
3. I. Balaban, A. Pnueli, and L. D. Zuck. Invisible safety of distributed protocols. In *ICALP (2)*, 2006.
4. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *POPL*, 2002.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 2007.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
7. I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *CAV*, 2007.
8. C. Braem, B. L. Charlier, S. Modart, and P. van Hentenryck. Cardinality analysis of Prolog. In *ILPS*, 1994.

9. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Log. Program.*, 1991.
10. M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. 1999.
11. D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.
12. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
13. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.
14. T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 2008.
15. T. Condie, D. E. Gay, B. T. Loo, et al. P2: Declarative networking website, 2008. `http://p2.cs.berkeley.edu/`.
16. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV*, 2006.
17. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 1977.
18. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 1992.
19. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
20. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 1992.
21. F. Gobert. *Towards Putting Abstract Interpretation of Prolog Into Practice*. PhD thesis, Université catholique de Louvain, 2007.
22. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV*, 1997.
23. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, 2009.
24. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program development using abstract interpretation (and the Ciao system preprocessor). In *SAS*, 2003.
25. C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahadat. Mace: Language support for building distributed systems. In *PLDI*, 2007.
26. C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
27. G. Lalire, M. Argoud, and B. Jeannet. The interproc analyzer. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.
28. L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998.
29. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *SIGMOD*, 2006.
30. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SIGOPS*, 2005.
31. A. Miné. The octagon abstract domain. *Higher-Order and Symb. Comp.*, 2006.
32. M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI*, 2004.

33. M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, 2002.
34. J. A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In *PADL*, 2009.
35. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.
36. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, 2002.
37. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
38. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 2002.
39. A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *NSDI*, 2008.
40. A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *EuroSys*, Leuven, Belgium, 2006.
41. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative networking verification. In *PADL*, 2009.
42. T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap Practical BFT Using Virtualization. Technical Report TR14-08, University of Massachusetts, 2008.
43. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.