DIONE: A protocol verification system built with DAFNY for I/O Automata

Chiao Hsieh and Sayan Mitra

University of Illinois at Urbana-Champaign, Champaign IL 61820, USA
{chsieh16,mitras}@illinois.edu

Abstract. Input/Output Automata (IOA) is an expressive specification framework with built-in properties for compositional reasoning. It has been shown to be effective in specifying and analyzing distributed and networked systems. The available verification engines for IOA are based on interactive theorem provers such as Isabelle, Larch, PVS, and Coq, and are expressive but require heavy human interaction. Motivated by the advances in SMT solvers, in this work we explore a different expressivity-automation tradeoff for IOA. We present DIONE, the first IOA analysis system built with DAFNY and its SMT-powered toolchain and demonstrate its effectiveness on four distributed applications. Our translator tool converts Python-esque DIONE language specification of IOA and their properties to parameterized DAFNY modules. DIONE automatically generates the relevant compatibility and composition lemmas for the IOA specifications, which can then be checked with Dafny on a per module-basis. We ensure that all resulting formulas are expressed mostly in fragments solvable by SMT solvers and hence enables Bounded Model Checking and k-induction-based invariant checking using Z3. We present successful applications of DIONE in verification of an asynchronous leader election algorithm, two self-stabilizing mutual exclusion algorithms, and CAN bus Arbitration. We automatically prove key invariants of all four protocols; for the last three this involves reasoning about arbitrary number of participants. These analyses are largely automatic with minimal manual inputs needed, and they demonstrate the effectiveness of this approach in analyzing networked and distributed systems.

1 Introduction

For modeling and verifying network and distributed systems, compositional approaches are considered essential in achieving modularity and scalability. A recent study on building verified industrial scale systems highlight the importance of compositional reasoning and connecting layers of abstractions, especially for integrating formal methods into software development practices [22]. The Input/Output Automata (IOA) [20] framework comes with an expressive modeling language [8], and a powerful set of simulation and substitutivity results that support reasoning about compositions, abstractions, and substitutions. The framework has been used to model and analyze a wide variety of distributed systems

ranging from high-level protocols for mutual exclusion, consensus, leader election [20], to implementation-level specifications for shared memory and group communication services [6,7,4], and communication protocols like TCP [25].

Currently available mechanized verification engines for IOA are mostly based on interactive theorem provers including Isabelle/HOL [21], Larch Shared Language [3], PVS [19], and recently CoqIOA [1]. All of these provide expressive specification languages based on higher order theories for modeling IOA, but they require a nontrivial amount of human interactions for constructing proofs[1].

In this paper, we aim to achieve higher automation for verification of distributed systems expressed in IOA, by restricting the specifications to less expressive theories with established decision procedures. One predominant option is First Order Theories supported by existing Satisfiability Modulo Theories (SMT) solvers. There has been a wave of advancements in SMT solvers, which in turn has led to the creation of widely diverse program verification engines like CBMC [5], Dafny [16], and SeaHorn [15]. We therefore believe that theories solvable with SMT solvers can be expressive enough for a broadly useful class of IOA, and an SMT-based IOA verification engine can advance the design and analysis of distributed systems.

Specifically, we build a toolchain based on Dafny [16], instead of directly interfacing with SMT solvers. Dafny provides a higher level of abstraction for developing both system specifications and for proof strategies [17]. The IronFleet project [11], for instance, demonstrates how practical consensus protocols can be modeled and verified in Dafny. The project further develops a Dafny implementation of the model with formal conformance guarantees, and ultimately generates correct-by-construction C# source code from the implementation. These results suggest that Dafny is a good candidate for supporting modeling and mostly-automatic verification for distributed protocols as IOA.

In this work, we propose Dione—a modeling language and verification system built using Dafny for IOA. The Dione workflow is similar to that of [3,19]. First, users model the protocol and specify the desired invariant in the Dione language. For example in Figure 1, a distributed algorithm from the textbook [9] can be faithfully modeled in Dione. Dione frontend then translates Dione to Dafny specification language. Additionally, Dione generates Dafny lemmas so that proving these lemmas is equivalent to Bounded Model Checking (BMC) and k-induction based invariant checking for the original IOA. There are three major contributions in this work:

(i) We have developed a Python-like language, Dione, for naturally specifying distributed systems as compositions of parameterized IOA. We have implemented the supporting tools for (a) translating Dione specifications to Dafny, and for (b) carrying out automatic proofs for Dione specifications. Our translator uses modules in Dafny to automatically generate (nontrivial) composition and compatibility checks for parameterized automata.
(ii) We have demonstrated that Dione can effectively model diverse distributed protocols: self-stabilizing mutual exclusion algorithms with different network

[1] Also, none of the tools appear to be maintained for at least two years.

```
# s[0] ∈ {1,3}, s[n-1] ∈ {2, 4}, and N(i) returns neighbors of i #
program StableArray;
{program for process i, i=0 or i=n-1}
do
∃j ∈ N(i): (s[j] = s[i]+1 mod 4) → s[i] := s[i]+2 mod 4
od
{program for process i, 0<i<n-1}
do
∃j ∈ N(i): (s[j] = s[i]+1 mod 4) → s[i] := s[j]
od
```

```
Status: type = IntRange[0:4]
@automaton
def StableArray(N: int):
   where= 2 <= N
   class signature:
      @output
      def trans(i: int): where=(0 <= i < N)
   class states:
      s: Seq[Status]
   initially= len(s)==N and(s[0]==1 or s[0]==3)and(s[N-1]==0 or s[N-1]==2)
   class transitions:
      @output
      @pre((i==0 and s[i+1]==incre(s[i]))or(i==N-1 and s[i-1]==incre(s[i])))
      def trans(i): s[i] = incre(incre(s[i]))
      @output
      @pre(0 < i < N-1 and s[i-1]==incre(s[i]))
      def trans(i): s[i] = s[i-1]
      @output
      @pre(0 < i < N-1 and s[i+1]==incre(s[i]))
      def trans(i): s[i] = s[i+1]
   invariant_of= len(s)==N and(s[0]==1 or s[0]==3)and(s[N-1]==0 or s[N-1]==2)
```

Fig. 1. Textbook description of a self-stabilizing mutual exclusion algorithm [9, Section 17.3.2] (top) and specification in DIONE (bottom).

topologies, leader election algorithms, and the CAN bus Arbitration. These showcase how protocols and requirements for an arbitrary number of processes can be modeled with parallel composition and collection types.

(iii) Finally, we have demonstrated that DIONE can achieve a promising level of proof automation. The ratio lengths of DIONE specifications to DAFNY models plus proofs is about 1:3 to 1:5. In the worst case only two out of five lemmas require user specified proof strategies. In particular, (a) for self-stabilization algorithms, our verifier is able to prove invariants for an arbitrary number of processes by k-induction with one or two manually added basic facts about a generic sequence. (b) For the leader election algorithm and CAN bus Arbitration, we show that auxiliary invariants or over-approximation of transitions can be easily integrated to fully automate verification, and over-approximation itself can be automatically validated by the verifier.

Related Work. In addition to the related works mentioned earlier, the translation by DIONE resembles the translation to Larch [3] as both are using First Order Theories. DIONE benefits from the advancement in SMT solving and program verification. With better understanding on the efficiency and decidability of SMT solving, DIONE is designed to achieve higher automation using DAFNY. In return,

it is difficult to specify user-defined sorts and theories not supported by Dafny in comparison with [3] and other works.

The design decision to translate IOA to Dafny is greatly influenced by the modeling techniques in IronFleet [11]. The objective of IronFleet project and our work however are very different. IronFleet focused on connecting layers of abstractions from specification to implementation. Their proofs of refinement based on Temporal Logic of Actions (TLA) were manually written. In comparison, our work aims to achieve higher automation on invariant checking for IOA and provides an IOA language with Dafny translation to reduce user effort.

More broadly, Tuttle and Goel [26] demonstrated how to use the SMT-based checker, DVF, to verify a consensus protocol. Ivy [23] recently explored using EPR, a decidable fragment of FOL, to model and verify infinite state transition systems. However, the DVF model for the protocol explicitly described the global state transition system instead of the local state of a participant of the protocol, and Ivy defined the execution of a protocol as one RML program within only one while loop; it is likely cumbersome to cleanly model parallel composition in both DVF and Ivy languages. Moreover, it is unclear how to specify which automaton has control over certain actions. On the other hand, our IOA-based language provides a language construct to explicitly specify components of a composition as well as **input** and **output** keywords to denote the controlling automaton of actions. For our purpose of verifying IOA, we believe DVF and Ivy can be used as alternative back-ends for Dione with proper translations.

Lastly, our work complements existing works in ensuring correct implementation of distributed protocols. Verdi [27], Chapar [18], Project Everest [2], to name a few, verified programs in high level language and synthesized low level implementations. As reported in Verdi [27], the ratio length of verified programs and manual proofs is at least 1:3. Our proof generation techniques may be used to reduce manual effort in proving high level programs.

2 Background: Brief overview of IOA

Mathematical Notations. Let X be a finite set of *variable* names. To model variables with static types, we assume a function $type(x)$ to return the set of possible values for x. A *valuation*, s, is a mapping from $x \in X$ to a value $s(x) \in type(x)$, and $val(X)$ denotes the set of all possible valuations of X. For a valuation $s \in val(X)$ and a subset of variable names $Y \subseteq X$, we use $s \lceil Y$ to denote the restriction of s to Y.

Input/Output Automata. An IOA [20], $\mathcal{A} = (\Sigma, X, Q, \Theta, \delta)$, is a tuple where (i) $\Sigma = \Sigma^I \dot\cup \Sigma^O \dot\cup \Sigma^H$ is the set of all actions partitioned into Σ^I, Σ^O, and Σ^H representing **input**, **output**, and **internal** actions, respectively. (ii) X is a finite set of *state variable* names. (iii) $Q \subseteq val(X)$ is the set of states (iv) $\Theta \subseteq Q$ is the set of initial states. (v) $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. An important requirement for an IOA $\mathcal{A}$ is that it is *input enabled*. That is, it cannot block

input actions. Formally,

$$\forall s \in Q, a \in \Sigma^I, \exists s' \in Q, (s, a, s') \in \delta$$

A pair of IOA $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if and only if the following holds: (i) Output actions are disjoint, i.e., $\Sigma_1^O \cap \Sigma_2^O = \emptyset$; (ii) Internal actions are only used by itself, i.e., $\Sigma_1^H \cap \Sigma_2 = \Sigma_2^H \cap \Sigma_1 = \emptyset$; and (iii) Variable names are disjoint, i.e., $X_1 \cap X_2 = \emptyset$. This ensures that a composition of compatible automata (defined next) is also an IOA.

Given two compatible automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we can construct the parallel composition automaton $\mathcal{A}_1 \parallel \mathcal{A}_2 = (\Sigma, X, Q, \Theta, \delta)$ where

(i) $\Sigma^I = (\Sigma_1^I \cup \Sigma_2^I) \setminus (\Sigma_1^O \cup \Sigma_2^O)$; $\quad \Sigma^O = \Sigma_1^O \cup \Sigma_2^O$; $\quad \Sigma^H = \Sigma_1^H \cup \Sigma_2^H$
(ii) $X = X_1 \cup X_2$
(iii) $Q = \{s \in val(X) \mid s \lceil X_1 \in Q_1 \wedge s \lceil X_2 \in Q_2\}$
(iv) $\Theta = \{s \in Q \mid s \lceil X_1 \in \Theta_1 \wedge s \lceil X_2 \in \Theta_2\}$
(v) The transition δ is defined as follows with shorthand expressions $s_1 = s \lceil X_1$, $s_2 = s \lceil X_2$, $s_1' = s' \lceil X_1$, and $s_2' = s' \lceil X_2$:

$$\{(s, a, s') \in Q \times \Sigma \times Q \mid (a \in \Sigma_1 \Rightarrow (s_1, a, s_1') \in \delta_1) \wedge (a \notin \Sigma_1 \Rightarrow s_1 = s_1') \\ \wedge (a \in \Sigma_2 \Rightarrow (s_2, a, s_2') \in \delta_2) \wedge (a \notin \Sigma_2 \Rightarrow s_2 = s_2')\}$$

It can be checked that the composed automaton $\mathcal{A}_1 \parallel \mathcal{A}_2$ is indeed an IOA. The theory of IOA also defines executions, traces, invariance, abstractions, refinements, forward-backward simulation relations, and substitutivity properties. We refer the interested reader to [20,8] for more complete overview.

IOA with Parameters. To concisely model a distributed protocol, a standard practice is to use parameterized automaton specifications. Formal parameters may appear in names of automata, actions, and in all the predicate and transition definitions. This interdependence of parameters and variables make compatibility and composition nontrivial to implement.

For example, it is standard to assume that each participating process is distinguished by a unique address *ip*, and that messages are tagged with source and destination *ip*s. We can specify the address as a parameter *ip* of *Addr* type and define a family of automata $\{\mathcal{A}_{ip}\}_{ip \in Addr}$. The complete behavior of the protocol is then modeled by composing automata instantiated with distinct *ip*s. For specifying one **output** action sending a message, we can use a tuple $(src, dst, ...)$ to represent a tagged message. However, in order to satisfy compatibility criteria, the set of **output** actions Σ_{ip}^O for sending messages has to be constrained by *ip*, e.g., $\Sigma_{ip}^O = \{(src, dst, ...) | src, dst \in Addr \wedge src = ip\}$, or else the action sets for different automaton instances may intersect. Consequently, it is not obvious whether the compatibility criteria are satisfied when the constraints are more complex, or when a composite automaton is used in another composition. Hence, it is important to have an automated check of the compatibility criteria.

3 Overview of DIONE

In this section, we discuss the design and features of DIONE as well as the translation from DIONE language to DAFNY for model checking.

3.1 DIONE Language

To provide users a familiar modeling language for IOA while allowing DIONE to be extensible, DIONE language borrows the syntax from one of the most popular programming languages, PYTHON. This design decision allows us to potentially embed DIONE into normal PYTHON programs and use existing PYTHON packages such as Z3Py for verification. Further, our tools benefit from existing and advanced compiler/interpreter infrastructure of PYTHON, and in the future an interpreter for DIONE programs could be built on top of PYTHON.

The syntax of DIONE is a strict subset of PYTHON 3.7 syntax. We interpret DIONE code as defining an IOA as in [8]. At the top level of a DIONE program, one can only define either a *type* or an *automaton*.

Data Types. A type definition is simply an assignment with type hints. The left hand side is annotated with the built-in base meta-type, `type`, to indicate the identifier represents a type. On the right hand side, one can provide an existing type to create a type synonym, or use a built-in type constructor to build new types, for example `Enum` and `NamedTuple` in Figure 2. The list of built-in types for DIONE can be found in our project website [12].

Automata. An automaton definition uses a PYTHON function definition syntax decorated with either `@automaton` or `@composition` to define a primitive or a composite automaton. The function parameters naturally serve as the parameters of the automaton. One can optionally add an assignment to the `where` variable to bound the parameter values for the automaton. For a primitive automaton, four DIONE program constructs, namely `states`, `initially`, `signature`, and `transitions`, are required to specify state variables, initial states, the signature, and the transition relation as discussed in Section 2. For a composition, `components` is required to specify the component automata. For either primitive or composite automaton, one can optionally assign an `invariant_of` variable to specify the invariant.

States. The `states` class is to specify variables X and consists of only variables annotated with types as shown in Figure 2. The `initially` variable should be assigned with a Boolean expression over state variables and/or automaton parameter. An allowed initial state is when `initially` is evaluated to true.

Signature and Transitions. The `signature` class is for declaring the set of actions used in the automaton. It should contain only functions decorated with `@input`, `@output`, or `@internal`, to declare an action in Σ^I, Σ^O, or Σ^H. Similarly, an assignment to `where` can be added to bound the parameters of

```
Loc: type = nat
Addr: type = Enum[A, B, C]
Msg: type = NamedTuple[src: Addr,
                       dst: Addr,
                       val: int]
@automaton
def Proc(ip: Addr, b: int):
    where = b>=10
    class states:
        pc: Loc
        x: int
    initially = (pc==1 and 0<=x<b)
```

Fig. 2. DIONE types and states.

```
type Loc = nat
datatype Addr = A | B | C
datatype Msg = Msg(src: Addr,
                   dst: Addr,
                   val: int)
datatype Parameter = Parameter(
    ip: Addr, b: int)
datatype State = State(pc: Loc, x: int)
predicate aut_where(p: Parameter)
{  p.b ≥ 10  }
predicate initially(
    s: State, p: Parameter)
{  s.pc = 1 ∧ 0 ≤ s.x < p.b  }
```

Fig. 3. Translated DAFNY from Figure 2.

the action. The `where` clause is crucial for sharing a same action name across multiple automata. For instance, the **output** action `send` defined in Figure 4 is constrained with messages sent from its own `ip`. Without the constraint, there can be no compatible automata using `send` as an **output** action because their output sets of actions are not disjoint. With the constraint, other automata using `send` as an **output** action are still compatible if the source address is not `ip`. To define transition relation for the automaton in `transitions` class, we follow [8] to specify preconditions and effects for actions. As show in Figure 4, the action is decorated with `@pre` as the precondition, and the effect of this action is specified within the function body. Currently, DIONE allows only some commonly used PYTHON statements and expressions in the effect. An updated list of supported statements and expressions is available on our website [12].

Composition. The `components` class is required for specifying a composite automaton. It is a list of component names annotated by the automaton instantiated with actual parameter values. For Figure 6 as an example, the `Sys` automaton is composed of three components, `p1`, `p2`, and `env`. Both `p1` and `p2` are instances of automaton `Proc`, and `env` is an instance of `Env`.

3.2 Translating DIONE to DAFNY

In this section, we informally describe the translation of program constructs from DIONE to DAFNY in our implementation. In particular, we illustrate the insights into systematically translating from notions of sets, subsets, and transitions in Section 2 to data types and predicates expressible in DAFNY. We first describe translating all constructs for one automaton with parameters and then we provide the composition of automata with compatibility checks by DAFNY.

Data Types. Benefitting from the rich types available in DAFNY, a broad class of types are supported in DIONE. Primitive types such as `bool`, `int`, `str`, `nat`, `real`, and `bv`N (bit-vectors with N bits) are translated to equivalent types in DAFNY. Python type `float` however is not supported due to the limitation of DAFNY. Collection types including `Mapping`, `Sequence`, `Set`, and `Counter` are available as `map`, `seq`, `set`, and `multiset` respectively. `NamedTuple`,

```
class signature:
    @output
    def send(m: Msg):
        where = (m.src==ip)
    @input
    def recv(m: Msg): pass
    @internal
    def hide(k: int): pass

class transitions:
    @output
    @pre(m==Msg(ip, B, 10)
         and pc==1)
    def send(m):
        pc = pc + 1
        x = x + m.val
```

Fig. 4. DIONE signature and transitions.

```
1  datatype Action=send(m: Msg)| recv(m: Msg)
2                 | hide(k: int)
3  predicate Output(act: Action,p: Parameter)
4  { act.send? ∧ act.m.src = p.ip }
5  predicate Input(act: Action,p: Parameter)
6  { act.recv? }
7  predicate Internal(act: Action,p: Parameter)
8  { act.hide? }
9  predicate Signature(act: Action,p: Parameter)
10 {Output(act,p) ∨ Input(act,p)
11                ∨ Internal(act,p)}
12 predicate pre_send(act: Action,s: State,
13                    p: Parameter)
14 { act.send? ∧ act.m=Msg(p.ip,B,10) ∧ s.pc=1 }
15 function eff_send(act: Action,s: State,
16                   p: Parameter): State
17   requires pre_send(act,s) {
18   var s: State := s.(pc := s.pc+1);
19   var s: State := s.(x := s.x+act.m.val); s}
```

Fig. 5. Translated DAFNY from Figure 4.

`Enum`, and `Union` are directly modeled with inductive data types in DAFNY. Currently, we do not allow other kinds of user defined types for simplicity.

States and Automaton Parameters. The `states` class is used to declare variables of an automaton, or formally $val(X)$. Instead of directly using a function or `map` in DAFNY for mapping from variable names X to values, we model the state with a record type, a special case of inductive data types with only one constructor. Each field of the record type then corresponds to a variable of the automaton. Similarly, we introduce a new record type `Parameter` to model the parameter space of the automaton. For the example in Figure 3, we simply define a `Parameter` type (line 6) and a `State` type (line 8). The bound over the parameter space is translated into the `aut_where` predicate (line 9). Initial set of states specified through `(pc==1 and 0<=x<b)` is translated to the `initially` predicate below:

```
predicate initially(s: State,p: Parameter)
{  s.pc = 1  ∧  0 ≤ s.x < p.b  }
```

Notice that `initially` refers to the parameter and bound the value of `s.x` with `p.b`. Hence, checking $s \in \Theta_p$ is equivalent to asking `initially(s,p)`.

Actions and Transitions. To model the set of actions for a network of automata, we collect all actions declared in the `signature` class from all automata. We then specify an algebraic data type `Action` with each action as an individual constructor. For example, we specify `Action` with two constructors `send` and `recv` at line 1 in Figure 5 representing sending and receiving messages where the message `m` is a parameter of the action. For each automaton, the three predicates `input`, `output`, and `internal` symbolically represent Σ_p^I, Σ_p^O, and Σ_p^H. Similarly, the translation makes sure $act \in \Sigma_p^O$ is equivalent to `output(act,p)` and so on. For example, `output` predicate (line 3) constrains the source `m.src` of sent messages with its own `ip`.

```
@automaton
def Proc(ip: Addr,
         b: int):
    ...

@automaton
def Env():
    ...

@composition
def Sys():
    class components:
        env: Env()
        p1:  Proc(A,10)
        p2:  Proc(B,20)

    invariant_of = \
        0 <= p1.x < 10
```

Fig. 6. DIONE composition.

```
module Type { type Msg = ...; type Action = ... }
module Proc { import Type
  type State = ...; datatype Parameter = ...
  {: All predicates and functions } }
module Env { import Type; type State = ...
  {: All predicates and functions } }
module Sys { import Env; import Proc
  datatype State = State(env: Env.State,
                         p1: Proc.State,
                         p2: Proc.State)
  predicate output(act: Action) {
     Env.output(act)
  ∨ Proc.output(act,Proc.Parameter(A,10))
  ∨ Proc.output(act,Proc.Parameter(B,20)) }
  // Similar for input,output,internal,etc.
  predicate invariant_of(s: State)
  {  0 ≤ s.p1.x < 10  }
} // End Sys
```

Fig. 7. Translated DAFNY from Figure 6.

The precondition `@pre` for each transition is represented by a **`predicate`**, and the body of each transition is rewritten to an *effect* function using LET expressions ("**`var`** ... ;" in DAFNY). The translation for the limited kinds of statements in DIONE, such as assignments and if-conditions, to a function producing a new state is not specific to DAFNY. It has been discussed thoroughly in other works such as [19] for PVS. We recommend interested readers to refer to [19] for details. The only major difference is that, we further specify the precondition of the effect function with **`requires`** clause. This instructs DAFNY verifier to only consider states and actions satisfying the precondition.

Lastly, given all pairs of translated (`pre_i`, `eff_i`) for the automaton, the whole transition relation $\delta \subseteq Q \times \Sigma \times Q$ is modeled by the `transitions` predicate over current state `s`, a given action `act`, and next state `s'` as follows.

```
predicate transitions(s: State, act: Action, s': State)
{ (pre_1(act,s) ∧ s'=eff_1(act,s)) ∨ (pre_2(act,s) ∧ s'=eff_2(act,s)) ∨... }
```

Primitive Automaton with Parameters. A primitive automaton is translated into a DAFNY **`module`** to group `Parameter` and `State` types as well as **`function`**s and **`predicate`**s defining actions and transitions. `Action` type and other types shared across multiple automata are declared as a separate **`module`** and imported to each automaton module. An example layout is shown in Figure 7. Note that both automata can define its own `Parameter` and `State` types without causing naming collision.

Composition and Invariants. Given all component automata and their corresponding **`module`**s, we define another module and **`import`** those component **`module`**s. We then can define `State` for the composition with `State` from each component **`module`**. The `State` essentially represents the Cartesian product of each component state space. With the new `State` type, we can then define the necessary predicates, namely `initially`, `input`, `output`, `internal`, and `transitions`, according to the composition operation defined in Section 2. For Figure 7 as an example, we can implement `output`

predicate (line 11) using the disjunction of `output` from each component to represent $\Sigma^O_{Sys} = \Sigma^O_{Env} \cup \Sigma^O_{Proc(A,10)} \cup \Sigma^O_{Proc(B,20)}$. Do notice that the parameter value for instantiating each component automaton needs to be passed down to the predicates from sub-modules. Finally, translating an invariant of an automaton becomes defining an `invariant_of` predicate with current state as the argument and is similar to defining `initially`.

Checking Compatibility Axioms for IOA. In specifying systems in IOA, several simple mistakes are easily overlooked when designing larger and more complicated systems. We list three required axioms when specifying IOA and provide translated lemmas below to detect simple violations with Dafny. For simplicity, we do not include parameters here. The first lemma simply states Σ^I, Σ^O, and Σ^H are mutually disjoint. The second lemma checks the IOA is input-enabled, i.e., $\forall s \in Q. a \in \Sigma^I. \exists s' \in Q. (s, a, s') \in \delta$. The third states that two component IOAs assigned with concrete parameters are compatible when $\Sigma^O_1 \cap \Sigma^O_2 = \emptyset$ and $\Sigma^H_1 \cap \Sigma_2 = \Sigma^H_2 \cap \Sigma_1 = \emptyset$.[2]

```
lemma disjoint_actions_proof?(a: Action)
  ensures ¬(input(a) ∧ output(a))
        ∧ ¬(input(a) ∧ internal(a))
        ∧ ¬(output(a) ∧ internal(a))

lemma input_enabled_proof?(a: Action, s: State)
  requires input(a)
  ensures ∃ s' • transitions(s,a,s')

lemma compatibility_proof?(a: Action)
  ensures ¬(P1.output(a) ∧ P2.output(a))
        ∧ ¬(P1.internal(a) ∧ P2.signature(a))
        ∧ ¬(P1.signature(a) ∧ P2.internal(a))
```

3.3 Bounded Model Checking and *k*-induction with Dafny

With the system automaton with invariant specification translated to Dafny, we now discuss how to perform Bounded Model Checking (BMC) to detect violations. Given a bound k, BMC intends to prove that the invariant holds at any state reachable from initial states in k transitions. Formally, given a candidate invariant φ, we check the validity of following proposition:

$$\forall s_0, a_1, s_1, ..., a_k, s_k, (s_0 \in \Theta \wedge \bigwedge_{i=0}^{i<k} \delta(s_i, a_{i+1}, s_{i+1})) \implies \bigwedge_{i=0}^{i<k+1} \varphi(s_i) \quad (1)$$

We can ask Dafny to prove the following equivalent **`lemma`**:

```
lemma bmc_proof?(s0: State, a1: Action, s1: State, ..., ak: Action, sk: State)
  requires initially(s0) ∧ transitions(s0,a1,s1) ∧ ... ∧ transitions(sk−1,ak,sk)
  ensures  invariant_of(s0) ∧ invariant_of(s1) ∧ ... ∧ invariant_of(sk)
```

The underlying engine of Dafny translates the lemma further into formulas in First Order Theories supported by SMT solver, Microsoft Z3. If the lemma is

[2] Question mark '?' and prime symbol ''' are allowed in identifiers in Dafny.

Table 1. Protocols verified with DIONE. **#A** is the number of primitive and composite automata. **#Ln** (resp., **#Ty**, **#Fn**, **#Lem**) is the total number of lines (resp., custom types, functions, lemmas) in DIONE or DAFNY code. Numbers inside parentheses indicate manually added items for proving. **#k** is the number of transitions for BMC and induction proof. **#PO** is the number of proof obligations reported by DAFNY. **Time** and **Mem** shows the time and peak memory usage to prove with DAFNY.

	DIONE			Dafny						Time	Mem
Protocol	**#A**	**#Ty**	**#Ln**	**#Ty**	**#Fn**	**#Lem**	**#Ln**	**#k**	**#PO**	**(s)**	**(MB)**
`StableArray`	1	1	24	4	17(0)	4(1)	115	0	32	4	84.5
`StableRing`	1	1	20	3	13(0)	5(2)	115	0	66	166	377.6
`AsyncLCR`	2	2	47	7	26(2)	5(0)	159	0	51	3	100.3
`CANArb`	3	2	62	6	38(5)	6(0)	333	24	60	287	169.7

proven, then it proves no violation within k transitions. Otherwise, Z3 returns a valuation disproving the formula, and we can use the debugger in the DAFNY tool chain to find the valuation of state and action variables. That is, we can reconstruct the execution from a counterexample of Proposition 1.

Similarly, to prove invariant φ by k-induction, we prove Proposition 1 to ensure the states reachable in k-steps are within the invariant, and then prove Proposition 2 to show the invariant is inductive at $(k+1)$-th step:

$$\forall s_0, a_1, s_1, ..., a_{k+1}, s_{k+1}, (\bigwedge_{i=0}^{i<k+1} \varphi(s_i) \wedge \delta(s_i, a_{i+1}, s_{i+1})) \implies \varphi(s_{k+1}) \quad (2)$$

Or in DAFNY:

```
lemma induction_proof?(s₀: State, a₁: Action, s₁: State, ...,
            aₖ: Action, sₖ: State, aₖ₊₁: Action, sₖ₊₁: State)
  requires invariant_of(s₀) ∧ transitions(s₀, a₁, s₁) ∧ ...
         ∧ invariant_of(sₖ) ∧ transitions(sₖ, aₖ₊₁, sₖ₊₁)
  ensures  invariant_of(sₖ₊₁)
```

4 Case Studies with DIONE

To study the capability of DIONE, we analyze four distributed protocols with different network topologies, applications, and invariant properties using DIONE: The first two, `StableArray` and `StableRing`, are self-stabilizing mutual exclusion algorithms on an array and a ring network, `AsyncLCR` is a classic leader election algorithm on a ring, and `CANArb` is the protocol that arbitrates the access to CAN bus. For each case study, we first describe our DIONE models and invariants, then explain necessary auxiliary lemmas and functions added in the translated DAFNY code for verification, and finally discuss verification results based on BMC and k-induction using the DAFNY verifier. All experiments are conducted with DAFNY 2.2.0 on Ubuntu 18.04 LTS running on Intel Xeon CPU E3-1240 v3 at 3.40GHz with 4 cores and 8 GB RAM. To obtain the verification

results, all invariants for one system are conjuncted together as one big invariant predicate. DAFNY would verify all lemmas specified in a case study including disjointness of actions for primitive automata, compatibility in compositions, BMC, and k-induction for invariant checking. Each reported time and memory usage is the average of running DAFNY three times. All our code, the DIONE translator, the input specifications for the examples in DIONE language,[3] their DAFNY translations, and the proofs[4] are available at our repository.

4.1 Self-stabilization Protocol on a Bidirectional Array

`StableArray` in Figure 1 is the self-stabilizing algorithm for mutual exclusion on a bidirectional array topology from [9, Section 17.3.2]. The system consists of an array of N processes with at least two processes, i.e., $N \geq 2$. All processes except for process 0 and $N-1$ can remain in any of $\{0,1,2,3\}$ states. Process 0 should stay in $\{1,3\}$, and process $N-1$ should stay in $\{0,2\}$. We model the system with a global state variable, s, of sequence type with the length $|s| = N$, and each process state $s[i]$ is of an enumeration type from 0 to 3, and we also specify that $s[0]$ should be 1 or 3 and $s[N-1]$ be 0 or 2. Formally, the following invariant should trivially hold for `StableArray`:

Invariant 1. $|s| = N \wedge (s[0] = 1 \vee s[0] = 3) \wedge (s[N-1] = 0 \vee s[N-1] = 2)$

A process i is considered holding a token if any of its neighbor process j satisfies $s[j] = incre(s[i])$ where $incre$ finds the next value in the enumeration type, equivalently, $incre(n) = (n+1)\%4$. A process i holding a token except for process 0 and $N-1$ can initiate a transition to copy the state from the above mentioned neighbor j. If process 0 or $N-1$ is holding a token, it increments twice, i.e., $s[0] = incre(incre(s[0]))$. For achieving mutual exclusion, a *legal configuration* of the system is when the number of processes holding tokens is exactly one. A desired invariant is that, once in a legal configuration, the system continues to be in the legal configuration. Here we prove a relaxed invariant that the number of processes holding a token is at most one. Formally,

Invariant 2. $$\left| \left\{ i \mid 0 \leq i < N \wedge \begin{pmatrix} i \neq 0 \wedge s[i-1] = incre(s[i]) \vee \\ i \neq N-1 \wedge s[i+1] = incre(s[i]) \end{pmatrix} \right\} \right| \leq 1$$

We can specify the conjunction of Invariant 1 and 2 in DIONE as below:

```
    (len(s)==N and (s[0]==1 or s[0]==3) and (s[N-1]==0 or s[N-1]==2))
and (len({i for i in range(0,len(s)) if (i!=0 and s[i-1]==incre(s[i]) or
                                 i!=N-1 and s[i+1]==incre(s[i]))}) <= 1)
```

Proof Strategy. As shown in Table 1, we are able to automatically prove that the conjunction of Invariant 1 and 2 is an inductive invariant for arbitrarily many processes (N) in 4 seconds. We however needed to manually introduce

[3] See https://github.com/cyphyhouse/Dione/tree/master/system_tests/ioa_examples
[4] See https://github.com/cyphyhouse/Dione/tree/master/system_tests/expected_dafny

one auxiliary fact (lemma) in the translated DAFNY code to make the proof go through by induction. The intuition behind this lemma is that, when some process i makes a transition, only processes $i-1$, i, and $i+1$ can either lose or gain tokens, and all other processes should remain the same. Therefore, the set of processes holding tokens before and after a transition could only differ for these three processes. This extra lemma simply enumerates all eight cases where the size of the set may either increase, decrease, or remain the same, and DAFNY is able to prove this lemma automatically. We then use this lemma over the two states before and after a transition in specifying the proof strategy, DAFNY verifier further infers that those cases where the size increases are impossible and successfully proves that the invariant is inductive.

4.2 Self-stabilization Protocol on a Ring

The second case study `StableRing` is the Dijkstra's famous self-stabilizing algorithm for mutual exclusion on a ring [9, Section 17.3.1]. The system is parameterized with an arbitrary number of N processes and K states where $N < K$. Each process always stays in one of states $0, ..., K-1$. Similarly, we model the system with a global sequence s as the state variable with $|s| = N$ and $0 \leq s[i] < K$ for each process i; the following invariant should be maintained (we skip DIONE versions from now on):

Invariant 3. $|s| = N \wedge \ \forall i(0 \leq i < N \implies 0 \leq s[i] < K)$

Process i is considered holding a token by checking its predecessor $i-1$ and see if it's in one of two conditions: (1) $i = 0 \wedge s[i] = s[N-1]$ or (2) $i \neq 0 \wedge s[i] \neq s[i-1]$. Any process except process 0 holding a token can initiate a transition to copy the state from its predecessor in the ring, i.e., $s[i] = s[i-1]$. If process 0 is holding a token, it can assign itself the value from process $N-1$ plus one, i.e., $s[0] = (s[N-1]+1)\%K$. Likewise, a legal configuration is defined as that only one process is holding a token. For `StableRing`, we prove the desired invariant that, once in a legal configuration, the system continues to be in the legal configuration. Formally, the following invariant should hold:

$$\textit{Invariant 4.} \quad \left| \left\{ i \mid 0 \leq i < N \wedge \begin{pmatrix} i = 0 \wedge s[i] = s[N-1] \vee \\ i \neq 0 \wedge s[i] \neq s[i-1] \end{pmatrix} \right\} \right| = 1$$

Proof Strategy. As reported in Table 1, two of five lemmas are manually written to prove Invariant 4. Our first lemma is to establish the axiom that, if an element i is in a set I, then $|I| = 1 \iff I = \{i\}$. This helps DAFNY infer from inductive hypothesis that only one process i was holding a token in the prestate s before a transition. Notice that after transition, process $i' = (i+1)\%N$ will replace process i to hold the token. This leads to only two possible scenarios in the state after transition s': (1) $i' = 0$ and every process state is the same as its predecessor OR (2) $i' \neq 0$ and only process state $s'[i']$ is different from its predecessor. Otherwise, at least two processes will be different from their predecessors and hence holding tokens. The difficulty for DAFNY here is to infer $s'[0] = s'[N-1]$ in scenario 1 to

show process 0 is holding a token and $s'[j] = s'[j-1]$ in scenario 2 so that every other process j does not hold a token. Based on this observation, the second simple lemma we add asserts that, if every element is equal to its predecessor in a sequence, then all elements are identical. This lemma can be proven by DAFNY by induction on the length of the sequence. Our proof strategy thus simply splits into two cases $i = N-1$ and $i \neq N-1$ and applies the lemma over appropriate (sub-)sequences of s'. DAFNY can then infer the intermediate result mentioned above and prove the invariant inductively.

4.3 Asynchronous Leader Election on a Ring

Our third case study explores the possibility to use DIONE to model parameterized systems via composition and check the correctness. `AsyncLCR` is a simplified version of the leader election algorithm from [20, Section 15.1]. In this system, each process is instantiated with a unique $u \in \mathit{UID}$ for voting. The main algorithm flow is the following: Each process maintains a queue q of votes to be sent to its successor and a *status* variable initialized as `UNKNOWN`. When a vote from its predecessor is delivered, only the vote greater than u is added into the queue. The leader is decided when a process receives a vote v where $v = u$ and sets its status to `CHOSEN`. A process with $\mathit{status} = \texttt{CHOSEN}$ can then report itself as the leader and set its *status* to `REPORTED`. The algorithm guarantees that the reported leader process should have the maximum *UID* over all processes. For simplicity, our model first assumes the vote is delivered instantaneously and removes the need of channel automata by merging send and receive into one `send_recv` action. Second, although it is straightforward to model an arbitrary number of processes in DIONE, we only consider three processes to reduce the verification effort in this study. Thus, our model for the system is a composition of three automata $\{P_0, P_1, P_2\}$. Each P_i is an instance of the same automaton design assigned with actual parameters, the index i and the *UID* value u_i. We prove the key invariant described in [20] that no process other than the process with the max *UID* can report itself as the leader. Formally,

Invariant 5. $\bigwedge_{i=0}^{2} (u_i \neq max(u_0, u_1, u_2) \implies P_i.status = \texttt{UNKNOWN})$

Proof Strategy. In Table 1, we reported that two functions and no extra lemmas are needed. These two functions, needed for an auxiliary invariant, are currently not expressible in DIONE. The first required function finds the greatest u_i and returns the index i; here we use a variable i_{max} to represent the return value for simplicity. The second function $between(lo, i, hi)$ checks if lo must pass thru i to reach hi in the ring topology where $lo \neq i$ and $lo \neq hi$. We then use both functions to build the invariant modified from [20]:

Invariant 6. $\bigwedge_{i=0}^{2} \bigwedge_{j=0 \wedge j \neq i}^{2} (i \neq i_{max} \wedge between(i, i_{max}, j)) \implies u_i \notin P_j.q$

[width=]fig/can-full-frame [width=.6]fig/can-frame-arbitration

Fig. 8. Complete CAN data frame structure (Top). Example arbitration with three nodes and CAN Data observed on bus (Bottom).

This invariant is to prove that, when i must reach i_{max} before reaching j, the vote u_i should have been dropped before reaching j; therefore u_i should never appear in the queue of process j. DAFNY is able to prove the conjunction of Invariant 5 and 6 to be inductive without any additional lemma.

4.4 CAN bus Arbitration

In our last case study, we consider a vastly different communication protocol, the arbitration protocol for Controller Area Network (CAN bus). CAN bus is a long lasting and extremely popular communication protocol for Electronic Control Units (ECUs) in automotive. Specifically, the data link layer of CAN bus implements a Carrier-Sense Multiple Access with Collision Detection (CSMA/CD) type of protocols to arbitrate between ECUs and grant access to CAN bus to only one ECU at a certain time. In this section, we first describe this arbitration protocol, then give our DIONE model for the protocol, and finally provide our proof strategy for checking invariant.

CAN bus Arbitration. According to the ISO standard [13, Section 10.4], it is assumed that all ECUs (or nodes) are synchronized with a global clock for bit transmission, and every node implements the same mechanism to *serially* transmit and receive CAN data frame as shown in Figure 8. The frame starts with a start of frame (SOF bit) bit that must be DOMINANT value (logical 0) followed by a 11-bit arbitration field (ID bits) representing the priority of this frame where a smaller value of ID bits represents a higher priority. Otherwise, a node keeps sending RECESSIVE value (logical 1) when it is not sending data. Further, all nodes are connected to the CAN bus. The CAN bus can be considered as a logical conjunction of an arbitrary number of inputs from all nodes so that it outputs logical 0 if any node sends logical 0 in a cycle.

Arbitration happens only when multiple nodes simultaneously attempt to send a data frame. More precisely, multiple nodes simultaneously transmit SOF and the 11-bit arbitration field of the frames bit by bit in each cycle. Figure 8 shows an arbitration between three nodes, Node 2, 5, and 14, sending CAN frames with priority 0010101010, 00000100110, and 00000110100. Bus row represents the bit values monitored on bus. When the bits sent by different nodes differs, nodes sending logical 0 win over those sending logical 1 such as at ID8 and ID4. Nodes who lost the arbitration then stop transmitting and only send logical 1 ever after. At the end of the arbitration, the node sending the data frame with the highest priority wins. For example, node 5 wins and transmits the rest of frame in Figure 8.

Finally, a major property of CAN bus Arbitration specified in [13, Section 6.3] states the following: "The transmitter with the frame of highest priority shall gain the bus access". The standard also explicitly explains that the priority

of different frames are assumed to be distinct. The statement can paraphrased and formalized as the following equivalent invariant: "The ECU with the frame of highest priority shall keep transmitting in every cycle".

DIONE *Model.* Our DIONE model is designed following specifications in [13]. The system is composed of a `NodeSeq` automaton and a `Bus` channel automaton. The `NodeSeq` automaton is an abstraction of an arbitrary number of nodes. Its state consists of a `pos` to indicate which ID bit is currently transmitted and a sequence of `NodeStates` for individual nodes. Each node has an `arb` variable for the 11-bit arbitration field and a `transmit` variable to denote if this node is transmitting. Initially, `pos` starts at 10 and `transmit` is `True` to model that all nodes start to send at the same time. The `Bus` automaton simply has one state variable `bus` to represent the current bus state at each cycle. This `Bus` automaton can be considered as a broadcast channel where all nodes `send` to and `recv` from the `Bus` automaton.

To model synchronized communication, a cycle is modeled with a `send` action followed by a `recv` action. At the `send` action, `NodeSeq` outputs `msgs` modeling the sequence of bits with each bit sent from one node. Each node either extracts the bit from `arb` at current `pos` via a built-in function `bv_extract`, or sends logical 1 if it already stopped transmitting. `Bus` automaton then reads `msgs`, computes the logical conjunction via universal quantification, and stores the result in `bus`. At the `recv` action, `Bus` automaton publishes the value of `bus` as `msg` back to each node. Upon receiving `msg` from `Bus` in the `recv` action, each node then compares if the received bit is what it transmitted. If a node observes unequal bit values, then this node will set its `transmit` to `False` to stop transmitting. Notice that this requires an iteration through all nodes. Here, we choose to support *list comprehension* syntax in PYTHON and create a new sequence of `NodeState` from the old sequence with each node updated accordingly. This eliminates the need for a loop structure and simplifies the translation. After all nodes are updated, `pos` is decremented until it is negative.

Lastly, given the index i_{min} of the node with the highest priority, i.e., the smallest arbitration field, the invariant simply means the `transmit` of node i_{min} should stay `True`. The invariant is formulated as below:

Invariant 7. $|nodes| \geq 1 \wedge nodes[i_{min}].transmit$

Proof Strategy. In order to prove the invariant for this case study, we have to manually introduce the following code in DAFNY. First, to reflect the synchronized communication, one `send` and one `recv` action compose a cycle; hence assumptions over actions are added to consider only the executions composed of alternating `send` and `recv` actions. Second, four auxiliary functions are added to support translating the aforementioned list comprehension expression. These functions are for generic type of lists and therefore can be reused in more cases. Finally, we have to manually figure an over-approximation of one effect of the `recv` action, and use this approximation in the transition relation instead. Fortunately, we can instruct DAFNY to automatically check whether the approximation is indeed an over-approximation.

With above mentioned manual efforts, DAFNY is able to prove both BMC and induction with $k = 24$. $k = 24$ is simply because there are only 11 bits in arbitration field, and hence the states should stay the same ever after 12 cycles, i.e., 24 transitions. This showcases the potential of DIONE in verifying parameterized system composed with different channel or environment models, and this can be achieved within a manageable amount of manual effort.

5 Conclusion

We presented DIONE, a formal framework for analyzing distributed systems with specification language based on I/O automata and verification methods powered by DAFNY. The key compatibility conditions for IOA models are encoded as lemmas that can be automatically discharged by DAFNY in all of our cases. Our case studies show that a range of different distributed protocols can be naturally modeled in DIONE for Bounded Model Checking and k-induction invariant checking. The translated DAFNY specifications in the case studies were analyzed with DAFNY verifier automatically, with little extra manual annotations. These results are encouraging, and suggest several exciting future directions. (1) In the spirit of [11], IOA specifications could be translated to synthesizable DAFNY code, and hence, to correct-by-construction C# implementations for protocols. (2) Uniform or parameterized verification for distributed systems using small model properties [24] or theory of arrays [10]. Finally, (3) DIONE could be extended to support timed, hybrid, and probabilistic I/O Automata [14].

Acknowledgement. The authors were supported in part by research grants from the National Science Foundation under the Cyber-Physical Systems (CPS) program (award number 1544901 and 1739966).

References

1. Athalye, A.A.R..: CoqIOA: a formalization of IO automata in the Coq proof assistant. Thesis, Massachusetts Institute of Technology (2017)
2. Bhargavan, K., Bond, B., et al.: Everest: Towards a Verified, Drop-in Replacement of HTTPS. In: SNAPL'17. vol. 71, pp. 1:1–1:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017)
3. Bogdanov, A.: Formal verification of simulations between I/O automata. Thesis, Massachusetts Institute of Technology (2001)
4. Chockler, G., Lynch, N., Mitra, S., Tauber, J.: Proving atomicity: an assertional approach. In: DISC'05. LNCS, vol. 3724, pp. 152 – 168. Springer (Sep 2005)
5. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: ASE'09. pp. 137–148 (Nov 2009)
6. Fekete, A., Kaashoek, M.F., Lynch, N.A.: Implementing sequentially consistent shared objects using broadcast and point-to-point communication. J. ACM **45**(1), 35–69 (1998)
7. Fekete, A., Lynch, N.A., Shvartsman, A.A.: Specifying and using a partitionable group communication service. ACM Trans. Comput. Syst. **19**(2), 171–216 (2001)

8. Garland, S.J., Lynch, N.A., et al.: IOA user guide and reference manual (2003)
9. Ghosh, S.: Distributed Systems: An Algorithmic Approach, Second Edition. Chapman & Hall/CRC, 2nd edn. (2014)
10. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based Verification of Parameterized Systems. In: FSE'16. pp. 338–348. ACM (2016)
11. Hawblitzel, C., Howell, J., et al.: IronFleet: Proving Practical Distributed Systems Correct. In: SOSP'15. pp. 1–17. ACM (2015)
12. Hsieh, C., Mitra, S.: Dione (2019), https://github.com/cyphyhouse/dione
13. ISO: Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization (Dec 2003)
14. Kaynar, D.K., Lynch, N., et al.: Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In: RTSS'03. pp. 166–. IEEE Computer Society (2003)
15. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: CAV'14. pp. 17–34. Springer International Publishing (2014)
16. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR'10. pp. 348–370. Springer Berlin Heidelberg (2010)
17. Leino, K.R.M.: Automating Theorem Proving with SMT. In: ITP'13. pp. 2–16. Springer Berlin Heidelberg (2013)
18. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: Certified Causally Consistent Distributed Key-value Stores. In: POPL'16. pp. 357–370. ACM (2016)
19. Lim, H., Kaynar, D., et al.: Translating Timed I/O Automata Specifications for Theorem Proving in PVS. In: FORMATS'05. pp. 17–31. Springer, Berlin, Heidelberg (Sep 2005)
20. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
21. Nipkow, T., Slind, K.: I/O automata in Isabelle/HOL. In: TYPES'95. pp. 101–119. Springer Berlin Heidelberg (1995)
22. O'Hearn, P.W.: Continuous Reasoning: Scaling the Impact of Formal Methods. In: LICS'18. pp. 13–25. ACM (2018)
23. Padon, O., McMillan, K.L., et al.: Ivy: Safety Verification by Interactive Generalization. In: PLDI'16. pp. 614–630. ACM (2016)
24. Pnueli, A., Rodeh, Y., et al.: The Small Model Property: How Small Can It Be? Inf. Comput. **178**(1), 279–293 (Oct 2002)
25. Smith, M.A.S.: Formal Verification of TCP and T/TCP. Ph.D. thesis (1997)
26. Tuttle, M.R., Goel, A.: Protocol Proof Checking Simplified with SMT. In: NCA'12. pp. 195–202 (Aug 2012)
27. Wilcox, J.R., Woos, D., et al.: Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In: PLDI'15. pp. 357–368. ACM (2015)