Parameterized Verification of Publish/Subcribe Protocols via Infinite-state Model Checking

Giorgio Delzanno

DIBRIS, University of Genova, {giorgio.delzanno}@unige.it

Abstract. We apply the Infinite-State Model Checking to formally specify and validate protocol skeletons for distributed systems with asynchronous communication and synchronous access to local data structures. More precisely, we validate the Redis Pub/Sub key-value Server. Redis is based on a publish-subscribe architecture used in Cloud Storage and Internet of Things ecosystems. For the considered protocol, we present a formal specification that combines ideas coming from round-based and shared-memory specification languages. The resulting model is validated via the SMT-based Infinite-state Model Checker Cubicle. In this setting we use unbounded arrays to model (1) arbitrary collections of publishers and subscribers, (2) unbounded shared memory used as a communication media between processes. Our model is validated using the symbolic backward reachability algorithm implemented in the tool. The peculiarity of the algorithm is that, upon termination, the resulting correctness proof is guaranteed to hold for every number of process instances.

CEUR-WS.org/Vol-2214/paper11.pdf

1 Introduction

Protocols designed to operate in distributed systems are often defined for an arbitrary number of components interacting via asynchronous communication. Formal specification languages like Petri nets can be used to model skeletons and abstractions of this kind of systems. In this setting the coverability decision problem [1] can be applied to specify potential violations of safety properties independently from the number of system components. For distributed systems it is often convenient to consider a generalization of the coverability problem existentially quantified over an infinite set of initial configurations [7,8]. Existential coverability has been considered, e.g., in [15,16,17,6,5] to reason about correctness of Broadcast Protocols. For fragments of Broadcast Protocols, (existential) coverability can be solved algorithmically via the symbolic backward reachability procedure defined for Well-structured Transition Systems in [3,21]. The procedure, together with a number of approximations and heuristics, has been implemented in the SMT-based Infinite-state Model Checker Cubicle [11]. Cubicle provides a specification language that combines both local and global update rules. The combination is particularly useful when dealing with specification of the internal behavior of protocols designed for client-server architectures.

In this paper we focus our attention on the application of Cubicle [11] to formally specify and validate distributed protocols that combine asynchronous communication and synchronous operations on data structures local to network nodes. As a case-study, we consider the Redis Pub/Sub key-value Server adopted in Cloud Storage and Internet of Things eco-systems. Redis Server is based on a publish-subscribe architecture that can be used to guarantee the consistency of updates in database systems of microservice architectures. Technically, in the paper we first present a formal model of the Pub/Sub protocol in the array-based specification language of Cubicle. Following recent works on verification of heard-of and round-based models of distributed protocols [9,18,19,20,24,13], we apply here a combination of round-based and shared-memory specification languages. Round-based models are quite useful to split complex protocol executions into a sequence of interconnected phases for which round numbers act as identifiers or timestamps. Shared-memory models can be used (1) to model synchronization steps of data structures local to server nodes and (2) to define asynchronous interaction between client nodes (publishers and subscribers).

Cubicle captures in a natural way the combination of these two types of models. First of all, it provides declaration of multi-dimensional unbounded arrays. For instance, the declaration `array Sub[proc,proc];` denotes a bidimensional array in which indexes range over an unbounded set of process identifiers. In our case study, each row of an unbounded matrix can be used to model a protocol phase, e.g., the current configuration of a subscriber group. Each cell in a row can then be used to represent the current state of a subscriber, publisher, message, and so on. Furthermore, Cubicle transitions can specify both local and global modifications to groups of array cells. Atomic global updates are obtained by using universally quantified transitions. Asynchronous updates can be obtained by splitting a global update in a series of local updates executed non-deterministically.

The resulting model is validated through the SMT-based verification engine of Cubicle. Cubicle implements a symbolic backward reachability algorithm in which sets of configurations are represented via formulas in fragments of First Order Logic that combine Presburger Arithmetics and the Theory of Arrays [11,25,4,22]. By construction, the Cubicle verification algorithm ensures that, upon termination, the resulting correctness proof is guaranteed to hold for any number of processes. In other words Cubicle can be applied as an automated engine for solving parameterized verification problems. For our case-study, we successfully validated different versions of the Redis Pub/Sub protocol and identify corner cases for the form of transition guards.

The paper is organized as follows. We first introduce our case-study. We then propose a formalization guided by the above described combination of ideas coming from round-based and shared memory specification models and by functional requirements of the protocol. We then discuss validation results obtained via the Cubicle engine. Finally, we discuss related work and address some future research directions.

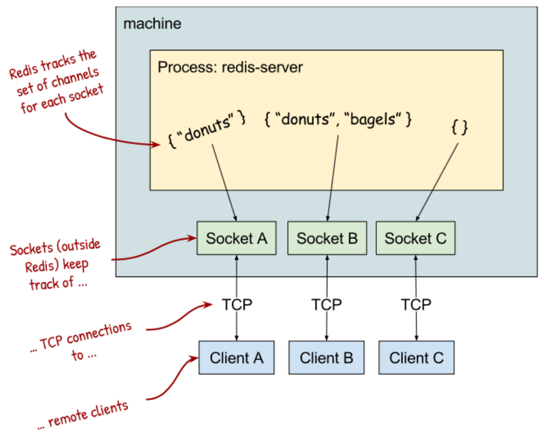

Fig. 1. Pub/Sub data structures of Redis

2 The Redis Pub/Sub Server

Redis is a Publish/Subscribe (Pub/Sub) system that can be used as key-value server. Redis is used in microservice applications to ensure consistency of updates of distributed databases. The underlying publish/subscribe protocol can indeed be used to propagate local updates to a given key to a set of connected services. In the original Pub/Sub implementation clients can use three types of commands: PUBLISH, SUBSCRIBE, and UNSUBSCRIBE. To track subscriptions, Redis uses a global variable *pubsub_channels* which maps channel names to sets of subscribed client objects. A client object is a virtual image of a TCP-connected client maintained in the server. When a client submits a SUBSCRIBE command for a given channel, its client object gets added to the set of clients for that channel name as shown in Fig. 1. To PUBLISH, Redis looks up the subscribers in the *pubsub_channels* variable, and for each client, it schedules a job to send the published message to the corresponding client socket. To optimize the management of connection failures (e.g. clients close their connections), Redis annotates each client with its set of subscribed channels, and keeps this in sync with the main *pubsub_channels* structure. This way, instead of iterating over every channel, Redis only needs to visit the channels the client was subscribed to. Fig. 2 illustrates the configuration obtained from Fig. 1 when adding the above mentioned auxiliary structure. At the implementation level the *pubsub_channels* is defined as an hash table to optimize key-search and hash table resizing. To summarize, the algorithm for the PUBLISH and (UN)SUBSCRIBE operations are informally defined as follows.

– To PUBLISH, hash the channel name to get a hash chain. Iterate over the hash chain, comparing each channel name to our target channel name. Once

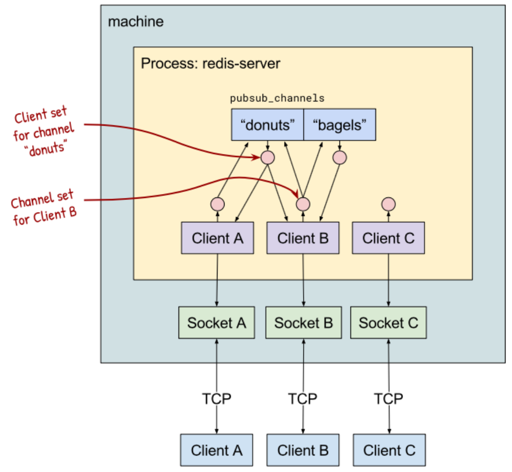

Fig. 2. Optimized Pub/Sub data structures of Redis

the target channel name is found, get the corresponding list of clients. Iterate over the linked list of clients, sending the published message to each client.
– To SUBSCRIBE, find the linked list of clients as before. Append the new client to the end of the linked list (constant time). Also, add the channel to the client's hash table (constant time).
– To UNSUBSCRIBE, find the linked list of clients for the channel, as before. Then iterate over the entire list and remove the client.

Further optimizations are related to lazy resizing strategies of hash tables and bit-level representation of hash tables and channel names. We do not describe these features here and focus instead on the protocol skeleton used in the Redis server.

3 Formal Specification of the Redis Pub/Sub Server

To model the Redis architecture and extract correctness requirements, we will first focus on some peculiarities and properties of the underlying Pub/Sub protocol. First of all, we remark that the object manipulated in the Server data structures represent TCP client connections. A TCP connection can be viewed as a perfect channel in which messages are delivered in the same order as they are sent. Under this assumption, we can restrict ourselves to failures due to network connections drops or to explicit client disconnections. Furthermore, we will assume that (1) the Redis implementation ensures the synchronization between the different data structures maintained in the server, (2) each data structure (e.g. *pubsub_channel*) is maintained consistent via standard concurrency control abstractions (concurrent data structures, synchronized blocks, etc). These assumptions allows us to focus on a model with two different layers: (1) a synchronous layer used to model updates of the data structure maintained by the Redis server. (2) an asynchronous layer used to model responses. More precisely, subscribe requests will be modeled via synchronous updates of an abstraction of the *pubsub_channel* data structure, whereas publish requests will be modeled in two phases: a synchronous step used to update the server data structures and and asynchronous one used to send notifications to subscribers.

Another important observation is related to the expected behavior of the Pub/Sub protocol. Subscribe and unsubscribe requests can be submitted any time to a Redis server. Thus, at least in principle, it is impossible to guarantee that, for a fixed topic, a given update will reach all subscribers that are in the *pubsub_channel* during the completion of a publish request. For instance, a subscriber can join the server right after the update has been distributed to all TCP connection objects but before it is actually received by all clients, an unsubscribe request can be received by the server before the completion of the publish request of another client, a TCP connection might drop during the protocol execution, etc. To model the above mentioned scenarios, it is convenient to adopt a round-based view of the protocol behavior. In each round we use a shared memory model to specify the current state of the *pubsub_channel*. More specifically, we use two unbounded arrays `Sub` and `Msg`, indexed on round and process indentifiers, to model: (1) the state of subscribers in different rounds of the protocol, and (2) messages waiting to be delivered to subscribers in that round. Notice that each row of the two matrices is associated to a distinct round. A round consists of synchronous steps in which either a (non deterministically selected) subscriber joins a group associated to a given topic or a publisher prepares the message to be sent to the current set of subscribers. The asynchronous phase of a round consists in the distribution of notifications to the subscriber group. Rounds are used here as a conceptual tool to model different phases of the protocol. In real execution traces several phases may overlap. Following the methodology discussed in [9,18,19,20], overlappings can be eliminated by using permutation schemes (e.g. left and right movers). To model the dynamics of subscribers groups, we operate as follows. Subscribers can non-deterministically create new groups by selecting a round number for which there are no pending messages prepared by publishers. As soon as a publisher associates to that round number a given message object (preparation of the notification phase), new subscribers need to create new groups using a fresh round identifier. The interesting point of a round-based model of the protocol is that the same idea can be applied to model network failures and reconfigurations. Indeed anytime any configuration of a subscriber group can be generated by selecting a fresh round identifiers and discharging the previous one. In other words for any possible real protocol executions we can rearrange protocol and group formation steps in order to define a sequence of rounds with the same groups and the same set of notifications. For instance, an execution in which several messages are sent to the same subscribers can be mimicked by a sequence of rounds in which a single message is sent to every subscriber and so on.

We do not consider here any order between round identifiers. The reason is that we rely on TCP connections for maintaining the order between a sequence of updates sent by the same publisher. However our goal is to show the correspondence between updates sent by a publisher and received by a subscriber within the same round. To ensure this property, we need to ensure freshness of every newly created subscriber group, and ensure that object preparation is done synchronously (or using some form of serialization step).

In our model we apply some additional abstractions. Firstly, we focus on a single channel name (topic), assuming that the synchronization of the data structure associating clients to topics and topics to clients are done correctly in the protocol implementation of the Redis server Secondly, we consider only notifications with two possible values, namely One and Two, since we two values are enough to show a possible inconsistency between a publisher and a subscriber view of the same update.

Figure 3 summarizes the intuition of the model that we will formally specify using Cubicle's input language in the rest of the section.

Array Variables

We first define types and array variables to model publishers, subscribers, and pending object messages in different rounds.

```
type pstate = Idle  | Busy | One  | Two
type sstate = SIdle | SOne | STwo
type mstate = Null  | MOne | MTwo

array Pub[proc]   : pstate
array P[proc,proc]  : pstate
array S[proc,proc]  : sstate

array Sub[proc,proc]  : bool
array Msg[proc,proc]  : mstate
```

More specifically, the interpretation of the declared data structures is as follows:

- `pstate` defines the publisher state,
- `sstase` defines the subscriber state,
- `mstate` is used to model the state of the shared-memory,
- `Pub` models the state of publishers: `Pub[n]` is initially `Idle` and it is set to `Busy` after the selection of a round number.
- `P` models the state of publishers after the selection of a round number, i.e., `Pub[t,n]=One` [resp. `Two`] means that, in round t, n has chosen to send *One* [resp. `Two`].
- `Sub` models a subscriber group in a specific round, i.e., $Sub[t,p] = True$ if, in round t, p belongs to the subscriber group.
- `S` models the state of subscribers after the selection of round numbers.
- finally, `Msg` models the flow of messages from publishers to subscribers, i.e., `Msg[t,p]` identifies a message that, in round t, has to be delivered to subscriber p.

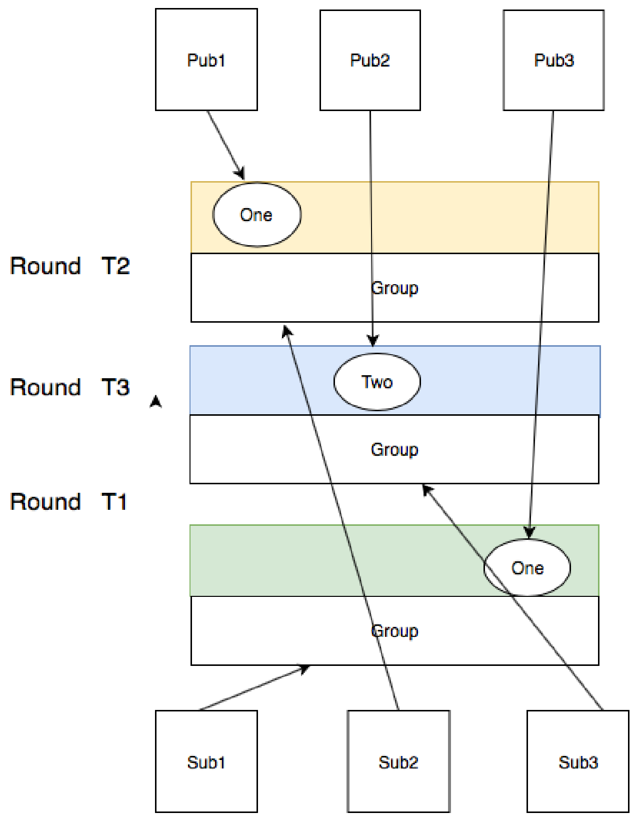

Fig. 3. Round-based model for publishers and subscribers: each rows denotes a different state of the *pubsub_channel* global variable.

Initial Configuration

The initial configuration is defined via the following Cubicle statement.

```
init (t n) {
   Pub[n] = Idle   &&
   P[t,n] = Idle &&
   S[t,n] = SIdle   &&
   Sub[t,n] = False &&
   Msg[t,n] = Null
}
```

The above statement must be interpreted as a conjunctive formula universally quantified over t and n. In other words in the initial configuration `Pub`,`P` and `S` are in idle state, there are no subscription groups and no message objects.

Subscribe

The first transitions specifies the subscription step of a given client.

```
transition subscribe(t n)
requires {
  Msg[t,n] = Null &&
  Msg[t,t] = Null  &&
  forall_other r. (Msg[t,r] = Null)
}
{
Sub[t,n] := True;
}
```

For a subscriber to join a group at round t, the rule requires the absence of message objects in the same round. Since the initial configuration does not put any constraint on groups, any subscriber group can be generated by using this rule until a given round identifier t remains fresh.

Publish

The second transition combines an asynchronous request done by a publisher with a synchronous step done by the Redis server in order to prepare message objects for every subscriber of the current round.

```
transition publish1(t n)
requires {
  Pub[n] = Idle &&
  Msg[t,n] = Null &&
  Msg[t,t] = Null  &&
  forall_other r. (Msg[t,r] = Null)
}
{
Pub[n] := Busy;
P[t,n]  := One;
Msg[p,q] := case
   | p=t  && q=n : MOne
   | p=t  && Sub[t,q]=True : MOne
   |  _                    : Msg[p,q];
}
```

In this rule a publisher with identifier n selects a fresh round number (in which subscribers could have formed a group with arbitrary shape) and then prepares message objects for every subscriber associated to that round number.
The first case `p=t  && q=n : MOne` of the update rule is used to insert at least a `MOne` message in the current round (in case there are no other subscribers).
The second case `p=t  && Sub[t,q]=True : MOne` assigns the state `MOne` to the

message object of every subscriber. The state of the message objects of other rounds or other indexes remains unchanged.

The following rule is used to select value *Two* and assign state *MTwo* to message objects.

```
transition publish2(t n )
requires {
  Pub[n] = Idle &&
  Msg[t,n] = Null &&
  Msg[t,t] = Null  &&
  forall_other r. (Msg[t,r] = Null)
}
{
Pub[n] := Busy;
P[t,n]  := Two;
Msg[p,q] := case
   | p=t  && q=n : MTwo
   | p=t  && Sub[p,q]=True : MTwo
   |  _ : Msg[p,q];
}
```

The combined effect of the previous rules is that of storing the type of notification (`One` or `Two`), in a non-deterministically chosen, unused round identifier t, and of initializing the state of every cell $\langle t, q\rangle$ (current round t, subscriber q) of the `Msg` matrix with the corresponding states (`MOne` for `One` and `MTwo` for `Two`).

Finally, we assume that publishers can reset their state and return to the `Idle` state in order to start a new round (and possibly send a different value).

```
transition reset(n)
requires {
  Pub[n] = Busy
}
{
  Pub[n] := Idle;
}
```

Notifications

We now come to the definition of the notification phase. Notifications are generated by non-deterministically delivering message objects to subscribers.

```
ttransition notify1(t n)
requires {
  S[t,n] = SIdle &&
  Msg[t,n] = MOne
}
{
  S[t,n] := SOne;
}

transition notify2(t n)
requires {
  S[t,n] = SIdle &&
  Msg[t,n] = MTwo
}
{
  S[t,n] := STwo;
}
```

The subscriber state `S` stores in a local state the value of the received message for the current round number.

4 Validation through Cubicle

Our correctness requirement must necessarily be formulated with respect to a given round. More specifically, we would like to show that only when a subscriber group remains stable for a long enough period (that we identify as a round), then every notification sent by the publisher will be received by every subscriber in the group. This kind of property is similar to correctness criteria used in consensus protocols like Paxos. Under this hypothesis, unsafe configurations can be specified using the following judgements adopted by Cubicle to specify bad configurations:

```
unsafe (t m n) {
  P[t,m]  = One  && S[t,n]  = STwo
}

unsafe (t m n) {
  P[t,m] = Two  && S[t,n] = SOne
}

unsafe (t n) {
  P[t,n]  = Two  && S[t,n] = SOne
}

unsafe (t n) {
  P[t,n]  = One  &&  S[t,n]  = STwo
}
}
```

Cubicle interprets this kind of formulas as existentially quantified over the parameter names. Therefore, the judgements specify configurations in which, through the shared-memory model of subscription and message objects, values read by subscribers are different from those published by publishers in the same round.

5 Cubicle Engine at Work

The Cubicle verification engine is based on symbolic backward exploration. Cubicle operates over sets of existentially quantified formulas called *cubes*. The search procedure maintains a set V and a priority queue Q resp. of visited and unvisited cubes. Initially, let V be empty and let Q contain the cubes representing bad states. At each iteration, the procedure selects the highest-priority cube Φ from Q and checks for intersection with the formula denoting the initial configurations (satisfiability of conjunction of Φ and formulas in the initial conditions). If the test fails, it terminates reporting a possibile error trace. If the test passes, the procedure proceeds to the subsumption check, i.e., implication between formulas. If subsumption fails, then add Φ to V, compute all cubes in $pred_t$ (for every t), add them to Q, and move on to the next iteration. If the subsumption check succeeds, then drop Φ from consideration and move on. The algorithm terminates when a safety check fails or Q becomes empty. When an unsafe cube is found, Cubicle actually produces a counterexample trace.

Formulas containing universally quantified formulas (generated during the computation of predecessors) are over-approximated by existentially quantified formulas. In presence of universally quantified guards the class of formulas manipulated by the backward reachability loop of Cubicle in not closed under pre-image. To handle such formulas, Cubicle implements a safe but over-approximate pre-image computation. Given a cube $\exists \bar{i}.\Phi$ and a guard G of the form $\forall \bar{j}.\Psi(\bar{j})$, the pre-image replaces G by the conjunction $\bigwedge_{\sigma \in \Sigma(\bar{j},\bar{i})} \Psi(\bar{j})\sigma$ of instances over the permutation of $\Sigma(\bar{j},\bar{i})$. In other words, in order to handle universally quantified guards, Cubicle applies a declarative (and syntactic) version of the monotone abstraction introduced in [2]. Safety checks, being ground satisfiability queries, are easy for SMT solvers. The challenge is in the subsumption check because of their size and the existential implies existential logical form. Cubicle applies the heuristics described in [11] to handle subsumption. The BRAB algorithm, introduced in [12], automatically computes over-approximations of backward reachable states that are checked to be unreachable in a finite instance of the system using the Murϕ model checker. The resulting approximations (candidate invariants) are model checked together with the original safety properties. Completeness of the approach is ensured by a mechanism for backtracking on spurious traces introduced by too coarse approximations.

5.1 Pub/Sub Protocol Validation

When applying Cubicle to our formal model of the Redis Pub/Sub protocol enriched with judgments expressing bad configurations, the tool proves the model correct in few seconds. More specifically, Cubicle visits 20 nodes with at most 3 process indexes, 970 fixpoint checks, and 806 calls to the Alt-Ergo SMT solver. Since Cubicle operates over unbounded arrays, the above result provides a correctness proof of the considered model for any number of publishers and subscribers. The proof certificate can be obtained by taking the (negation of the) set of assertions (formulas) collected during the fixpoint computation. When

Model	dfs	brab	V	F	S	M	D	I	C
Redis			20	970	806	3	0	0	Yes
”	$\surd$		20	962	820	3	0	0	Yes
”		$\surd$ (2)	12	518	291	3	0	6	Yes
”		$\surd$ (3)	12	520	301	3	0	8	Yes
Redis1			20	970	806	3	0	0	No

Fig. 4. Experimental results: **V**=visited nodes, **F**=fixpoint tests, **S**=solver calls, **M**=max process number, **D**=deleted node, **I**=number of invariants (brab), **C**=property checked (Yes/No).

invoked using the BRAB algorithm with parameter 2 the tool infers 6 invariants that reduce the visited nodes to 12, the fixpoint checks to 518, and the number of calls to the solver to 291. The execution time remains unchainged. When invoked using the BRAB algorithm with parameter 3 the tool infers 8 invariants at the cost of a significant increase of the execution time. The table in Fig. 4 summarizes the results obtained with the different heuristics provided by Cubicle. In Fig. 4, in order to check robustness, we also consider variations, as such as Redis1, of the above discussed model. Redis1 is obtained by removing from the publisher rules the the first case in the `Msg` update action, namely `| p=t  && q=n : MOne` in `publish1` and `| p=t  && q=n : MTwo` in `publish2`. When validated in Cubicle, this modification leads to the following error trace (in the Cubicle output notation):
`pub1(#1,#2)->Sub(#1, #3)->pub2(#1,#3)->notify2(#1,#3)->unsafe[1]`.
In the trace we can find two phases (rounds) that are not separated via distinct identifiers. This leads to the following undesired behaviour: a publisher sends a message, say One, when no subscriber is registered to the considered topic, yet. As a consequence no message object is inserted in the shared-memory (Redis server data structure). Subscribers can then re-use the same round number and, in the second part of the session, join the group before a second message of type Two is brodcast to the subscriber group. The guard in the update rule forbids this behavior by marking the round after the first publish message as used. This way, we enforce the correspondence between round numbers and protocol phases.

6 Related Work and Conclusions

We have presented an application of the SMT-based Infinite-state Model Checker Cubicle to the verification of a parameterized model of the Pub/Sub architecture implemented in the Redis server. The formal specification is based on the combination of a round-based interpretation of the protocol behavior with a shared memory representation of the updates to its internal data structures. This work is strictly related, although applied in a different class of protocols, with our recent work on the application of logic-based declarative methods for the verification of distributed systems [14,10]. More specifically, in [14,10] we proposed to apply a logic-based language called GLog and Cubicle [11,23] as a declarative verification framework for distributed systems. GLog is based on a quantified predicate logic in a finite relational signature with no function symbols. Configurations are represented as sets of ground atomic formulas (instances of unary and binary predicates). Update rules consist of a guard and two sets of first order predicates that define resp. deletion and addition of state components. Rules can be applied to update a global configuration component by component and to test global conditions on the vicinity of a node by restricting updates to given predicates. Termination of an update subprotocol can then be checked via a global condition. Similar specification patterns have been applied to model non-atomic consistency protocol and mutual exclusion protocols with non-atomic global conditions. GLog has been applied to manually analyze distributed protocols in [14]. Cubicle [11,23], an SMT-based infinite-state model checker based on work by Ghilardi et al. [4], can be applied as an automated verification engine for existentially quantified coverability queries in GLog. In Cubicle parameterized systems can be specified as unbounded arrays in which individual components can be referred to via an array index. The Cubicle verification engine performs a symbolic backward reachability analysis using an SMT solver for computing intermediate steps (preimage computation, entailment and termination test) and applies overapproximates predecessors using upward closed sets as in monotone abstractions [2]. A peculiar feature of Cubicle w.r.t. MCMT [4] is that the tool can handle unbounded matrices. This is particularly relevant when modeling topology-sensitive protocols as done in GLog using binary relations defined over component identifiers. Furthermore, existentially quantified coverability decision problems can be mapped into Cubicle. Indeed, classes of initial configurations are specified by using partial specifications of initial configurations in Cubicle verification judgements. Infinite sets of bad configurations can be expressed as unsafe configurations in Cubicle verification judgements. We believe that the proposed methodology (array-based specifications of round-based protocols with complex local data structures, verification via symbolic exploration) is applicable to other classes of distributed systems such as consensus protocols or protocols that use consensus protocols as building blocks (e.g. distributed objects/ledgers).

References

1. P. A. Abdulla and G. Delzanno. Parameterized verification. *STTT*, 18(5):469–473, 2016.
2. P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *Int. J. Found. Comput. Sci.*, 20(5):779–801, 2009.
3. P. A. Abdulla and B. Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems. *Theor. Comput. Sci.*, 256(1-2):145–167, 2001.
4. F. Alberti, S. Ghilardi, and N. Sharygina. A framework for the verification of parameterized infinite-state systems. *Fundam. Inform.*, 150(1):1–24, 2017.
5. N. Bertrand, G. Delzanno, B. König, A. Sangnier, and J. Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In

RTA'12, volume 15 of *LIPIcs*, pages 101–116. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

6. N. Bertrand, P. Fournier, and A. Sangnier. Distributed local strategies in broadcast networks. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 44–57, 2015.
7. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
8. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.
9. B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
10. S. Conchon, G. Delzanno, and A. Ferrando. Parameterized verification of topology-sensitive distributed protocols goes declarative. In *Proceedings of NETYS 2018, to appear*, 2018.
11. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 718–724, 2012.
12. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68, 2013.
13. H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012.
14. G. Delzanno. A logic-based approach to verify distributed protocols. In *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016.*, pages 86–101, 2016.
15. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, pages 313–327, 2010.
16. G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 441–455, 2011.
17. G. Delzanno, A. Sangnier, and G. Zavattaro. Verification of ad hoc networks with node and communication failures. In *FORTE/FMOODS'12*, volume 7273 of *LNCS*, pages 235–250. Springer, 2012.
18. C. Dragoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 161–181, 2014.
19. C. Dragoi, T. A. Henzinger, and D. Zufferey. The need for language support for fault-tolerant distributed systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 90–102, 2015.
20. C. Dragoi, T. A. Henzinger, and D. Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 400–415, 2016.
21. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
22. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
23. A. Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés. (Invariants inference for model checking of parameterized systems)*. PhD thesis, University of Paris-Sud, Orsay, France, 2014.
24. T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.
25. `http://users.mat.unimi.it/users/ghilardi/mcmt/`.