Verification of Contact Tracing Protocols via SMT-based Model Checking and Counting Abstraction*

Sylvain Conchon[1], Giorgio Delzanno[2], and Arnaud Sangnier[3]

[1] LRI, Universitè Paris, France sylvain.conchon@lri.fr
[2] DIBRIS, University of Genova, Italy giorgio.delzanno@unige.it
[3] IRIF, Universitè Paris Denis Diderot, France sangnier@irif.fr

Abstract. We present an automata-based model specifically devised to formalise abstractions of distributed protocols used by contact-tracing applications that combine Bluetooth and TCP/IP communication with a centralised server. The model provides pure names, store and read operations on both value and set variables, synchronous and asynchronous communication primitives for both kind of variables. A protocol configuration consists of the current state of a finite set of local states containing the states of individual devices. The transition system models the interaction between devices in the same physical location and between a single device and possible distributed servers. We will use the resulting model to specify the logic underlying contact tracing protocols. To automatically validate our formal models, we employ an extension of the Cubicle infinite-state model checker based on the Alt-Ergo SMT solver. To overcome spurious results due to the application of monotone abstraction, we propose to refine the predecessor computation adopted in Cubicle by combining predicates on the Theory of Arrays (as provided by Cubicle) with Presburger predicates inferred via a counting abstraction applied on a subset of control states of individual processes.

Keywords: Formal Verification, SMT, Infinite-state Model Checking, Contact Tracing Protocols

1 Introduction

The goal of contact tracing is to make people aware of possible contacts with positively diagnosed people so that they should possibly have an infection test [16]. For pandemic diseases, reporting a positive diagnosis is mandatory. Contact tracing methods, e.g., smartphone apps, are aimed at investigating who could have been contaminated by a positively diagnosed person and alert them. After the COVID-19 pandemic, several protocols have been proposed as an automated support for this task. The Pan-European Privacy-Preserving Proximity

Tracing (PEPP-PT) proposed both centralized and decentralized solutions. As a decentralized solution, they propose the Decentralized Privacy-Preserving Proximity Tracing (DP3T), and the specifications of ROBERT4 and NTK5. Both centralized and non-centralized systems require a central server for alerts and typically differ in the way ephemeral identifiers are generated. In general the system architecture is based on a smart app, a notification server as in Publish/Subscribe architectures, and a possibly trusted server. An important notion here is that of ephemeral identifiers, i.e., identifiers that are frequently changed either by a trusted authority/server or by the app itself in order to reduce the risk of a malicious use of private user data from third parties such as the attacks presented in [5, 16]. Ephemeral identifiers are kept locally in the smart app. In centralized versions of the protocol, ephemeral identifiers are downloaded from a trusted server. In decentralized versions, ephemeral identifiers are created by the app. The smart app installed on the user device, e.g. the Immuni app in Italy, makes use of the Bluetooth interface to broadcast the currently valid ephemeral identifier. The communication range of the Bluetooth interface is tuned so as to reach the distance for a possible contagious contact. The app collects all identifiers received by other apps located in the same physical place and stores them locally. Identifiers whose temporal marks fall out of the incubation period are deleted from memory. When users are diagnosed to be infected, they can release a report to the shared server. The server is used to make the report (a log containing ephemeral identifiers) available to each user interested in checking potential contacts with infected users. This step can be implemented in several different ways depending on the system architecture and on the role of the authority in the whole process. One possibility is to send the report to a central log database. Users can then consult the database to check if they crossed their way with infected users, i.e., they share some ephemeral identifier with one of them.

Building up on the formalization of decentralized protocols in [1], in this paper we present an automata-based formal model of distributed systems specifically devised to formalise abstractions of Contact Tracing Protocols that combine Bluetooth and network communication. The model combines pure names, read/write operations on first-order and higher-order variables and synchronous communication primitives. The transition system models the interaction between devices in the same physical location and between a single device and a notification server.

To automatically validate protocols in our formalism, we resort to an extension of the Cubicle SMT-based infinite-state model checker. Cubicle is a model checker that can be applied to verify safety properties of array-based systems, a syntactically restricted class of parameterized transition systems with states represented as arrays indexed by an arbitrary number of processes [7, 15]. Cubicle integrates the SMT solver Alt-Ergo [4]. Cubicle input language is a typed version of Murϕ [12]. A system is describe in Cubicle by: (1) a set of type, variable, and array declarations; (2) a formula for the initial states; and (3) a set of transitions. A system execution is defined by an infinite loop that at each

iteration: (1) non-deterministically chooses a transition instance whose guard is true in the current state; and (2) updates state variables according to the action of the fired transition instance.

Properties of contact tracing protocols are strongly related to the consistency of global and local state information, e.g., ensure that the local user state reflects data stored in the server. Violation of this kind of property are specified via universally quantified assertions, the more difficult type of assertions for infinite-state model checking. This kind of assertions or transitions is a potential cause of undecidability already in simpler infinite-state models such as Petri Nets.

To deal with universally quantified conditions, Cubicle applies monotone abstraction in order to over-approximate the symbolic computation of predecessors states turning universally quantified preconditions into postconditions. Unfortunately, this approximation does not work well for the class of properties of interest for contact tracing protocols. Our proposed solution is based on the strengthening of the symbolic computation performed in Cubicle by combining array-based assertions with Presburger formulas generated via a Counting Abstraction.

Plan of the paper In Section 2 we introduce Data Automata as a formal model for protocols such as those used in contact tracing In Section 3 we discuss an example of contact tracing protocol specification via Data Automata. In Section 4 we present an encoding of Data Automata in the Cubicle input language with the help of our case study. In Section 5 we discuss the problems encountered with the current search strategy implemented in Cubicle and propose our solution based on a combination of array and counting assertions. In Section 6 we address conclusions and related works.

2 Contact Tracing Systems

In this section we introduce a possible formalization of a decentralized contact tracing protocol. To specify the protocol rules and behaviour (computation), we will use a transition system defined on configurations consisting of a set of states of individual users. User configurations contain a local memory needed to store the list of broadcasted and collected identifiers. System configurations contain a global memory that gathers user logs and a set of user states that corresponds to the current number of registered individuals. To simplify the model, we fix the number of agents and simulate dynamic injection by randomly selecting the initial value of local clocks used to define validity of ephemeral identifiers (i.e. local clocks may have different values).

Syntax We will use the following general definitions:

- $\mathcal{I}$ is a denumerable set of pure names (identifiers) containing the undefined label $\perp$;
- M is a finite set of message labels $(a, b, c, \dots)$;

– V is a finite set of first order variables (identifiers) that are part of the local state of users $(x, y, \dots)$;
– S is a finite set of second order variables (a set of identifiers) that are part of the local state of users $(s, t, \dots)$.

We then define $\texttt{Act}(M, V, S)$ the set containing the following elements:

1. *Conditions:*
   – $eq(x, y)$ and $noteq(x, y)$, where $x, y \in V$;
   – $in(x, m)$ and $notin(x, m)$, where $x \in V$ and $m \in M$;
2. *Operations:*
   – $local$;
   – $new(x)$, where $x \in V$;
   – $bcast(m, x)$ and $rec(m, x)$, where $m \in M, x \in V$;
   – $add(x, s)$, where $x \in V, s \in S$;
   – $reset(s)$, where $s \in S$;
   – $reg(m, s)$, where $m \in M, s \in S$.

A protocol is defined as an automaton $P = \langle Q, M, V, S, R, q_0 \rangle$, where $q_0 \in Q$ and $R \subseteq Q \times \texttt{Act}(M, S, V) \times Q$. We assume here that $bcast$ and reg actions are always defined on distinct labels in M.

Semantics Consider a protocol $P = \langle Q, M, V, S, R, q_0 \rangle$. A process configuration of P is a triple $pc = \langle q, \delta, \gamma \rangle$ where:

– $q \in Q$ is the current user state;
– $\delta : V \to \mathcal{I}$ is the current evaluation of first order variables;
– $\gamma : S \to \mathcal{P}(\mathcal{I})$ is the current evaluation of second order variables.

We denote by $\mathcal{PC}$ the set of process configurations. A global configuration of P is then a pair (K, Θ), where $K \in \mathbb{N}^{\mathcal{PC}}$ is a finite multiset of process configurations and $\Theta : M \mapsto \mathcal{P}(\mathcal{I})$ corresponds to a centralized memory. $\mathcal{GC}$ denotes the set of global configurations.

For $\sigma \in \mathcal{GC}$, we will use $ids(\sigma)$ to denote the set of pure names occurring in σ. The operational semantics is then defined as the minimal relation $\to \subseteq \mathcal{G} \times \mathcal{G}$ that satisfies the following property definitions in which will use $\uplus$ to denote multiset union:

– $\langle \{\langle q, \delta, \gamma \rangle\} \uplus K, \Theta \rangle \to \langle \{\langle q', \delta, \gamma \rangle\} \uplus K, \Theta \rangle$ if one of the following conditions is satisfied
  • $\langle q, local, q' \rangle \in R$,
  • $\langle q, eq(x, y), q' \rangle \in R$ and $\delta(x) = \delta(y)$ [for $noteq(x, y)$ $\delta(x) \neq \delta(y)$],
  • $\langle q, in(x, m), q' \rangle \in R$ and $\delta(x) \in \Theta(m)$ [for $notin(x, y)$ $\delta(x) \notin \Theta(m)$].
– $\sigma = \langle \{\langle q, \delta, \gamma \rangle\} \uplus K, \Theta \rangle \to \langle \{\langle q', \delta', \gamma \rangle\} \uplus K, \Theta \rangle$ if $\langle q, new(x), q' \rangle \in R$ and $\delta'(x) = id \notin ids(\sigma), id \neq \bot$, and $\delta'(y) = \delta(y)$ for $x \neq y$.
– $\langle \{\langle q, \delta, \gamma \rangle\} \uplus K, \Theta \rangle \to \langle \{\langle q', \delta, \gamma' \rangle\} \uplus K, \Theta \rangle$ if $\langle q, add(x, s), q' \rangle \in R$ and $\gamma'(s) = \gamma(s) \cup \{\delta(x)\}$ and $\gamma'(t) = \gamma(t)$ for $t \neq s$.

- $\langle \{\langle q, \delta, \gamma\rangle\} \uplus K, \Theta\rangle \rightarrow \langle \{\langle q', \delta, \gamma'\rangle\} \uplus K, \Theta\rangle$ if $\langle q, reset(s), q'\rangle \in R$ and $\gamma'(s) = \emptyset$ and $\gamma'(t) = \gamma(t)$ for $t \neq s$.
- $\langle \{\langle q, \delta, \gamma\rangle\} \uplus K, \Theta\rangle \rightarrow \langle \{\langle q', \delta, \gamma\rangle\} \uplus K, \Theta'\rangle$ if $\langle q, reg(m, s), q'\rangle \in R$ and $\Theta'(m) = \Theta(m) \cup \gamma(s)$ and $\Theta'(n) = \Theta(n)$ for $n \neq m$.
- $\langle \{\langle q, \delta, \gamma\rangle, \langle q_1, \delta_1, \gamma_1\rangle, \ldots, \langle q_k, \delta_k, \gamma\rangle_k\} \uplus K, \Theta\rangle \rightarrow$ $\langle \{\langle q', \delta, \gamma\rangle, \langle q'_1, \delta'_1, \gamma_1\rangle, \ldots, \langle q'_k, \delta'_k, \gamma_k\rangle\} \uplus K, \Theta\rangle$ if $k \geq 0$ holds, the transition $\langle q, bcast(m, x), q'\rangle \in R$, and, for all $i \in [1, k]$, we have that $\langle q_i, rec(m, x_i), q'_i\rangle \in R$ and $\delta'_i(x_i) = \delta(x)$ and $\delta'_i(y) = \delta_i(y)$ for $y \neq x_i$.

2.1 Computations

Given a protocol P and an initial global configuration σ_0, a computation is a possibly infinite sequence of configurations $\sigma_0\sigma_1 \ldots$ s.t. $\sigma_i \rightarrow \sigma_{i+1}$ for $i \geq 0$. In this paper we are particularly interested in verification problems for protocol instances with a finite but arbitrary number of processes. In other words we will consider the infinite set of initial configurations I consisting of initial configurations containing any (finite) number of processes and study reachability problems that might expose anomalies during protocol execution.

2.2 Verification Problems

For the resulting model, we are interested in safety properties for parameterized version of a protocol instance. More specifically, given a protocol definition P and a given process state $q_e \in Q$ that denotes an error state, we would like to verify that global configurations that contain occurrences of q_e are unreachable starting from any possibile initial configuration (i.e. independently from the number of process instances). A similar property can be formulated by adding an error flags to the server state instead of selecting a special error state in Q. We are also interested in properties expressed by violations in which global configurations contain a process in state q_e while all other processes are in states contained in a given set of control locations. We will refer to these properties as control state reachability and constrained control state reachabiliy, respectively. Violations of control state reachability can be defined using quantified assertions, such as there exists a process i in control state q_e, without further constraints on global memory of local states. Similarly, violations of constrained properties can be expressed by adding universally quantified conditions on the control states of other processes.

In the rest of the paper we will illustrate, with the help of a case study, how to encode our formalism in the Cubicle input language and how to verify control state reachability problems by a refinement of the symbolic exploration procedure provided by the tool.

3 Case Study: Contact Tracing Protocols

We present below a formal specifications of the protocol scheme of Contact Tracing applications abstracting away identifier creation, i.e., assuming that user app

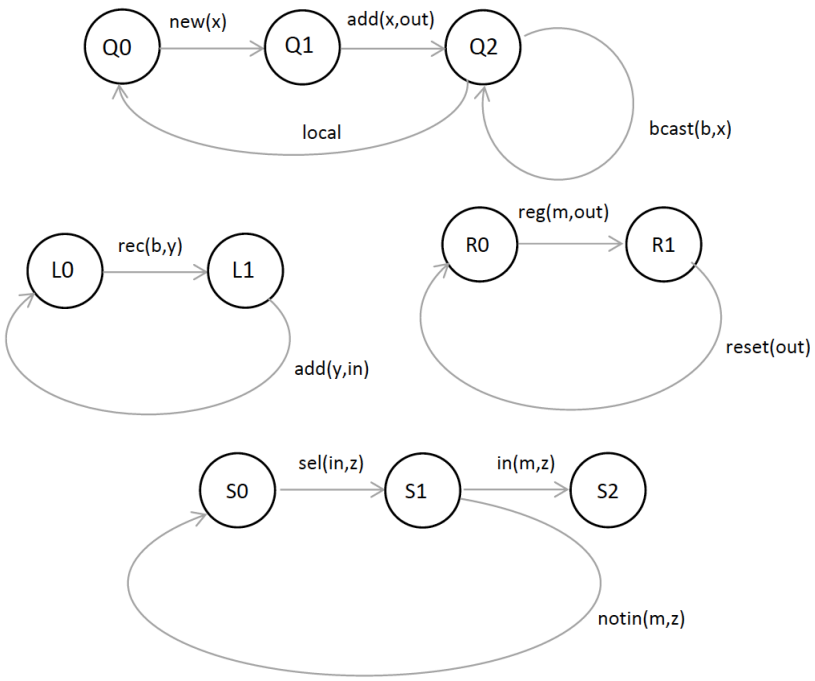

Fig. 1. Basic behavior of the different phases.

generate fresh identifiers, and reasoning within the validity epoch of identifiers, i.e., those maintained in local memory. The system is composed of a finite but arbitrary number of user apps and of a server modeled via a centralized memory. Each user apps has local variables $V = \{x, y, z\}$ and $S = \{in, out\}$. Furthermore, $M = \{b, m\}$ where b denotes beacon messages broadcasted by user apps, and m denotes report messages sent to the centralized memory. The user app starts their part of protocol in state q_0 and operate in three different modes. In emitter mode the app generates and emits fresh beacons:

- $\langle q_0, new(x), q_1 \rangle$, the user app generates a fresh identifier and stores it in the local variable x.
- $\langle q_1, add(x, out), q_2 \rangle$, in q_1 the fresh identifier stored in x is added to the local memory out.
- $\langle q_2, bcast(b, x), q_2 \rangle, \langle q_2, local, q_0 \rangle$, in q_2 the user app either emits the message (b, x) or returns to the initial state in order to apply a rotation scheme for the beacon identifiers or perform other steps.

In reception mode the app, while in any state $l_0 \in Q \setminus \{r_1, s_2\}$, receives a beacon and stores in the local memory in and returns the current state:

- $\langle l_0, rec(b, y), l_1 \rangle$ in state l_0 the user is always ready to receive beacons.

– $\langle l_1, add(y, in), l_0 \rangle$, in state l_1 the beacon is stored in the local memory in.

In report mode, enabled only in state $r_0 = q_0$, the app sends its local memory out to the server and moves to the halt state r_1: $\langle q_0, send(m, out), r_1 \rangle$.

In query mode, enabled only in state $s_0 = q_0$, the user app selects a beacon z from the local memory in, i.e., $\langle s_0, sel(in, z), s_1 \rangle$, sends it to the server to check if it has been reported by another app or not. In the former case it moves to the halting state s_2, i.e, $\langle s_1, in(m, x), s_2 \rangle$. In the latter case it returns to the initial state s_0.

4 From Data Automata to Cubicle

Let us now discuss how to model a protocol definition $P = \langle Q, M, V, S, R, q_0 \rangle$ in Cubicle. A global configuration is a pair (K, Θ), where K is a finite multiset of process configurations and Θ corresponds to a centralized memory. The encoding if Θ is quite immediate since we can represent the global memory as a finite set of unbounded arrays $A_{m_1}, \ldots, A_{m_n}$ with cells of Boolean type one for each $m_i \in M$. The array A_m is such that $A_m[x] = True$ if and only if $x \in \Theta(m)$ for each pair x, m.

For the encoding of K we can proceed in several different ways. Remember that a process configuration is a triple $pc = \langle q, \delta, \gamma \rangle$, in which $q \in Q$ but is the current user state, δ is the current evaluation of first order variables; and γ is the current evaluation of second order variables. One possible encoding is defined as follows:

– To represent control states, we introduce an unbounded array $state$ that associates an enumerative type associated to Q to represent the current state of each process, i.e., $state[i] = q$ if process i is in state q.
– To represent the current state of a first order variable v we can introduce an unbounded array val_v such that $val_v[i] = d$ if $\delta(v) = d$ holds in pc_i (in process i).
– To represent the current state of a second order variable s we can introduce an unbounded array val_s such that $val_s[i, d] = True$ if and only if $d \in \gamma(s)$ holds in pc_i (in process i).

Transition rules can then be expressed via Cubicle transition rules. In particular, to model send operation involving second order variables, we can use global operations on arrays, in which all cells of an arrays are copied into another one (whole-place operations in Petri net languages). For the sake of brevity, we illustrate the idea with the help of an example inspired to the contact tracing protocols described in the previous sections.

4.1 Encoding of Contact Tracing Protocol

To encode our protocol specification, we introduce the following array declarations for modelling individual users:

```
array Pos[proc] : bool
array Contact[proc]  : bool
array In[proc,proc] : bool
array Out[proc,proc] : bool
```

`Pos[i]` defines the current value of positive status of user `i`. `Contact[i]` defines the current value of the variable that defines whether or not user `i` has detected a contact with a positive user. `In[i]` defines the current value of the `In` set variable. Namely, `In[i,b]` is true if and only if b is in the In local memory of process i. `Out[i]` defines the current value of the `Out` set variable. Namely, `Out[i,b]` is true if and only if b is in the Out local memory of process i.

The server can be modelled using a single array

```
array Server[proc] : bool
```

in such a way that `Server[b]` is `True` only when b is in the global memory Θ. Finally, we can model freshness of pure names (identifiers) but using an additional array in which we mark used names:

```
array Used[proc] : bool
```

We remark here that all above introduce declarations define potentially unbounded mono- or bi-dimensional data structures. In other words the array-based formalism allows us to symbolically represent and reason on an infinite family of transition systems, each one with finitely many process instances.

The emission of a beacon requires a broadcast operation that involves a set of non-deterministically selected nodes. To model this step, we introduce a special global variable `Lockstep` that is initially set to `False` and used to split transitions in locksteps (e.g. to perform broadcast operations within a single lockstep).

```
array Lockstep[proc] : bool
```

Let us now illustrate the way we can encode the Protocol transitions using Cubicle parametric rules. The set of initial global configurations is defined as follows:

```
init (x y) {
  Pos[x]     = False  &&
  Contact[x] = False  &&
  In[x,y]    = False  &&
  Out[x,y]   = False  &&
  Used[x]    = False  &&
  Server[x]  = False &&
  Lockstep[x] = False
}
```

Here x, y are universally quantified variables that define constraints on the array cells in every possible instance of an initial configuration, i.e., with one formula

we state a constraint over infinitely many initial configurations. Initially, all cells are set to $False$ (no used names, empty memory, no reported positive user, no reported contact).

Emission To model the emission of a beacon via a broadcast message sent to a set of nodes, non deterministically selected from those in the current global states, we introduce a special `Lockstep` array indexed on beacon identifiers that will be used to split a single broadcast operations in a sequence of steps that do not interfere with other operations. In other words the cell `Lockstep[b]` is set to `True` only during the emission of beacon `b`. Reception rules will use this cell as a guard for firing the rule. All other operations will use the `Lockstep` flags to avoid to interfere with the broadcast operation.

More specifically, we introduce the following ignition rule:

```
transition out(u b)
requires {
 Pos[u]  = False  &&
 Used[b] = False  &&
 Lockstep[b] = False &&
 forall_other x. (Lockstep[x] = False)
}
{
 Used[b]  := True;
 Out[u,b] := True;
 Lockstep[b] :=True
}
```

Here u, b are existentially quantified variables ranging on pairs of distinct values. Notice that the guard requires the process u to be non positive, freshness of beacon b, and b to be not present in the local Out memory. We avoid interferences with other (broadcast or non broadcast) operations via a guard that ensures that no lockstep have been activated in the current state. This guard is expressed using the universally quantified formula of the Cubicle language:

```
Lockstep[u] = False && forall_other x. (Lockstep[x] = False)
```

The effect of the rule is to emit the beacon, i.e., set it to $Used$, and store it in the Out memory. We also set `Lockstep` to `True` to start a lockstep in which to include all subsequent reception steps with other nodes in the current global state.

We can also add the following rule in order to cover the range of all identifiers as a potentially emitted beacon.

```
transition out(u)
requires {
 Pos[u]  = False  &&
 Used[u] = False  &&
```

```
 Out[u,u] = False &&
 Lockstep[u] = False &&
 forall_other x. (Lockstep[x] = False)
}
{
 Out[u,u] := True;
 Lockstep[u] :=True;
 Used[u]  := True
}
```

The reception of a beacon, within a given lockstep, is modeled via the following rules:

```
transition in(u b)
requires {
 Lockstep[b] = True &&
 Pos[u]  = False &&
 In[u,b] = False
}
{  In[u,b] := True }

transition in(b)
requires {
 Lockstep[b] = True &&
 Pos[b]  = False &&
 In[b,b] = False
}
{
  In[b,b] := True
 }
```

Each lockstep is non deterministically terminated via the following rule:

```
transition in(b)
requires  {
  Lockstep[b] = True
}
{
  Lockstep[b] := False
 }
```

This rule terminates the selection of the finite set of receivers of a certain beacon emitted at the beginning of the lockstep.

Report The report rule involves a global operation in which the local memory of a process is copied to the global memory of the server. The copy operation can be expressed using the following operation

```
Server[b] := case
   | Out[u,b] = True : True
   |  _              : Server[b];
```

This action set to `True` every cell of the array Server associated to a beacon `b` that is included in the `Out` local memory of the sender that sends the report.

```
transition report(u)
requires {
 Pos[u] = False &&
 Contact[u] = False &&
 Lockstep[u] = False &&
 forall_other x. (Lockstep[x] = False)
}
{
 Pos[u] := True;
 Server[b] := case
   | Out[u,b] = True : True
   |  _              : Server[b];
}
```

This action set to `True` every cell of the array Server associated to a beacon `b` that is included in the `Out` local memory of the sender that sends the report.

Query In the query rule a process and the server are supposed to synchronize on a given beacon request. We can simplify the encoding of the selection and rendezvous step by simply checking if there exists a beacon `b` in the local memory that also occurs in the global memory.

```
transition query(u b)
requires {
 Pos[u]     = False &&
 Contact[u] = False &&
 In[u,b]    = True &&
 Server[b]  = True &&
 Lockstep[b] = False &&
 forall_other x. (Lockstep[x] = False)
}
{
  Contact[u] := True;
 }
```

This action set to `True` every cell of the array Server associated to a beacon `b` that is included in the `Out` local memory of the sender that sends the report. Although this rule does not mimic all steps specified in the protocol (selection of a beacon from `In`, request to the server, ack or nack to the user) it captures the essence of these operations in a more concise form.

5 Validation in Cubicle

The Cubicle verification engine is based on symbolic backward exploration. Cubicle operates over sets of existentially quantified formulas called cubes. Formulas containing universally quantified formulas (generated during the computation of predecessors) are over-approximated by existentially quantified formulas. In other words Cubicle applies monotone abstraction [2] and over-approximates predecessors via upward-closed sets of configurations. Cubicle applies different strategies and heuristics during backward search such as mixing breadth and depth-first search or injecting over-approximations of backward reachable states that are checked to be unreachable in a finite instance of the system. Infinite sets of unsafe states (bad configurations) are symbolically defined by using $unsafe$ constraints in which all variables are implicitly existentially quantified. In our case studies we are interested in checking the consistency between local and global information. More specifically, when user U queries the server, she/he supposed to receive a notification only in presence of at least one positive user who emitted a beacon that has been received by U. The server does not maintain any association between identities and beacons collected in the log. Consistency of global and local states combined with the privacy preservation property of the protocol can be tested by adding an Error state and a special rule in which we only compare local states of different users. Namely, the rule generates an error whenever there exist a user U in the `Contact` state without any positive user in the global system. If this happens the protocol does not trace physical contacts in correct way. The property is expressed by adding a global `Error` flag

```
Error: bool
```

together with the following rule:

```
transition bad(v)
requires {
  Contact[v] = True &&
  forall_other u. (Pos[u] = False)
 }
{
  Error := True;
}
```

To validate the considered properties, we define the following assertion that specifies bad configurations in the sense of control state reachability,

```
unsafe () { Error = True }
```

i.e., configurations of any size in which `Error` is `True`.Due to the presence of universally quantified conditions both in the protocol rule and in the precondition of the error rule, there is no termination guarantee on the execution of the verification task in Cubicle. In our case-study Cubicle, after visiting 41 nodes and computing 364 fixpoint tests with 864 solvers calls, returns a spurious trace. The

problem is due to the over-approximation introduced via monotone abstraction. Preconditions with universally quantified guards are transformed in postconditions and thus they are not propagated through different steps of predecessor computations. Since in the correctness property of the protocol local states are put in relation via hidden data (not explicitly mentioned in control states) stored in the server memory, it seems necessary to strengthen the symbolic reasoning procedure with some form of propagation of universally quantified conditions.

One solutions comes from the combination of the existentially quantified formulas used to represent sets of states in Cubicle (cubes) with other forms of constraints on the global configuration e.g. constraints inspired to counting abstraction used in Petri nets.

More in detail, we enrich our assertions by adding counters that keep track of the number of users in a given state. For our purposes, we will focus only on a counter associated to the `Pos` flag. We now add a global variable `Count` whose initial state is defined as `Count=0`. The counter is updated in the report rule and used as precondition in the error rule as specified below.

```
transition report(u)
requires {
 Lockstep[u] = False &&
 forall_other x. (Lockstep[x] = False) &&
 Pos[u] = False &&
 Contact[u] = False &&
 0 <= Count
}
{
 Count := Count + 1;
 Pos[u] := True;
 Server[b] := case
   | Out[u,b] = True : True
   |  _              : Server[b];
}

transition bad(v)
requires {
  Pos[v] = False  &&
  Contact[v] = True &&
  Count = 0
}
{
 Error := True;
}
```

The enriched assertional language combining array formulas (for expressing precise properties of individual processes and of global variables) and Presburger constraints (for expressing global constraints via the counting abstraction) is the key point in order to enforce termination and prove correctness for the desired

property. Cubicle indeed proves the protocol correct for any number of users and beacons after visiting 10 nodes, computing 138 fixpoint tests and 41 solver calls.

6 Conclusions

In this paper we have presented a formal model of distributed systems that can be used to specify protocols that combine bluetooth and network communication and make use of notification systems based on a central server. Our formalism is an extension of automata-based formalisms in which individual process have a local memory with first-order and second-order variables. Broadcast communication is inspired to Broadcast Protocols [13, 8], related formalisms as those discussed in [6, 11], and versions with data [10, 14, 3]. The use of a global memory is somehow related to automata based model used for software specification.

Differently from more foundational works, see again [6], in this paper we focus on a possible encoding of the formalism in existing infinite-state model checkers that can deal with both first-order and second-order variables in the local memory of individual processes. More specifically, we focus on the SMT-based tool Cubicle that provides a logic over nested unbounded arrays as input specification language. To handle verification problems such as constrained control state reachability, needed to specify consistency protocols in contact tracing protocols, we proposed an extension of the symbolic backward procedure that can deal with complex assertions with universally quantified formulas. The proposed approach can be viewed as fully declarative, since it uses predicates in a combination of logical formalisms, counterpart of ad hoc approaches such as those proposed in [9] in which constraints on global configurations are embedded into the verification algorithm instead of being expressed via assertions combining different types of predicates. Although not yet implemented, the above mentioned strategy could be incorporated into the Cubicle algorithm for instance by associating counters to specific control states and by generating counter manipulation operations in each transition rule via a preliminary static analysis of the Cubicle specification as done in our example via human ingenuity.

References

1. P. A. Abdulla, M. F. Atig, G. Delzanno, M. Montali, and A. Sangnier. On the Formalization of Decentralized Contact Tracing Protocols. In *Proceedings of the 2nd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the Bolzano Summer of Knowledge 2020 (BOSK 2020), September 25, 2020*, volume 2785 of *CEUR Workshop Proceedings*, pages 65–70. CEUR-WS.org, 2020.
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, G. Delzanno, and A. Rezine. Approximated parameterized verification of infinite-state processes with global conditions. *Formal Methods in System Design*, 34(2):126–156, 2009.

4. http://alt-ergo.lri.fr Alt-Ergo.
5. G. Avitabile, V. Botta, V. Iovino, and I. Visconti. Towards Defeating Mass Surveillance and SARS-CoV-2: The Pronto-C2 Fully Decentralized Automatic Contact Tracing System. *IACR Cryptol. ePrint Arch.*, 2020:493, 2020.
6. 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.
7. 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.
8. G. Delzanno, J. Esparza, and A. Podelski. Constraint-Based Analysis of Broadcast Protocols. In *CSL'99*, volume 1683 of *LNCS*, pages 50–66. Springer, 1999.
9. G. Delzanno and A. Rezine. A Lightweight Regular Model Checking Approach for Parameterized Systems. *STTT*, 14(2):207–222, 2012.
10. G. Delzanno, A. Sangnier, and R. Traverso. Parameterized Verification of Broadcast Networks of Register Automata. In *RP*, pages 109–121, 2013.
11. 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.
12. David L. Dill. The Mur*phi* Verification System. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, 1996.
13. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 352–359, 1999.
14. R. Lazic, T. Newcomb, J. Ouaknine, A. W. Roscoe, and J. Worrell. Nets with Tokens which Carry Data. *Fundam. Inform.*, 88(3):251–274, 2008.
15. 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.
16. Serge Vaudenay. Centralized or Decentralized? The Contact Tracing Dilemma. *IACR Cryptol. ePrint Arch.*, 2020:531, 2020.