Modelling and Verification of a Health Cloud Management Protocol

Almo Cuci[1], Umar Ozeer[2], and Gwen Salaün[1]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France
[2] Euris Cloud Santé, Paris, France

Abstract. The digitisation of personal health information (PHI) through electronic health record (EHR) is now widely adopted due to their efficiency in terms of cost, storage, processing, and the subsequent quality of delivering patient care. However, security concerns remain one of its major setback. In order to handle EHR, institutions need to comply with their local government security regulations. These regulations control to which extent health data can be processed, transmitted, and stored as well as define how misuses are addressed. ϕ-comp has been proposed as an industrial solution for monitoring, assessing, and evaluating the compliance of health applications with respect to defined security regulations. ϕ-comp is able to assess the level of security risk of an application at runtime and to automatically perform the required mitigation actions to recover a compliant environment. Since the risk associated to sensitive health data is critical, there is a need of guarantees in terms of correctness of the ϕ-comp approach. In this paper, we first present a formal specification of ϕ-comp representing all the components of the solution as well as their behaviour, that is, the way they all interact together to implement the whole approach from monitoring to mitigation. In a second step, some important properties of interest are formalised and analysed using model checking techniques on several realistic applications.

1 Introduction

An Electronic health record (EHR) consists of a digital version of a patient's personal health information (PHI) such as medications and laboratory test results. The digitisation of healthcare has revolutionised the efficiency of the industry in terms of storage, transmission, and processing capabilities as well as in terms of quality, cost, and time effectiveness of patient care. In order to host an environment (infrastructure, application, services, solutions) that manipulates PHI via EHR, institutions need to comply to the security regulations enforced by the local regulating authorities. These regulations control to which extent personal health information can be processed, transmitted, shared, and stored. In addition, control audits can take place at any given time to check whether the hosted platforms respect the high security protocols imposed. In such a context, all activities (system logs, application logs, user activities, operations performed, etc.) should be logged and stored in a secure manner.

ϕ-comp [14, 15] is a security compliance monitoring and management solution for sensitive health data environment, which respects existing security regulations. It monitors, computes, and evaluates the security compliance of health applications and their underlying infrastructures. The monitored data are reported and classified with respect to four security areas (confidentiality, integrity, availability, traceability). These data are analysed by security area and further evaluated into three levels of risk (identified by colors), namely nominal behaviour (blue), potential threat (orange), and non-compliant behaviour (red). In the latter case, risk mitigation actions are automatically performed so as to attenuate the level of risk and restore a compliant behaviour. System administrators are also notified in case of non-compliance, so that manual interventions can be carried out if required. Monitored data as well as performed mitigation actions are logged for *a posteriori* auditing. These logs are used for generation of compliance reports for the hosting institution and for on-demand request from supervising authorities.

Since health data and associated security risks are crucial, there is a need of guarantees in terms of correctness of the ϕ-comp approach. In this paper, we first present a formal specification of ϕ-comp representing all the components of the solution's architecture as well as the behaviour of all components including the way they interact together. This specification step is achieved using the LNT process algebraic specification language [3, 9]. In a second step, some important properties of interest are formalised using the MCL logic [13] and analysed using model checking techniques on several realistic applications. As far as analysis is concerned, we rely on the CADP verification toolbox [8], which provides powerful model checking tools for automating these analysis steps. The experiments we carried out on these applications confirmed that all properties were satisfied, thus convincing the protocol's designers of the correctness of the ϕ-comp solution.

The rest of the paper is organized as follows. Section 2 presents the ϕ-comp solution with more details. Section 3 describes the formal specification of the ϕ-comp protocol. Section 4 introduces the definition of properties and their verification using model checking techniques. Section 5 surveys related work and Section 6 concludes the paper.

2 Health Management Protocol

ϕ-comp [14, 15] is a security compliance monitoring and management solution, which was designed to target cloud computing environments that house sensitive health data. In the rest of this section, we successively present the application model, the architecture of the solution and we give a short description of the components involved in the ϕ-comp approach.

2.1 Application Model

The application model corresponds to an abstraction of the computing environment and consists of two levels: the infrastructure and the application.

An infrastructure is modelled by a set of virtual machines (VM) and the physical network connecting them. The infrastructure hosts the application, that is, it provides the physical resources such as CPU, memory, disk, and bandwidth for executing applicative entities. Figure 1 illustrates an instantiation of an application model consisting of three VMs and two physical networks (net1 and net2) connecting them.

An application is modelled by a set of software entities and bindings. A binding implements the communication between a couple of software entities and is directed according to the functional dependency between the entities. A binding can be optional or mandatory. Each infrastructure and applicative entity is uniquely identified. A software entity is functional and compliant when all of its mandatory bindings are running and are compliant. Therefore, the non-compliance of a software entity impacts other entities which depend on it. The compliance of a software entity is not affected by its optional dependencies. Note that there are no cycle of dependencies between software entities. Figure 1 depicts three software entities. VM1 hosts a front-end HTTP server (e.g., Apache), VM2 hosts an application server (e.g., Tomcat), and VM3 hosts a relational database system (e.g., MariaDB). These software entities are connected through bindings (b1 and b2) which allow communication between them and give the direction of the functional dependencies.

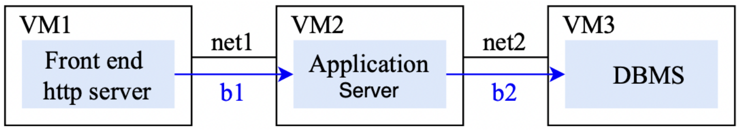


Fig. 1: An Instance of an Application Model

2.2 Compliance Management Protocol

Figure 2 gives an overview of the architecture of the ϕ-comp solution. The components involved in this architecture are first introduced. In a second step, we describe how they interact together in order to fulfil the compliance management's goals.

Probes and Enforcers are the only participants which are deployed on the target environment to be managed. The role of Probes is to report monitoring data whereas Enforcers perform actions on the environment so as to enforce security measures and attenuate the level of risk. The rest of the participants are deployed on a dedicated infrastructure.

The Preliminary Data Analyser (PDA) retrieves data reported by Probes, and subsequently formats and enriches these data. The PDA computes a preliminary security risk based on these monitoring data.

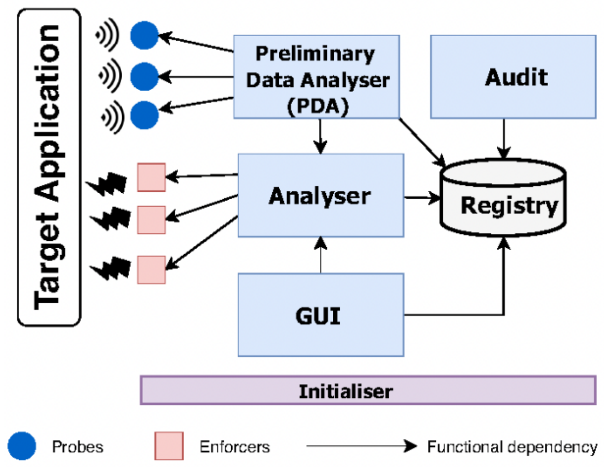


Fig. 2: Architecture of the Protocol

The Analyser evaluates the risk of the target environment and determines whether an entity is compliant. The Analyser decides the mitigation actions in case of non-compliance.

The Registry stores the data reported by probes, the changes in security risk, and all the mitigation decisions taken by the Analyser. This aims at keeping a log of the compliant/non-compliant behaviour of the target environment as well as the decisions and mitigation actions performed in case of non-compliance. The data stored on the registry are kept for a customisable period of time with respect to the required regulations (e.g., one year for HDS regulation).

The Graphical User Interface (GUI) displays the current security risk of each entity of the target environment in real time and indicates whether they are compliant. The GUI allows system administrators to work collaboratively for handling security incidents and carrying out investigations.

The Audit component allows the generation of security reports based on the behaviour of the target application with respect to the different security areas. Reports can be generated at a regular period (e.g., at the end of every week) or on demand.

The Initialiser component is responsible for configuring and setting up ϕ-comp. It determines the placement and deployment of probes and enforcers on the target environment. The Initialiser ensures the consistent start up of the different participants of ϕ-comp with respect to their functional dependencies. It is worth noting that the Initialiser is not involved in the initial setup of the target environment.

Let us now focus on how all these components work together to monitor the application, analyse identified risks, and mitigate them in order to maintain compliance. Figure 3 gives an overview of the procedure for compliance management by the participants of ϕ-comp. Monitoring is performed by Probes which report information to the PDA. These data are subsequently classified into the corresponding security risk areas, and a preliminary risk is established by the PDA for each security area. Assuming that a target environment is initially compliant, for each change in its preliminary risk, the Analyser is notified. The Analyser evaluates the current security risk relative to each security area for each infrastructure and applicative entity into three levels, namely blue, orange, and red. If the risk evaluated by the Analyser is blue, no actions are undertaken. A risk evaluated as orange represents a compliant behaviour but a potential threat. In this case, risk mitigation actions are highly recommended but not compulsory until the evaluated risk becomes red. On the other hand, if the risk evaluated is red, the Analyser computes the impact of the risk. In the case of an infrastructure entity, the impact is localised whereas in the case of a software entity, the impact is propagated to all mandatory dependent entities. A red risk for a software entity thus means that its dependent entities are also assigned a red risk. A red risk implies that mitigation actions have to be performed to attenuate the level of risk and recover a compliant environment. These mitigation actions are determined by the Analyser. The GUI is then updated with the current risk and system administrators are notified of the non-compliance. Mitigation actions are forwarded to Enforcers which act on the target environment.

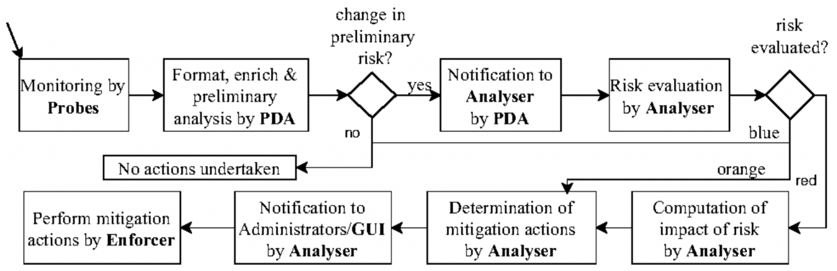

change in preliminary risk?
risk evaluated?
Monitoring by Probes
Format, enrich & preliminary analysis by PDA
yes
no
Notification to Analyser by PDA
Risk evaluation by Analyser
blue
orange
red
No actions undertaken
Perform mitigation actions by Enforcer
Notification to Administrators/GUI by Analyser
Determination of mitigation actions by Analyser
Computation of impact of risk by Analyser

Fig. 3: Overview of Compliance Management

3 Formal Specification

In this section, we introduce the LNT specification for the ϕ-comp protocol. We chose LNT [3, 9] as specification language because it is expressive enough and adequate for formally describing communication protocols as the protocol presented beforehand in this paper. Moreover, it is equipped with CADP [8], a rich toolbox for analysing LNT specifications using model checking techniques.

LNT is an extension of LOTOS [10], an ISO standardised process algebra, which allows the definition of data types, functions, and processes. Processes define actions (that can come with incoming or outgoing parameters) and these actions can be organised using several operators, among which: sequential composition (**;**), conditional statement (**if-then-else**), hiding (**hide**) that hides some action in a behaviour, non-deterministic choice (**select**), parallel composition (**par**) where the communication between the involved processes is carried out by rendezvous on a list of synchronised actions, looping behaviours described using process calls or explicit operators (**while**, **for**, **loop**), and assignment (**:=**) where the variable should be defined beforehand (either in a **var** block or as a formal parameter). LNT is formally defined using operational semantics based on Labelled Transition Systems.

In the rest of this section, we present how the different elements of the ϕ-comp protocol are specified using LNT. The specification consists of about 2000 lines of LNT.

3.1 Application Model

First of all, a model of the application is specified in LNT. Data types are used to describe the application model, that is, VMs, software entities or components, and bindings between components. More precisely, an application consists of a set of VMs, a physical network connecting these VMs, and a set of bindings connecting the components hosted on the VMs. Each binding also has a Boolean parameter indicating whether this binding is mandatory (if not, it is optional). A VM consists of an identifier and a set of components.

Figure 4 gives an excerpt of LNT specification describing the virtual machines and bindings corresponding to the application shown in Figure 1. One can see that each VM hosts a single component (C1, C2, etc.). The physical network is not made explicit in this example for simplification purposes.

```
var C1 , C2 , C3 : COMPONENT,
    ALLVM        : VMs,
    ALLBDN       : BDNs,
    APP          : APPLICATION
in

 C1 := COMPONENT(1) ;
 C2 := COMPONENT(2) ;
 C3 := COMPONENT(3) ;

 ALLVM := VMs ( { VM( 1, "FRONT", {C1}),
                  VM( 2, "SERVER", {C2}),
                  VM( 3, "DBMS", {C3}) } ) ;

 ALLBDN := BDNs (
   { BINDING( VM( 1, "FRONT", {C1}) , VM( 2, "SERVER", {C2}), TRUE),
     BINDING( VM( 2, "SERVER", {C2}) , VM( 3, "DBMS", {C3}), TRUE),
     BINDING( VM( 1, "FRONT", {C1}) , VM( 3, "DBMS", {C3}), FALSE) } );

 APP := APPLICATION( "model-316", ALLVM, ALLBDN ) ;

end var
```

Fig. 4: Example of Application Model in LNT

3.2 Management Protocol

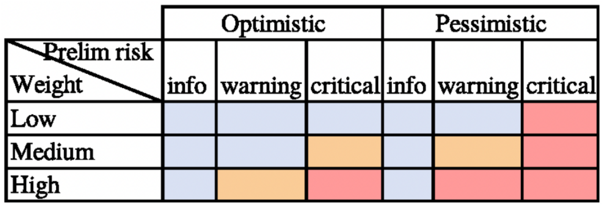
In this section, we first present the specification of each participant of the management protocol, and we terminate with the main process describing how all participants interact together to model the whole management protocol. It is worth noting that several components (Audit, Registry, GUI and Initialiser) are left outside of the specification because their role does not impact the behaviour of the protocol. In contrast, the following entities are crucial and are made explicit in the specification: Probes, PDA, Analyser and Enforcers.

Probe processes mostly capture observation messages obtained by monitoring the application and are useful to identify problems for each security area. The Probe acts as a listener mechanism. Every probe is identified by a unique identifier and has as parameter the unique identifier of the entity being monitored. Every time a probe raises an alert, this observation comes with a timestamp and a triple (obs, val, ra) corresponding respectively to the observation made by the probe, its value, and the risk area associated with this observation.

The *PDA* process receives observation messages issued by the Probe component. Each message contains information about the security areas, the policy used to evaluate the risk (optimistic or pessimistic) and an entity weight value (low, medium, high). Based on these inputs, the PDA process computes a preliminary risk level. This risk level is specified using multiple *if-else* conditions for each parameter value observed in the respective messages. For the values out of the normal scope, a warning or critical preliminary risk score is assigned as result. More precisely, the preliminary risk has four fields to store the score for each security risk area, and three levels of criticity (information, warning, critical). Once computed, this preliminary risk level is submitted to the next component (Analyser) for a finer analysis, and this message contains the preliminary risk score type, information about the security areas, the policy used for evaluation and the entity weight.

The *Analyser* process can be seen like a decision-making mechanism. It receives a preliminary risk message from the PDA process, and goes further in its evaluation of the risk and the compliance of the entity. Based on the information received, the Analyser process can either decide to ignore the message or to trigger a mitigation action by issuing a message to the enforcers with precise countermeasures to be initiated. The mechanism of taking mitigation action relies on three inputs: the preliminary risk score of a single-dimension security area, the policy used (pessimistic, optimistic) and the entity weight. The output is a final risk label of the color blue, orange, or red. This output is computed using the table given in Figure 5.

If the final score is blue, the Analyser process ignores it. If the compliance score is orange, the Analyser process keeps monitoring the progress of the specific parameter in this security area. If it is red, then mitigation actions are required. For each security area parameter of the entity that is violated, a countermeasure is required in order to mitigate the risk and set the entity back to compliant behaviour.



	Optimistic			Pessimistic		
Prelim risk / Weight	info	warning	critical	info	warning	critical
Low	🟦	🟦	🟦	🟦	🟦	🟥
Medium	🟦	🟦	🟧	🟦	🟧	🟥
High	🟦	🟧	🟥	🟦	🟥	🟥

Fig. 5: Compliance Evaluation Results

Mitigation actions are computed via a dedicated process (called 'mitigate'). This process takes as input the decision issued by the Analyser process (with all the aforementioned parameters), and compute the corresponding mitigation action per security area. As a result, a mitigation action consists of the entity identifier and the action to be taken (e.g., restart VM, add CPU, isolate from internet, log modifications, restart probe, etc.).

These mitigation actions have as target the *enforcers*, which are in charge of actually triggering these actions. Enforcers are able to access to the source of the problem and implement the mitigation actions. Since the specification is abstract, there is no real change of the application. Therefore, the specification can issue and propose countermeasures but they are not actually exploited because the application is not a real one, but just a model of the real one.

In our model, there is one more process called *pbinjector*. This process is used for issuing potential problems occuring on the application and thus for simulating the execution of the ϕ-comp solution when examples of risks occur. To do so, this process simulates different states of the software entities and particularly abnormal values are of interest to us. The states of the entity represent the compliance of the security areas. This process is very useful to simulate many scenarios, that turn out to be interesting in terms of exhaustive verification as we will see in Section 4.

Finally, the main process represents all processes (probes, PDA, analyser, enforcers, and problem injector) in parallel as shown in Figure 6. Beyond putting all processes in parallel, the main process also makes explicit how these processes interact together, thus making explicit the architecture given in Figure 2. As an example, the PDA process synchronize with the other processes on the actions PROBEVENT and ANALYSISEVENT.

4 Verification

In this section, in a first step, we introduce the properties that must be preserved by the health management protocol. These properties are then formally specified using the MCL language and automatically analysed using the CADP model checker. MCL [13] is an extension of the alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. CADP [8] is a rich verification toolbox that implements the results of concurrency theory

```
process MAIN [ PROBEVENT: any, ANALYSISEVENT: any, DECISION: any,
    COUNTERMEASURE: any] is
 var app: APPLICATION in
    app:= application() ; use app;
 par
    PROBEVENT -> pbinjector [PROBEVENT]
 ||
    PROBEVENT -> probe [PROBEVENT]
 ||
    PROBEVENT, ANALYSISEVENT -> pda [PROBEVENT , ANALYSISEVENT ]
 ||
    ANALYSISEVENT , DECISION -> analyser [ANALYSISEVENT , DECISION]
 ||
    DECISION , COUNTERMEASURE -> mitigate [DECISION , COUNTERMEASURE]
 end par
 end var
end process
```

Fig. 6: Main Process in LNT

and is used for the design of asynchronous concurrent systems, such as communication protocols, distributed systems, asynchronous circuits, multiprocessor architectures or web services. It provides a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking. The toolbox offers a compiler for several input formalisms, one of which is LNT. In this work, we particularly rely on the Evaluator model checker, which takes as input an LNT specification and an MCL formula, and returns a Boolean verdict and a counterexample if the property is violated.

Let us now introduce the properties, which were identified as important for the protocol, and are listed below:

- The protocol systematically detects and handles every raised problem (probe).
- Every problem corresponding to a non compliance is followed by a mitigation action.
- A specific non-compliance problem of one entity should not affect the other entities connected to that entity.
- Entities subject to mitigation actions can keep communicating normally with the other connected entities.
- Mitigation actions are only performed when there is a non-compliance
- The correct mitigation action is performed when there is a non-compliance
- When a mitigation action is performed on an entity, the latter becomes compliant

In the rest of this section, we present with more details the two first properties, which are the most important ones.

The first property (Figure 7) is specified as a liveness property (inevitable property), and checks whether every probe raised by the environment is systematically detected. This is written in MCL as follows using the inevitable pattern and indicating using "?any" that any parameter can come with the PROBEVENT action:

```
"MODEL–316.bcg" |= with evaluator4
      INEVITABLE ({ PROBEVENT ?any ... });
expected TRUE;
```

Fig. 7: MCL Property for Probe Detection

The second property (Figure 8) states that every time an action is raised by a probe indicating a non-compliant behaviour for a specific software entity, there must be a reaction from ϕ-comp in the form of a mitigation action. This reaction is actually justified only if the risk is assessed as critical (red) by the analysis process. To verify this, we first specify that the action ANALYSISEVENT has a parameter with a risk score equivalent to the red label and observe if there exists a parameter in the next action (DECISION) that updates the value for the detected problem. This corresponds to the first part of the MCL property given below for illustration purposes, where we can see that a traceability problem is detected and the decision is taken of restarting the corresponding service.

The second part of the property aims at checking whether the response to mitigate the risk is the correct one. To do so, we check that the parameters appearing in the ANALYSISEVENT action are the same as in the DECISION action. In particular, the Probe identifier is extracted from the observation message and we compare if it has the same value as the identifier issued with the mitigation action. If this is the case, it means that every non-compliant behaviour is mitigated by the correct action.

```
"MODEL–316.bcg" = total rename
  "ANALYSISEVENT !PRELIMINARY_RISK (\([0-4]\), \([0-4]\), \([0-4]\),
      \([0-4]\))
                  !MESSAGE2_TRACEABILITY (\([0-4]\), \([0-4]\), .*) .*"
                  -> "ANALYSISEVENT !\1 !\2 !\3 !\4 !\5 !\6",
  "DECISION !RESTART_SERVICE (\([A-Z]*\), .*)
              !MESSAGE2_TRACEABILITY (\([0-4]\), \([0-4]\), .*), .*"
              -> "DECISION !\1 !\2 !\3"
in generation of "INSTANCE4.lnt";

"MODEL–316.bcg" |= with evaluator4 [ true* . { ANALYSISEVENT !3 !3 !3
    !4 !1 !1 } ]
        < (not 'PROBEVENT !.*')* . { DECISION !TRUE !1 !1 } > true;
    expected TRUE;
```

Fig. 8: MCL Property for Systematic Mitigation Action

These properties were analysed on a set of realistic applications where we vary the number of virtual machines, software entities, and bindings. In order to simulate real scenarios, we added a new process (pbinjector, see Section 3) whose role is to inject problems of any kind to the application (confidentiality, integrity, availability, traceability). In its current version, the ϕ-comp approach can support several problems but not at the same time, it handles them one after the other. Therefore, in our verification scenarios, we respected this specific assumption as well.

As a result, all the properties were analysed and turn out to be satisfied on the aforementioned concrete applications (with 4-5 virtual machines, which is usually the number of machines handled by the ϕ-comp framework). Moreover, it is worth noting that this is not necessary to use large applications for verification purposes, because most problems are usually found on small yet pathological applications. It takes up to a few minutes to verify all properties on a realistic application, which is reasonable because the model checking of the protocol is executed off-line.

5 Related Work

In this section, we first present several works dedicated to the specification and verification of management protocols in cloud computing or Fog computing / IoT. At the end of the section, we compare our work with respect to these related works.

In [1,2], the authors present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfiguration steps aim at (dis)connecting ports and changing component states (stopped or started). This protocol supports failures that may occur during the reconfiguration process. The protocol is robust in the sense that all the steps of this protocol preserves a number of architectural invariants. This was proved using the Coq theorem prover. We preferred model checking techniques here, because they are convenient at design time in order to detect possible issues. Theorem proving is interesting when the developers have already at their disposal a stable version of a protocol, and they ultimately want to prove its correctness.

In [6,7,17], the authors present a self-deployment protocol that was designed to automatically configure cloud applications consisting of a set of software elements to be deployed on different virtual machines. This protocol starts the software elements in a certain order, using a decentralised algorithm. It works in a decentralised way, i.e., there is no need for a centralised server. It also starts the software elements in a certain order, respecting important architectural invariants. This protocol supports virtual machine and network failures, and always succeeds in deploying an application when faced with a finite number of failures. A formal specification of the protocol allowed the successful verification of important properties to be preserved.

[5] presents a protocol for reconfiguring applications running in the cloud. The protocol supports the addition of new components and VMs as well as the removal of components and VMs. All these reconfiguration operations are posted through a cloud manager, which is in charge of guiding the reconfiguration of the whole application. This protocol also detects the occurrence of failures and in those situations makes the application restore a global consistent state. In order to ensure its correctness and robustness, the protocol was specified and verified Maude's analysis tools. The set of properties in this work focused on reconfiguration's correctness whereas the main goal here is to ensure the correct monitoring and application of mitigation actions.

[11, 12] propose verification of IoT applications before deployment using model checking techniques. [4] applies infinite-state model checking to formally verify IoT protocols such as the Pub/Sub consistency protocol adopted by the REDIS distributed file system. The verification method is based on a combination of SMT solvers and overapproximations as those implemented in the Cubicle verification tool. This work focuses on the verification of the communication techniques used in IoT systems. Since these protocols involve infinite data structures, the authors chose to use analysis techniques capable of reasoning on infinite state spaces.

[16] focus on a failure management protocol, which allows the supervision of IoT applications and the management of failures. This protocol targets stateful IoT applications in the sense that those applications handle and store data during their execution. When a failure occurs, the protocol detects the failure and restores a consistent pre-failure state of the application to make it functional again. Since designing such distributed protocol is error-prone, it was specified and analysed using formal specification techniques and model checking tools in order to ensure that the protocol respects some important properties. These analysis steps helped to detect several issues and clarify some subtle parts of the protocol.

In this paper, we decided to rely on model checking techniques, as it was the case in [2, 7, 16], because these techniques turn out to be effective in order to validate the correctness of important properties on representative applications. It is worth noting that this work was achieved in collaboration with a company (Euris), and formal verification techniques applied on a protocol used in an industrial context. Last but not least, to the best of our knowledge, this is the first time formal verification techniques are used for cloud platforms dealing with healthcare.

6 Concluding Remarks

In this paper, we have focused on a health management protocol, which allows the storage, monitoring and supervision of health cloud applications. When a problem is detected, it is analysed and, if necessary, a decision is taken to apply a mitigation action. Since this protocol targets health data and applications, it makes it critical and it is therefore crucial that specific properties of correctness are preserved by the protocol. It was decided to rely on formal specification techniques and verification tools in order to ensure that the protocol respects some important properties. In particular, we used a process algebraic specification language and model checking techniques for verifying these properties. The analysis of several applications and scenarios show that the aforementioned properties were satisfied, thus showing that the protocol works as expected.

The main perspective of this work is to improve the current management protocol by making use of blockchains in order to store the health data in a secure way while providing traceability and transparency of the approach. Developing such a solution consists first in designing a protocol for the distributed storage of health data in the cloud by using blockchain technologies. Similarly to what we have done in this paper, we will also make use of model checking techniques for validating the protocol. Beyond analysis and certification, we finally plan to implement the solution using cloud and blockchain technologies.

Acknowledgements. The authors would like to thank Frédéric Lang for his help in the specification and verification of the protocol.

References

1. F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, pages 13–22. IEEE Press, 2013.
2. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer-Verlag, 2011.
3. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA/VASY and INRIA/CONVECS, 153 pages, 2018.
4. G. Delzanno. Formal Verification of Internet of Things Protocols. In *Proc. of FRIDA'18*, 2018.
5. F. Durán and G. Salaün. Robust and reliable reconfiguration of cloud applications. *J. Syst. Softw.*, 122:524–537, 2016.
6. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. Reliable Self-deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM, 2014.
7. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. Reliable Self-deployment of Distributed Cloud Applications. *Softw., Pract. Exper.*, 47(1):3–20, 2017.
8. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
9. H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In J.-P. Katoen, R. Langerak, and A. Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500, pages 3–26, Oct. 2017.
10. ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, ISO, 1989.
11. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. IoT Composer: Composition and Deployment of IoT Applications. In *Proc. of ICSE'19*, pages 19–22. IEEE / ACM, 2019.

12. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. Rigorous Design and Deployment of IoT Applications. In *Proc. of FormaliSE'19*, pages 21–30, 2019.
13. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
14. U. Ozeer. φ comp: An architecture for monitoring and enforcing security compliance in sensitive health data environment. In *Proc. of ICSA'21*, pages 70–77. IEEE, 2021.
15. U. Ozeer and B. Pouye. Risk analysis based security compliance assessment and management for sensitive health data environment. In *Proc. of HealthCom'20*, pages 1–7. IEEE, 2020.
16. U. Ozeer, G. Salaün, L. Letondeur, F. Ottogalli, and J. Vincent. Verification of a Failure Management Protocol for Stateful IoT Applications. In *Proc. of FMICS'20*, volume 12327 of *LNCS*, pages 272–287. Springer, 2020.
17. G. Salaün, X. Etchevers, N. D. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, volume 7740 of *LNCS*, pages 60–79. Springer, 2013.