Modelling and Verification of an Application for Managing Sensitive Health Data

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 [15, 16] 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 [2, 7]. 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 [6], 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 Presentation of φ-comp

φ-comp [15, 16] 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 provides an abstract view of the computing environment, and consists of the infrastructure and the application.

An infrastructure consists of a set of virtual machines (VM) and the physical network connecting them. The infrastructure hosts the application. To do so, it provides the physical resources (CPU, memory, disk, bandwidth, etc.) for executing applicative entities. For illustration purposes, we give in Figure 1 an example of application model consisting of three VMs and two physical networks (net1 and net2) connecting them.

An application consists of a set of software entities and a set of bindings implementing the communication between software entities. A binding is directed according to the functional dependency between the entities and can be optional or mandatory. A software entity is functional and compliant when all of its mandatory bindings are satisfied, that is, connected to functional entities. Consequently, when a software entity is non-compliant, this may impact other entities depending on it. The compliance of a software entity only depends on its mandatory dependencies. To ensure deployment of a compliant application, cycles of (mandatory) dependencies are forbidden.

Figure 1 shows an example of application consisting of 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). The front-end server is connected to the application server via binding b1 and the application server is connected to the database via binding b2. These bindings allow communication between these entities and give the direction of the functional dependencies.

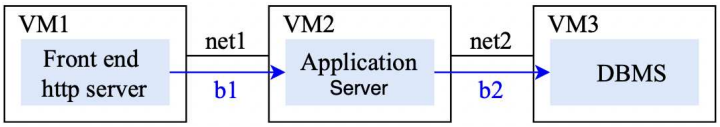

Fig. 1: An Instance of an Application Model

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 deployed on the environment or application to be managed whereas the other ones are part of the proposed solution and thus deployed on a dedicated infrastructure. Probes monitor the application and report observation data periodically. Enforcers perform actions on the environment when asked, in order to apply mitigation actions and attenuate the level of risk.

The Preliminary Data Analyser (PDA) receives observation data reported by Probes. The main goal of the PDA is to compute a preliminary security analysis based on these data.

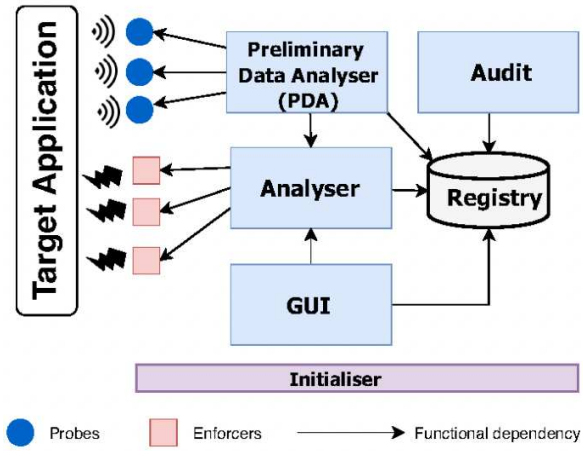

Fig. 2: Architecture of the Protocol

The role of the Analyser is to evaluate the level of risk and to check whether all entities remain compliant. When the Analyser detects a problem or a non-compliance, it computes the required mitigation actions to be applied on the environment.

The Registry aims at storing all information received or computed by φ-comp, that is the data reported by probes, the compliant/non-compliant behaviour of the target environment and all the mitigation actions taken by the Analyser in case of non-compliance. The data stored in the registry are preserved for a determined period of time depending on the required regulations (e.g., one year for HDS regulation).

The Graphical User Interface (GUI) displays information in real time about the monitored application, particularly the current security risk and level of compliance of each entity in the target environment. The GUI allows administrators to work together for handling security issues and carrying out investigations regarding the source of these problems.

The Audit component generates periodically or on demand security reports based on the behaviour of the target application.

The Initialiser component is used for configuring φ-comp by defining the placement and deployment of probes and enforcers on the target environment. The Initialiser is also in charge of ensuring the consistent start up of the different participants of φ-comp by considering their functional dependencies.

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 describes the way these components work together using a workflow description. Probes monitor the application and report observation data to the PDA. Several levels of security risks are computed from these data and a preliminary analysis is established by the PDA for each security area.

Assuming that a target environment is initially compliant, the Analyser is notified whenever there is risk identified in the preliminary analysis. The Analyser then evaluates the current security risk relative to each security area for each infrastructure and applicative entity into three levels represented as colors (blue, orange, and red). If the risk is blue, no mitigation actions is taken. If the risk is orange, there is a potential threat and mitigation actions are recommended but not mandatory. If the risk is red, the Analyser carries out a deeper analysis of the problem and its potential impact on other entities. If the risk concerns a software entity, the impact is propagated to all mandatory dependent entities. In the case of a red risk for a software entity all its dependent entities are also assigned a red risk. A red risk systematically implies the execution of mitigation actions in order to reduce the level of risk and recover a compliant environment. The precise definition of mitigation actions are decided by the Analyser. In case of changes in the compliance level, the GUI is updated and administrators are notified. Mitigation actions are submitted to Enforcers which apply them on the target environment.

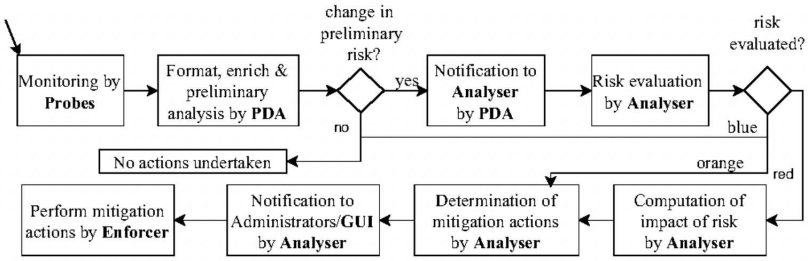

Fig. 3: Overview of Compliance Management

3 Formal Specification

In this section, we introduce the LNT specification for the φ-comp solution. We chose LNT [2, 7] as specification language because it is expressive enough and adequate for formally describing communicating systems as the solution presented beforehand in this paper. Moreover, it is equipped with CADP [6], a rich toolbox for analysing LNT specifications using model checking techniques. It is worth noting that other languages (and verification tools) with similar specificities could have been used instead (e.g., Promela/Spin [8] or mCRL2 [1]).

LNT is an extension of LOTOS [9], 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. Last but not least, it is worth saying that the default communication model in LNT is synchronous communication (handshake communication), which is fine since the implementation of φ-comp relies on TCP/IP.

In the rest of this section, we present how the different elements of the φ-comp framework 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, and C3). This excerpt of LNT also shows there are three bindings, there is for instance a binding between C1 hosted on the front-end server and C2 hosted on the application server. The physical network is not made explicit in this example for simplification purposes. The final part of Figure 1 shows that the application consists of three parts: an identifier, a set of VMs, and a set of bindings.

3.2 Management Protocol

This section presents successively the specification of each participant of the management protocol, and 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.

```
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

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

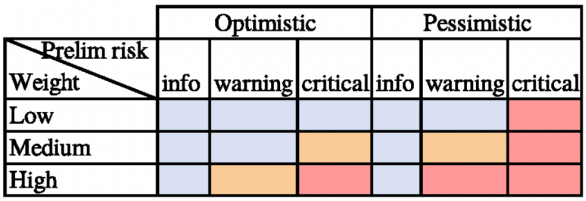

Fig. 5: Compliance Evaluation Results

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.

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. However, this is fine enough, because the main goal of the analysis steps is to show that the decision process is correct (in case of detected issue, the right decision is taken).

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. In the parallel composition, we can see the names of the processes (e.g., 'probe' or 'pda'). Before these names, there is an arrow preceded by a list of actions. These actions correspond to the actions on which a process has to synchronize with the other processes. As an example, one can see that the action ANALYSISEVENT appears before the 'pda' process and before the 'analyser' process, meaning that

both processes synchronize on this action and by doing so exchange information of interest with respect to the analysis of the detected problem.

```
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

4 Verification

In this section, we first explain how properties are verified. Then, we present the list of properties to be ensured. Finally, we comment on the results of analysis of these properties on realistic scenarios.

4.1 Model Checking

In this section, in a first step, we introduce the properties that must be preserved by the health management solution. 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 [6] is a rich verification toolbox that implements the results of concurrency theory 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.

4.2 Properties

The main goal of the analysis steps is to ensure that the decision process is performed correctly when a problem is detected, resulting at the end by the right decision taken by the φ-comp solution for mitigating this problem. Keeping this idea in mind, let us now introduce the properties, which were identified as important for the approach, and are listed below:

- The solution 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.

The MCL property formalising this requirement relies on regular expressions. These expressions are required to match the parameters coming with the actions used in the LNT specification. As an example, PRELIMINARY_RISK is accompanied with several numbers (between 0 and 4) giving information about

potential risks, and this is expressed using the regular expressions appearing in Figure 8. Moreover, we use exclamation marks followed by a number to indicate repetition of a same parameter. The number is used like an identifier (exclamation marks with the same number indicate the same parameter).

More technically speaking now, the MCL property given in Figure 8 for illustration purposes consists of two parts. The first part shows that a traceability problem is detected (MESSAGE2_TRACEABILITY) and the decision is taken of restarting the corresponding service (DECISION !RESTART_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

4.3 Results

In this section, we comment on the analysis of realistic scenarios using the CADP model checker. The 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 φ-comp solution is executed off-line.

5 Related Work

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

In [4,5,18], 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.

[11, 12] propose verification of IoT applications before deployment using model checking techniques. [3] 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.

[14] focuses on large applications relying on widely distributed and replicated storage of data for scalability, availability, and disaster tolerance. Maintaining high degrees of data consistency requires costly communication across distant sites, applications over such distributed and partially replicated data is complex. This paper proposes to use rewriting logic and its associated Maude tool environment to formally model and analyze both the correctness and the performance of state-of-the-art distributed transaction systems designs, as well as on how to automatically obtain a correct-by-construction distributed implementation of a promising design.

[10] proposes BiiMED, which is a Blockchain framework for Enhancing Data Interoperability and Integrity regarding EHR-sharing. This solution consists of an access management system allowing the exchange of EHRs between different medical providers and a decentralized Trusted Third Party Auditor for ensuring

data integrity. This work also introduces two validation techniques for enhancing the quality and correctness of the proposed solution: Formal verification checks the correctness of a mathematical model describing the behaviour of the system whereas model-based testing derives test suites from the model and executes them to validate its correctness.

[17] 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 [5, 17], 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 that formal verification techniques were applied on a software solution 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 framework, 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 solution targets health data and applications, it makes it critical and it is therefore crucial that specific properties of correctness are preserved by the solution. It was decided to rely on formal specification techniques and verification tools in order to ensure that the solution 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 approach 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 new solution 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 solution. Beyond analysis and certification, we finally plan to implement the proposal using cloud and blockchain technologies.

Acknowledgements. The authors would like to thank Frédéric Lang for his help in the specification and verification of the φ-comp solution.

References

1. M. Atif and J. F. Groote. *Understanding Behaviour of Distributed Systems Using mCRL2, 1*, volume 458. Springer, 2023.
2. 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.
3. G. Delzanno. Formal Verification of Internet of Things Protocols. In *Proc. of FRIDA'18*, 2018.
4. 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.
5. 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.
6. 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.
7. 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.
8. G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
9. ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, ISO, 1989.
10. R. Jabbar, M. Krichen, N. Fetais, and K. Barkaoui. Adopting Formal Verification and Model-Based Testing Techniques for Validating a Blockchain-based Healthcare Records Sharing System. In *Proc. of ICEIS'20*, pages 261–268. SCITEPRESS, 2020.
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. P. C. Ölveczky. Design and Validation of Cloud Storage Systems Using Rewriting Logic. In *Proc. of SYNASC'19*, pages 17–21. IEEE, 2019.
15. 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.
16. 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.

17. 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.
18. 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.