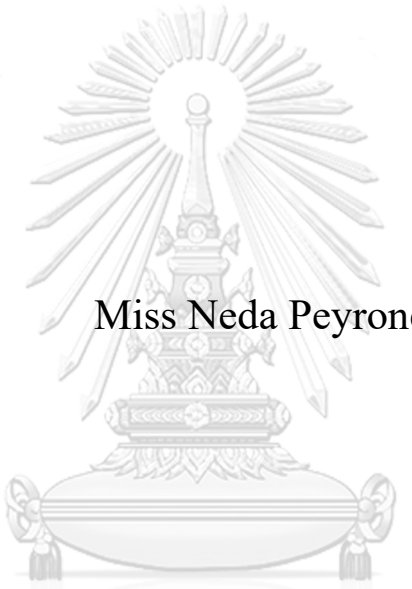
# Formal Models for Consent Management in Healthcare Software System Development

Miss Neda Peyrone





A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Computer Engineering
Department of Computer Engineering
FACULTY OF ENGINEERING
Chulalongkorn University
Academic Year 2022

# แบบจำลองเชิงรูปนัยสำหรับการจัดการความยินยอมในการพัฒนาระบบซอฟต์แวร์กลุ่มให้บริการทางสุขภาพ

น.ส.เนด้า เปอิโรเน



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2565

| | |
|---|---|
| Thesis Title | Formal Models for Consent Management in Healthcare Software System Development |
| By | Miss Neda Peyrone |
| Field of Study | Computer Engineering |
| Thesis Advisor | Associate Professor DUANGDAO WICHADAKUL, Ph.D. |

Accepted by the FACULTY OF ENGINEERING, Chulalongkorn University in Partial Fulfillment of the Requirement for the Doctor of Philosophy

..... Dean of the FACULTY OF ENGINEERING
(Professor SUPOT TEACHAVORASINSKUN, D.Eng.)

DISSERTATION COMMITTEE

..... Chairman
(Assistant Professor Chanon Dechsupa, Ph.D.)
..... Thesis Advisor
(Associate Professor DUANGDAO WICHADAKUL, Ph.D.)
..... Examiner
(Associate Professor CHOTIRAT RATANAMAHATANA, Ph.D.)
..... Examiner
(Associate Professor TWITTIE SENIVONGSE, Ph.D.)
..... Examiner
(Associate Professor WIWAT VATANAWOOD, Ph.D.)

เนด้า เปอิโรเน : แบบจำลองเชิงรูปนัยสำหรับการจัดการความยินยอมในการพัฒนาระบบซอฟต์แวร์กลุ่มให้บริการทางสุขภาพ. ( Formal Models for Consent Management in Healthcare Software System Development) อ.ที่ปรึกษาหลัก : รศ. ดร.ดวงดาว วิชาดากุล

ในยุคแห่งโอกาสของการขับเคลื่อนด้วยข้อมูล ธุรกิจจำนวนมากเผชิญความท้าทายด้านการจัดการดูแลข้อมูลส่วนบุคคล ซึ่งนำไปสู่ความเสี่ยงในการปกป้องข้อมูลของลูกค้า เพื่อให้บุคคลทั่วไป (data subjects) มีอำนาจในการควบคุมข้อมูลของตน สหภาพยุโรปได้ออกกฎหมายคุ้มครองข้อมูลส่วนบุคคล หรือ จีดีพีอาร์ โดยกำหนดให้ธุรกิจหรือองค์กร (data controllers) จะต้องปกป้องข้อมูลของแต่ละบุคคล (personal data) ภายใต้กฎหมายคุ้มครองข้อมูลส่วนบุคคล อย่างไรก็ตาม ธุรกิจจำนวนมากยังคงประสบปัญหาในการปรับปรุงและพัฒนาระบบซอฟต์แวร์ของตนให้สอดคล้องกับจีดีพีอาร์ เนื่องจากเป็นการยากที่จะตีความและนำไปใช้เป็นแนวทางในการพัฒนาซอฟต์แวร์ นอกจากนี้การประมวลผลข้อมูลส่วนบุคคลจะเริ่มขึ้นได้ ก็ต่อเมื่อเจ้าของข้อมูลส่วนบุคคลจะต้องให้ความยินยอมโดยชัดแจ้งแก่ผู้ควบคุมข้อมูลส่วนบุคคล ซึ่งทำให้การจัดการความยินยอม (CM) มีความจำเป็นสำหรับการจัดการวงจรชีวิตของข้อมูลส่วนบุคคล วิทยานิพนธ์นี้มีวัตถุประสงค์เพื่อเติมเต็มช่องว่างนี้โดยการเสนอแบบจำลองเชิงรูปนัยและการแปลงไปเป็นแผนภาพคลาสสำหรับการจัดการความยินยอมในระบบรวมศูนย์ และการแบ่งปันข้อมูลในระบบกระจาย เพื่อเป็นแนวทางสำหรับวิศวกรซอฟต์แวร์ นอกจากนี้แบบจำลองเชิงรูปนัยที่เสนอได้รับการตรวจสอบและอธิบายพฤติกรรมโดยใช้เมธอดอีเวนต์บี



| | | |
|---|---|---|
| สาขาวิชา | วิศวกรรมคอมพิวเตอร์ | ลายมือชื่อนิสิต ..... |
| ปีการศึกษา | 2565 | ลายมือชื่อ อ.ที่ปรึกษาหลัก ..... |

## 6371021321 : MAJOR COMPUTER ENGINEERING
KEYWORD: General Data Protection Regulation, GDPR, Privacy by Design, Consent Management, Formal method, Event-B, Smart Contracts

Neda Peyrone : Formal Models for Consent Management in Healthcare Software System Development. Advisor: Assoc. Prof. DUANGDAO WICHADAKUL, Ph.D.

In the era of data-driven opportunities, many businesses are missing the data-privacy challenge, which leads to risks in safeguarding their customers' data. To empower individuals (data subjects) to control their data, the General Data Protection Regulation (GDPR) mandated businesses or organizations (data controllers) to protect individuals' data (personal data) within data protection law. Nevertheless, many businesses still struggle to enhance and develop their software systems to comply with the GDPR because it is difficult to interpret and apply to software development practices. Besides, the processing of personal data begins when the data subject provides explicit consent to the data controller, which makes consent management (CM) essential for conducting the personal data lifecycle. This thesis aims to fill this gap by proposing formal models and translating them into class diagrams for consent management in centralized systems and data sharing in distributed systems as guidelines for software engineers. Moreover, the proposed models have been verified and described behavior using the Event-B method.



| | | |
|---|---|---|
| Field of Study: | Computer Engineering | Student's Signature ..... |
| Academic Year: | 2022 | Advisor's Signature ..... |

# ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Assoc. Prof. Dr. Duangdao Wichadakul. She encouraged me to pursue research interests in data protection and privacy for health information and helped me develop good research practices. Whenever I lose confidence or face problems, my advisor is always there to listen and give me sincere advice.

I am deeply grateful to Assoc. Prof. Dr. Wiwat Vatanawood, Assoc. Prof. Dr. Chotirat Ratanamahatana, Assoc. Prof. Dr. Twittie Senivongse, and Asst. Prof. Dr. Chanon Dechsupa for being my thesis committee and giving all valuable comments and suggestions.

I thankfully acknowledge the support and inspiration from my teachers, especially Assoc. Prof. Dr. Vara Varavithya for instilling the courage and diligence to achieve my ambitious goals.

Lastly, I would express a deep sense of gratitude to my family and friends for their unconditional love and support. This dissertation could not have been completed without them.

Neda Peyrone

# TABLE OF CONTENTS

**Page**

# LIST OF TABLES

**Page**

# LIST OF FIGURES

**Page**

# CHAPTER I
# INTRODUCTION

Data privacy concerns have become more critical since the General Data Protection Regulation (GDPR) went into effect on May 25, 2018. The GDPR is the data privacy law in the European Union (EU) that empowers people ('data subjects') with various rights to control their personal data [1]. It motivates people to be aware of how their data is being used. On the other hand, businesses must rethink and redesign their software systems to embrace data protection. However, the GDPR is written in natural language, and most data protection articles are described in generic terms. Therefore, it causes many businesses to struggle with identifying appropriate technical solutions for their development process to demonstrate GDPR compliance [2-4]. Nevertheless, developers find the GDPR difficult to interpret and adopt into software systems [5, 6]. Besides, the lack of clear guidelines on how to implement data protection as a component of software systems leads to risks of confidentiality and privacy breaches [7, 8]. Moreover, software systems that fail to comply with GDPR requirements face heavy penalties and fines, which becomes a significant research challenge [9].

Most modern software systems (e.g., banking, online shopping, social media) rely on customers' data. Moreover, they may probably share customers' data among third-parties services to improve their products and services. The growth of data created and processed by software systems continues increasing, as businesses should be concerned about customers' privacy to handle their data with ethical and legal integrity. By designing a data protection mechanism for software systems, developers need to transform GDPR requirements into software specifications. However, the developers should incorporate data privacy by design to guarantee that all software systems embed a data protection mechanism. Privacy by Design (PbD) is an approach to developers that considers data protection upfront and integrates it as a core functionality into software systems [10-12]. The benefit of incorporating PbD is to make compliance with GDPR requirements easier [13].

Based on Article 6 GDPR, there are the six legal bases for data processing as follows: 1) the consent indicates the data subject's agreement that he/she has given clearly approval for personal data processing, 2) the contract indicates that the purpose of the data processing is essential to perform a contract with data subjects, and data controllers need to examine which provisions regarding the legal basis of processing personal data, e.g., the contract between customers of payment services, 3) the legitimate interest indicates that data processing is essential in manners data subjects commonly expect, and organizations use their personal data to meet its objectives, e.g., fraud prevention, 4) the vital interest indicates that data processing is essential to

protect individuals' life, e.g., emergency medical treatment, 5) the legal requirement indicates that data processing is essential to perform with a legal obligation, e.g., consumer transaction law, and 6) the public interest indicates that data processing is essential to perform public functions undertaken by public authorities, e.g., a public body's tasks.

The consent under GDPR ensures data subjects' freedom to make decisions about their personal data. Hence without data subjects' consent, a software system conducts their personal data unlawfully or unauthorized. From a practical perspective, scientific communities, private companies, and the Cyber Security Network of Competence Centres for Europe (CyberSec4Europe) are pointing out that the consent and security services successfully enforce the data protection regulation [14]. Daoudagh et al. [2] concurred that the consent service enables organizations to manage personal data lifecycles. In contrast, security services such as authorization modules ensure that only the authorized user can access a specific resource (i.e., Access Control (AC)), which brings personal data into protection within a regulatory regime (e.g., data usage purpose, user consent, data retention period). Therefore, incorporating consent and security services overcomes the challenge of designing software systems to support GDPR requirements. Sforzin et al. [15] revealed that there are many research studies for defining and implementing privacy knowledge and rules, but there is still no generic solution.

Consent management (CM) is a software component used to manage the entire personal data lifecycle [16]. With its capability, consent management helps build software systems that meet the GDPR requirements [17]. The key roles involved in consent management under the GDPR are as follows: 1) the data controller is the entity, e.g., person or organization, responsible for defining policies on collecting and processing data (Articles 4(7) & 24 GDPR), and 2) the data processor is the entity, e.g., person, organization, responsible for collecting and processing data upon the data controller's policies (Articles 4(8) & 28 GDPR). Furthermore, based on the data subject's consent (Articles 6(1a) & 7 GDPR), which are composed of four elements: 1) the data subject shall give his/her consent voluntarily, 2) the purpose of processing data must be specific and transparent, 3) the data controller must inform the data subject for the purpose before gathering and processing his/her personal data, and 4) the data subject gives explicit consent for enabling the processing of his/her personal data.

Nevertheless, the GDPR expects organizations to implement privacy into technology solutions at their earliest stages of process development [18], as stated in Article 25. At its core, the GDPR mandates only a baseline set of guidelines, not how to embed data protection into software design [19]. To ensure consent management

mechanism as a primary component in software systems, we thus adopt privacy by design (PbD). PbD is a concept that emphasizes how to integrate data protection into technology as default settings, but PbD cannot be accomplished solely by data protection laws [19]. It is the philosophy proposed by Cavoukian. Besides, PbD outlines the seven Foundational Principles, which define a set of the following guidelines: 1) it is crucial to incorporate data protection as part of software design, 2) data protection must be embedded as core functionality in software systems by default, 3) the system must adopt a data protection mechanism into its architecture, 4) the system must conduct personal data accurately and securely without decreasing the system's ability, 5) the system must keep personal data and destroy it for an appropriate retention period, 6) the system must provide privacy notices for fulfilling the purpose, and it should be clear and transparent to individuals about their personal data, and 7) the system must respect and protect individuals' data with regard to a high level of security.

In addition, we have addressed key issues and requirements of consent management for centralized systems (Table 1) and data sharing in distributed systems (Table 2) related to GDPR. This study aims to fulfill the requirements derived from the literature partially. Data controllers can gather consent from different types of channels, including websites, mobile applications, web forms, and various marketing platforms, which makes it difficult to process the collecting of informed consent from the data subject [20]. In this thesis, we focus on constructing formal models divided into two primary purposes: 1) consent management for centralized systems and 2) consent management for data sharing in distributed systems. These two types of consent management are essential for businesses collecting, processing, and sharing personal data. Centralized consent management enables privacy processes centralization to conduct the lifecycle of individuals' data concerning data protection regulations. In contrast, distributed consent management enables secure data sharing by limiting accessing personal data within given consent and capturing audit logs for every activity. We thus adopt blockchain technology to conduct data-sharing processes with higher reliability and security. In doing so, we set out primitive CM operations to fulfill issues and requirements for both centralized and distributed consent management, including manipulating data subjects' consent, restricting access to authorized personal data based on the data subject's consent, enabling data subjects to revoke consents, enabling data subjects to request portable their personal data, and allowing data subjects to renew their consent for continued use of services and products offered by service providers.

Formal methods are essential and reliable for achieving data protection. They use a mathematical approach to model and verify a software system specification to guarantee its correctness [21, 22].

To guarantee the correctness of the models, we used the Event-B formal method [23, 24]. The benefit of using Event-B is that it provides an automated tool called Rodin Platform, which supports developing and checking various models [25]. The Rodin Platform is a model development tool based on Eclipse-IDE that offers useful plugins such as a proof obligation generator, provers, a model-checker (ProB), etc [26]

Table 1: Consent management-related issues as requirements for centralized systems.

| Topic | Issue | Requirement |
|---|---|---|
| Rules for collection and processing of personal data based on a specific purpose | The lack of management of the personal data lifecycle possibly violates individuals' rights [17, 27, 28]. | The system shall define the consent management functionality mandated for further collecting and processing of personal data. |
| | The data subject can control and consent over his/her personal data being collected and processed [29-31]. | |
| Access control | The ambiguous role and responsibility of the entities involved in the collecting and processing personal data beyond originally specified purposes may lead to risks and freedoms for individuals [32-34]. | The system shall allow a data controller to assign stakeholders involved according to their roles and responsibilities of collecting and processing personal data within a given consent. |
| Restricted records data retrieval | The risk of individual rights violations has increased with the collection and processing of unnecessary personal data [35]. | The system shall define a one-time request per an individual's data, and it will be collected and processed according to predefined data fields within a given consent. |
| | The risk of a data breach has increased with the retrieval of a massive amount of personal data [36, 37]. | |
| Withdrawal | The data subject has the right to withdraw his/her consent at any time (Article 7(3) GDPR) [38, 39]. | The system shall provide a withdrawal approval mechanism. |
| | The data controller must demonstrate that services and products can be refused or withdrawn without any detrimental consequences to the data subject (Recital 42 GDPR) [39]. | |
| | If the personal data is unnecessary after withdrawal, then the data controller should remove personal data from the system [17, 38, 39]. | |
| Portable | The data subject has the right to receive a copy of their personal data (Article 20 GDPR) [18, 33, 40]. | The system shall provide a portable approval mechanism. |
| | The data controller is also allowed to refuse a data subject's request only if the refusion can be justified to the data subject (Article 12(5) GDPR) [41]. | |
| Renewal | The data controller may offer a data subject to extend the retention period to continue using the products and services [20]. | The system shall provide a consent renewal mechanism. |
| | The data subject has the right to object to the data controller for continuing to process his/her personal data (Article 21 GDPR) [41]. | |
| | To re-consent the data subject, an investigator may notify the data subject in person of the circumstances. In the case of re-signing consent, there was no conflict or legal problem with the original consent [42]. | |
| | If the personal data is unnecessary after rejection to extend the retention period, then the data controller should remove personal data from the software system [17, 38, 39]. | |

Table 2: Consent management-related issues as requirements for data sharing in distributed systems.

| Topic | Issue | Requirement |
|---|---|---|
| Rules for sharing of personal data based on a specific purpose | The challenge of consent management in data sharing is to obtain and maintain consent and personal data in a distributed environment securely, effectively, and transparently [20, 43-45]. | The system shall define the consent management functionality enabling decentralized security and transparency of recording and sharing data within a network. |
| | The data subject can control and consent over his/her personal data being shared [20, 43, 45, 46]. | |
| Access control | The data subject shall provide his/her consent to transfer personal data between data controllers [20, 43]. | The system shall enable participant data controllers to request and share personal data within a given consent. |
| Restricted records data retrieval | The risk of individual rights violation has increased with the sharing of unnecessary personal data [35, 47]. | The system shall define a one-time request-response interaction per an individual's data, and it will be shared according to predefined data fields within a given consent. |
| | The data controller only shares the minimum amount of personal data [36, 37, 43]. | |
| Withdrawal | The data subject can revoke his/her consent to stop sharing personal data at any time [20, 48]. | The system shall provide a consent revocation mechanism by which the data subject can withdraw his/her consent. |
| Renewal | The data controller may ask the data subject to extend the retention period to continue sharing his/her personal data [20, 48]. | The system shall provide a consent renewal mechanism by which the data subject can renew his/her consent. |
| Audit logs track activities | Sharing personal data between the participants as data controllers recorded at each transmission stage should be immutable and transparent on the history of transactions [20, 43, 47]. | The system shall provide an audit mechanism to record a request-response interaction among the data controllers on every shared data. |
| | The data subject shall be able to perform audits based on access to his/her personal data within the given consent [20, 43, 46]. | |
| Personal data | The risk of direct identification in data sharing may lead to recognizing individuals [47, 49-51]. | The system shall use pseudonymized data to minimize the risk associated with using personal data. |

## 1.1. Objective of the Work

The objectives of this study are as follows:

1.1.1. To construct formal models used as guidelines for software development on the aspects of consent management based on centralized systems to fulfill GDPR requirements.

1.1.2. To construct formal models used as guidelines for software development on the aspects of consent management based on data sharing in distributed systems to fulfill GDPR requirements.

## 1.2. Contributions

- This study reduces the ambiguity of software design in consent management functionality according to the GDPR, which can lead to broader and more consistent adoption of the standards outlined in the law.
- This study provides class diagrams as clear guidance on how to incorporate consent management functionality into healthcare systems.
- This study provides a Python REST API accessible to smart contracts for enabling consent management in data sharing, called SmartDataTrust.

## 1.3. Research Methodology

- Conduct a literature review.
- Identify recent literature trends related to formal consent management models according to GDPR compliance.
- Study related works in formal models for consent management, GDPR requirements, and use cases cover the lifecycle of consent management.
- Set up the Rodin Platform for the Event-B method and practice how to construct a model to verify its correctness.
- Define state machines and identify GDPR articles that they covered.
- Develop complete formal models for the research question.
- Verify formal models' correctness using the Rodin Platform with no invariant violations and deadlocks found.
- Transform formal models into class diagrams.
- Publish two journal articles relating to the work.
- Prepare and engage in a thesis defense.

# CHAPTER II
# RELATED WORK

Data privacy is becoming increasingly important to consumer data protection as technology gathers so much data. One significant privacy issue is that developers lack an understanding of GDPR and PbD concepts, which leads to software systems not being designed and developed from the perspective of data protection requirements [5, 6]. There are numerous studies on the challenge of implementing data protection into software systems from the perspective of laws [4, 9, 18, 19], computer science [3, 16, 17], and software engineering [2, 5-8]. Schupp [52] pointed out that formal methods play a significant role in supporting PbD, but half of the academic papers proposed formal methods without demonstrating the implementation of their approach. As for the other half, they demonstrated a few examples that could guide developers to implement privacy-preserving systems. Hence, there remains a lack of clear software development guidelines for implementing data protection.

Several relevant publications on using formal methods in data privacy did not consider GDPR and consent management as part of their models. To begin with, Matwin et al. [53] proposed an approach that empowers individuals to take control of their privacy in data-mining programs. This privacy-preserving data mining approach used the Coq theorem prover [54] to prove the properties of data-mining programs, e.g., Weka [55]. The Coq is an interactive formal proof to assist in developing mathematical theories and formalizing the system's correctness. The authors first translated programs into logic expressions of theorem provers to specify the privacy properties. Then, they constructed a model and defined a set of permissions for limiting access to a program's properties according to the owner permissions. Figure 1 shows the architecture of their proposed model. It begins with the user *C* assigning permissions $P_c(D, A)$ to an algorithm *A* for determining whether actions can take with his/her data *D*. When the developer modifies *A* with its source code *S* and builds it into a binary executable *B*, the trusted organization *Veri* checks whether $R(P_c, S)$ is a proof of a theorem $T(P_c, S)$. *B* is the executable of *S* with respect to the user's permissions by the organization responsible for processing the user's data. The limitation of the approach is that it cannot express data properties syntactically in formal logic. However, this proposed model can be used as a starting point for verifying privacy policies in data-mining programs.

Stouppa & Studer [56] revealed that the main challenge of data privacy is to share a portion of data while protecting personal data. The authors proposed a theoretical framework to protect personal data exposed to public views by restricting the privileges of all users in relational databases and ontology-based information systems. They defined the query answering problems in first-order ontologies under the logical

entailment and explained how to apply their model in a telecommunication company. To begin with, the company offers end-users to find phone numbers through search engines, but some customers want to keep their phone numbers private. Therefore, the model should define a set of queries *Owns(cust$_i$, Tel)*, where *Owns* indicates the relationship between a customer *cust$_i$* and his/her phone number *Tel*. Then, when a user executes a query to retrieve a customer's phone number, no result is returned by the query for every possible interpretation, indicating that the customer's phone number has been protected. However, the proposed framework does not cover the case of boolean queries because it does not apply to ontology.

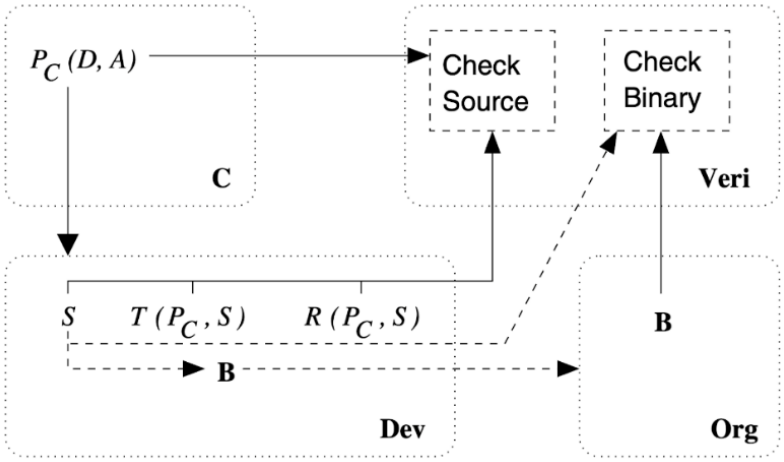



Figure 1: Demonstrating the architecture of the Matwin et al.'s model (Figure 1 of [53]).

According to Ni at el. [57] data privacy has become increasingly important for consumers, organizations, researchers, and legislators. The study aimed to address the problem of using traditional access control over data privacy. The authors proposed Privacy-aware Role Based Access Control (P-RBAC) to enable the authoring and conducting of privacy-aware access control policies. The P-RBAC extended from RBAC to provide fully supporting complex privacy-related policies. The RBAC is a security approach restricting system access to all users with their roles to perform on specific resources but does not endorse privacy protection requirements. As for P-RBAC, the privacy policies were mapped as permission assignments (PAs) which belonged to roles (Figure 2). However, the relationships between PAs and roles are many-to-many and may cause conflicts among PAs within user roles in various conditions. The authors then provided an algorithm to solve the conflict of PA by improving the rules of Enterprise Privacy Authorization Language (EPAL) [58].

In the data-driven age, big data has become one of the major areas of data management to deal with massive data sets for supporting analysts and decision-makers. The organizations involved in processing vast amounts of data are concerned with

privacy issues, and data breaches may affect their businesses. Blake & Saleh [36] suggested that formal methods significantly impact privacy-preserving in big data and its applications. The authors argued that the challenge of protecting sensitive data in big data is that misconduct with pieces of data causes to violate users' privacy. Data integration is the essential process in big data for combining heterogeneous data from multiple sources into a data warehouse using the Extraction, Transformation, and Loading (i.e., ETL) process. In the data integration staging area (Figure 3), the authors suggested adding test procedures based on formal methods to validate the conformance of data protection in four specific areas: 1) Pre-Hadoop process validation, which determines what data is sensitive and how long to keep data in the data preprocessing step, 2) Map-Reduce process validation, which lowers the risk of a data breach by retrieving massive data and limits sharing only the minimum amount of data among processes where is necessary, 3) ETL process validation, which verifies privacy-related policies and unlinks personally identifiable information before loading into a data warehouse and, 4) Report testing process, which verifies the visibility permission of sensitive data in report forms based on particular purposes.

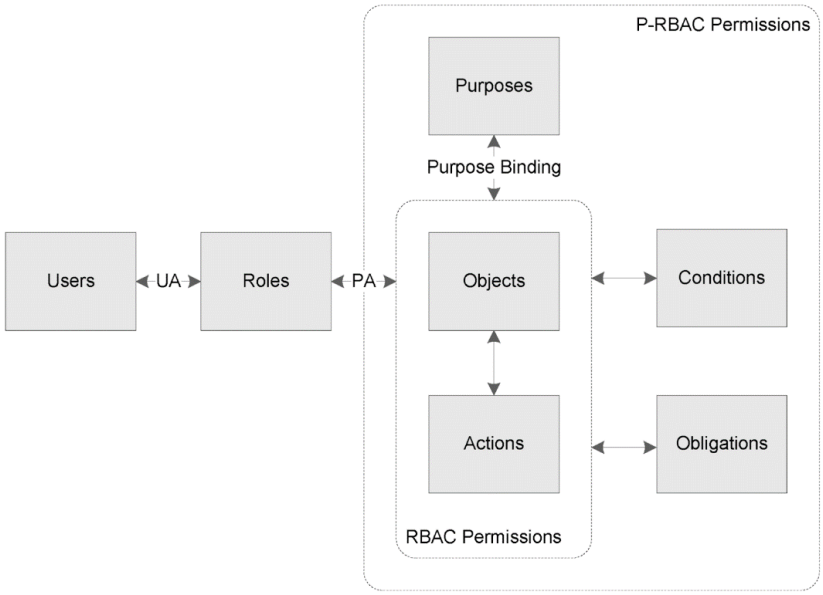



Figure 2: The P-RBAC model architecture (Figure 3 of Ni at el. [57]).

In another study, Abe & Simpson [59] pointed out that the concept of privacy has captured more attention in people's lives but needs to be more specific. The authors argued that formal methods play a significant role in certifying a variety of data privacy contexts. They proposed a formal model to protect against unauthorized access for sharing data among processes in distributed systems based on the United Kingdom's Data Protection Act (DPA) 1998 [60]. They first defined the disclosure processing based on a single system that works internally related to a data controller.

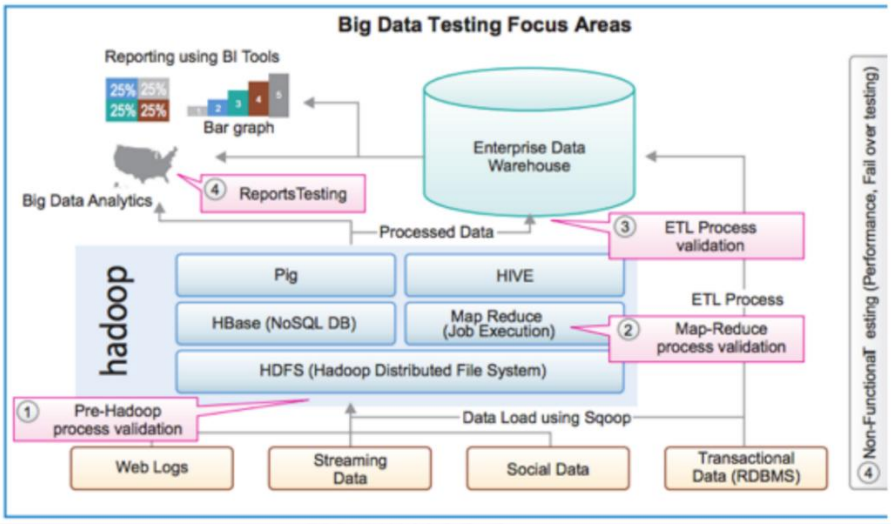



Figure 3: The architecture of big data testing areas (Figure 2 of Blake & Saleh [36])

The model was composed of five processes (Figure 4):

1. Parameterisation (PAR) defines the parameters as a guideline in each aspect of individuals' data processing designated by the data controller, e.g., extract parameter (extparam), render parameter (renparam), test parameter (tesparam), and disseminate parameter (disparam).

2. Extraction (EXT) extracts personal data according to the variable *extparam*, e.g., the data source's location, the characteristics of data extraction, the workload applied during the extraction, and the method used for extraction. After the processing task, the result produces the extraction of personal data and holds in the variable *extdata*.

3. Rendering (REN) controls the visibility permission in personal data based on the variables *extdata* and *renparam* (e.g., the methods used for rendering and intensity visible and the characteristics of visible data). After processing, the result produces the personal data visible and held in the variable *rendata*.

4. Testing (TES) evaluates the data quality according to the data controller's policies and uses the *extdata*, *rendata*, and *tesparam* variables as inputs. The *tesparam* is used to control the testing process for determining the risk of violating individuals' privacy. After the processing task, the result produces the testing results that perform on the *extdata* or *rendata* variables.

5. Dissemination (DIS) performs the data transmission based on the *extdata*, *rendata*, and *disparam* variables. The *disparam* is used to determine the data transfer location and mode of transfer. After the processing task, the result indicates that personal data has been transferred.

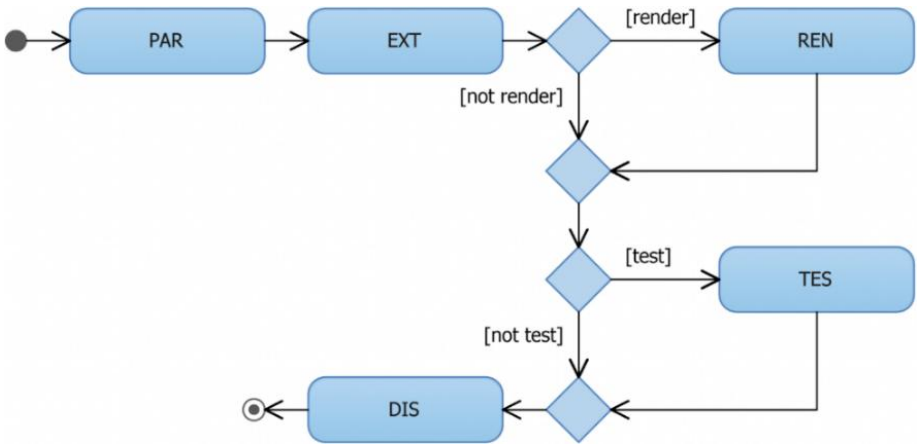



Figure 4: The architecture of disclosure-processing (Figure 1 of Abe & Simpson [59]).

The authors added security constraints into the model, which were composed of three major parts:

1. Determining permissions to prevent inappropriate disclosure, which combines the relationship between actions, resources, and process identifiers (PID).

2. Restricting system access to authorized users based on Role-Based Access Control (RBAC).

3. Determining a designated source of personal data to be processed.

In doing so, the authors formalized the model using Z notation and verified its model with ProZ. The Z notation is a modeling-oriented method used to describe the behavior of systems in mathematical terms [61], while ProZ is a model checker to generate test cases, and check reachability, deadlock-free and invariant violations [62]. Therefore, the model result indicated that the data controller's obligations were satisfied by system specifications.

Consent is one of the primary lawful bases for processing personal information under the GDPR. Many studies have shown that consent is essential to allow individuals to track their personal data being used and revoke consent at any time they desire. Besides, there are numerous publications about consent management on centralized and distributed systems, but most of the studies do not apply formal methods. On the other hand, several studies incorporated consent management into software systems using formal methods to ensure correct behavior. For example, Besik & Freytag [63] focused on healthcare privacy and utilized Business Process Model and Notation (BPMN) to model clinical workflows. This study aimed to employ privacy-preserving mechanisms in existing non-privacy-aware workflows for a newborn screening scenario. In the model, privacy awareness was defined as privacy rules of workflows

based on privacy concepts, e.g., GDPR principles, privacy policies, and privacy preferences.

Figure 5 demonstrates the overview of their proposed solution, which divided into three parts:

1. Creating ontology based on privacy concepts that represents knowledge-based systems. The BPMN is used to connect activities, events, and gateways of clinical workflows related to the privacy ontology. In this study, there are three sources of privacy concepts:

    1.1. The GDPR principles, which cover some articles, e.g., purpose limitation (Article 5(1b)), data minimization (Article 5(1c)), consent validation (Article 6(1a)), and data retention (Article 5(1e)).

    1.2. Privacy policies, in the context of software design, a statement that specifies the data to be processed, for what purpose, who is responsible for processing data, and how long data can be obtained.

    1.3. Privacy preferences, which allows patients to grant who can or cannot access their data based on given consent. Besides, patients can determine their consent duration.

2. Formalizing privacy rules based on privacy policies and privacy preferences.

    2.1. Formalizing privacy rules, which states as follows:



        2.3.1. Privacy policies of consent *PC*, which contains consent rules defined as 2-tuple (purpose, requiresConsent). The *purpose* indicates the objective of data processing, while *requiresConsent* is a member of the boolean (i.e., true, false) indicating whether the processing of personal data requires consent.

        2.3.2. Privacy policies of retention PR, which contains rules of retention upon specific purpose defined as 4-tuple (user, purpose, data, retention). The purpose is defined the same as consent privacy policies, while the other three variables represent as follows: 1) the user indicates end-users which can be either individuals or organizations, 2) the data indicates a set of data objects, and 3) the retention indicates the duration of data to be stored.

        2.3.3. Privacy policies of data minimization PD, which contains data minimization rules defined as 4-tuple (user, purpose, data, condi-

tion). The first three variables are defined as retention privacy policies, while the condition indicates additional constraints regarding the data-usage objective.

2.2. Formalizing privacy preferences rules

Privacy preferences R, which contains data subjects' preferences defined as 8-tuple (dataSubject, user, purpose, data, condition, duration, status, entryDate). The variables user, purpose, data, and condition are defined the same as data minimization privacy policies; while the other four variables define as follows: 1) the dataSubject indicates a set of individuals whose personal data is being used, 2) the duration indicates the period of the data subjects' preference, 3) the status indicates whether the data subject allows the user to access his/her personal data, and 4) the entryDate indicates the creation date of the privacy preference.

3. Verifying compliance with GDPR principles and integrating privacy awareness into existing clinical workflows.

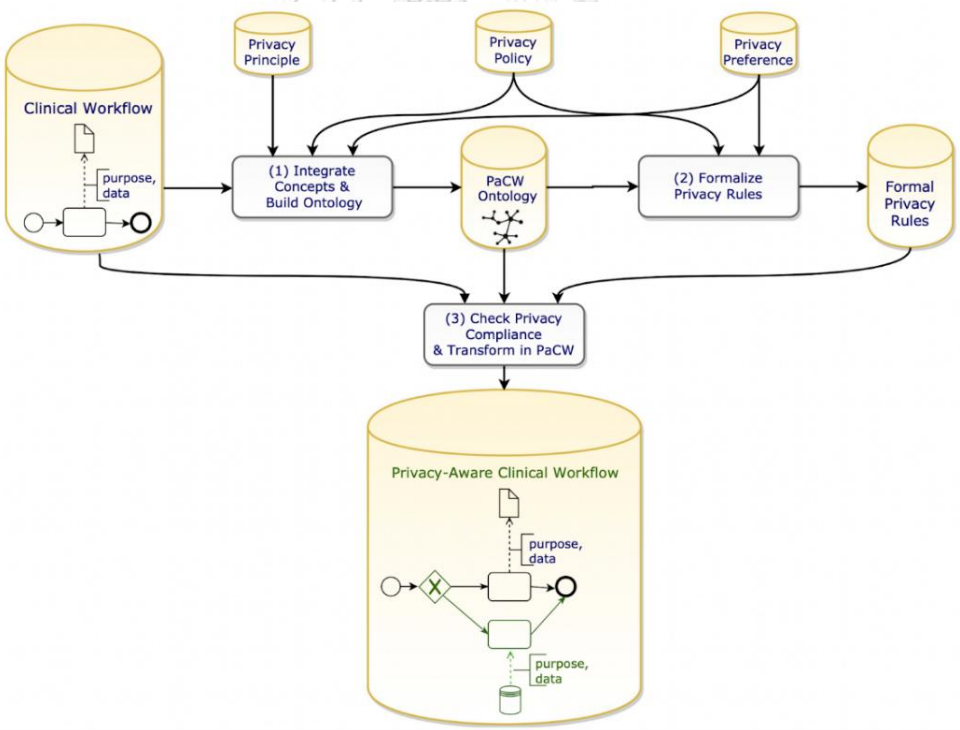



Figure 5: Conceptual diagram of privacy-awareness clinical workflows (Figure 1 of Besik & Freytag [63]).

The authors first formalized data-aware workflows in process modeling notations of BPMN. The data-aware workflow is a directed graph with vertices (compo-

nents) *C* and edges (sequence flows) *F*. The *C* represents a set of components and contains disjoint sets of tasks *T*, events *E*, data objects *D*, and gateways *G*, while the *F* is a subset of C × C, representing the connection between source and destination components. Besides, each task *T* is linked to a data object *D*, and every access is required to verify the given purpose *p*, representing an ordered pair (D, p). Finally, the authors created algorithms written by formal annotations to fulfill privacy concepts.

In the study conducted by Tokas & Owe [64], they proposed a formal framework for consent management that enables data subjects to modify their privacy preferences through a distributed system. In addition, the framework partially covered some of the GDPR articles, which comprise data protection principles (Article 5 GDPR), lawful bases for processing (Article 6 GDPR), data protection embedded into design (Article 25 GDPR), and data subjects' right to request access to their personal data (Article 15 GDPR). The authors defined the relationship between a data subject and a specific purpose as a 2-tuple (subject, purpose), called data tagging. Data tagging was defined to restrict personal data based on purpose in methods associated with privacy-preserving. The privacy policy is a statement written in natural language. However, it is difficult for machines to understand. So, it needs to be transformed into program entities or machine-readable code with the policy and consent specification defined as the relationship between principals *P*, purposes *R*, and access rights *A* as 3-tuple (P, R, A). First, the *P* represents a principal that denotes personal data that can be accessible, and its object or interface corresponds to a principal. An interface is a contract among classes with the inheritance hierarchy to be publicly exposed. Second, the *R* represents the purpose of conducting personal data. Third, The *A* represents an access right that denotes permission to perform a specific operation (e.g., read, write, modify, full control) on the object.



For example, consider the personal health data with a tag {(*Lilly*, *treatm*)}, and consented policies in the object *Lilly* are (pos(*Doctor*, *treatm*, *full control*); neg(*Sompong*, *treatm*, *read*)). However, in the positive policy, this setting indicates a *Doctor* has complete control of Lilly's health data within the *treatm* purpose. On the other hand, in the negative policy, Sompong is a *Doctor* and cannot read Lilly's health data.

Therefore, the policy and consent specification is a set of rules that aim to protect individuals by limiting the use of their personal data, written in Backus-Naur (BNF) notation [65]. The framework provided classes and interfaces to obtain individuals' privacy settings. The developers are required to implement the interfaces and classes to incorporate consent management into the system. In addition, the framework ensured that each access request to personal data corresponds to the current consent policies.

For other aspects of research consent management, Hyysalo et al. [66] proposed Consent Management Architecture (CMA) which provides authorization context of different data sources for securing access to health services following the strategy and principles of MyData [67, 68]. The CMA was designed to fill the gap in the following requirements: 1) data subjects own the right to control their personal data, 2) data should be easily accessible and usable, 3) there should be a means to transform business entities exposed to a useful resource as new services that are identified via URIs, 4) the infrastructure shall provide personal data sharing and guarantee that personal data can be shared safely between public and private organizations comply with the GDPR, and 5) data subjects can switch service providers.

Figure 6 describes the CMA framework and its APIs, which is divided into three major parts:

1. Operator(s) are responsible for managing accounts composed of Authorization and Protection APIs. The Operator(s) here provides interfaces for account verification across different data sources, service providers, and individuals. The Authorization API provides an interface for Data sink API to generate/refresh the proof key of the authorization based on active consent, while the Protection API provides an interface for Data source API to validate consent.

2. Sink(s) provides the Data sink API as an interface for end-users to manage consent and access their personal data. For any request, the Data sink API shall be executed, after verifying the proof key of the authorization via Authorization API.

3. Source(s) are responsible for managing consent and personal data composed of Data and Data source APIs. The Data source API provides an interface for other data sources to manage consent, while the Data API enables an interface for Data sink API to retrieve personal data from the source with the resource identifier.

Therefore, the authors implemented minimum operations for proof of concept of CMA. The framework was developed using Flask in Python to build REST API and a web application. As for data management, they used the SQLAlchemy toolkit for managing connectivity and mapping table columns to object properties in an SQLite [69] database.

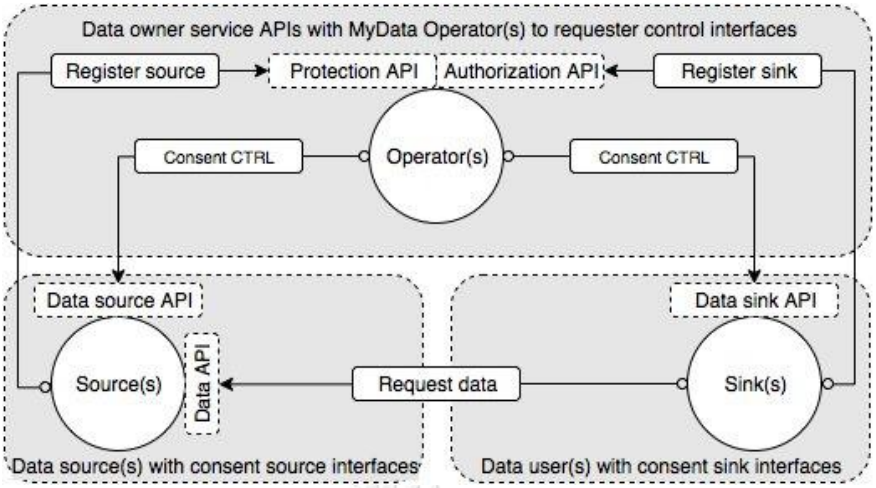



Figure 6: The CMA framework and its APIs interaction comply with GDPR and MyData approach (Figure 1 of Hyysalo et al. [66]).

Similarly, Marillonnet et al. [70] proposed human-centric architecture for supporting consent management by accessing e-government services of the Territorial Collectivities and Public Administration (TCPA). These TCPA are local and national government officials that provide e-government services for their citizens. Citizens shall submit some regulated document requests, e.g., renewing official documents, requesting allowance documents, and registering for local services. The benefit of e-government services is to provide citizens the ability to ease access to digital public services. In doing so, citizens shall give Personally Identifiable Information (PII) to TCPA with the required data for personal data processing. The authors argued that existing solutions did not address issues related to PII in the context of TCPA. Such issues are that the user's consent must be strictly considered regardless of PII's original location. In addition, the solutions must address the heterogeneous system cooperation with various sources, and the verification of remote sources needs to be determined if sources are reliable in providing users' PII. This study aimed to design consent management incorporating the PII data lifecycle to fulfill TCPA requirements.

The authors defined the system model with four major parts:

1. Actors in the use case are defined along with their roles in an involved environment, which is divided into four actors:

    1.1. The citizen with a user account can submit regulated document requests to TCPA online services. In addition, the user can keep track of his/her request through the platform.

    1.2. The PII manager is responsible for enforcement of the user's consent and verifying the trusted sources.

1.3. The TCPA User-Relationship Management (URM) is a service provider to help create trust among users and PII managers.

1.4. TCPA or third-party service providers maintain the data sources.

2. Environment Hypotheses indicate the use of experiments for enforcing data protection regulations based on production environments. The authors separated into two different hypotheses:

2.1. There should be rules and policies for accessing PII operators, which many PII managers host. Besides, the users would be asked voluntarily to select the operator of their PII manager.

2.2. The TCPA should arrange PII managers' authority using a public-key infrastructure (PKI).

3. Functional Requirements describe the product features that systems shall offer. In this study, they defined a list of non-exhaustive functional requirements related to PII management as follows:

3.1. Usage definition allows the data subject to specify the purposes designating the PII collection.

3.2. Consent management allows the PII manager to monitor access to PII only if users provide their consent.

3.3. Usage monitoring allows the data subject to designate his/her own metrics for PII consumption on any TCPA service. This monitoring provides a view of users' PII usage on any TCPA service.

3.4. Delegation capabilities provide the PII manager to decide whether to grant access to the PII based on the user's consent, even if the user does not connect to the platform.

3.5. PII location abstraction allows the PII manager to assure the management of PII regardless of the actual source of the PII.

3.6. Protocol standardization enables the PII manager inquiries with a generic interface leaning on standard protocols of PII management.

3.7. Access uniformization facilitates data access from multiple PII data sources in the same manner.

3.8. Authorization protocol interoperability provides identity management protocols based on access mechanisms and authorization schemes by enabling multiple remote sources.

4. Technical Hypotheses indicate the use of experiments for technical support in platform development, which considers four types of sources:

4.1. Plain OAuth 2.0 provides a user to authorize TCPA services to load the PII from remote sources.

4.2. SAML 2.0 identity providers enable a mechanism for passing user authentication and authorization across multiple secure domains along with Single Sign On (SSO).

4.3. HTTP basic authentication is used to restrict access to REST sources by an identified user.

4.4. Kerberos protocol is a network authentication protocol that verifies the identity of resource servers using a basis of tickets.

Figure 7 and Figure 8 demonstrate the design of a multi-service-based architecture for consent management. To begin with, a user has access to a user-centric PII management zone (Figure 7) to manage his/her PII, authorized sources, and their consent for any URM platform. The user first gets a ticket granted from the PII manager (Figure 8). When the user's identity and consent are specified, the PII manager issues the access token by the ticket, which scopes on the requested resource. The authors implemented a prototype design of the PII manager in a URM platform with minimum operations for proof of concept using the Django web framework in Python.



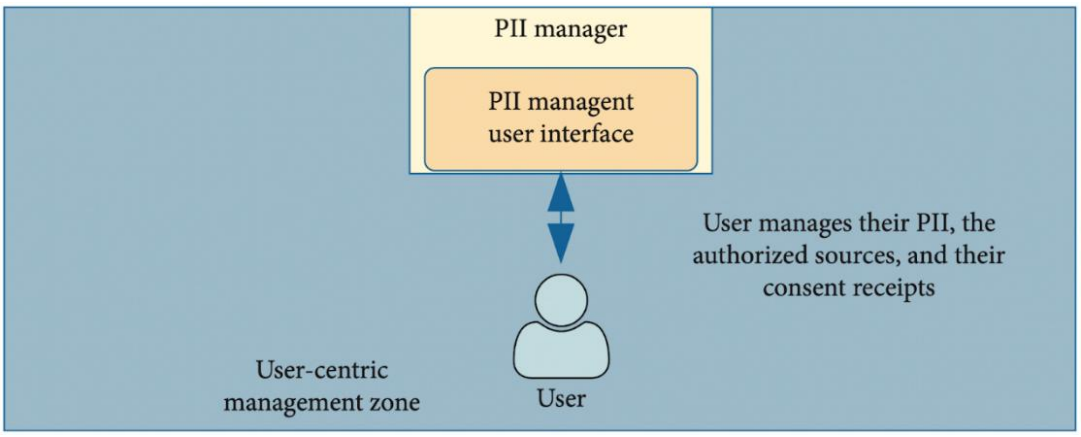



Figure 7: The interaction between a user and a PII manager (Figure 1(d) of Marillonnet et al. [70]).

Data accountability is crucial for data sharing in distributed systems. Nevertheless, data access and sharing come with the risk of privacy breaches. According to the IBM Security Report [71], the global average cost of data breaches has risen to a new high of $4.35 million, a climb of 13% over the past two years. The increased data breaches cause people to question existing personal data collection techniques. In addition, each audit record can potentially point to the causes of data breaches. The difference between distributed systems and blockchain is that distributed systems require trusted machines that administrators control, while blockchain technology enables a distributed ledger that records and shares immutable transactions between untrusted parties in a verifiable way and is permanently visible to all parties.

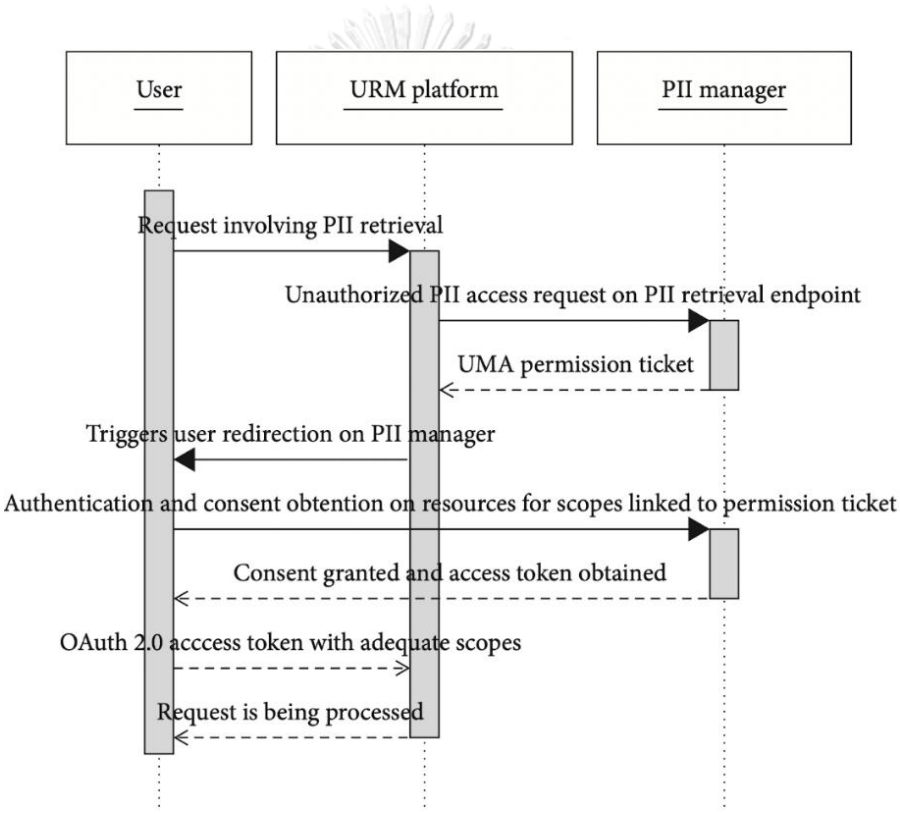



Figure 8: The sequence diagram of user authentication and consent collection on the PII manager (Figure 3 of Marillonnet et al. [70]).

Numerous studies are based on blockchain-enabled smart contracts to fulfill the privacy gap and mitigate trust concerns in consent management. Daudén-Esmel et al. [72] argued that the text of legislation regulations does not demonstrate how transparently the data subjects have signed this consent. Besides, most data subjects are unaware of their rights, nor do regulations provide guidelines to respond when their privacy has been violated. The authors proposed a lightweight blockchain-based GDPR-compliant personal data management system to fill this gap. This study focused on a human-centric approach, which allows data subjects to determine data

usage permissions based on their consent leveraged using smart contracts in blockchain. Smart contracts are programs live on the blockchain, which execute once specific objective criteria are met [73]. Therefore, the authors presented a conceptual design and system architecture for personal data management under GDPR requirements. Hence, this proposed architecture enables open-access permanent evidence that records the agreement between data subjects and service providers relevant to personal data usage.

Three requirements drive the proposed architecture:

1. GDPR requirements cover some articles, e.g., data controllers and data processors need data subjects' consent to begin processing personal data (Articles 6 and 12), systems need to identify who is responsible for processing the personal data (Article 13), data controllers must be able to prove that they obtain data subjects' valid consent (Article 7), data subjects shall be able to adjust which personal data can be collected (Article 18), data subjects shall be able to revoke their consent at any time (Articles 21 and 22), and data subjects shall be able to request for deleting their personal data (Article 17).

2. Functional requirements consist of three elements: 1) the architecture shall decrease the number of interactions between the system and its actors (i.e., lightweight interactions), 2) the architecture shall support consent management on distributed systems, and 3) the consent agreement has been activated and cannot be deleted except for modification.

3. Security and Privacy Requirements consist of six elements: 1) no actors can process any personal data without permission from data subjects, 2) actors have to prove themselves who they are, 3) active consent agreements cannot be unaltered, 4) no actors can disclaim their action on the system, 5) the system must enable audit logs of all events and provide unmodifiable logs to demonstrate its transparency, and 6) the system shall not obtain personal data and neither provide any information leading to identifying data subjects.

Figure 9 shows the system architecture overview. First, a data subject subscribes to a data controller to use its services. The data controller then creates a new consent smart contract indicating whose personal data can be collected and how long to keep it. As for access to the service, the data subject has to grant his/her permission to the data controller to collect personal data via the consent smart contract. The data controller then has permission to obtain this personal data in off-chain data storage. After receiving the request from a data processor, the data controller creates a new purpose smart contract. If the data subject accepts the

agreement of processing purpose via the created smart contract, then the data processor has permission to process personal data. Finally, the supervisory authority shall be able to look into the audit logs to check whether the data controller and data processor have violated data protection regulations.

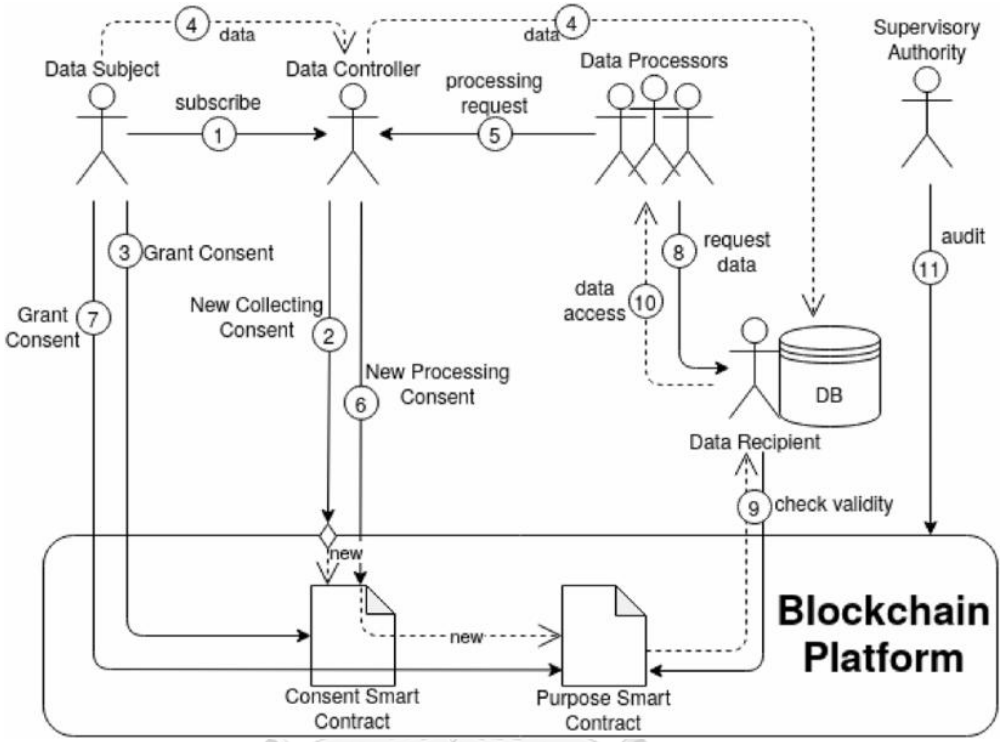



Figure 9: System architecture of personal data management on the blockchain (Figure 1 of Daudén-Esmel et al. [72]).

Therefore, the authors implemented a prototype of the proposed architecture by using smart contracts and deploying them on the local blockchain.

Similarly, Merlec et al. [74] worked on a human-centric approach to design dynamic consent management to enable data subjects to control their personal data usage purposes through smart contracts on a blockchain. Besides, the authors pointed out that centralized systems lack trusted data provenance, transparency, and accountability. The main contribution of this study is the proposed smart-contract-based dynamic consent management system (SC-DCMS) that adheres to the legal use of personal data under GDPR requirements. The proposed architecture covered some articles, e.g., the definition of personal data (Article 4(1) GDPR) indicates a piece of information that could lead to identifying a living person, the operations performed on personal data must rely on the basic principles for processing personal data (Article 5 GDPR). Moreover, consent is a legal basis that empowers data subjects to control their personal data (Articles 4(11) and 7 GDPR).

Figure 10 shows the system architecture overview, which is divided into three layers:

1. Personal data layer enables decentralized applications (Dapps) to provide a user interface for end-users to manage personal data and easily interact with smart contracts. Dapps are applications that have their own smart contracts operating on peer-to-peer blockchain networks [75].
2. Dynamic consent management layer is a smart contract-based middleware for managing dynamic consent, including four main components:

    2.1. User profile management manages user identities, profiles, and roles. As such, it separates modularity purposes into two sub-components: 1) the identity and profile manager is responsible for managing the identity and profile of participant users, and 2) the profile role manager is responsible for managing user roles in request, approval, and revocation processes.

    2.2. Consent agreement management manages data subjects' consent all over the personal data life cycle, which divides into four sub-components: 1) the consent requester handles the request for the collecting and processing of personal data, 2) the consent agreement allows data subjects to manage their consent agreement on each requested personal dataset, 3) the consent tracker enables traceable consent transaction logs on the blockchain, and 4) the consent updater provides data subjects to modify their consent agreement preferences (i.e., consent withdrawal) upon the processing purpose.

    2.3. Smart contract code generator is used to generate smart contracts upon predefined contract templates (i.e., through JSON policy format for the XACML), which comprise four sub-components:

        2.3.1. The data/transaction format examines data provision and common transactional structures.

        2.3.2. The source code generator translates consent agreement policies into smart contracts source code, which indicates one consent agreement per one smart contract.

        2.3.3. The code verifier and validator are used to validate the correctness of generated smart contracts without errors and security exploits.

        2.3.4. The compliance checker is used to verify generated smart contracts against privacy policies and GDPR compliance before deploying them on the blockchain.

2.4. Security and privacy management are divided into four components:

2.4.1. The security manager enables protection mechanisms for protecting the system's resources, e.g., authentication, authorization, and accountability.

2.4.2. The access control manager restricts access to personal data within privacy and access control policies specified in smart contracts.

2.4.3. The privacy manager facilitates data subjects to manage their privacy preferences.

2.4.4. The audit manager handles the logging of all events regarding who requested access to personal data, when personal data was processed, and by whom.

3. Distributed ledger technology and a secure storage layer provide a Quorum blockchain and off-chain data storage using InterPlanetary File System (IPFS) protocol. The Quorum [76] is a permissioned blockchain implemented from the Ethereum [77] codebase, while the IPFS protocol is a peer-to-peer file sharing in decentralized storage [78]. In addition, this layer provides blockchain oracle service (BOS) to expose a secure channel to exchange data between the outside world and blockchain [79].

According to its design, a data subject or a third-party organization first creates a dataset profile which obtains a hashed index. The hashed index directs personal data to off-chain data storage. Second, peer data controllers receive the request for dataset profile publication. Finally, peer data controllers approve the request and publish the data profile into the blockchain.

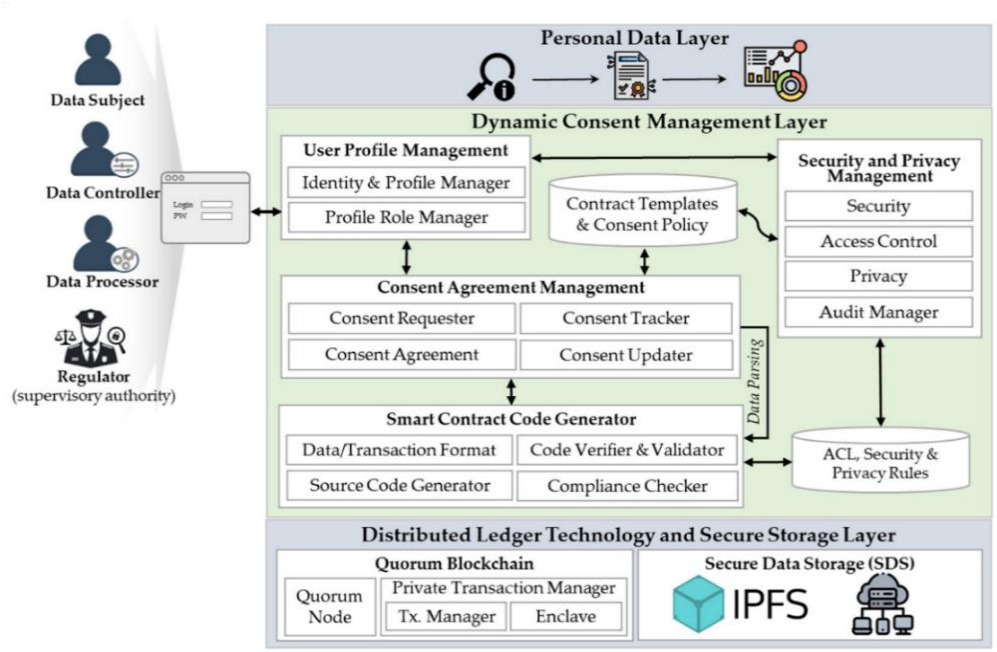



Figure 10: The layered system architecture of SC-DCMS (Figure 3 of Merlec et al. [74]).

Therefore, the authors implemented a prototype of the proposed architecture by using smart contracts and deploying them on the local blockchain, smart contracts written in Solidity language [80], and the local blockchain using the Cakeshop sandbox. As a performance evaluation, the authors examined the impact of workload transactions between IBFT [81] and RAFT [82] consensus protocols. The evaluation results indicate that the proposed system gained high transaction throughputs and minimal latencies for utilizing storage network bandwidth and moderate resources.



The growth of the Internet of Things (IoT) affects individuals' lives, and some devices gather personal data, including behavioral, fingerprint, or biometric data, e.g., gait characteristics and voice. According to Rantos et al. [83], applying GDPR to the IoT is a real challenge. Therefore, the authors proposed the ADvoCATE using a human-centric approach to enable data subjects to manage privacy preferences in the IoT ecosystem upon GDPR requirements.

Their proposed architecture, demonstrated in a cloud service platform (Figure 11), comprises three components:

1. Consent management (CM) component provides data subjects to manage their consent and privacy preferences, including creation, modification, and revocation. ADvoCATE used an ontology to model data protection requirements for ease of data controllers fulfilling GDPR compliance.

2. Consent notary (CN) component offers data integrity and data versioning of data subjects' consent by adopting digital signatures and blockchain technology. This component is responsible for mediating the CM component and blockchain infrastructure. It guarantees consent agreements are complete, accurate, and up-to-date with protection against unauthorized changes. The ADvoCATE focused on the Ethereum blockchain for smart contracts implementation. Figure 12 shows the CN component's workflow. First, the CN component received a new entry consent agreement from the CM component. This consent agreement could be for adding a new one, editing an existing one within policies among the parties, or revocation. Next, the data controller and processor are independently requested to sign the data subject's consent. These digital signatures or hashes are obtained in the blockchain and used when detecting unauthorized modifications. The smart contract (SC) interacts with both the data controller and data processor for initiating, updating, or withdrawing a specific consent agreement regarding a particular IoT device. Moreover, this SC is responsible for managing changes to a consent agreement from consent initiation to final withdrawal, while the various consent versions are represented as data contracts. To check consent integrity, this logic of the SC restricts only the latest version of the consent agreement. Finally, the CN component returns the latest signed consent with its signatures and the SC's address to the CM component.

3. Intelligence component enables conflict detection and suggestion of data subjects' policies incorporated with ontology, which consists of two mechanisms:

    3.1. Intelligent policies analysis mechanism (IPAM) offers conflict detection on data subjects' privacy statements using Fuzzy Cognitive Maps (FCM). The FCM is a learning method used to represent knowledge of systems and causal inference [84].

    3.2. Intelligent recommendation mechanism (IReMe) offers suggestions based on personalized policies to safeguard the privacy of data subjects in real-time using Cognitive Filtering (CF). The CF is rule-based collaborative filtering with the contents of the items and the data subject's consent to avoid any privacy violations [85].

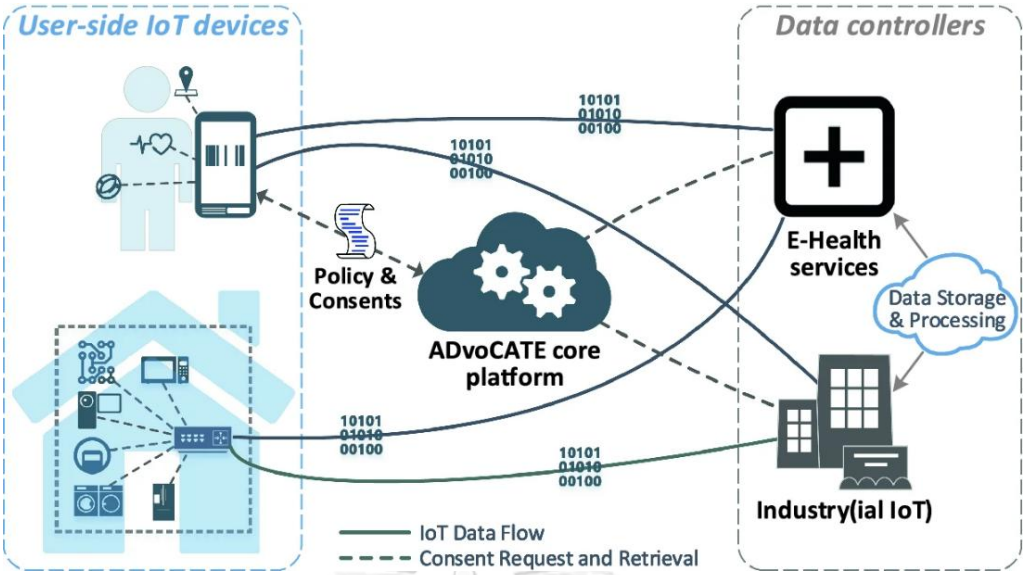



Figure 11: The ADvoCATE architecture (Figure 1 of Rantos et al. [83]).

By its design, a data subject first registers his/her IoT device via the ADvoCATE platform. Then, the data controller places a request on the data subject. Afterward, the data controller and data subject independently send the request to intelligence policies analysis, except for the data controller sending the signed request. The result of the privacy policies analysis is represented as a consent agreement. This consent agreement will be informed to the data subject. If the data subject accepts the condition, then both the data subject and data controller independently sign consent using the SC to obtain his/her consent agreement in the blockchain. Thus, for each access to data collected in an IoT device, the data controller and data processor must verify the data subject's consent validity.



The authors implemented the device registration, consent management component, and smart contract using Node.js, MongoDB database, and Solidity.

The challenge of data sharing receives heightened attention in academic research and business sectors.

Specifically, research in blockchain-based medical data sharing and many studies have been published. For instance, Azaria et al. [86] proposed MedRec as a decentralized electronic medical record (EMR), allowing service providers to share data with others through smart contracts on the Ethereum blockchain. The authors mentioned that the challenge of healthcare interoperability is managing fragments of health records. Data sharing brings much to medical research, such as discovering new treatments, specifying public health issues, and enabling personalized medicine. To bring trust and encourage patients to cooperate by disclosing their medical records, the authors thus designed MedRec to achieve these issues. The use of blockchain

provides a secure way for sharing and auditing data in a distributed manner. Based on MedRec, smart contracts are programmed to manage access privilege control of patients' EMRs. Figure 13 shows MedRec smart contracts and interactions between service providers.

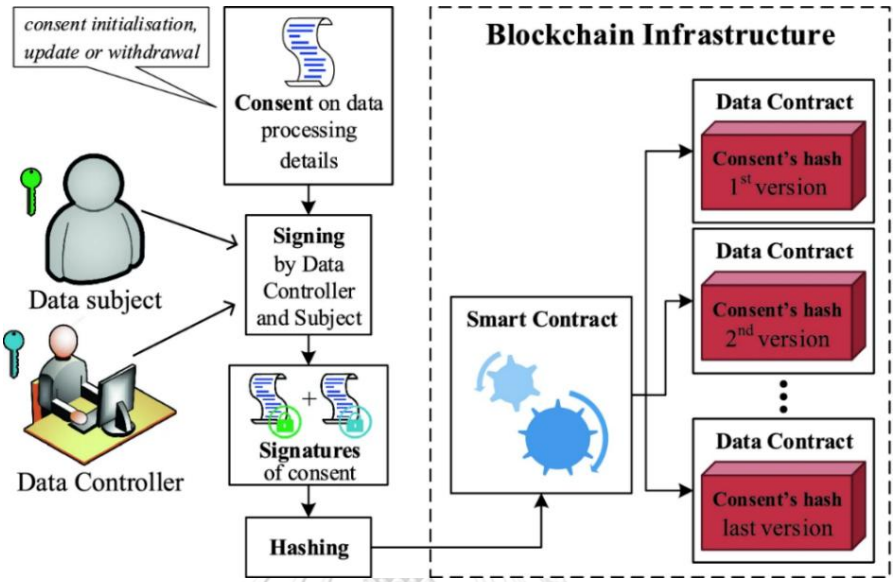



Figure 12: The CM component's workflow (Figure 3 of Rantos et al. [83]).

The system first creates Ethereum addresses and maps to participants' identification via Registrar Contract (RC) to exchange the data between participants (i.e., patients and service providers). Then, the system executes Patient-Provider Relationship Contract (PPR) to establish a peer-to-peer data exchange between patients and service providers. Besides, the PPR determines the pointer of data that specifies where a patient's EMRs are collected and manages the restriction of service providers who wish to access data. The latter is the Summary Contract (SC) employed to track the engagement of participants in data exchange. Therefore, the authors implemented a prototype of the proposed system to prove its functionality.

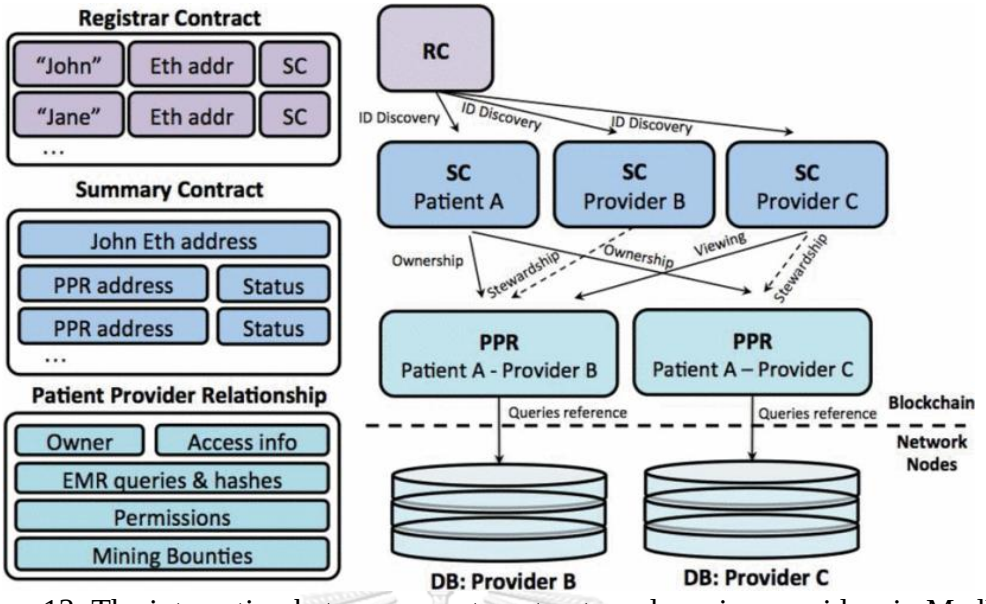



Figure 13: The interaction between smart contracts and service providers in MedRec (Figure 1 of Azaria et al. [86]).

Similarly, Hu et al. [87] stated that the lack of managing fragmented data causes the problem of patient information retrieval from various service providers. Therefore, the authors introduced CrowdMed-II as a framework for managing and sharing data in healthcare by utilizing the Ethereum blockchain. CrowdMed [88] improved this proposed framework to support large-scale adoption.

CrowdMed-II allows patients to maintain ownership over their health data by providing and revoking consented permission. In addition, blockchain in this framework enables transparency, auditability, and incentives, which motivates patients to incorporate into research by sharing their valuable data to improve health outcomes.



The authors separated the proposed framework into three layers:

1. The data storage layer is responsible for managing existing providers' healthcare databases.

2. The central management layer is responsible for conducting a user's identity by mapping the original identity (ID) into a digital signature represented as a virtual ID. This virtual ID is used in blockchain transactions and helps minimize the risk of exposing the patient's real identity. The central management layer is composed of two components:

    2.1. The central query manager handles the query execution on the user's local database and the data storage layer.

2.2. The blockchain obtains patients' permissions and logs every activity that they perform on health data.

3. The user layer comprises four participant roles: patients, data creators, data viewers, and data reviewers.

The proposed framework was designed with two smart contract structures:

1. Patient-Viewer Relationship (PVR)-Centric contract (Figure 14) has a structure similar to the PPR in MedRec [86]. The difference between the PPR and the PVR structures is the number of smart contracts at which to be executed for gathering a patient's health records. For example, the PPR must execute multiple smart contracts to retrieve a patient's health records among service providers. As a result, it causes high gas consumption and low efficiency. On the other hand, the PVR structure has to execute only a PVR to retrieve all health records for one patient.

2. Provider-Patient-Viewer Relationship (PPVR)-Centric contract structure (Figure 15) improves from the PPR and PVR structures. Moreover, the proposed framework designed two more smart contracts:

   2.1. The Provider Contract (PC) is used by a medical service provider and obtains health records for all patients which providers give.

   2.2. The ReViewer Contract (RVC) has a function similar to PC; the responsible role is data reviewer, who acts as a provider to review remarks on the health data of each provider to improve its quality. However, there are no databases for data reviews because all remarks have been stored in the health data-sharing system.

3. In addition, the proposed framework enables assigning a role to a group of users (i.e., group-based access) instead of assigning a role to a user, which eases management access rights. As a performance evaluation, the authors determined two experiments. First, they evaluated gas consumption in every transaction after executing transactions sequentially on six smart contracts into a personal Ethereum network. The six smart contracts of this experiment are as follows: 1) the PPR-centric, 2) the PPR-centric with group-based access, 3) the PVR-centric, 4) the PVR-centric with group-based access, 5) the PPVR-centric, and 4) the PPVR-centric with group-based access. The first experiment results indicated that the PVR-centric contract structures with group-based access consumed the lowest gas. As for the second

experiment, the authors then executed the PVR-centric with group-based access in the same sequence as the first experiment by measuring throughput and latency, while the second experiment results indicate that the registration transactions caused latency significantly higher than average.

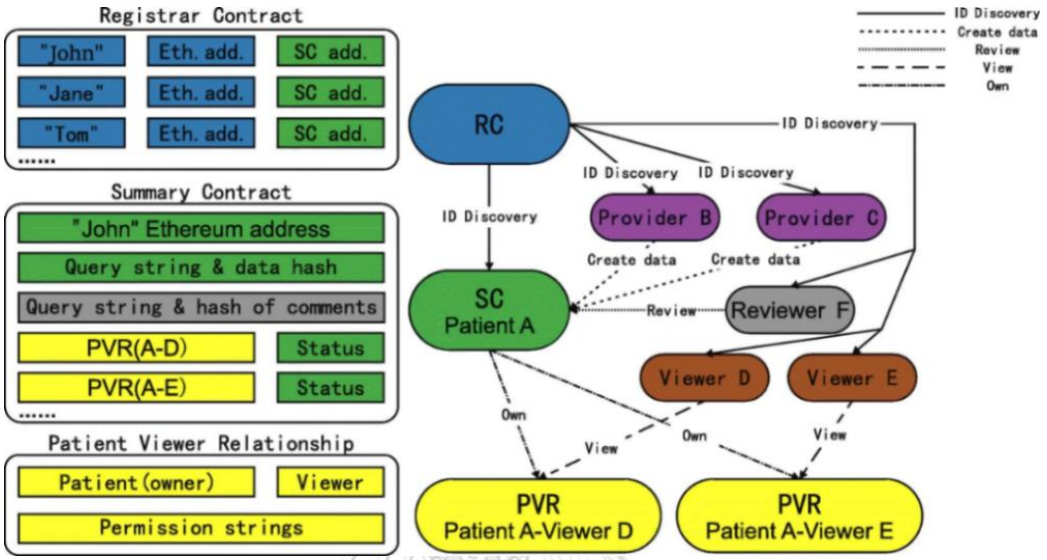



Figure 14: The PVR-centric contract structure in CrowdMed-II (Figure 2 of Hu et al. [87]).

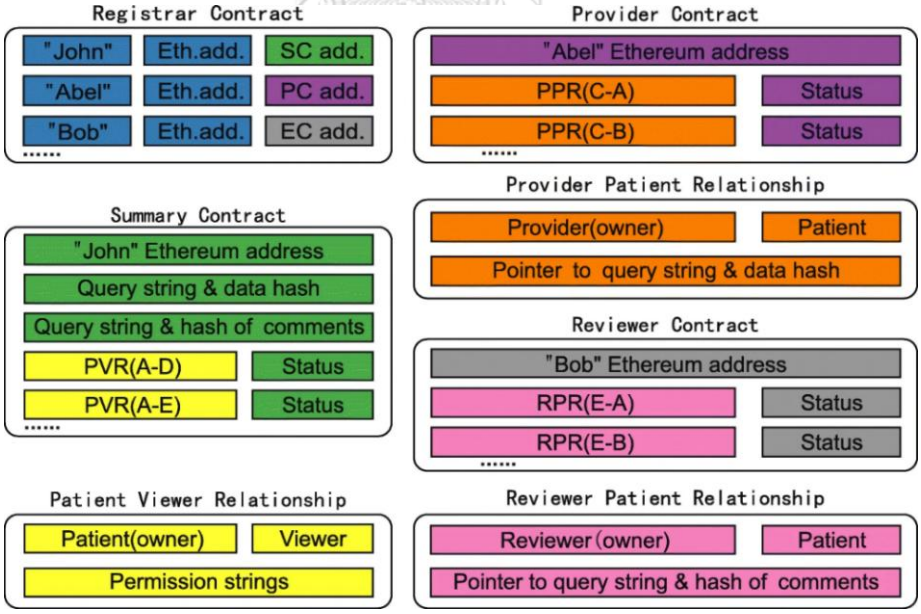



Figure 15: The PPVR-centric contract structure in CrowdMed-II (Figure 3 of Hu et al. [87]).

Table 3 demonstrates the difference between related works within data privacy and consent management contexts. The first six studies used formal methods for modeling the system's behavior according to desired privacy policies [53, 56, 57, 59,

63, 64]. They formalized a portion of the process, which makes it unclear how to implement an entire process. Besides, two of the first six studies used model checking to verify model correcstness.

On the other hand, the rest of the studies focused on conceptual and architectural frameworks rather than logical ones, which makes it difficult to build software systems based on these frameworks; more than half of the studies considered GDPR as part of software design [63, 64, 66, 70, 72, 74, 83], but it is still unclear which GDPR articles they covered. Furthermore, studies have separated into two groups: centralized and distributed systems; most of the studies proposed frameworks based on distributed systems (e.g., microservices, blockchain).

The distributed system is a group of software components that are located on different networked computers [89], while the centralized system is one unified system that maintains the entire operation [89, 90]. Both systems are managed by a central authority, except for blockchain. The studies that employed blockchain technology [70, 72, 74, 83, 86, 87] have integrated off-chain data storage for collecting personal data instead of on-chain, so they can delete personal data where necessary. As for a security service, most studies determined access control based on the notion of purpose or consent, which help identify the security access of an individual data within the purpose or given consent; less than half of the studies integrated consent service as part of software design, which comprises only two functionalities, such as manipulation and withdrawal consents [63, 64, 70, 72, 74, 83]. Indeed, the audit trail is essential for data protection to defend against data breaches, and several studies included audit logs as part of their proposed frameworks [70, 72, 74, 83, 86, 87]. Finally, no studies specify the records restriction of data retrieval for minimizing data breaches.

Table 3: Comparison with related works in the context of data privacy and consent management.

| | [53] | [56] | [57] | [59] | [63] | [64] | [66] | [70] | [72] | [74] | [83] | [86] | [87] | Our study |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Formal method** | √ | √ | √ | √ | √ | √ | | | | | | | | √ |
| **End-to-End process** | | | | | | | | √ | √ | √ | √ | | | √ |
| **Data protection regulation** | | | | | | | | | | | | | | |
| GDPR | | | | | √ | √ | √ | √ | √ | √ | √ | | | √ |
| DPA | | | | √ | | | | | | | | | | |
| **Target system** | | | | | | | | | | | | | | |
| Centralized | √ | √ | √ | | √ | | | | | | | | | √ |
| Distributed | | | | √ | | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| **Security service** | | | | | | | | | | | | | | |
| Access control based on the notion of purpose or consent | | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| **Consent service** | | | | | | | | | | | | | | |
| Manipulation | | | | | √ | √ | √ | √ | √ | √ | √ | | | √ |
| Withdrawal | | | | | √ | √ | √ | √ | √ | √ | √ | | | √ |
| Portability | | | | | | | √ | | | | | | | √ |
| Renewal | | | | | | | | | | | | | | √ |
| **Audit logs track activities** | | | | | | | | √ | √ | √ | √ | √ | √ | √ |
| **Restricted records data retrieval** | | | | | | | | | | | | | | √ |

# CHAPTER III
# BACKGROUND

The relevant theories of this thesis include consent management (CM), Event-B, blockchain technology, and smart contract.

## 3.1. Consent Management

According to the literature, consent management represents a software component that provides a mechanism for managing consent and controlling personal data lifecycle based on a given consent under data protection regulations. However, standardizing consent management is a complex challenge. Consent is the legal basis for personal data processing activities and is used in most cases [91]. The GDPR mandates that data controllers must be able to prove the validity of data subjects' consent and could face a fine of up to 20 million euros or 4% of annual revenue (Article 83 GDPR) if they fail to comply. In the survey research conducted by Kurteva et al. [92], they presented solutions based on ontologies to improve an understanding of consent management implementation. The use of ontology provides the knowledge ground upon which the consent and personal data lifecycle relate to GDPR requirements. The present study introduced a model of the consent lifecycle (Figure 16), which derives from the approaches related to consent.

The consent lifecycle describes the process of conducting consent in CM, which comprises four key steps:

3.1.1. Manipulation of consent, e.g., consent has been changed, confirmed, and reaffirmed.

3.1.2. Checking consent validity, if the data subject's consent is invalid (i.e., consent is revoked, expired, invalidated, or refused), then the CM system sends a consent request to inform the data subject. Otherwise, the data controller or data processor is allowed to process personal data.

3.1.3. Comprehension of informed consent represents the data subject must have adequate information to understand the consent agreement of what he/she agrees.

3.1.4. Decision-making on informed consent indicates that the data subject has the right to accept or refuse the consent agreement to process his/her personal data.

To help better understand the context of consent, the authors summarized the classes and attributes essential for modeling consent from existing ontologies [17, 93-95]. We thus analyzed and recategorized these classes and object attributes according to Table 1 and Table 2, used as a guideline for our study, presented in Table 4. Furthermore, based on the list of competency questions for consent management (Table 5) defined by the authors, used as the comparison of baseline between existing studies [17, 93, 95-99], we added additional questions to Table 5. These questions are represented in our work, including question numbers 7, 8, 14, 16, 20 and 21.

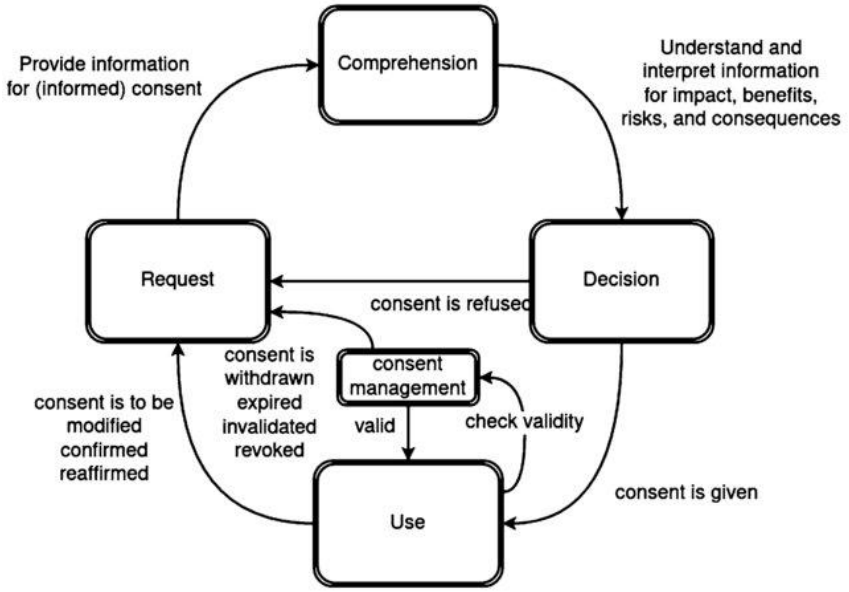



Figure 16: The consent lifecycle within consent-based approaches (Figure 1 of Kurteva et al. [92]).

Table 4: The comparison between classes and object attributes of existing ontologies in consent context.

| | [17] | | [93] | | [94] | | [95] | |
|---|---|---|---|---|---|---|---|---|
| | **Class** | **Attribute** | **Class** | **Attribute** | **Class** | **Attribute** | **Class** | **Attribute** |
| **Consent service** | | | | | | | | |
| Manipulation | Consent, ConsentingParty, ConsentObligation, ConsentFormat | given_at, given_by, given_for, data_has_format | Consent | isConsentForDataSubject, forPersonData, forPurpose, forProcessing, hasStatus, hasExpiry | ConsentAssertion | | Consent | hasConsentNotice, hasExpiry, hasExpiryTime, hasExpiryCondition, isExplicit, hasProvisionBy |
| | ConsentingParty | gives_consent | DataSubject | hasConsent | DataSubject | | DataSubject | |
| | | | Purpose | isPurposeForConsent | | | Purposes | |
| | Data | category | PersonalData | isPersonalDataForConsent | DataEvent | splog:instanceData | | |
| | ConsentObligation | activity | Processing | isActionForPurpose | Process | splog:performedBy, skos:member | | |
| | Activity | | | | Activity | splog:performedBy | | |
| | | | Status | isStatusForConsent | | | | |
| Withdrawal | Consent, ConsentingParty | given_at, given_by | Consent | hasStatus | ConsentRevocation | splog.revoke | Consent | hasWithdrawalMethod, hasWithdrawalTime |
| Portability | | | | | | | | |
| Renewal | | | | | | | | |
| **Security service** | | | | | | | | |
| Access Control | Permission | validity_time given_to given_for_data, given_action, given_for_condition | | | | | | |
| **Logging** | | | | | | | | |
| Activity audit log | | | | | LogEntry | spl:hasData, spl:hasProcessing, spl:hasPurpose, spl:validityTime, splog:dataSubject, splog:activity, splog:immutableRecord | | |
| | | | | | ImmutableRecord | splog:contentHash, splog:userHash, splog:hashAlgorithm | | |

Table 5: The competency questions for consent management in which relevant to GDPR articles, extended from Kurteva et al. [92] (cont'd).

| No. | Question | Relevant entity/process | GDPR article |
|---|---|---|---|
| **Questions regarding consent** | | | |
| 1 | Who is responsible for gathering consent agreements? | Data Controller, Data Processor | Articles 4(7), 6(1a) and 28 |
| 2 | For what purposes does a consent agreement cover? | Purpose | Articles 4(4), 6(1a) and 7 |
| 3 | How to revoke consent agreement? | Consent Withdrawal | Article 7<br>Recitals 63 and 66 |
| 4 | How long does a consent agreement last? | Consent Retention/Validity/Expiration | Article 5(1)<br>Recitals 32 and 42 |
| 5 | When has consent been granted? | Consent Retention | Articles 4(11), 7 and 6(1a) |
| 6 | When has consent been withdrawn? | Consent Withdrawal | Articles 17 and 19 |
| 7 | When is consent permitted data to be portable? | Consent Data Portability | Article 20 |
| 8 | When has consent been renewed? | Consent Retention/Validity/Expiration | Articles 4(11), 7 and 6(1a) |
| **Questions regarding personal data** | | | |
| 9 | What is regarded as personal data? | Categories of Personal Data | Articles 4(1) and 9 |
| 10 | How has personal data been used? | Data Processing | Article 4(2) |
| 11 | How has personal data been gathered? | Data Collection | Articles 12, 13 and 14<br>Recitals 39, 58, 62 and 73 |
| 12 | To whom personal data is disclosed? | Recipient, Data Sharing | Articles 4(7), 6, and 28 |
| 13 | Who is in charge of personal data? | Data Controller | Article 24<br>Recitals 74 and 79 |
| 14 | How to minimize the data collection? | Data Collection | Article 5(1c) |
| 15 | Where has personal data been obtained? | Data Storage | Article 5(1e) |
| 16 | When should personal data be pseudonymized? | Pseudonymization | Article 4(5)<br>Recital 26 |
| **Questions regarding the data controller** | | | |
| 17 | Who has been identified as the data controller? | Data Controller | Articles 4(7) and 28 |
| 18 | How to reach out to the data controller? | Data Controller, Contact Information | Articles 4(7), 14 and 28 |
| 19 | What is the data controller in charge for? | Data Controller, Responsibilities, Obligations | Articles 4(7), 14, 28 and 37 |
| 20 | How to embed data protection as a default setting for processing personal data? | Data Controller | Article 25 |

Table 5: The competency questions for consent management in which relevant to GDPR articles, extended from Kurteva et al. [92].

| No. | Question | Relevant entity/process | GDPR article |
|---|---|---|---|
| **Questions regarding the data processor** | | | |
| 21 | Who has been identified as the data processor? | Data Processor | Article 4(8) |
| **Questions regarding the data subject** | | | |
| 22 | Who has been identified as the data subject? | Data Subject | Article 4(1) |
| **Questions regarding third party** | | | |
| 23 | Whom to reach out to? | Contact Information, Third Party | Articles 12, 13 and 14 |

### 3.2. Event-B

Event-B is a formal model development method in mathematical terms to prove that a formal model fulfills a set of defined specifications [23, 24]. Event-B is separated into two parts: 1) contexts, the static specification is used to define static properties of the model, containing carrier sets *s*, constants *c*, and axioms *A(s, c)*, and 2) machines, the dynamic specification is used to define behavioral properties of the model, containing state variables *v*, invariants *I(s, c, v)*, and events *evt*. The refinement process in Event-B is a crucial feature for modeling a complex system [100, 101], as presented in Figure 17.

It begins with an abstract model and gradually adds features one at a time until a concrete model is completed [102, 103]. This technique makes the model more straightforward to prove than modeling an entire system at once. This technique makes the model more straightforward to prove than modeling an entire system at once. However, the Event-B model's consistency requires proof obligations, which must be proved to guarantee that all invariants are preserved within every event occurrence [102, 103].

The Event-B constructs Proof obligations (POs) from the invariants *I*, the local concrete invariants *J* (i.e., gluing invariants), and the specifications of abstract and concrete operations (Figure 18 A). There are various types of proof obligations [104]. For example, Invariant Preservation (INV) ensures that each invariant is preserved within each event occurrence. Event-B produces an INV when an action modifies variable values directly into a specific invariant. For example, Figure 19, shows that the Login event comprises three guards and one action, as shown on the left-hand side of the figure. The guard grd1 indicates that the current session has not been created. The guard grd2 means a user must be authorized to access the system and is not currently logged on. Finally, the guard grd3 guarantees that inserting an ordered pair (s ↦ u) into the variable sessions must satisfy inv1. If all guards are valid, the action act1 inserts an ordered pair (s ↦ u) to the sessions directly to inv1, Event-B thus generates Login/inv1/INV to ensure that the values of the session change preserve inv1, as shown on the right-hand side of the figure.

Well-Definedness of an event Guard (GRD) ensures that a guard has been formulated well-defined. Event-B generates GRD when there are some potentially ill-defined expressions (e.g., partial, modulo, and max-min functions) in a guard condition. For example, Figure 20 shows that the AddPatient event comprises four guards and one action, as shown on the left-hand side of the figure.

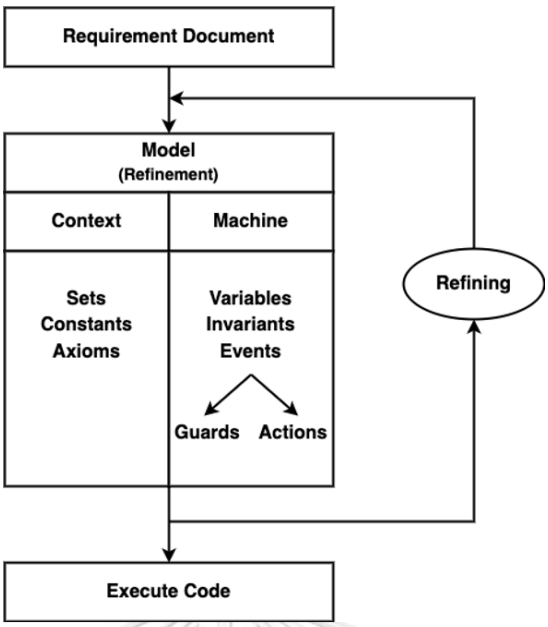



Figure 17: The process of refinement in Event-B (Figure 1 of Jarrar & Balouki [101]).

The guard grd1 indicates the user is an authorized user and is currently logged on. The guard grd2 indicates the user must have a NursingStaff role, while the guard grd3 means a new patient has not been added to the system. Finally, the guard grd4 guarantees that this event must be deactivated if any states that enter query states. If all guards are valid, the action act1 inserts the patient *p* into the variable patients. However, guards grd1 and grd4 are involved in the sessions as a potentially ill-defined expression. To ensure that these two guards are Well-Definedness (WD) conditions, Event-B thus generates AddPatient/grd1/WD and AddPatient/grd4/WD, as shown on the right-hand side of the figure.



The generated POs must be discharged to prove the correctness of the given properties in the Event-B model. The guards are a set of predicates indicated as pre-conditions that should be true before executing the event. An event consists of local variables *l*, guards, and actions. Each state machine event may have one or more guards *G(l, s, c, v).* When guards are valid, the actions *S(l, s, c, v)* will modify the state variable v, as shown in equation (1).

$$\text{evt} \triangleq \textbf{any } l \textbf{ when } G(l, s, c, v) \textbf{ then } v :| S(l, s, c, v) \textbf{ end} \quad (1)$$

The POs in the Event-B model guarantee that each event must be shown to preserve the model invariants, where $v'$ is the state variables after executing the event, and *BA(l, c, v, v')* is the before-after predicate of the assignment event, as shown in equation (2).

$$I(s, c, v) \wedge A(s, c) \wedge G(l, s, c, v) \wedge BA(l, c, v, v') \Rightarrow I(s, c, v') \quad (2)$$

For each event, the post-condition will automatically be derived from its guards and actions [105-107]; an Event-B model is deadlocked if all events are disabled in a particular state [100, 108, 109]. Besides, there is an open-source tool that supports Event-B, called Rodin Platform [24]. The Rodin Platform is an Eclipse-based IDE that enables a variety of plug-ins for developing models, such as a proof obligation generator, provers, model-checker (ProB), etc. Nevertheless, Event-B does not provide deadlock detection [110, 111]. So, we must plug ProB into the Rodin Platform to enable deadlock detection, test-case simulation, and state reachability [110-112]. Figure 18 demonstrates the process of model checking in ProB to prove whether a given model satisfies given specifications. If the output is true, a given model is valid. Otherwise, ProB produces a counterexample.

To start developing an Event-B model, we need to install Java and Rodin Platform following these instructions: http://www.event-b.org/install.html. To enable a model checker, it needs to install the ProB plug-in. First, open the Help menu and click "Install new software." Then, select the update site project, which begins with the title "ProB - " and click on "ProB for rodin2". Finally, enter the Next button and complete the installation.

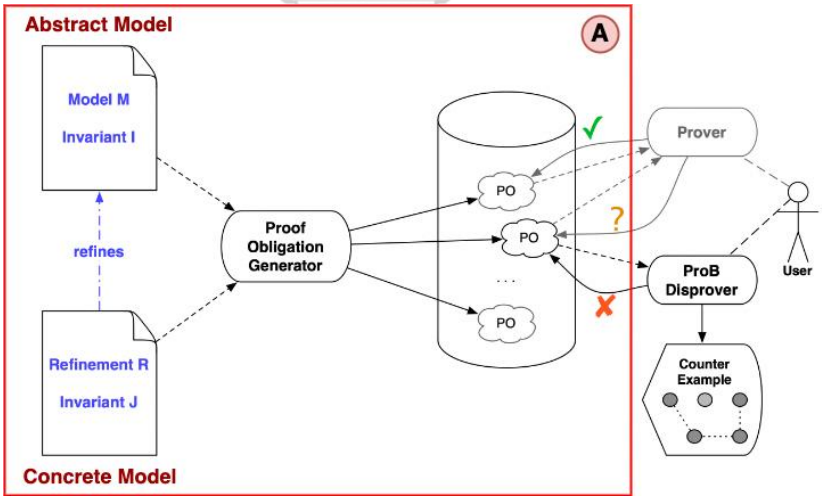



Figure 18: The process of model checking in ProB (Figure 1 of Ligot et al. [111]). (A) demonstrating the generation of proof obligations in compliance with the abstract and concrete models.

To begin developing Event-B models, we recap some set notations used in our study. We thus determine set predicates by P and Q, set expressions by S, T, and E, single variables by x and y, a list of variables by z, and the relation by r, $r_1$, and $r_2$. The set notations are as follows: 1) the predicate logic, e.g., conjunction ($P \wedge Q$), disjunc-

tion (P ∨ Q), and existential quantification ((∃z·P) ∧ Q), 2) the pre-defined sets, e.g., booleans (BOOL), i.e., TRUE or FALSE, and empty set (∅), 3) the set operators, e.g., membership (E ∈ S), union (S ∪ T), intersection (S ∩ T), powerset (ℙ(S)), a subset (S ⊆ T), not a subset (S ⊈ T), ordered pairs (x ↦ y), set difference (S \ T), cartesian product (S × T), and 4) the relations identifying the connection between sets, e.g., relations (S ↔ T), domain (dom(r)), range (ran(r)), partial functions (S ⇸ T), partial injections (S ⤔ T), domain restriction (S ◁ T), domain subtraction (S ⩤ T), range restriction (S ▷ T), range subtraction (S ⩥ T), relational image (r[S]), and overriding $r_1 \Leftarrow r_2$. More detailed information regarding Event-B notation is publicly accessible at [113].

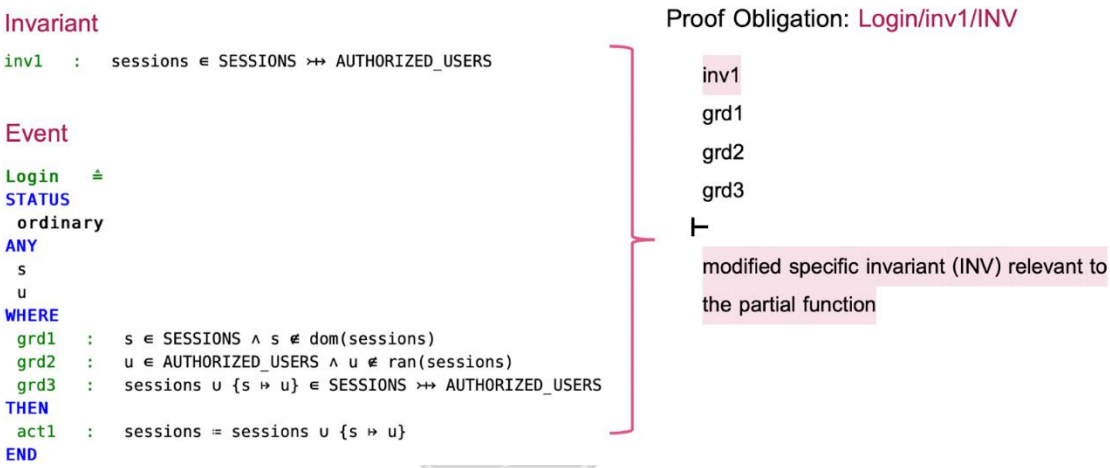



Figure 19: The example of generating INV proof obligation from the Login event.

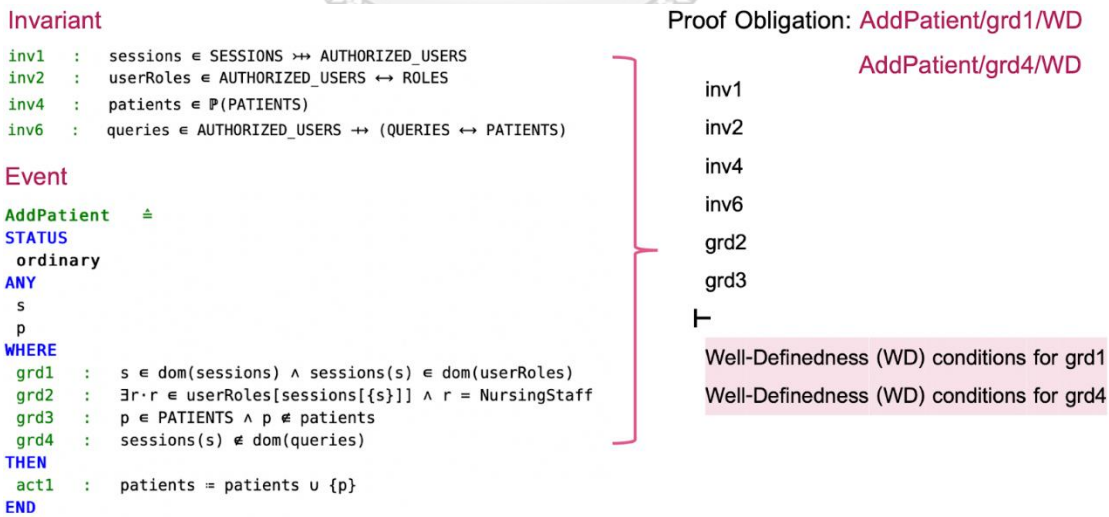



Figure 20: The example of generating GRD proof obligation from the AddPatient event.

### 3.3. Blockchain Technology

Blockchain technology is the innovation of distributed ledger technology, which enables the secure transfer and storage of digital assets without central authority management [114-116]. On the other hand, blockchain provides potential solutions to safeguard data owners from unauthorized or unlawful collecting and processing of personal information [48, 114, 115]. For instance, blockchain enables security and tamper-proof transactions among untrusted participants, eliminates the management of center entities, and utilizes cryptographic hash functions to protect the integrity of data stored in the distributed ledger [117, 118]. The blockchain structure lists ordered transactions [115, 119, 120] called blocks (Figure 21). To begin with, adequate participants have confirmed the transaction, it is permanently inserted into the list of blocks, and each block is securely attached using cryptography. The elements of a block are as follows: 1) the data which contains information depends on the objective of using blockchain, 2) the hash is a unique identifier in which generated by a nonce (i.e., a nonce is a random 32-bit number for ensuring the validity of the block hash), 3) the previous hash is a hash value of the parent block except for a genesis block that does not contain a previous hash, and 4) the metadata contains descriptive information about data, e.g., block number, and timestamp.

The consensus mechanism in blockchain represents a set of methodologies used to verify and confirm the legitimacy of a new transaction before being added to a distributed ledger to ensure fault tolerance and security [73, 120, 121]. Besides, it involves the assignment of participants to work on tasks or activities to maintain blockchain infrastructure by devoting necessary resources, e.g., crypto-asset and energy. Indeed, there have been two common consensus mechanisms: Proof of Work (PoW) and Proof of Stake (PoS). The PoW is a mechanism that outlines the difficulty or rules (e.g., the cryptographic math problem) to which the mining competitors must dedicate their computing resources to process transactions. The mining competitors who first solve math problems receive a fee for mining as a reward, while the PoS is similar to PoW but the better version. The PoS is a mechanism that allows participants who own cryptocurrency and are randomly selected to validate transactions and earn rewards. Therefore, the differences between PoW and PoS are the means they determine who gets the privilege to validate transactions and energy usage.

The PoS is more energy efficient than PoW because it eliminates duplicate tasks. Furthermore, the blockchain is divided into three types [121-123]:

3.3.1. The private blockchain requires participants to be granted before entering the network ecosystem, e.g., Hyperledger Fabric (HF) [124]. It is a centralized system with a central authority to manage user access control and

permissions. Besides, it may offer a token or not, depending on blockchain preferences.

3.3.2. The public blockchain is publicly accessible and has no restrictions on particular participants and existing validators in the network, e.g., Bitcoin and Ethereum. It guarantees that no central authority controls the network and is a fully distributed system.

3.3.3. The consortium or hybrid blockchain comprises two types: 1) some nodes are partially private, and 2) all the rest are public. This characteristic is called a hybrid blockchain, e.g., the Ripple network [125], and there are two types of users: 1) the users who have complete control over the blockchain and determine the access privileged for individual users, and 2) the others who only have access to the blockchain.

In our study, we focus on the Ethereum blockchain. Ethereum is an open-source, public, and blockchain-based distributed system. It supports the PoS consensus mechanism and smart contract functionality. The Ethereum blockchain enables a peer-to-peer network with a trusted ledger of transactions and facilitates smart contracts to share data securely.

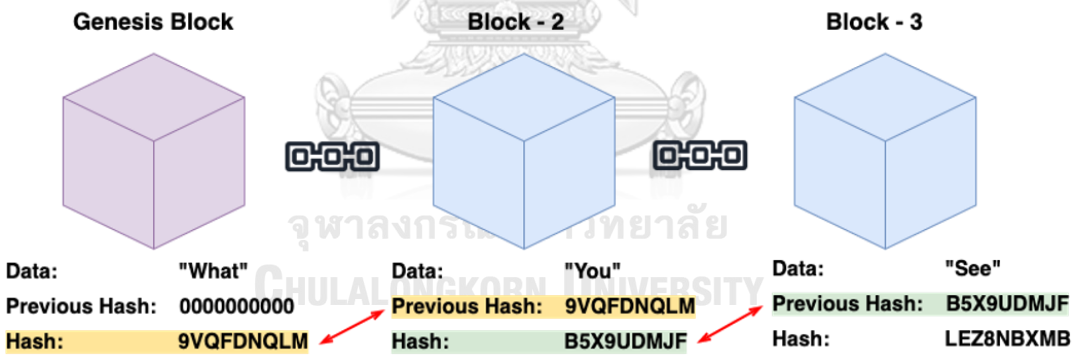


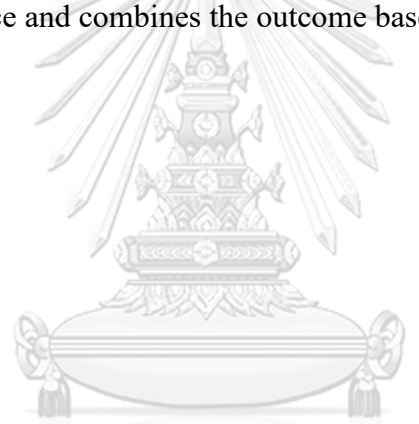

Figure 21: List of blocks of transactions in a blockchain data structure, modified from Figure 1 of Chinnasamy et al. [115].

### 3.4. Smart Contract

Smart contracts are programs based on certain logic and agreements that automatically execute transactions if conditions are met [73, 126, 127]. They are hosted on a blockchain network, and all participants can access results without third parties involved. Moreover, smart contracts are composed of three types [128]: 1) smart legal contracts are used to create legally binding agreements on the parties which derive from legal requirements, 2) decentralized autonomous organizations (DAOs) are used to create a set of rules by a group of people to self-govern themselves, and 3) applica-

tion logic contracts are used to contain an application-specific code in cooperation with other blockchain contracts.

In real-world development, smart contracts may need to retrieve information outside the blockchain, but they cannot accomplish that [79, 129]. So, the oracle has been introduced to solve this problem. The oracle is a middleware that constructs a secure connection between the blockchain and various resources outside the chain, called off-chain. There are five types of the oracle as follows [129]: 1) the hardware oracle is used to collect data from physical devices (e.g., heat sensors, geolocators) and push it to smart contracts, 2) the software oracle is used to retrieve information from online resources, such as public transport, temperature, and supply up-to-date information to smart contracts, 3) the inbound oracle enables a function for receiving external data and forwarding it to smart contracts, 4) the outbound oracle allows smart contracts for sending data to external data sources outside the chain, and 5) the consensus-based oracle provides the query of multiple oracle sources to reduce the risks of using only one source and combines the outcome based on their consensus.



# CHAPTER IV
# FORMAL MODELS FOR CONSENT MANAGEMENT IN CENTRALIZED SYSTEMS

This chapter is a slightly modified version of a manuscript published in the *Journal of Logical and Algebraic Methods in Programming*, Volume 128, August 2022, and has been reproduced here with the permission of the copyright holder.

To develop CM for centralized systems, we reviewed GDPR articles from a system design perspective to build GDPR-aware system models related to PbD [130]. The key roles in GDPR include 1) data subject, 2) data controller, and 3) data processor. A data subject has full control of his/her data [Article 4(1) describes personal data as information that leads to the recognition of an individual]. The data controller is the organization or person who establishes policies for managing a life cycle of personal data processing, as described in Article 4(7). Finally, the data processor is the organization or person who manipulates individual data according to the policies given by the data controller, as described in Article 4(8). We then defined a set of primitive state machines that cover the basics of consent management functionality, consisting of four state machines: 1) the restricted processing state machine (RPSM), 2) the withdrawal approval state machine (WASM), 3) the portable approval state machine (PASM), and 4) the consent renewal state machine (CRSM). Moreover, we mapped each state machine to GDPR articles (Table 6), which helps developers better understand how to translate GDPR articles into system requirements and design.

To define a set of states and transitions in RPSM, we determined the logic involved in processing activities by following privacy methods included in Article 5. This article outlines the context of personal data processing that respects six data protection principles as follows: 1) it requires that personal data are stored and processed legitimately (‘lawfulness, fairness and transparency’), 2) the purpose of data processing must be clearly defined before beginning the process (‘purpose limitation’), 3) personal data should only include a minimum amount of data that is strictly necessary to accomplish a specific purpose (‘data minimization’), 4) personal data must be complete and kept up-to-date (‘accuracy’), 5) the data controller must ensure that personal data will be only retained for a necessarily limited period (‘storage limitation’), and 6) personal data must be ensured with consistency and confidentiality over its life cycle (‘integrity and confidentiality’). The responsibilities of the data controller must comply with these fundamentals.

Table 6: List of proposed state machines and GDPR articles they covered.

| Machine name | GDPR article | Summary |
|---|---|---|
| **RPSM** | Article 4(1) | Personal data is any information (e.g., full name, social security number, medical records) that can directly or indirectly identify a person. |
| | Article 4(11) | The data subject's consent indicates that the data subject gives his/her unambiguous consent. |
| | Article 5 | Any processing of personal data must be done following the GDPR's six legal bases in Article 5, including 1) personal data processing must always be legitimate, explicit, and genuine with data subjects, 2) personal data must be collected and processed for a specific purpose, 3) personal data must be collected only the data necessary upon the consent, 4) personal data must be accurate and kept up-to-date, 5) personal data must be collected for a specific purpose upon expiration of the time period, and 6) personal data must be accurate and consistent over its entire lifecycle. |
| **WASM** | Article 7(3) | Data subjects have the right to revoke their consent for processing at any time. |
| | Articles 17 and 19 | Data subjects have the right to request to erase their personal data. |
| **PASM** | Article 20 | Data subjects have the right to request a portable copy of their personal data. |
| **CRSM** | Article 6(1) | Data subjects need to sign consent before processing their personal data. |

Furthermore, we built WASM as a model dealing with the right to withdraw consent. Article 7(3) describes that the data subjects are able at all times to revoke consent for the processing of their data. After revoking the consent, personal data should be erased automatically [39]. This revoking is also known as the right to erasure (‘right to be forgotten’) under Articles 17 and 19.

The right to data portability, GDPR Article 20, permits data subjects to control their data by receiving and transferring personal data in a machine-readable format across controllers. We modeled this discrete behavior through the state transitions in PASM.

For the renewal of consent effects within GDPR Article 6(1), the data controller may offer a data subject to extend the retention period to continue using the products and services. If the data subject accepts the retention offer, the data controller or the data processor can legitimately process his/her data.

However, if the data subject declines the retention offer, the data controller must revoke the data subject’s consent. We also modeled this discrete behavior through the state transitions in CRSM.

### 4.1. CM State Machines in Centralized Systems

This thesis proposes a set of formal models integrating privacy concerns into software development under the GDPR. According to Article 4(11) GDPR, consent is a data subject’s voluntary agreement to permit either a data controller or a data processor to process his/her personal data under specific conditions. We considered consent management an essential component of the system design [131, 132]. This means a system must not process personal data without the validity of a data subject’s consent. In this thesis, we built state machines to depict the dynamic behavior of privileged permissions based on the relationships of the data subject’s consents, user roles, and data subject’s data fields.

Following PbD concepts and GDPR guidelines present in Table 6, demonstrated via a software platform for cancer precision medicine called RUN-ONCO [133]. RUN-ONCO allows users (i.e., oncologists, nurses, researchers) to manage and analyze patient clinical and genetic data, which assists oncologists in designing treatment plans for patients with cancers. Patients need to sign consent before an authorized user enters their clinical and genetic data into the platform. Figure 22A shows how RUN-ONCO supports authentication based on roles but lacks the consent management functionality. The informed consent process for clinical trials has been paper-based and outside the platform. Without built-in consent management functionality, a platform is difficult to control and maintain patients’ privacy preferences. To implement a

consent management functionality for an existing system without clear guidelines, developers will need to spend much time analyzing and redesigning the system without knowing if the redesigned platform covers GDPR requirements. To enhance RUN-ONCO support consent management (Figure 22B and Figure 22C), by following RPSM, we first need to alter the *Patient* class structure to support dynamic access attributes within role-based consent. Second, we further create the *PatientConsent* class to hold patients' consent. To manage the right to withdraw consent (WASM), the right to data portability (PASM), and consent renewal (CRSM), we then update the *PatientConsent* class by adding methods that obtain the logic of the following state machines. Third, we must modify logic in the *AuthenticationService* class to manage the authorized access patients' attributes within role-based consent. Fourth, the *PatientService* class needs to modify the logic for restricting patient information retrieval according to given authorization.

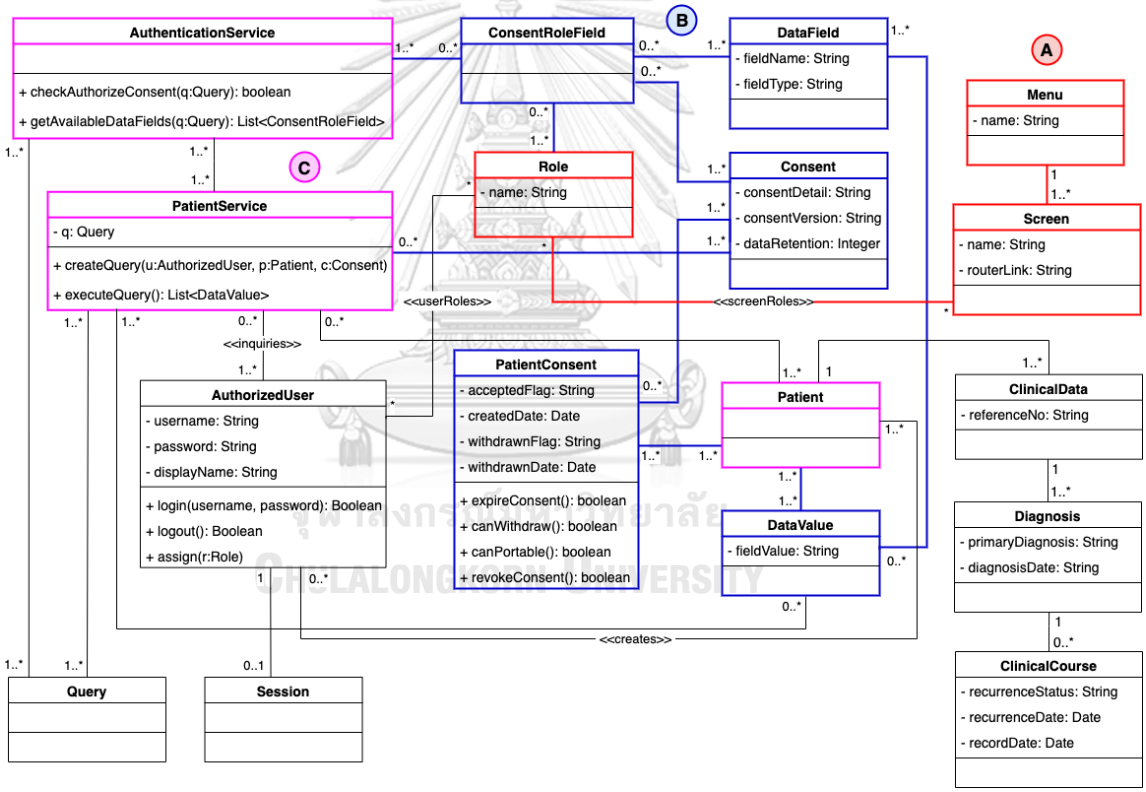



Figure 22: Class diagram demonstrating how a software platform for cancer precision medicine manages roles and permissions to restrict users' access to screens. (A) an authentication module associated with users, roles, and screens. (B) new classes added to RUN-ONCO for supporting dynamic access attributes within role-based consent. (C) relevant classes needed to be enhanced to support consent management.

We provided four state machines that cover the main aspects of consent management. First, the RPSM explains the behavior of restricting unauthorized user access from storing and processing personal data (Figure 23). Based on RPSM, a user

must first login to access the platform. By logging in, the user with the NursingStaff role will be able to add a new patient and informed consent. Moreover, to access the patient's personal data, a user has been granted a role based on the patient's consent.

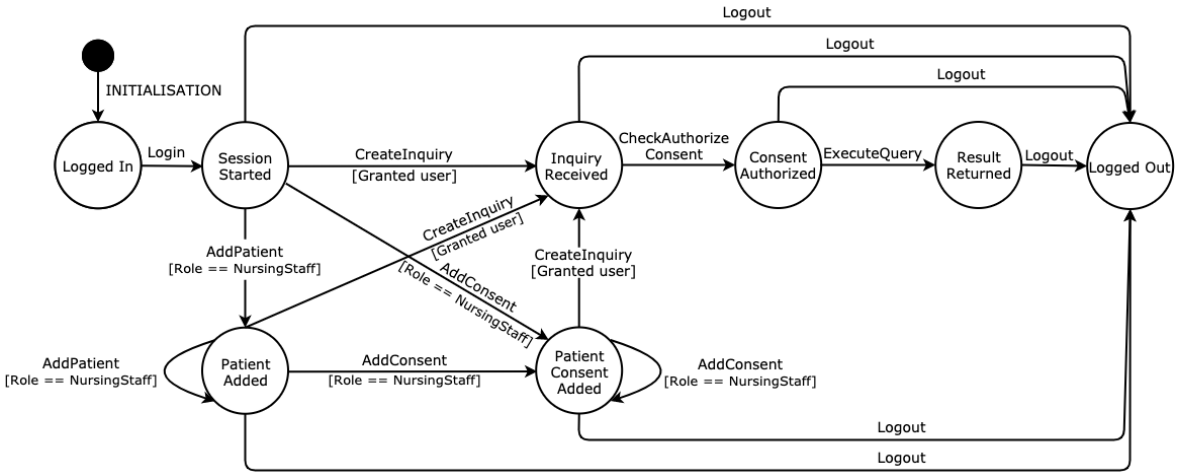



Figure 23: Restricted Processing State Machine (RPSM) describing the transition states and events used to restrict the processing of personal data.

Second, WASM explains the behavior of approval for withdrawing a data subject's consent and deleting his/her personal data (Figure 24). Based on WASM after a patient requests to withdraw consent, the user with the LegalStaff role login to the platform and initiates a withdrawal process. A user with the LegalApprover role will then review a withdrawal request based on the initiated process The platform allows patients to withdraw consent at any time, as long as the patient has the adequate capacity to make decisions about medical treatment. After assessing a patient's capacity, if a patient can make his/her own treatment decisions, the approver will approve to revoke consent and submit a delete request to erase the patient's personal data. Otherwise, the approver will reject the withdrawal request.

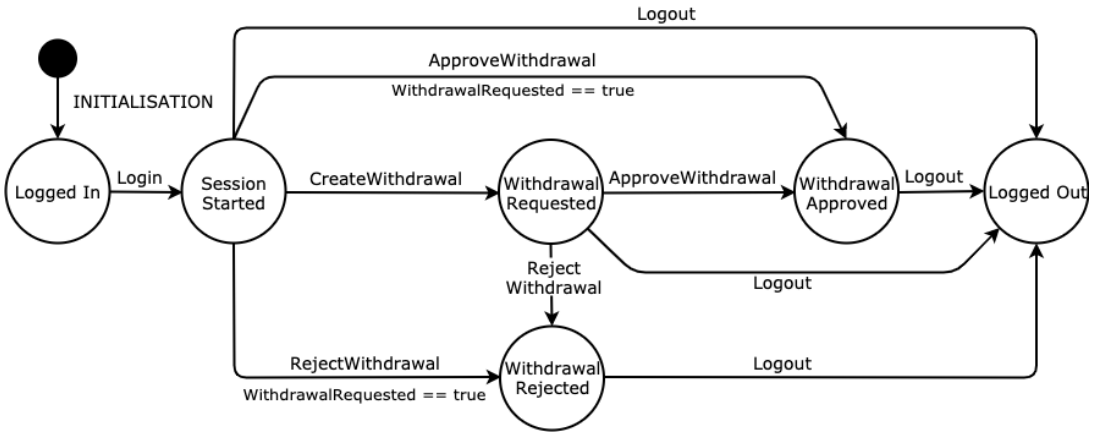



Figure 24: Withdrawal Approval State Machine (WASM) describing the transition states and events used to manage a consent revocation request.

Third, PASM explains approval behavior for transferring a data subject's personal data (Figure 25). Based on PASM, after a patient requests a portable copy of the personal data, the user with the LegalStaff role login to the platform and initiates a portable process. The platform offers data portability that allows patients to request all relevant health and genetic data, as long as the patient accepts prerequisite conditions (e.g., a fee for preparing and transmitting personal data to other data controllers). The approver will approve the portable request if the patient accepts prerequisite conditions. Otherwise, the request will be rejected.

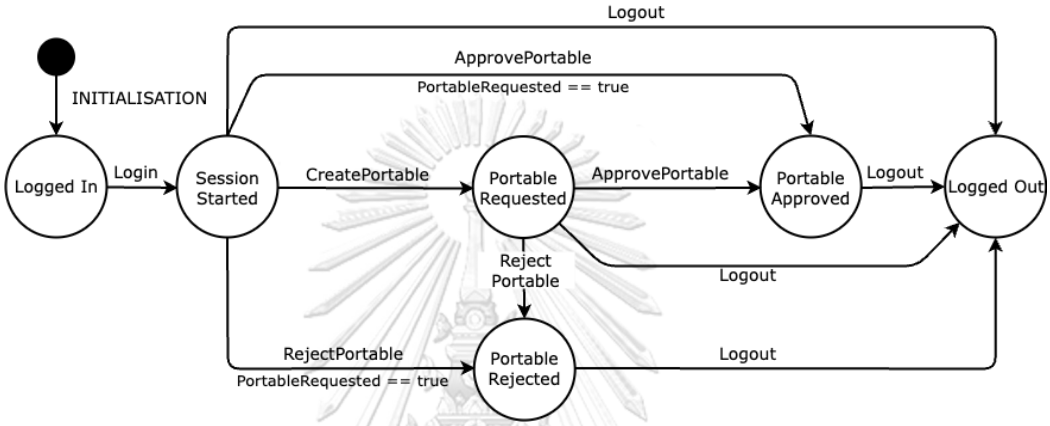



Figure 25: Portable Approval State Machine (PASM) describing the transition states and events used to manage a data transferring request.

Fourth, CRSM explains approval behavior for extending the retention period of a data subject's consent (Figure 26). Based on CRSM, the user with the LegalStaff role login to the platform and initiates a renewal process. The patient will then review a renewal request based on the initiated process.



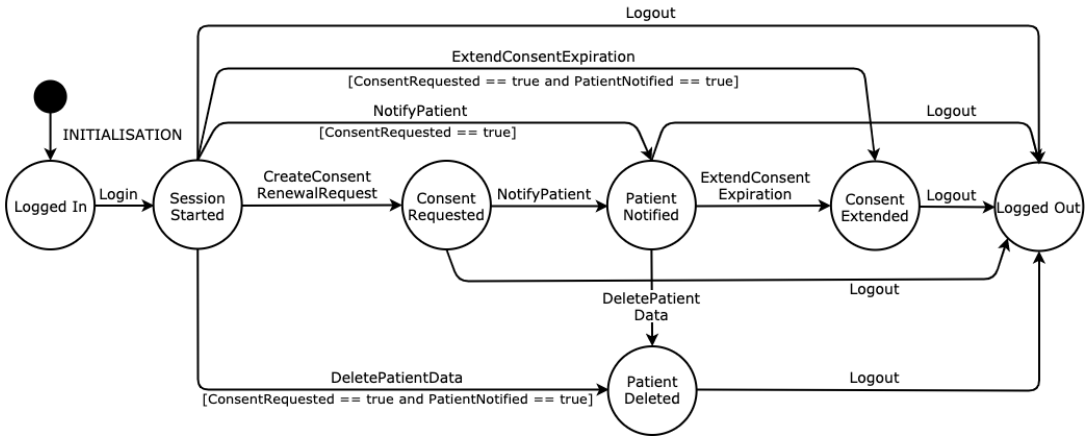



Figure 26: Consent Renewal State Machine (CRSM) describing the transition states and events used to manage a data retention request.

The platform offers a mechanism that allows patients to increase the retention period for keeping the personal data it collects and processes. After the patient replies accept status (i.e., approve, reject) to the platform, the legal staff responds, followed by accept status. If the patient approves, the legal staff increases the retention period within informed consent. Otherwise, the legal staff submits a delete request to erase the patient's personal data.

## 4.2. Formal Development in Event-B

We created an Event-B context and defined necessary sets, constants, and axioms that are relevant to health information privacy as follows: 1) PATIENTS is a set of data subjects, 2) SESSIONS represents a set of sessions associated with an authorized user (i.e., AUTHORIZED_USERS), 3) ROLES (e.g., NursingStaff, Oncologist, LabStaff) specifies a set of user permissions to prevent unauthorized access attempts, 4) FIELDS is a set of patient data fields (e.g., HN, Name, Age, Gender), and 5) STATUSES is a set of workflow statuses (e.g., Void, Approved, Rejected). The state machines will refer to this context, which contains global static variables to construct the states and transitions. We built the state machines and defined preserved invariants as the properties of the states using common naming, e.g., inv1, inv2. Events represent state transitions in Event-B. For each event, we defined guards as preconditions and actions as state variable assignments using the common naming, e.g., grd1, grd2, and act1, act2, respectively.

### 4.2.1. Restricted Processing State Machine (RPSM)

The RPSM (Figure 23) created based on the Event-B method, describes the dynamic behavior of restricted data processing in terms of events. For this state machine, we defined invariants that hold all possible states as follows:



```
inv1: sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
inv2: userRoles ∈ AUTHORIZED_USERS ↔ ROLES
inv3: pc ∈ PATIENTS ↔ CONSENTS
inv4: patients ∈ ℙ(PATIENTS)
inv5: crf ∈ CONSENTS ⇸ (ROLES ↔ FIELDS)
inv6: queries ∈ AUTHORIZED_USERS ⇸ (QUERIES ↔ PATIENTS)
inv7: pf ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
inv8: authorizedConsent ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ CONSENTS)
```

The variable *sessions* holds the one-to-one relationship between SESSIONS and AUTHORIZED_USERS, which means a single session can contain only one user. To limit the data breach risk, we applied role-based access control (RBAC) in the model and defined the *userRoles* as a relationship between AUTHORIZED_USERS and ROLES. It indicates that each user can have multiple roles. The variable *patients* contains the set of PATIENTS during the refinement

process. According to GDPR, we need a patient's consent to process data. Hence, we declared the *pc* as a set of ordered pairs (p ↦ c) where p ∈ PATIENTS and c ∈ CONSENTS. The use of *pc* here specifies that a patient can have more than one consent. The *crf* defines (c ↦ rf) as a set of ordered pairs where c ∈ CONSENTS, rf ∈ ROLES ↔ FIELDS, which combines the relationships of consents, roles, and data fields to restrict user's access over the specific fields of data based on the given consent of data subjects. The model allows a data controller or a data processor to execute a query per data subject to minimize the risk of retrieving large amounts of personal data by creating the variable named *queries*. The *queries* defines (u ↦ qp) as a set of ordered pair where u ∈ AUTHORIZED_USERS, qp ∈ QUERIES ↔ PATIENTS to hold personal data inquiries. We stored the result of a query in variable *pf*, which is a set of ordered pairs (u ↦ pf) where u ∈ AUTHORIZED_USERS and pf ∈ PATIENTS ↔ FIELDS. The *pf* represents the final output of RPSM that describes how the model provides consent-based permission for each user to perform on specified data fields. We defined *authorizedConsent* (u ↦ pc) as a set of ordered pairs where u ∈ AUTHORIZED_USERS and pc ∈ PATIENTS ↔ CONSENTS, indicating the valid consent for the authorized user.

The INTIALISATION is an event that was fired first. It allows the initialization of arbitrary values and establishes invariants before other events are executed. Listing 1 introduces the Login event. The guards are defined with three preconditions. First, the guard grd1 ensures that any session *s* is a member of SESSIONS and *s* does not exist in the domain of sessions. Second, the guard grd2 ensures that any user *u* is a member of AUTHORIZED_USERS and *u* does not exist in the range of sessions. Third, the guard grd3 ensures that adding an ordered pair (s ↦ u) into *sessions* must satisfy the invariant inv1. Whenever all guards of the Login event are valid, the action act1 adds an ordered pair (s ↦ u) to the sessions, which indicates that the user has successfully logged in.

```
Login ≙
Any s,u Where
  grd1 : s ∈ SESSIONS ∧ s ∉ dom(sessions)
  grd2 : u ∈ AUTHORIZED_USERS ∧ u ∉ ran(sessions)
  grd3 : sessions ∪ {s ↦ u} ∈ SESSIONS ⇸ AUTHORIZED_USERS
Then
  act1 : sessions ≔ sessions ∪ {s ↦ u}
End
```

Listing 1: The Login event.

Listing 2 shows how we formally modeled the adding of a new patient using Event-B. The guards are defined with four preconditions. First, the guard grd1 ensures that the user successfully got the *session* and the user role is within

the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a nursing staff. Third, the guard grd3 ensures that the patient does not exist in the variable patients. Fourth, the guard grd4 ensures that the AddPatient event does not fire after entering the inquiry states. Whenever all guards are valid, the action act1 adds the patient *p* to the patients.

```
AddPatient ≙
Any s,p Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = NursingStaff
  grd3 : p ∈ PATIENTS ∧ p ∉ patients
  grd4 : sessions(s) ∉ dom(queries)
Then
  act1 : patients ≔ patients ∪ {p}
End
```

Listing 2: The AddPatient event.

Listing 3 shows the formal model of how a new patient's consent is added to the system. The guards are defined with six preconditions. First, the guard grd1 ensures that the user is successfully logged in with the user role known by the system. Second, the guard grd2 ensures that one of the user roles is a nursing staff. Third, the guard grd3 ensures that any patient *p* is a member of patients and consent *c* is a member of the domain *crf*. Fourth, the guard grd4 ensures that a new ordered pair (p ↦ c) does not exist in the *pc*. Fifth, the guard grd5 ensures that adding an ordered pair (p ↦ c) into variable *pc* must satisfy the invariant inv3. Sixth, the guard grd6 ensures that the AddConsent event does not fire after entering the inquiry states. Whenever all guards are valid, the action act1 adds an ordered pair (p ↦ c) to the *pc*.



```
AddConsent ≙
Any s,p,c Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = NursingStaff
  grd3 : p ∈ patients ∧ c ∈ dom(crf)
  grd4 : p ↦ c ∉ pc
  grd5 : pc ∪ {p ↦ c} ∈ PATIENTS ↔ CONSENTS
  grd6 : sessions(s) ∉ dom(queries)
Then
  act1 : pc ≔ pc ∪ {p ↦ c}
End
```

Listing 3: The AddConsent event.

Listing 4, Listing 5, and Listing 6 show how we formally model the handling of a user inquiry, starting from creating an inquiry (Listing 4), verifying the consent validation (Listing 5), and executing the inquiry (Listing 6). The

CreateInquiry event (Listing 4) is used to prepare a new query under the currently logged on user. The guards are defined with three preconditions. First, the guard grd1 ensures that the user is successfully logged in with the user role known by the system. Second, the guard grd2 ensures that any query *q* is a member of QUERIES, patient *p* is a member of the domain *pc*, and session(s) does not exist in the domain *queries*. Third, the guard grd3 ensures that when adding an ordered pair (q ↦ p) to the *queries(sessions(s))*, the invariant inv6 must be satisfied. Whenever all guards are valid, the action act1 adds an ordered pair (q ↦ p) to the *queries(sessions(s))*.

```
CreateInquiry ≙
Any s,q,p Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : q ∈ QUERIES ∧ p ∈ dom(pc) ∧ sessions(s) ∉ dom(queries)
  grd3 : queries ⩤ {sessions(s) ↦ {q ↦ p}s} ∈
         AUTHORIZED_USERS ⇸ (QUERIES ↔ PATIENTS)
Then
  act1 : queries(sessions(s)) ≔ {q ↦ p}
End
```

Listing 4: The CreateInquiry event.

The CheckAuthorizeConsent event (Listing 5) is used to verify if the patient's consent does not expire. The guards are defined with six preconditions. First, the guard grd1 ensures that the user is successfully logged in and the user has created queries. Second, the guard grd2 ensures that *consentExpired* is a member of the boolean and *consentExpired* is FALSE. Third, the guard grd3 ensures that the consent *c* is a member of *pc[{p}]* and *c* is a member of the domain *crf*. Fourth, the guard grd4 ensures that one of the user roles of the logged on user is a member of the domain *crf(c)*. Fifth, the guard grd5 ensures that a new ordered pair (p ↦ c) does not exist in the domain *authorizedConsent*. Sixth, the guard grd6 ensures that when adding an ordered pair (p ↦ c) to the *authorizedConsent(sessions(s))*, the invariant inv8 must be satisfied. Whenever all guards are valid, the action act1 adds an ordered pair (p ↦ c) to the *authorizedConsent(sessions(s))*.

```
CheckAuthorizeConsent ≙
Any s,p,c,consentExpired Where
   grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(queries)
   grd2 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
   grd3 : c ∈ pc[{p}] ∧ c ∈ dom(crf)
   grd4 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r ∈ dom(crf(c))
   grd5 : sessions(s) ∉ dom(authorizedConsent)
   grd6 : authorizedConsent ⩤ {sessions(s) ↦ {p ↦ c}} ∈
          AUTHORIZED_USERS ⇸ (PATIENTS ↔ CONSENTS)
Then
```

```
  act1 : authorizedConsent(sessions(s)) ≔ {p ↦ c}
End
```

Listing 5: The CheckAuthorizeConsent event.

The ExecuteQuery event (Listing 6) is used to get the result of a query. The guards are defined with five preconditions. First, the guard grd1 ensures that the user is successfully logged in and the user has created queries. Second, the guard grd2 ensures that any patient *p* is a member of the range of *queries(sessions(s))* and *c* is a member of the domain *crf*. Third, the guard grd3 ensures that *sessions(s)* is a member of the domain *authorizedConsent* and an ordered pair (p ↦ c) is a member of *authorizedConsent(sessions(s)*. Fourth, the guard grd4 ensures that sessions(s) does not exist in a domain pf. The grd4 represents that the query has not yet been executed within the user session. Fifth, the guard grd5 ensures that when adding a cartesian product *{p} × ran( userRoles[sessions[{s}]] ⩤ crf(c))* to the *pf(sessions(s))*, the invariant inv7 must be satisfied. The variable pf represents the result of the query based on consent-permission which is defined in the variable *crf*. Whenever all guards are valid, the action act1 adds a cartesian product *{p} × ran(userRoles[sessions[{s}]] ⩤ crf(c))* to the *pf(sessions(s))*.

```
ExecuteQuery ≙
Any s,p,c Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(queries)
  grd2 : p ∈ ran(queries(sessions(s))) ∧ c ∈ dom(crf)
  grd3 : sessions(s) ∈ dom(authorizedConsent) ∧ p ↦ c ∈
         authorizedConsent(sessions(s))
  grd4 : sessions(s) ∉ dom(pf)
  grd5 : pf ⩤ {sessions(s) ↦ {p} × ran(userRoles[sessions[{s}]] ⩤
         crf(c))} ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
Then
  act1 : pf(sessions(s)) ≔ {p} × ran(userRoles[sessions[{s}]] ⩤
         crf(c))
End
```

Listing 6: The ExecuteQuery event.

The Logout event (Listing 7) is fired when a user signs out of the system. The guards are defined with five preconditions. First, the guard grd1 ensures that the user is successfully logged in. Second, the guard grd2 ensures that removing *sessions(s)* from *queries* must satisfy the invariant inv6. Third, the guard grd3 ensures that removing *sessions(s)* from *authorizedConsent* must satisfy the invariant inv8. Fourth, the guard grd4 ensures that removing sessions(s) from pf must satisfy the invariant inv7. Fifth, the guard grd5 ensures that removing *sessions(s)* from *sessions* must satisfy the invariant inv1. Whenever all guards of the Logout event are valid, the action act1 removes *sessions(s)* from *queries*, ac-

tion act2 removes *sessions(s)* from *authorizedConsent*, action act3 removes *sessions(s)* from *pf*, and action act4 removes *sessions(s)* from *sessions*.

```
Logout ≙
Any s Where
  grd1 : s ∈ dom(sessions)
  grd2 : {sessions(s)} ⩤ queries ∈ AUTHORIZED_USERS ⇸
         (QUERIES ↔ PATIENTS)
  grd3 : {sessions(s)} ⩤ authorizedConsent ∈
         AUTHORIZED_USERS ⇸ (PATIENTS ↔ CONSENTS)
  grd4 : {sessions(s)} ⩤ pf ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
  grd5 : sessions ⩥ {sessions(s)} ∈ SESSIONS ⤖ AUTHORIZED_USERS
Then
  act1 : queries ≔ {sessions(s)} ⩤ queries
  act2 : authorizedConsent ≔ {sessions(s)} ⩤ authorizedConsent
  act3 : pf ≔ {sessions(s)} ⩤ pf
  act4 : sessions ≔ sessions ⩥ {sessions(s)}
End
```

Listing 7: The Logout event.

### 4.2.2. Withdrawal Approval State Machine (WASM)

The WASM (Figure 24) was created based on the Event-B method to describe the dynamic behavior of the model for revoking an individual consent and automatically deleting personal data. We defined the invariants for the WASM model as follows. The first three invariants are the same as of RPSM.

```
inv1: sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
inv2: userRoles ∈ AUTHORIZED_USERS ↔ ROLES
inv3: pc ∈ PATIENTS ↔ CONSENTS
inv4: withdrawalState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
inv5: markAsDeleted ∈ PATIENTS ↔ CONSENTS
```

Additionally, we declared two more variables in the context to support the refinement of WASM. First, the *withdrawalState* defines (pc ↦ status) as a set of ordered pairs, where pc ∈ PATIENTS ↔ CONSENTS, and status ∈ STATUSES that holds the status of the withdrawal request. Second, the *markAsDeleted* contains the relationship between PATIENTS and CONSENTS that represents the patient as deleted under the consent.

The INTIALISATION event gets fired first to initialize the variables. Then the Login event starts to get a new session which holds a user role. The CreateWithdrawal event (Listing 8) is used to initiate a withdrawal request. The guards are defined with four preconditions. First, the guard grd1 ensures that the user successfully got the session and the user role is within the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a legal

staff. Third, the guard grd3 ensures that any patient *p* is a member of the domain *pc*, where consent *c* is a member of the range *pc*, and the ordered pair (p ↦ c) does not exist in the domain *withdrawalState*. Fourth, the guard grd4 ensures that when adding Void status to the *withdrawalState({p ↦ c})*, the invariant inv4 must be satisfied. Whenever all guards are valid, the action act1 adds a status Void to the *withdrawalState({p ↦ c})*, which will trigger the approval workflow.

```
CreateWithdrawal ≙
Any s,p,c Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧ {p ↦ c} ∉ dom(withdrawalState)
  grd4 : withdrawalState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
Then
  act1 : withdrawalState({p ↦ c}) ≔ Void
End
```

Listing 8: The CreateWithdrawal event.

Listing 9 shows the formal model of how to approve the consent withdrawal. The guards are defined with six preconditions. First, the guard grd1 ensures that the user successfully got the session and the user role is within the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a legal approver. Third, the guard grd3 ensures that *pc1* is a member of the domain *withdrawalState* and the status of the *withdrawalState(pc1)* is Void. Fourth, the guard grd4 ensures that when updating Void to Approved status must satisfy the invariant inv4. Fifth, the guard grd5 ensures that *canWithdraw* is a member of a boolean and *canWithdraw* is TRUE. The TRUE boolean here indicates that all required activities before withdrawal were done. Sixth, the guard grd6 ensures that when adding *pc1* to the *markAsDeleted*, the invariant inv5 must be satisfied. Whenever all guards are valid, the action act1 updates the withdrawalState({p ↦ c}) from Void to Approved status, and act2 adds *pc1* to *markAsDeleted*.



```
ApproveWithdrawal ≙
Any s,pc1,canWithdraw Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(withdrawalState) ∧ withdrawalState(pc1) = Void
  grd4 : withdrawalState ⩤ {pc1 ↦ Approved} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canWithdraw ∈ BOOL ∧ canWithdraw = TRUE
  grd6 : markAsDeleted ⩤ pc1 ∈ PATIENTS ↔ CONSENTS
Then
  act1 : withdrawalState(pc1) ≔ Approved
  act2 : markAsDeleted ≔ markAsDeleted ⩤ pc1
```

```
End
```

Listing 9: The ApproveWithdrawal event.

Otherwise, the *RejectWithdrawal* event (Listing 10) will be fired if the variable *canWithdraw* is FALSE, assuming that some required activities were not completed. The status of *withdrawalState(pc1)* will then be changed from Void to Rejected according to the action act1. In both cases, the request must be approved or rejected by the legal approver. Especially in the ApproveWithdrawal event, we defined the *markAsDeleted* to hold the deleted patients for the approved cases. The Logout event is fired to indicate that the user is no longer in the system.

```
RejectWithdrawal ≙
Any s,pc1,canWithdraw Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(withdrawalState) ∧ withdrawalState(pc1) = Void
  grd4 : withdrawalState ⩤ {pc1 ↦ Reject} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canWithdraw ∈ BOOL ∧ canWithdraw = FALSE
Then
  act1 : withdrawalState(pc1) ≔ Rejected
End
```

Listing 10: The RejectWithdrawal event.

### 4.2.3. Portable Approval State Machine (PASM)

The PASM (Figure 25) created based on Event-B describes the dynamic behavior of the model allowing patients to port their personal data. The first three invariants of the model are the same as the previous two models and a new variable named *portableState* was introduced to hold the status of data portability request.

```
inv1: sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
inv2: userRoles ∈ AUTHORIZED_USERS ↔ ROLES
inv3: pc ∈ PATIENTS ↔ CONSENTS
inv4: portableState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

The behavior of PASM is similar to the WASM but is used for different purposes. After initializing the variables and creating a new session, the CreatePortable event (Listing 11) will be started. The guards are defined with four preconditions. First, the guard grd1 ensures that the user successfully got the session and the user role is within the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a legal staff. Third, the guard grd3 ensures that any patient *p* is a member of the domain *pc*, consent *c* is a member of the

range *pc*, and the new ordered pair (p ↦ c) does not exist in the domain *portableState*. Fourth, the guard grd4 ensures that when adding Void status to the *portableState({p ↦ c})*, the invariant inv4 must be satisfied. Whenever all guards are valid, the action act1 adds the status Void to the *portableState({p ↦ c})*.

```
CreatePortable ≙
Any s,p,c Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧ {p ↦ c} ∉ dom(portableState)
  grd4 : portableState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
Then
  act1 : portableState({p ↦ c}) ≔ Void
End
```

Listing 11: The CreatePortable event.

After the CreatePortable event is done, the ApprovePortable event (Listing 12) will be fired if the variable *canPortable* is TRUE. The status of *portableState(pc1)* will then be changed from Void to Approved according to the action act1.

```
ApprovePortable ≙
Any s,pc1,canPortable Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(portableState) ∧ portableState(pc1) = Void
  grd4 : portableState ⩤ {pc1 ↦ Approved} ∈
          (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canPortable ∈ BOOL ∧ canPortable = TRUE
Then
  act1 : portableState(pc1) ≔ Approved
End
```

Listing 12: The ApprovePortable event.

Otherwise, the RejectPortable event (Listing 13) will be fired to change the status from Void to Rejected. In both cases, the portability request must be determined by the legal approver.

```
RejectPortable ≙
Any s,pc1,canPortable Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(portableState) ∧ portableState(pc1) = Void
  grd4 : portableState ⩤ {pc1 ↦ Rejected} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

```
  grd5 : canPortable ∈ BOOL ∧ canPortable = FALSE
Then
  act1 : portableState(pc1) ≔ Rejected
End
```

Listing 13: The RejectPortable event.

### 4.2.4. Consent Renewal State Machine (CRSM)

The CRSM model (Figure 26) created by Event-B describes the dynamic behavior of the model to extend the renewal period of a consent. The first three invariants of the model are the same as the previous three models. We also defined four more invariants and variables to cover the refinement of CRSM as follows.

```
inv1: sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
inv2: userRoles ∈ AUTHORIZED_USERS ↔ ROLES
inv3: pc ∈ PATIENTS ↔ CONSENTS
inv4: isConsentExpired ∈ (PATIENTS ↔ CONSENTS) ⤖ BOOL
inv5: markAsDeleted ∈ PATIENTS ↔ CONSENTS
inv6: markAsReceived ∈ PATIENTS ↔ CONSENTS
inv7: consentRenewalState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

The first variable *isConsentExpired* is a set of ordered pairs represents by one-to-one relationship (pc ↦ expired) where pc ∈ PATIENTS ↔ CONSENTS, and expired ∈ BOOL (i.e., TRUE or FALSE). The second variable *markAsDeleted* contains the relationship between PATIENTS and CONSENTS that represents the patient as deleted under the consent. The third variable *markAsReceived* contains the relationship between PATIENTS and CONSENTS that keeps track of the patient's incoming response to the renewal request. The fourth variable is *consentRenewalState*, which has held the status of consent renewal. It is a set of ordered pairs (pc ↦ status), where pc ∈ PATIENTS ↔ CONSENTS, and status ∈ STATUSES that holds the status of patient's consent.

By default, the INTIALISATION event is fired to initialize the variables before executing a renewal request. The Login event is triggered to retrieve the user login information, and the session has started. The CreateConsentRenewalRequest event (Listing 14) is used to initiate a consent renewal request. The guards are defined with seven preconditions. First, the guard grd1 ensures that the user successfully got the session and the user role is within the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a legal staff. Third, the guard grd3 ensures that any patient *p* is a member of the domain *pc*, consent c is a member of the range *pc*, and a new ordered pair (p ↦ c) does not exist in the domain *consentRenewalState*. Fourth, the guard grd4 ensures that *expired* is a member of a boolean and *expired* is TRUE. Fifth, the guard

grd5 ensures that *isWithdrawn* is a member of a boolean and *isWithdrawn* is FALSE. Sixth, the guard grd6 ensures that when adding Void status to the *consentRenewalState({p ↦ c})*, the invariant inv7 must be still satisfied. Seventh, the guard grd7 ensures that when adding TRUE to the *isConsentExpired({p ↦ c})*, the invariant inv4 must be satisfied. Whenever all guards are valid, the action act1 adds a status Void to the *consentRenewalState({p ↦ c})*, and act2 adds TRUE to the *isConsentExpired({p ↦ c})*.

```
CreateConsentRenewalRequest ≙
Any s,p,c,expired,isWithdrawn Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧ {p ↦ c} ∉ dom(consentRenewalState)
  grd4 : expired ∈ BOOL ∧ expired = TRUE
  grd5 : isWithdrawn ∈ BOOL ∧ isWithdrawn = FALSE
  grd6 : consentRenewalState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd7 : isConsentExpired ⩤ {{p ↦ c} ↦ TRUE} ∈
         (PATIENTS ↔ CONSENTS) ⤖ BOOL
Then
  act1 : consentRenewalState({p ↦ c}) ≔ Void
  act2 : isConsentExpired({p ↦ c}) ≔ TRUE
End
```

Listing 14: The CreateConsentRenewalRequest event.

The NotifyPatient event (Listing 15) is used to notify the patient about extending the time period of consent. The guards are defined with five preconditions. First, the guard grd1 ensures that the user successfully got the session and the user role is within the domain *userRoles*. Second, the guard grd2 ensures that one of the user roles is a legal staff. Third, the guard grd3 ensures that *pc1* is not a subset of *markAsReceived*, *pc1* is a member of the domain *consentRenewalState*, and *consentRenewalState(pc1)* is equal to Void. Fourth, the guard grd4 ensures that the *acceptStatus* is a member of STATUSES but excludes Void. Fifth, the guard grd5 ensures that when updating the *acceptStatus* to the *consentRenewalState(pc1)*, the invariant inv7 must be satisfied. Whenever all guards are valid, the action act1 adds the *acceptStatus* to the *consentRenewalState(pc1)*, and act2 adds *pc1* to the *markAsReceived*.

```
NotifyPatient ≙
Any s,pc1,acceptStatus Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ⊈ markAsReceived ∧ pc1 ∈
         dom(consentRenewalState) ∧ consentRenewalState(pc1) = Void
  grd4 : acceptStatus ∈ STATUSES \ {Void}
  grd5 : consentRenewalState ⩤ {pc1 ↦ acceptStatus} ∈
```

```
        (PATIENTS ↔ CONSENTS) ⤖ STATUSES
Then
  act1 : consentRenewalState(pc1) ≔ acceptStatus
  act2 : markAsReceived ≔ markAsReceived ∪ pc1
End
```

Listing 15: The NotifyPatient event.

After receiving the patient's response, the ExtendConsentExpiration event (Listing 16) will be fired if the variable *consentRenewalState(pc1)* is Approved and *isConsentExpired(pc1)* is TRUE. The *isConsentExpired(pc1)* as a boolean will then be changed from TRUE to FALSE according to the action.

```
ExtendConsentExpiration ≙
Any s,pc1 Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ∈ dom(consentRenewalState) ∧
         consentRenewalState(pc1) = Approved
  grd4 : pc1 ⊆ markAsReceived ∧ pc1 ∈ dom(isConsentExpired) ∧
         isConsentExpired(pc1) = TRUE
  grd5 : isConsentExpired ⩤ {pc1 ↦ FALSE} ∈
         (PATIENTS ↔ CONSENTS) ⤖ BOOL
Then
  act1 : isConsentExpired(pc1) ≔ FALSE
End
```

Listing 16: The ExtendConsentExpiration event

Otherwise, the DeletePatientData event (Listing 17) will be fired to add the *pc1* to *markAsDeleted*. In both cases, the consent renewal request is determined by the legal staff.



```
DeletePatientData ≙
Any s, pc1 Where
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ∈ dom(consentRenewalState) ∧
         consentRenewalState(pc1) = Rejected
  grd4 : pc1 ⊆ markAsReceived ∧ pc1 ∈ dom(isConsentExpired) ∧
         isConsentExpired(pc1) = TRUE
  grd5 : markAsDeleted ∩ pc1 = ∅
Then
  act1 : markAsDeleted ≔ markAsDeleted ∪ pc1
End
```

Listing 17: The DeletePatientData event.

### 4.3. Model Evaluation in Event-B

The refined models are formalized and proved correct using the Rodin Platform. The Rodin Platform generates the POs that can be proved automatic or manual. Moreover, it guarantees that all events preserve invariants whenever state variables have changed. The proving results (Table 7) demonstrate that all models were proved automatically by Atelier B provers. Moreover, there were no invariant violations or deadlocks found. The Event-B models are presented in APPENDIX A.

Table 7: The summary of proof statistics by the Rodin platform for the proposed four consent management state machines based on Event-B models.

| Machine name | Number of proof obligations | Automatic (%) | Manual (%) |
|---|---|---|---|
| RPSM | 42 | 42 (100%) | 0 (0%) |
| WASM | 16 | 16 (100%) | 0 (0%) |
| PASM | 16 | 16 (100%) | 0 (0%) |
| CRSM | 22 | 22 (100%) | 0 (0%) |

### 4.4. Event-B Model Transformation to Class Diagram

The proposed models implemented by Event-B can be used as a guideline for software development on the aspects of consent management. According to the object-oriented approach, a class diagram is a static structural model which describes the system's classes, attributes, operations, and associations. It helps developers understand a system's overall structure. Here, we give an example of how to transform our Event-B models into a class diagram. First, identify the primary classes of the system which appear in static variables of Event-B (e.g., sets, constants, variables, and definitions). Second, identify the relation between self or other sets which appear in invariants, which indicate the association between classes. Third, identify events as operations in classes. Also, notice that a transition can be fired, and only if guard conditions are true, an event occurs. Each guard condition must be implied as a precondition of a method in a class. The globally declared static variables can be mapped to concrete classes (e.g., *AuthorizedUser*, *Role*, *Consent*, *DataSubject*, *DataSubjectConsent*), as shown in Figure 27. The set of PATIENTS represents data subjects under GDPR. We could define a *DataSubject* associated with *DataField*, and *DataValue* classes to hold patient personal data. Moreover, GDPR requires the systems to get consent from data subjects before processing data. So, the Consent class needs to be created with a set of properties (e.g., *consentDetail*, *dataRetention* (in months), *consentVersion*, *createdDate*). In the CheckAuthorizeConsent event of RPSM, the variable *consentExpired* is a flag indicating if the data's age exceeds the applicable data retention, defined inside the *ConsentPolicyAccess* class. We need to create a *DataSubjectConsent* class to hold the properties required for calculating the *consentExpired* flag. For example, suppose that we define properties as follows: 1) the *acceptedFlag* indicates a data subject's response to the consent extension, which can be either approved ("Y") or rejected

("N"), 2) the *createdDate* represents the data subject's last response date, 3) the *dataSubject* object indicates this data subject, and 4) the consent object indicates the consent that has been approved or rejected by the data subject.

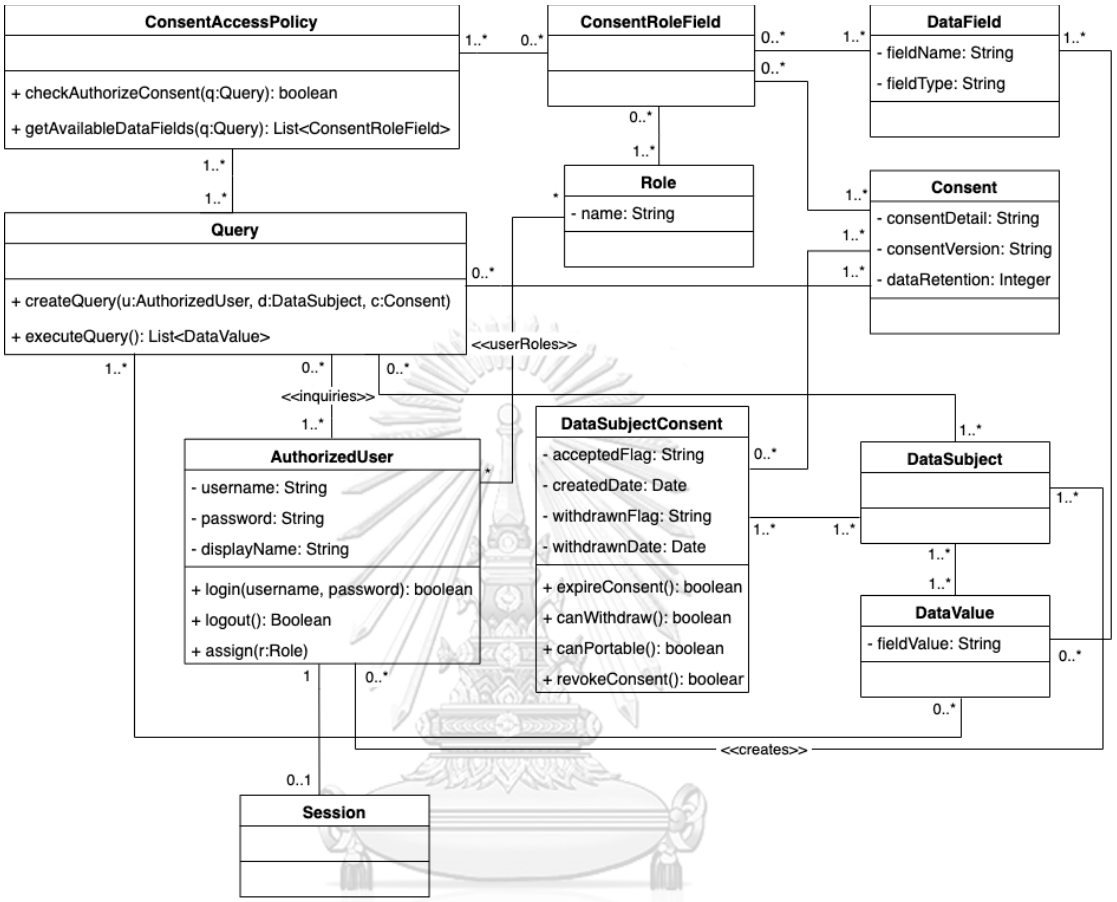



Figure 27: A class diagram transformed from the proposed consent-based models in Event-B.



To calculate the *consentExpired* flag, we need to retrieve the *DataSubjectConsent* object associated with a specific data subject and consent. After getting the object, check if the *acceptedFlag* = "Y" and *getSystemDate() > addMonths(createdDate, consentObject.dataRetention)*, then set the *expiredFlag* = "Y", otherwise set the *expiredFlag* = "N". Our proposed models based on Event-B method are designed to be simple and applicable, which could be easily mapped to the real codes. In the case of RUN-ONCO, a web-based application, we adopted the functionality from the *ConsentPolicyAccess* class (Figure 27) and enhanced it into *AuthenticationService* and *PatientService* classes (Figure 22) to make clean and reusable codes.

In addition, particular businesses or systems can also use these models. According to the class diagram in Figure 27, the *DataSubject* class represents an individual that can recognize a person's uniqueness (e.g., customers, patients, employees). Hence a system has to define a set of data fields of personal data on which can dynamically

be added into the *DataField* class (e.g., full name, social security number, birthdate). Since data fields have been defined, a stakeholder who is involved in a software system (e.g., an individual, team, organization) needs to add consent into the *Consent* class and establish a relationship between these data fields. To limit data access precisely, a stakeholder needs to assign suitable user roles based on consent data. When collecting personal data, a system needs to obtain the value of personal data in the *DataValue* class followed by predefined data fields according to a given consent. This thesis showed that our formal models support the commonly used features of consent management.

# CHAPTER V
# A FORMAL MODEL FOR BLOCKCHAIN-BASED CONSENT MANAGEMENT IN DATA SHARING

This chapter is a slightly modified version of a manuscript published in the *Journal of Logical and Algebraic Methods in Programming*, Volume 134, 2023, 100886, and has been reproduced here with the permission of the copyright holder.

Sharing data can lead to a potential loss of control over personal data, as data are across boundaries between software services. The use of blockchain technology enables to manage of data subjects' informed consent for data sharing to build trust, transparency, and traceability to share data across software services. Nevertheless, cooperation between data privacy and blockchain technology benefits protecting data against manipulation.

To develop CM for distributed systems in data sharing, we reviewed data-sharing issues (Table 8) from the view of system design to build a GDPR-aware system model on blockchain related to PbD [20, 36, 43-46].

In this thesis, we defined the data sharing state machine (DSSM) upon requirements in Table 8 that covered blockchain-enabled consent management in data sharing and created a mapping of GDPR articles relevant to DSSM in Table 9. This state machine aims to help developers address GDPR requirements in software engineering practices.

To define a set of states and transitions in DSSM, we determined the logic within consent management functionality comprises the following fundamental features: 1) the consent authorization feature is used to restrict access to share personal data based on the given consent (Articles 5 & 20 GDPR), 2) the consent withdrawal feature is used to revoke permission to share personal data (Articles 17 & 19 GDPR), and 3) the consent renewal feature is used to keep data sharing functionality available (Article 6(1a) GDPR). The consent authorization feature is essential in data-sharing processing activities to check whether consent is expired or withdrawn based on the data subject's consent. If consent is expired or removed, data transfer is not permitted. Otherwise, the system can proceed with data-sharing activities, i.e., transfer data to another service.

Table 8: Data sharing-related issues as requirements for blockchain-based consent management.

| Topic | Issue | Requirement |
|---|---|---|
| Rules of data sharing upon a particular purpose | The challenge of consent associated with sharing personal data is to manage consent and personal data effectively and transparently [20, 43-45]. | **RQ1:** The system shall determine the consent management functionality based on decentralized security, which enables an immutable audit log and transparent data-sharing over a network. |
| | The data subject can control and provide his/her consent over personal data being shared [20, 43, 45, 46]. | |
| Access restriction based on the purpose or consent | The data subject has the right to give his/her consent to transmit personal data among data controllers [20, 43]. | **RQ2:** The system shall allow data controllers to request and disclose individuals' data only if the data subject has provided his/her consent. |
| Limited number of records in data selection | The sharing of unnecessary personal data puts the confidentiality and privacy of individuals at risk [35, 47]. | **RQ3:** The system shall determine that one request-response interaction is only provided for one individual's data and it will be disclosed in accordance with predefined data fields in this given consent. |
| | The data controller is restricted to sharing only the minimum amount of personal data necessary [36, 37, 43]. | |
| Consent withdrawal | The data subject has the right to revoke consent to discontinue sharing personal data as he/she wishes [20, 48]. | **RQ4:** The system shall provide a mechanism for consent revocation in which the data subject can revoke consent at any time. |
| Consent renewal | The data controller may request the data subject an extension of the retention period for continuing to share his/her personal data [20, 48]. | **RQ5:** The system shall provide a mechanism for consent renewal in which the data subject can renew consent. |
| Auditability | The sharing of personal data among data controllers should be documented at each transmission step in immutable and transparent data storage [20, 43, 47]. | **RQ6:** The system shall provide a mechanism for audit logging to document a request-response interaction between participant data controllers on every disclosure. |
| | The data subject shall have access to audit log activities for tracking at each transmission step based on the consent that he/she provided [20, 43, 46]. | |
| Personal data masking | The risk of direct personal identification in data sharing may cause the recognition of individual persons [47, 49-51]. | **RQ7:** The system shall provide a mechanism for pseudonymized data to reduce the risk of identifying individual persons through data sharing. |
| Secure distributed data storage | The sharing of personal data among data controllers should not duplicate any individuals' data in secure distributed data storages, reducing IT costs and operational burdens. | **RQ8:** The system shall enable async callback to manage the request-response interaction with dynamic configuration endpoint callback URLs for eliminating the use of secure distributed data storage. |

Table 9: The proposed model and GDPR articles it covered (cont'd).

| | DSSM | | | Class Diagram | | | GDPR article |
|---|---|---|---|---|---|---|---|
| | **Event** | **Set/ Constant** | **Local/ State variable** | **Operation** | **Class** | **Attribute** | |
| **RQ1** | AddConsent | CONSENTS | consents | addConsent | Consent:struct, ConsentContract | consentCode, consentDetail, consentVersion, dataRetention, requesterId, requesterUrl | Article 4(1), Article 4(4), Article 4(7), Article 5(1b), Article 24, Article 28, Article 37 |
| | | CONSENTS, FIELDS | dataFields | addDataField | DataField:struct, DataFieldContract | consentCode, consentVersion, fieldName | |
| | AddDataSubjectConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | dataSubjectConsents | addDataSubjectConsent | DataSubjectConsent:struct, DataSubjectConsentContract | responderId, responderUrl, pseudonym, consentCode, consentVersion | |
| **RQ2** | | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | consentExpired:bool, dataSubjectConsents | isConsentValid | DataSubjectConsent:struct, DataSubjectConsentContract | responderId, pseudonym, consentCode, consentVersion, dataRetention, createTimestamp | Article 5(1a), Article 5(1c), Article 5(1d), Article 6(1a) |
| **RQ3** | SubmitRequest | REQUESTS, PARTICIPANTS, DATA_SUBJECTS, CONSENTS | dataAccessRequests | submitRequest | DataAccessRequest:struct, DataAccessRequestContract | requestExists, requestId, pseudonym, consentCode, consentVersion | Article 5(1e), Article 5(1f), Article 6(1a), Article 20 |
| | SubmitResponse | RESPONSES, REQUESTS | dataAccessResponses | submitResponse | DataAccessResponse:struct, DataAccessResponseContract | responseExists, responseId, requestId, pseudonym, consentCode, consentVersion | |
| **RQ4** | RevokeConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | dataSubjectConsents | revokeConsent | DataSubjectConsent:struct, DataSubjectConsentContract | pseudonym, responderId, consentCode, consentVersion, withdrawnTimestamp | Article 7(3), Article 17, Article 19 |

Table 9: The proposed model and GDPR articles it covered.

| | DSSM | | | Class Diagram | | | GDPR article |
|---|---|---|---|---|---|---|---|
| | **Event** | **Set/ Constant** | **Local/ State variable** | **Operation** | **Class** | **Attribute** | |
| **RQ5** | RenewConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | dataSubjectConsents | renewConsent | DataSubjectConsent:struct, DataSubjectConsentContract | pseudonym, responderId, consentCode. consentVersion, createTimestamp | Article 6(1a) |
| **RQ6** | | | | LogAddedConsent, LogInactivatedConsent | ConsentContract | | |
| | | | | LogAddedDataField | DataFieldContract | | |
| | | | | LogAddedDataSubjectConsent, LogFiredRequesterCallback, LogReturnedRequesterCallback, LogRevokedConsent, LogRenewedConsent | DataSubjectConsentContract | | |
| | | | | LogSubmittedRequest, LogFiredResponderCallback, LogReturnedResponderCallback | DataAccessRequestContract | | |
| | | | | LogSubmittedResponse, LogFiredDataTranferCallback, LogReturnedDataTransferCallback | DataAccessResponseContract | | |
| **RQ7** | | | | | DataSubjectConsent:struct | pseudonym | Article 4(5) |
| **RQ8** | CallbackRequester | balanceOf(this), PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | oraclizeFee:number, dataSubjectConsents, callbackRequesterStates | callbackRequester | DataSubjectConsent:struct, DataSubjectConsentContract | responderId, pseudonym, consentCode, consentVersion, resquesterUrl | |
| | CallbackResponder | balanceOf(this), REQUESTS, PARTICIPANTS, DATA_SUBJECTS, CONSENTS | oraclizeFee:number, dataAccessRequests, callbackResponderStates | callbackResponder | DataAccessRequest:struct, DataAccessRequestContract | requestId, pseudonym, consentCode, consentVersion, responderUrl | |
| | CallbackDataTransfer | balanceOf(this), RESPONSES, REQUESTS | oraclizeFee:number, dataAccessResponses, callbackDataTransferStates | callbackDataTransfer | DataAccessResponse:struct, DataAccessResponseContract | responseId, responderUrl, transferUrl | |

## 5.1. CM State Machine for Data Sharing in Distributed Systems

This chapter proposes a formal model for data sharing in distributed systems which embeds data protection into software development upon the GDPR. Based on Article 4(11) GDPR, for the consent to be valid, the data subject voluntarily agrees to enable either a data controller or a data processor to process his/her personal data for a specific purpose. We considered consent management essential for promoting privacy awareness in the system design [131, 132]. Furthermore, it indicates that the system cannot process or share personal data without the data subject's consent. In this chapter, we built a state machine for data sharing to depict the dynamic behavior of a requester sending requests to access personal data on the blockchain relevant to the relationships of a data subject's consent, a requester, a responder, and a smart contract's balance. We followed PbD concepts and GDPR guidelines presented in Table 9 and provided the example of request-response interaction through the data-sharing sequence diagram (Figure 28 and Figure 29) and the DSSM (Figure 30) that covers the main aspects of blockchain-based consent management in data sharing.

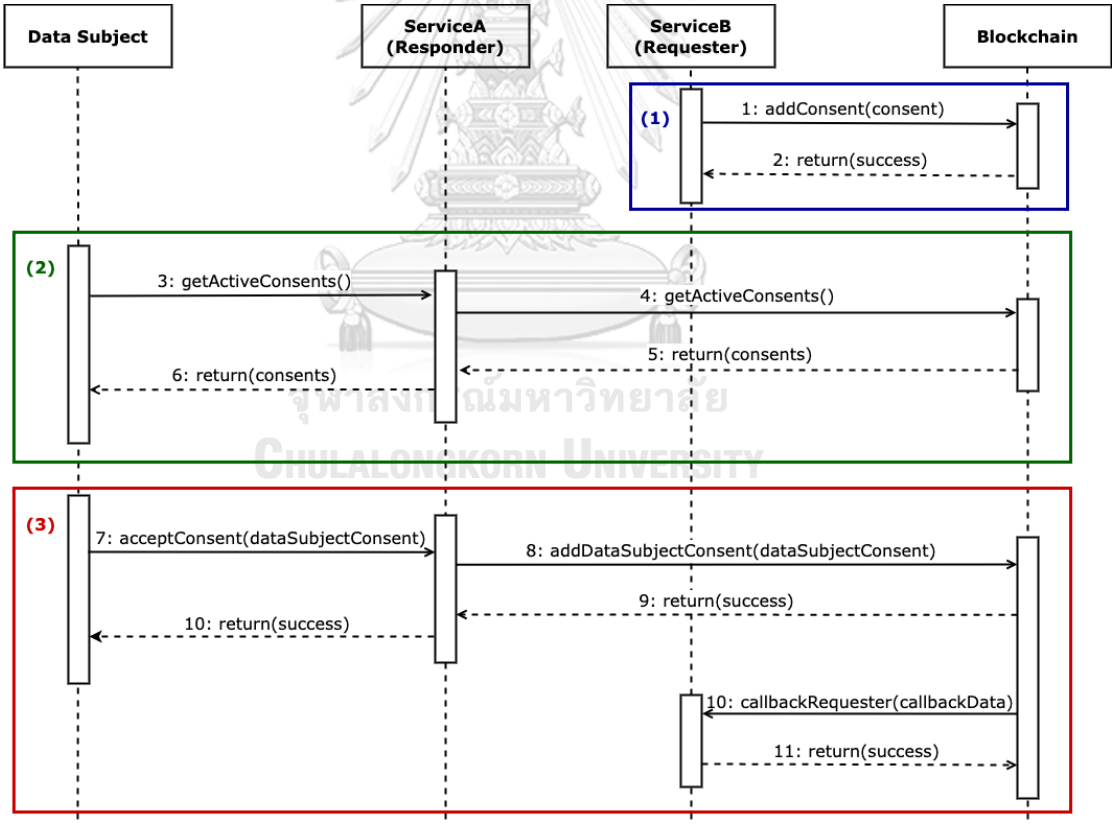



Figure 28: Data sharing sequence diagram illustrating the request-response interaction between ServiceA (responder) and Service B (requester).

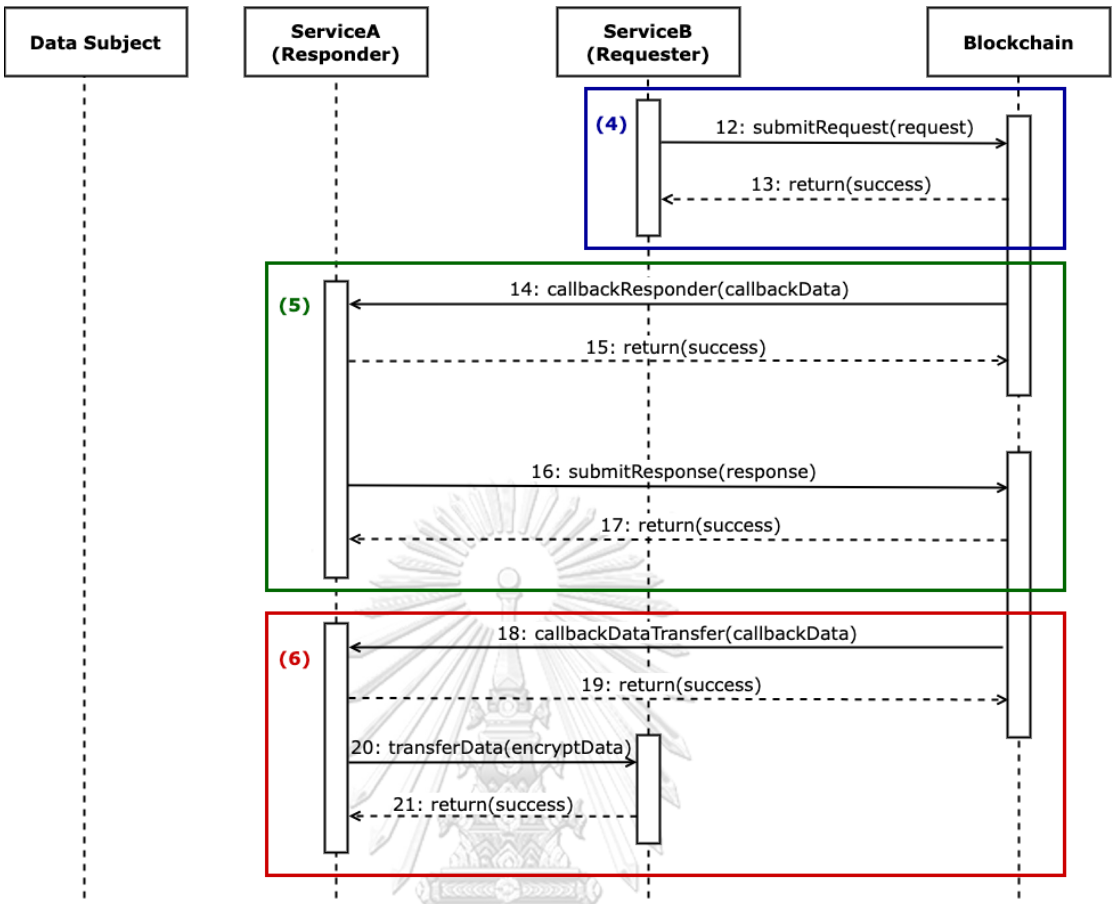



Figure 29: Data sharing sequence diagram continued from the previous diagram (Figure 28), which illustrates the request-response interaction between ServiceA and ServiceB.

We utilize the blockchain to obtain records of all request-response interactions without storing personal data. Moreover, the requester and responder communicate through blockchain, which is strictly forbidden to communicate directly with each other. The interactions between the requester and the responder begin with the requester requesting to access personal data through smart contracts (i.e., providing consent management) live on a blockchain. Then smart contracts automatically check if the data subject has authorized access to their personal data. If the request is approved, the blockchain makes a callback to trigger the responder. Finally, the responder sends the response back to the blockchain and transmits personal data to the requester through an off-chain channel (i.e., the channel allowing transactions to occur outside the blockchain).

Based on sequence diagrams, the requester (ServiceB) first adds its new consent into the blockchain (Figure 28(1)). Second, the data subject accesses the front-end of his/her data provider, a responder (ServiceA), and retrieves from the blockchain all available consents required by the requester (ServiceB) offering new products or services (Figure 28(2)). The data subject must accept before using its products or ser-

vices. Third, after the data subject agrees with a requester's consent, the responder (ServiceA) sends back the data subject's acceptance status into the blockchain (Figure 28(3)). Fourth, when the new data subject's consent has been stored on the blockchain, the blockchain makes a callback to trigger the requester (ServiceB), which can prepare a request for accessing personal data (Figure 29(4)). Fifth, when the request has been stored on the blockchain, the blockchain makes a callback to trigger the responder (ServiceA), which can respond to access the personal data within the retention period (Figure 29(5)). Sixth, when the response has been stored on the blockchain, the blockchain makes a callback to trigger the responder (ServiceA), which can transfer personal data directly to the requester (ServiceB) via an off-chain channel (Figure 29(6)). One request will get only one response in our model, as tracking all requests and responses on the blockchain is easier.

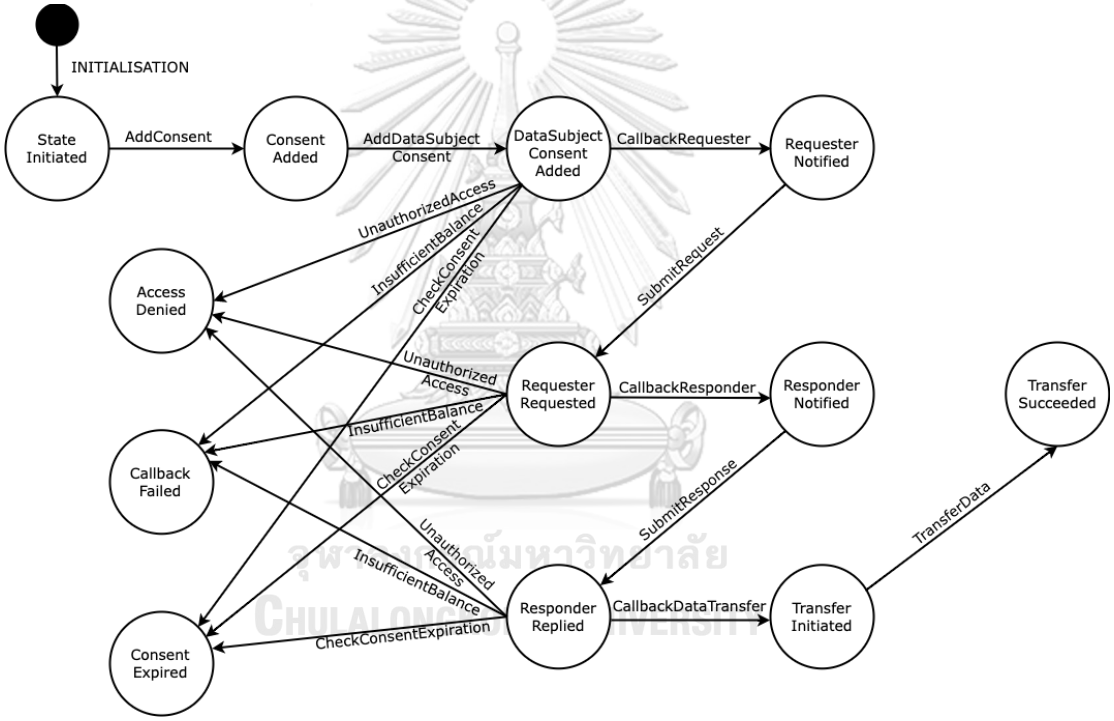



Figure 30: Data Sharing State Machine (DSSM) illustrating the transition states and events used to share personal data between a requester and a responder through blockchain.

## 5.2. Formal Development in Event-B

To build the data sharing model, first, we created the data sharing context (DSCX) to define carrier sets, and constants associated with blockchain as follows: 1) CONSENTS is a set of personal data sharing agreements (e.g., ConsentA, ConsentB) between the services sending and receiving data, 2) FIELDS is a set of data fields (e.g., Name, BirthDate, BirthDefects) that identifies individuals, 3) DATA_SUBJECTS is a set of data subjects (e.g., DataSubject1), 4) PARTICIPANTS is a

set of services (e.g., ServiceA, ServiceB) that require data sharing based on blockchain technology, 5) ADDRESSES is a set of contract addresses to interact with deployed smart contracts, 6) the constant *this* is a member of ADDRESSES, which refers to the contract address itself, 7) the constant *initialBalance* is a natural number representing the initial balance of contract address *this*, 8) REQUESTS is a set of data requests, the requesting services (requesters) create requests (e.g., Request1) to the responding services (responders) for accessing personal data, and 9) RESPONSES is a set of data responses, the responders check whether the requests have authorized access to personal data and return the responses (e.g., Response1) back to the requesters.

Second, we created DSSM and referred to DSCX; the state machine can directly access the defined global static variables. State machine naturally encapsulates states and behaviors related to variables, invariants, and transitions. In the Event-B model, variables represent the states of the system, and invariants, e.g., inv1, inv2, represent the preserved properties of the states. A transition represents the change from one state to another according to an event. Every event comprises guards as preconditions and actions for variable modification, labeled, e.g., grd1, grd2, and act1, act2, respectively. A transition will take place only if it satisfies all invariants and guards.

### 5.2.1. Data Sharing State Machine (DSSM)

The DSSM (Figure 30) was modeled and formally proved for blockchain-based data sharing. It depicts the dynamic behavior of a requester sending requests to access personal data on the blockchain that provides consent-based access control. If the request is authorized, the responder will send the response back to the blockchain and transmit personal data to the requester through an off-chain channel. For this state machine, we defined the preserved invariants as follows:

#### 5.2.1.1. Invariants in DSSM

**inv1:** consents ∈ ℙ(CONSENTS)
**inv2:** dataFields ∈ CONSENTS ⇸ ℙ1(FIELDS)
**inv3:** dataSubjectConsents ∈
PARTICIPANTS × DATA_SUBJECTS × CONSENTS ⇸ BOOL
**inv4:** addresses ⊆ ADDRESSES
**inv5:** balanceOf ∈ addresses ⟶ ℕ
**inv6:** callbackRequesterStates ∈
ℙ(PARTICIPANTS × DATA_SUBJECTS × CONSENTS)
**inv7:** dataAccessRequests ∈
REQUESTS ⇸ PARTICIPANTS × DATA_SUBJECTS × CONSENTS
**inv8:** callbackResponderStates ∈ ℙ(REQUESTS)
**inv9:** dataAccessResponses ∈ RESPONSES ⤖ REQUESTS

```
inv10: callbackDataTransferStates ∈ ℙ(RESPONSES)
inv11: encryptedData ∈ RESPONSES ⇸ ℙ(DATA_SUBJECTS × FIELDS)
inv12: dataTransferStates ∈ RESPONSES ⇸ BOOL
```

The variable *consents* contains a set of CONSENTS, which holds all consents offered by the requesters. According to PbD, the requester must demonstrate that data subjects agreed to process their personal data for a specific purpose on the defined data fields. Hence, we declared the variable *dataFields* as a set of ordered pairs (consent ↦ dataField) where consent ∈ CONSENTS and *dataField* ∈ ℙ1(FIELDS). The dataFields specifies that consent can have one or more data fields. The *dataSubjectConsents* defines (pdc ↦ active) as a set of ordered pairs, where pdc ∈ PARTICIPANTS × DATA_SUBJECTS × CONSENTS, and active ∈ BOOL (i.e., TRUE or FALSE). The variable *dataSubjectConsents* represents a record of the data subject's consent that allows a requester to process his/her personal data under the purpose of the consent. The *addresses* is a subset of the ADDRESSES set where each represents a unique smart contract address on the blockchain. We defined the *balanceOf*(address ↦ balance) where address ∈ addresses and balance is a natural number, keeping track of the contract address balance. The variable *callbackRequesterStates* contains a set of ordered triples (responder ↦ dataSubject ↦ consent) where responder ∈ PARTICIPANTS, dataSubject ∈ DATA_SUBJECTS, consent ∈ CONSENTS to track which requesters have been successfully invoked after the data subjects have given their consents for their data processing. The variable *dataAccessRequests* is a set of ordered pairs (request ↦ dataSubjectConsent) where request ∈ REQUESTS, and dataSubjectConsent ∈ dom(dataSubjectConsents), represents a record of data request of a requester to the blockchain after receiving a callback of data subject's permission. The *callbackResponderStates* contains a set of REQUESTS to track which responders have been successfully invoked after the requesters have initiated their requests. Hence, we declared the *dataAccessResponses* that holds the one-to-one relationship between RESPONSES and REQUESTS. This mapping allows transferring data between the responder and requester. The variable *callbackDataTransferStates* contains a set of RESPONSES to track which responders have been successfully invoked for starting an off-chain data transfer after accepting the requests. We stored the encrypted data in variable *encryptedData*, a set of ordered pairs (response ↦ personalData) where response ∈ RESPONSES, and personalData ∈ DATA_SUBJECTS × FIELDS. Furthermore, we defined a variable *dataTransferStates* to hold the status of successful data transfer as a set of ordered pairs (response ↦ success) where response ∈ RESPONSES, and success ∈ BOOL (i.e., TRUE or FALSE).

#### 5.2.1.2. Events in DSSM

The DSSM state machine is executed starting from the INITIALISATION event, then all variables of DSSM are initialized. Listing 18 shows the formal model of how a new requester's consent is added to the blockchain. The guards are defined with three preconditions. First, the guard grd1 ensures that the consent does not exist in the variable consents. Second, the guard grd2 ensures that any *dataField* is a member of ℙ1(FIELDS). Third, the guard grd3 ensures that adding an ordered pair (consent ↦ dataField) into variable *dataFields* must satisfy the invariant inv2. Whenever all guards are valid, action act1 adds the consent to the consents, and action act2 adds an ordered pair (consent ↦ dataField) to the *dataFields*.

```
AddConsent ≙
Any consent, dataField Where
  grd1 : consent ∈ dom(consents) ∧ consent ∈ consents
  grd2 : dataField ∈ ℙ1(FIELDS)
  grd3 : dataField ⩤ {consent ↦ dataField} ∈ CONSENTS ⇸ ℙ1(FIELDS)
Then
  act1 : consents ≔ consents ∪ {consent}
  act2 : dataFields(consent) ≔ dataField
End
```

Listing 18. The AddConsent event.

Listing 19 shows how to formally model the addition of a new data subject's consent. The guards are defined with five preconditions. First, the guard grd1 ensures that the responder is a member of PARTICIPANTS. Second, the guard grd2 ensures that the *dataSubject* is a member of DATA_SUBJECTS. Third, the guard grd3 ensures that the consent is a member of the variable consents and within the domain *dataFields*. Fourth, the guard grd4 ensures that a new ordered triple (responder ↦ dataSubject ↦ consent) does not exist in the domain *dataSubjectConsents*, which means no active data subject's consent is already granted for the requester on the blockchain. Fifth, the guard grd5 ensures that when adding TRUE to the *dataSubjectConsents(responder ↦ dataSubject ↦ consent)*, the invariant inv3 must be satisfied. Finally, whenever all of the guards are valid, the action act1 adds TRUE to the *dataSubjectConsents(responder ↦ dataSubject ↦ consent)*.

```
AddDataSubjectConsent ≙
Any responder, dataSubject, consent Where
  grd1 : responder ∈ PARTICIPANTS
```

```
  grd2 : dataSubject ∈ DATA_SUBJECTS
  grd3 : consent ∈ consents ∧ consent ∈ dom(dataFields)
  grd4 : responder ↦ dataSubject ↦ consent ∉ dom(dataSubjectConsents)
  grd5 : dataSubjectConsents ⩤
         {responder ↦ dataSubject ↦ consent ↦ TRUE} ∈
         (PARTICIPANTS × DATA_SUBJECTS × CONSENTS) ⇸ BOOL
Then
  act1 : dataSubjectConsents(responder ↦ dataSubject ↦ consent) ≔ TRUE
End
```

Listing 19. The AddDataSubjectConsent event.

Listing 20 shows how we formally model the handling of the request-response mechanism on the blockchain. After adding a new data subject's consent, the blockchain creates a callback to the requester (Listing 20) via an outside API call. When the requester receives a callback, it will prepare a request to access personal data. The guards are defined with three preconditions. First, the guard grd1 ensures that the constant this is a member of the domain *balanceOf*, the *oraclizeFee* is the charge for sending a payload to an API call outside the blockchain, which is a member of a set of natural numbers, and the *oraclizeFee* must be less than or equal to *balanceOf(this)*. Second, the guard grd2 ensures that the decreased *balanceOf(this)* with the *oraclizeFee* must satisfy the invariant inv5. Third, the guard grd3 ensures that the *dataSubjectConsent* is a member of the domain *dataSubjectConsents*, *dataSubjectConsent* does not exist in the callbackRequesterStates, and the active status of the *dataSubjectConsents(dataSubjectConsent)* is TRUE. Whenever all guards are valid, action act1 charges oraclizeFee from the *balanceOf(this)*, and action act2 adds the *dataSubjectConsent* to the *callbackRequesterStates*.



```
CallbackRequester ≙
Any oraclizeFee, dataSubjectConsent Where
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee ≤ balanceOf(this)
  grd2 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
         addresses ⟶ ℕ
  grd3 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsent ∉ callbackRequesterStates ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
Then
  act1 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
         oraclizeFee}
  act2 : callbackRequesterStates ≔ callbackRequesterStates ∪
         {dataSubjectConsent}
End
```

Listing 20: The CallbackRequester event.

The SubmitRequest event (Listing 21) allows a requester to create a request for accessing personal data. In this event, we defined a set of constraints to restrict the request access: 1) the consent has not expired, 2) the consent has not been withdrawn, and 3) the request ID has not been submitted. These constraints were then described as five guards of the event. First, the guard grd1 ensures that the *consentExpired* is a member of the boolean and *consentExpired* is FALSE. Second, the guard grd2 ensures that the *dataSubjectConsent* is a member of the *dataSubjectConsents*, and the range of *dataSubjectConsents(dataSubjectConsent)* is TRUE. Third, the guard grd3 ensures that the *dataSubjectConsent* is a member of the variable *callbackRequesterStates*. Fourth, the guard grd4 ensures that the request is a member of REQUESTS and the request does not exist in the domain *dataAccessRequests*. Fifth, the guard grd5 ensures that adding an ordered pair (request ↦ dataSubjectConsent) into variable *dataAccessRequests* must satisfy the invariant inv7. Whenever all guards are valid, action act1 adds an ordered pair (request ↦ dataSubjectConsent) to the *dataAccessRequests*.

```
SubmitRequest ≙
Any consentExpired, dataSubjectConsent, request Where
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : dataSubjectConsent ∈ callbackRequesterStates
  grd4 : request ∈ REQUESTS ∧ request ∉ dom(dataAccessRequests)
  grd5 : dataAccessRequests ⩤ {request ↦ dataSubjectConsent} ∈
         REQUESTS ⇸ PARTICIPANTS × DATA_SUBJECTS × CONSENTS
Then
  act1 : dataAccessRequests(request) ≔ dataSubjectConsent
End
```

Listing 21: The SubmitRequest event.

The CallbackResponder event (Listing 22) handles a callback from the blockchain to the responder. When the responder receives a callback, it will respond to a request to access the personal data within the retention period. The guards are defined with four preconditions. The first two guards are the same as in the CallbackRequester event. Additionally, we declared the guard grd3 to ensure that the request is a member of the domain *dataAccessRequests*, and the request does not exist in the *callbackResponderStates*. Finally, through the guard grd4, we specified that the *dataAccessRequests(request)* as a *dataSubjectConsent* is a member of the domain *dataSubjectConsents*, and the range of the *dataSubjectConsents(dataAccessRequests(request))* as a boolean is TRUE. Whenever all

guards are valid, action act1 charges *oraclizeFee* from the *balanceOf(this)*, and action act2 adds the request to the *callbackResponderStates*.

```
CallbackResponder ≜
Any oraclizeFee, dataSubjectConsent Where
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee ≤ balanceOf(this)
  grd2 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
         addresses ⟶ ℕ
  grd3 : request ∈ dom(dataAccessRequests) ∧  request ∉
         callbackResponderStates
  grd4 : dataAccessRequests(request) ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataAccessRequests(request)) = TRUE
Then
  act1 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
         oraclizeFee}
  act2 : callbackResponderStates ≔ callbackResponderStates ∪ {request}
End
```

Listing 22: The CallbackResponder event.

The SubmitResponse event (Listing 23) is used to handle the response of a responder to a requester. Before returning the response to the requester, the event must check the following constraints: 1) the consent has not expired, 2) the consent has not been withdrawn, and 3) the response ID has not been submitted. Based on these constraints, guards are defined with five preconditions. The first two guards are the same as in the SubmitRequest event. Additionally, we declared guards grd3 to ensure that the request is a member of the variable *callbackResponderStates* and grd4 to ensure that the response is a member of RESPONSES and the response does not exist in the domain *dataAccessResponses*. Finally, the last guard grd5 ensures that adding an ordered pair (response ↦ request) into variable *dataAccessResponses* must satisfy the invariant inv9. Whenever all guards are valid, action act1 adds an ordered pair (response ↦ request) to the *dataAccessResponses*.

```
SubmitResponse ≜
Any consentExpired, request, response Where
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : request ∈ callbackResponderStates
  grd4 : response ∈ RESPONSES ∧ response ∉ dom(dataAccessResponses)
  grd5 : dataAccessResponses ⩤ {request ↦ response} ∈ RESPONSES ⤖
         REQUESTS
Then
  act1 : dataAccessResponses(response) ≔ request
```

```
End
```

Listing 23: The SubmitResponse event.

The CallbackDataTransfer event (Listing 24) is used to handle a callback from the blockchain to trigger the responder for data transfer. When the responder receives a callback, it will transfer personal data to the requester directly. The guards are defined with four preconditions. The first two guards are the same as in the CallbackRequester event. Additionally, we declared the guard grd3 to ensure that the response is a member of the domain *dataAccessResponses*, and the response does not exist in the *callbackDataTransferStates*. By means of the guard grd4, we specified that the *dataAccessResponses(response)* as a request is a member of the domain *dataAccessRequests*, *dataAccessRequests(dataAccessResponses( response))* as a *dataSubjectConsent* is member of the domain *dataSubjectConsents*, and the range of the *dataSubjectConsents( dataAccessRequests(dataAccessResponses(response)))* as a boolean is TRUE. Whenever all guards are valid, action act1 charges *oraclizeFee* from the *balanceOf(this)*, and action act2 adds the response to the *callbackDataTransferStates*.

```
CallbackDataTransfer ≙
Any oraclizeFee, request Where
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee ≤ balanceOf(this)
  grd2 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
         addresses ⟶ ℕ
  grd3 : response ∈ dom(dataAccessResponses) ∧ response ∉
         callbackDataTransferStates
  grd4 : dataAccessResponses(response) ∈ dom(dataAccessRequests) ∧
         dataAccessRequests(dataAccessResponses(response)) ∈
         dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataAccessRequests(dataAccessResponses(
         response))) = TRUE
Then
  act1 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
         oraclizeFee}
  act2 : callbackDataTransferStates ≔ callbackDataTransferStates ∪
         {response}
End
```

Listing 24: The CallbackDataTransfer event.

The TransferData event (Listing 25) transmits personal data from the responder to the requester via an off-chain channel. Before the personal data is transmitted, all constraints must be satisfied. The guards are defined with five preconditions. First, the guard grd1 ensures that the response is a member of the *callbackDataTransferStates* and the domain *dataAccessRe-*

*sponses*, and the response does not exist in the domain *dataTransferStates*. Second, the guard grd2 ensures that consent is a member of the domain *dataFields*. Third, the guard grd3 ensures that the data subject's consent is active and exists in the variables *dataAccessRequests*. Fourth, the guard grd4 ensures that adding an ordered pair (response ↦ {dataSubject} × dataFields(consent)) into variable *encryptedData* must satisfy the invariant inv11. Fifth, the guard grd5 ensures that adding an ordered pair (response ↦ TRUE) into variable *dataTransferStates* must satisfy the invariant inv12. Whenever all guards are valid, the action act1 adds an ordered pair (response ↦ {dataSubject} × dataFields(consent)) to the *encryptedData*, and action act2 adds an ordered pair (response ↦ TRUE) to the *dataTransferStates*.

```
TransferData ≙
Any responder, dataSubject, consent, response Where
  grd1 : response ∈ callbackDataTransferStates ∧ response ∈
         dom(dataAccessResponses) ∧ response ∉
         dom(dataTransferStates)
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
          dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : ∃x·x ∈ dataAccessRequests[{dataAccessResponses(response)}] ∧
         x = responder ↦ dataSubject ↦ consent ∧ responder ↦
         dataSubject ↦ consent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(x) = TRUE
  grd4 : encryptedData ⩤ {response ↦ {dataSubject} ×
         dataFields(consent)} ∈ RESPONSES ⇸ ℙ(DATA_SUBJECTS ×
         FIELDS)
  grd5 : dataTransferStates ⩤ {response ↦ TRUE} ∈ RESPONSES ⇸ BOOL
Then
  act1 : encryptedData(response) ≔ {dataSubject} × dataFields(consent)
  act2 : dataTransferStates(response) ≔ TRUE
End
```

Listing 25: The TransferData event.

The RevokeConsent event (Listing 26) is fired when a data subject requests to withdraw his/her consent. The guards are defined with two preconditions. First, the guard grd1 ensures that *dataSubjectConsent* is a member of the domain *dataSubjectConsents* and the active status of the *dataSubjectConsents(dataSubjectConsent)* is TRUE. The second guard grd2 ensures that when updating FALSE to the *dataSubjectConsents(dataSubjectConsent)*, the invariant inv3 must be satisfied. Whenever all guards are valid, action act1 assigns FALSE to the *dataSubjectConsents(dataSubjectConsent)*.

```
RevokeConsent ≙
Any dataSubjectConsent Where
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
```

```
  grd2 : dataSubjectConsents ⩤ {dataSubjectConsent ↦ FALSE} ∈
         (PARTICIPANTS × DATA_SUBJECTS × CONSENTS) ⇸ BOOL
Then
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ FALSE
End
```

Listing 26: The RevokeConsent event.

The RenewConsent event (Listing 27) is fired when a data subject requests to renew his/her consent. The guards are defined with two preconditions. First, the guard grd1 ensures that *dataSubjectConsent* is a member of the domain *dataSubjectConsents* and the active status of the *dataSubjectConsents(dataSubjectConsent)* is FALSE. The guard grd2 ensures that when updating TRUE to the *dataSubjectConsents( dataSubjectConsent),* the invariant inv3 must be satisfied. Whenever all guards are valid, the action act1 assigns TRUE to the *dataSubjectConsents( dataSubjectConsent)*.

```
RenewConsent ≙
Any dataSubjectConsent Where
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = FALSE
  grd2 : dataSubjectConsents ⩤ {dataSubjectConsent ↦ TRUE} ∈
         (PARTICIPANTS × DATA_SUBJECTS × CONSENTS) ⇸ BOOL
Then
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ TRUE
End
```

Listing 27. The RenewConsent event.

The InsufficientBalance event (Listing 28) handles the insufficient balance within a smart contract. An insufficient balance occurs when a smart contract's balance is too low to cover fees. The guards are defined with three preconditions. First, the guard grd1 ensures that the constant *this* is a member of the domain *balanceOf*, the *oraclizeFee* is a member of the set of natural numbers, and the *oraclizeFee* must be greater than *balanceOf(this)*. Second, the guard grd2 ensures that *dataSubjectConsent* is a member of the domain *dataSubjectConsents* and the active status of the *dataSubjectConsents(dataSubjectConsent)* is TRUE. Third, the guard grd3 ensures that insufficient balance occurs in callback events. Whenever all of the guards are valid, the process ends.

```
InsufficientBalance ≙
Any oraclizeFee, dataSubjectConsent, request, response Where
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee > balanceOf(this)
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
```

```
        dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : (dataSubjectConsent ∉ callbackRequesterStates) ∨
        (request ↦ dataSubjectConsent ∈ dataAccessRequests ∧
        request ∉ callbackResponderStates) ∨
        (response ↦ request ∈ dataAccessResponses ∧
        response ∉ callbackDataTransferStates)
Then
  skip
End
```

Listing 28. The InsufficientBalance event.

The CheckConsentExpiration event (Listing 29) is used to handle when data subjects' consent is expired. The guards are defined with three preconditions. First, the guard grd1 ensures that the *consentExpired* is a member of the boolean and *consentExpired* is TRUE. Second, the guard grd2 ensures that *dataSubjectConsent* is a member of the domain *dataSubjectConsents* and the active status of the dataSubjectConsents( dataSubjectConsent) is TRUE. Third, the guard grd3 ensures that when updating FALSE to the dataSubjectConsents(dataSubjectConsent), the invariant inv3 must be satisfied. Whenever all guards are valid, the action act1 assigns FALSE to the dataSubjectConsents(dataSubjectConsent).

```
CheckConsentExpiration ≙
Any consentExpired, dataSubjectConsent Where
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = TRUE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : dataSubjectConsents ⩤ {dataSubjectConsent ↦ FALSE} ∈
         PARTICIPANTS × DATA_SUBJECTS × CONSENTS ⇸ BOOL
Then
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ FALSE
End
```

Listing 29. The CheckConsentExpiration event.

The UnauthorizedAccess event (Listing 30) is used to handle when there is a request to access the data of a data subject, but the data subject's consent has been revoked or expired. The guard grd1 ensures that *dataSubjectConsent* is a member of the domain *dataSubjectConsents* and the active status of the *dataSubjectConsents(dataSubjectConsent)* is FALSE. Whenever the guard is valid, the process ends.

```
UnauthorizedAccess ≙
Any dataSubjectConsent Where
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = FALSE
Then
```

```
    skip
End
```

Listing 30. The UnauthorizedAccess event.

### 5.3. Model Evaluation in Event-B

The DSSM was formalized with Event-B, and its correctness was verified using the Rodin Platform. The Rodin Platform produces and discharges a set of POs automatically or manually to ensure that all events preserve all invariants. The resulting model (Table 10) demonstrates that the DSSM was proved automatically by Atelier B provers. As a result, there are no invariant violations or deadlocks found. The Event-B model are presented in APPENDIX B.

Table 10: The summary of proof statistics by the Rodin platform for the proposed state machine based on the Event-B model.

| Machine name | Number of proof obligations | Automatic (%) | Manual (%) |
|---|---|---|---|
| DSSM | 42 | 42 (100%) | 0 (0%) |

### 5.4. Event-B Model Transformation to Class Diagram

The proposed model constructed by Event-B assists developers as a guideline in applying consent management functionality among distributed services based on blockchain technology. In an object-oriented approach, a class diagram depicts a static view of a system, which is described by modeling its classes, attributes, operations, and associations. Moreover, a class diagram makes it easier for developers to understand how to implement smart contracts to support consent management. Here is an example of transforming our Event-B model into a class diagram. First, identify a system's classes that appear in static variables of Event-B (e.g., carrier sets, constants, and variables). Second, identify a system's class associations among itself or other sets that appear in invariants. Third, identify a system's operations in classes. Besides, each of the transitions has guard conditions, and it can be fired when the guard conditions are evaluated to be true, then an event occurs. Each guard condition represents a precondition based on state variables inside a method within classes. Figure 31 and Figure 32 show the class diagram designed based on Ethereum smart contracts using Solidity. We mapped the static variables in Event-B to concrete classes, which are divided into two groups: 1) classes used in consent management functionality, e.g., *Consent*, *ConsentContract*, *DataSubjectConsent*, *DataSubjectConsentContract*, as shown in Figure 31, and 2) classes used in request-response interactions between services, e.g., *DataAccessRequest*, *DataAccessRequestContract*, *DataAccessResponse*, *DataAccessResponseContract*, as shown in Figure 32. The set of DATA_SUBJECTS and CONSENTS represents data subjects and consents under GDPR, respectively. According to GDPR, the system requires gaining data subjects' consent before processing data. Therefore, we first defined structs (i.e., user-defined data types that ob-

tain related data items, probably of different data types); for example, Consent is used to hold a set of properties (e.g., consentCode, consentVersion, consentDetail, dataRetention (in days), createTimestamp, requesterUrl). Within our proposed model, the system should inform data subjects which piece of personal data is being used. So, we created DataField to hold predefined data fields upon the requesters' consent and used it to specify the personal data to be transferred. Moreover, DataSubjectConsent must be created to keep the relationship between data subjects and requesters' consent.

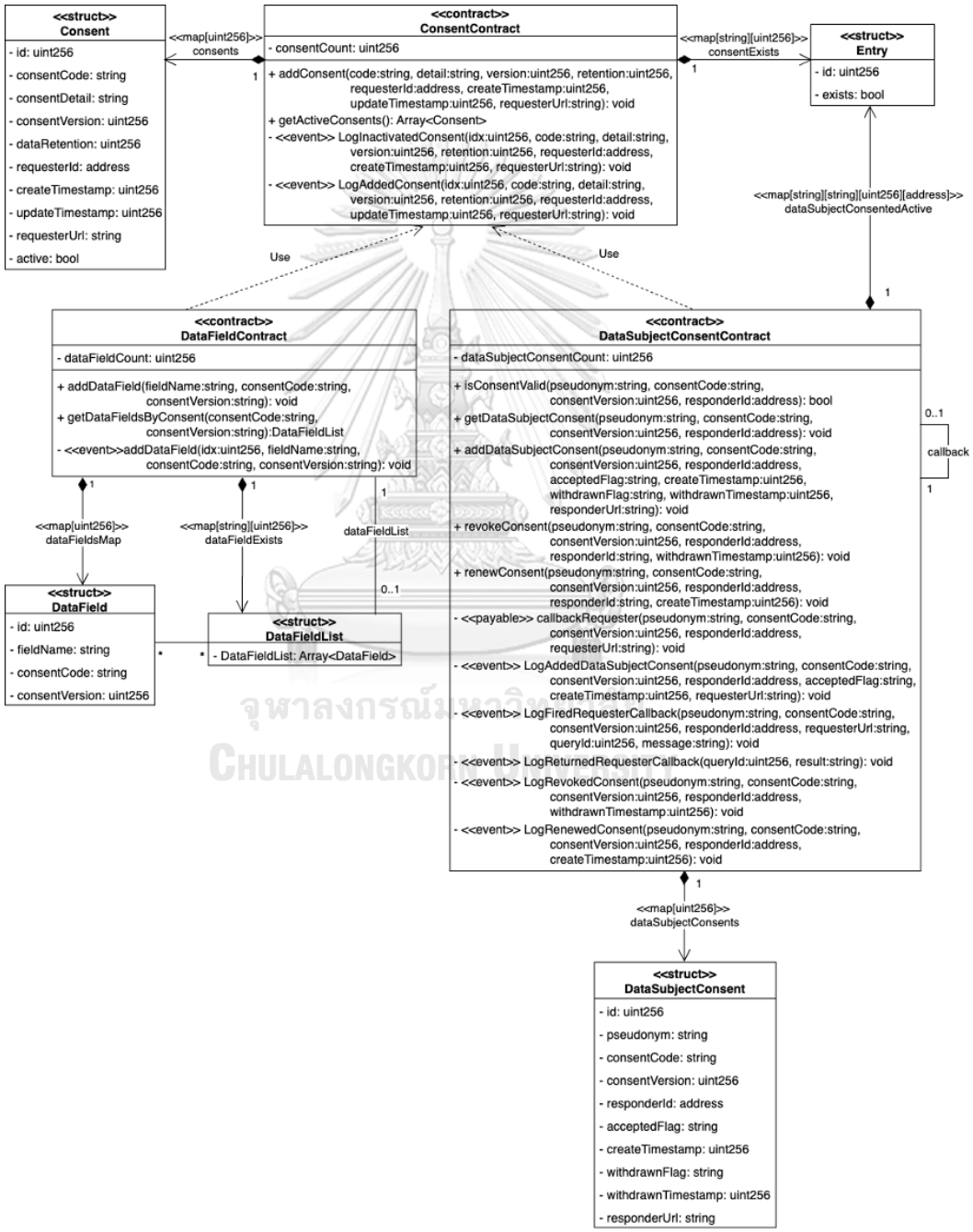



Figure 31: Class diagram resulted from mapping the proposed model in Event-B to code for supporting consent management in the context of data sharing.

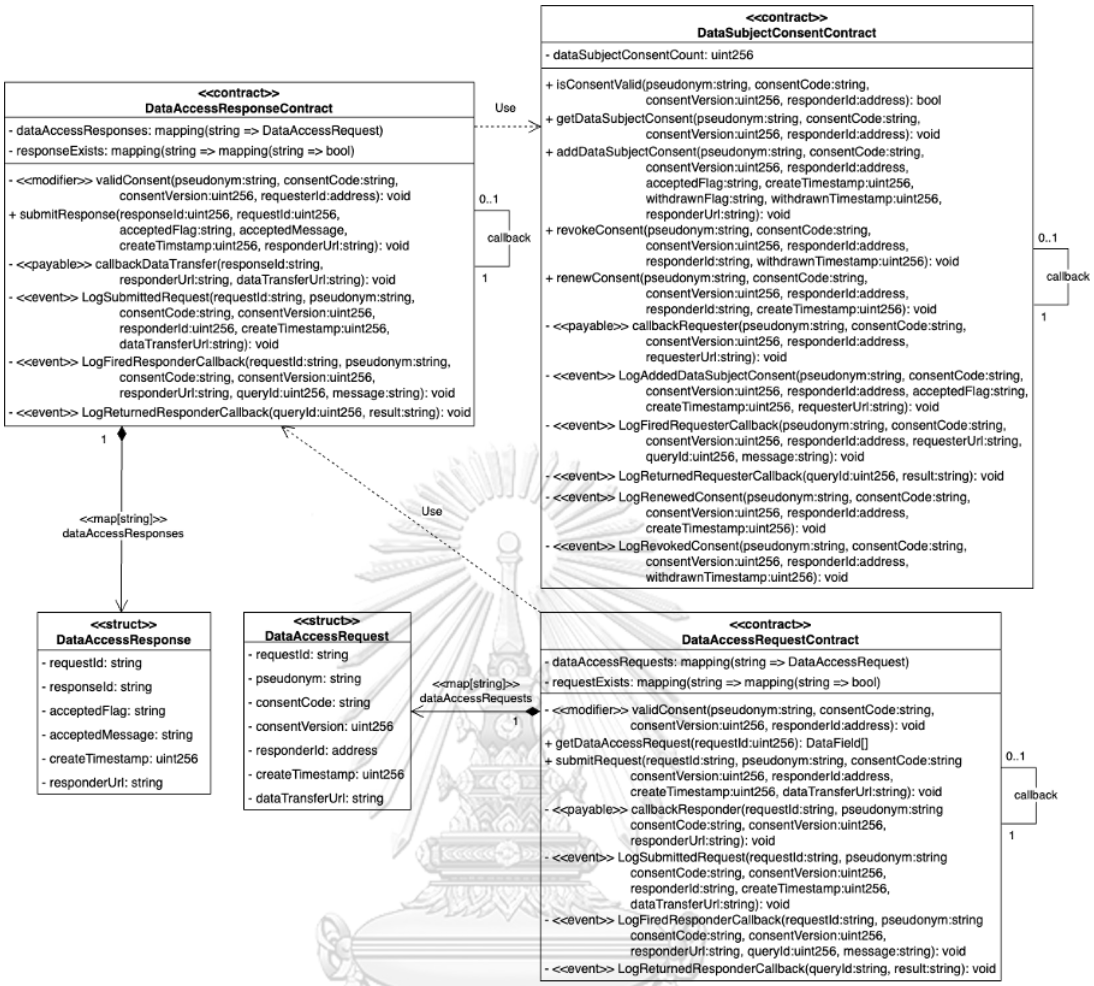



Figure 32: Class diagram continued from the previous diagram (Figure 31) demonstrating how to transform the proposed model in Event-B for supporting request-response interactions.



Second, we defined contracts (i.e., classes that obtain state variables and methods); for example, *ConsentContract* is to provide the addConsent() method, which is mapped to the AddConsent event. In the AddDataSubjectConsent event, we created *DataSubjectConsentContract* to obtain the addDataSubjectConsent() method. In addition, we added isConsentValid() as a common method to check whether consent is expired or withdrawn. As for the request-response interactions, we created *DataAccessRequestContract* and *DataAccessResponseContract* to obtain *DataAccessRequest* and *DataAccessResponse* structs, respectively.

For the SubmitRequest and SubmitResponse events, the model is required to check whether consent is valid. Hence, we then added the isConsentValid() method into these two contracts by calling *DataSubjectConsentContract*.isConsentValid(). According to the class diagram in Figure 31 and Figure 32, our proposed model was designed for generic usage and can be applied to any business model.

To allow the developers to quickly adopt the model, we developed SmartData-Trust that implemented smart contracts based on these class diagrams and exposed a REST API to interact with the blockchain. The requester and responder services only need to focus on implementing a REST API for consuming SmartDataTrust API and providing the callback URLs made by the blockchain.

### 5.5. SmartDataTrust Implementation

The SmartDataTrust API is a middleware that interacts with smart contracts live on the Ethereum blockchain by exposing REST services to the outside world (Figure 33). Implementing this API aims to provide a set of consent functionality for requester and responder services, which minimizes the effort of incorporating GDPR-compliant consent management in their interacting services. Moreover, it supports scalability by separating configuration from code in the YAML format (i.e., config.yaml), which is easily configured to deploy as Docker containers [134] with Kubernetes [135]. The API was designed based on a three-layer architecture [136] partitioned into REST controllers (i.e., consent_controller.py, data_subject_controller.py, data_access_request_controller.py, data_access_response_controller.py), application services (i.e., consent_service.py, data_field_service.py, data_subject_service.py, data_access_request_service.py, data_access_response_service.py), and the blockchain connector (i.e., blockchain_connector.py). The REST controllers handle incoming HTTP requests from requester and responder services and pass them through the application services. As for application services, they encapsulate data validation and conversion. Finally, the blockchain connector uses web3 frameworks [137] (e.g., Web3.py, Ethers.js, Infura API) for connecting smart contracts on the Ethereum blockchain through their contacts' addresses and contracts' schema files, which are configured in config.yaml.



In smart contract development, we first plug Truffle Suite [138] into SmartDataTrust API for building and deploying smart contracts on the Ethereum blockchain. Second, we implemented smart contracts with Solidity followed by the class diagram, as shown in Figure 31 and Figure 32. Third, we deployed smart contracts using Truffle's command (i.e., truffle migrate). After successful deployment, Truffle Suits generates the contracts' address and contracts' schema in JSON format files. Fourth, we configured the contracts' address and schema path into config.yaml. Finally, we start the Python REST API.

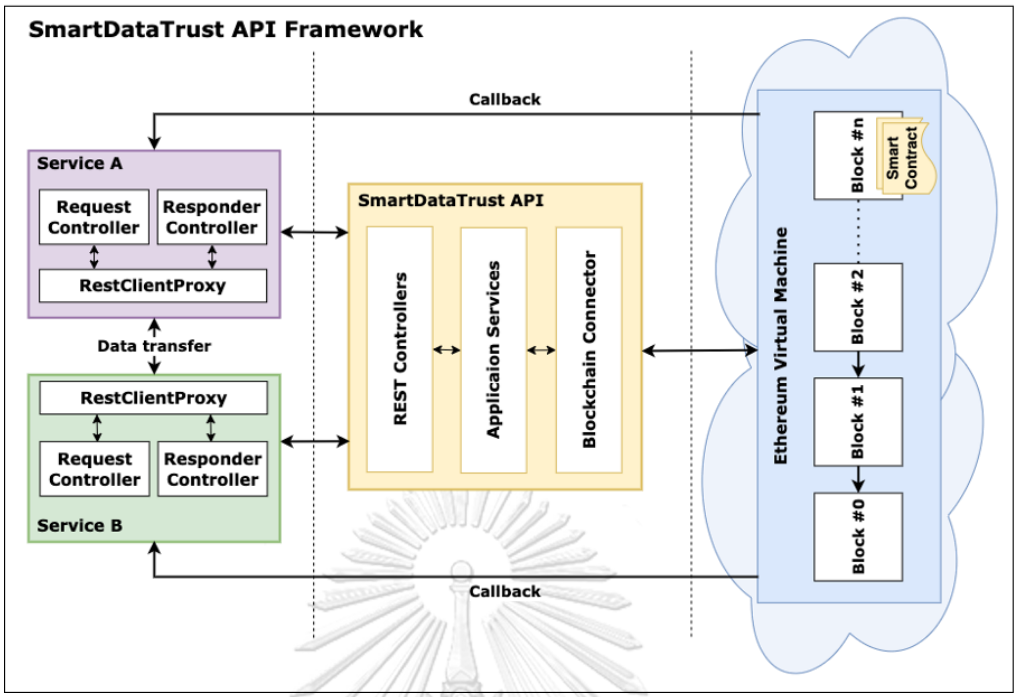



Figure 33: Overview of SmartDataTrust API framework.

Unfortunately, a smart contract is an immutable program. Once it is deployed on the blockchain, it preserves a new address. However, the multiple times of deployments of the smart contract lead to difficulty managing addresses and increasing execution time. We then designed reusable smart contracts to keep only states of data subjects' consent and request-response interactions between services. To create a callback URL outside the blockchain, we use blockchain oracles [79], e.g., Provable, Chainlink, and Astraea. In particular, we chose Provable for integrating into smart contracts because it is easy to implement and support dynamic data retrieval from trusted sources in large-scale applications. As for any service, it can be either a requester or a responder. We then created RequesterController and ResponderController classes following the available services in the SmartDataTrust API, and to handle API calls and HTTPS GET/POST requests among blockchain; we created RestClientProxy class.

To enhance an existing system integrated with the SmartDataTrust API, we demonstrate via a software platform for cancer precision medicine called RUN-ONCO [133]. RUN-ONCO allows users (i.e., oncologists, nurses, and researchers) to manage and create their own data analyzes to examine clinical, biospecimen, and genetic data, which assists oncologists in making specific treatment plans for individual cancer patients based on their genetics. To engage in research on cancer precision medicine, we need more patient data to help discover how to improve patient outcomes, such as genetic data and drug response. Therefore, we need to enhance RUN-ONCO to enable data sharing to exchange health data across organizations and be-

tween services. We then divided services into two types: 1) the service which manages its own patients' data, e.g., health information systems, and 2) the service which does not contain any patients' data, e.g., third-party API. RUN-ONCO and other services only focused on implementing a REST API for consuming the SmartDataTrust API to manage consent requests/responses on the blockchain and handling the requester and responder callbacks made by the blockchain. To enhance RUN-ONCO support consent management in data sharing (Figure 34A and Figure 34B), by following DSSM, we first need to alter the ConsentService class by adding the addConsent() method. Second, we need to add the encryptData() and decryptData() methods into the PatientService class to support secure data transfer between services.

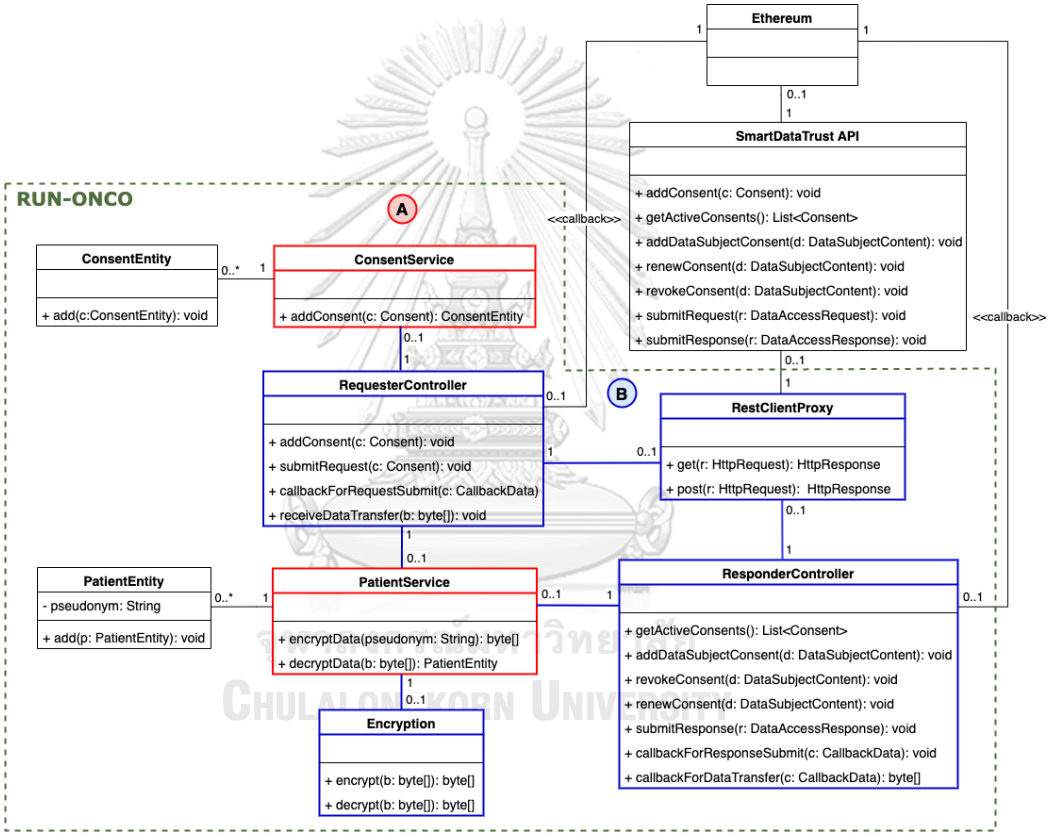



Figure 34: Class diagram demonstrating how a software platform for cancer precision medicine handles GDPR-compliant blockchain-based consent management in data sharing. (A) relevant classes needed to be enhanced to support data sharing. (B) new classes added to RUN-ONCO for supporting managed consent into the blockchain and handling the requester and responder callbacks made by the blockchain.

# CHAPTER VI
# ANALYSIS AND INTERPRETATION OF RESULTS

To justify our formal models corresponding to the competency questions in Table 5, we used the ProB for generating test cases to ensure that formal models fulfill a given coverage criteria. The ProB generates test cases based on non-deterministic choice in Event-B separated into three places [139]: 1) the choice derives from different events, 2) the choice derives from local variables of events, and 3) the choice derives from the non-deterministic assignment. The ProB executes events to perform test scenarios based on the non-deterministic choice corresponding to current state variables, invariants, and guards restricted to small finite sets. Besides, if unsatisfied guards exist in any events during the model checking simulation, then these events will be absent from the choice of the possible events on the next ones.

We thus specified test cases in both CM for centralized systems and CM for distributed systems in data sharing.

## 6.1. Test Cases in CM for Centralized Systems

### 6.1.1. Test Cases in the RPSM Model

This RPSM model describes the dynamic behavior of how the system manipulates patients' consent and how to restrict privileged permissions of authorized users (e.g., doctors, nurses, lab staff) for processing personal data within patients' consent.



We then specify the test case objectives as follows:

- **RP1**: In the AddPatient and AddConsent events, a user who does not obtain a nursing staff role shall not perform these events.
- **RP2**: A user who does not obtain any role granted in consents shall not perform the ExecuteQuery event.
- **RP3**: In the ExecuteQuery event, the local variable *consentExpired* shall be FALSE (i.e., the patient's consent is valid), and the user shall obtain the role granted in the consent configuration and hold in the variable *crf*.
- **RP4**: After the ExecuteQuery event firing, the variable *pf* (i.e., query results) shall contain only selected data fields corresponding to consent configuration.

- **RP5**: If a user has more than one role to access a patient's data under the same given consent, the value of variable *pf* shall contain all selected data fields corresponding to a user's roles.

First, we determine the variable value of *crf* and *userRoles*, before running the ProB simulation.

The variable value *crf* is:

```
{(CONSENTS1 ↦ {(NursingStaff ↦ HN)}),
 (CONSENTS2 ↦ {(Oncologist ↦ HN),
               (Oncologist ↦ Name),
               (Oncologist ↦ Age),
               (Researcher ↦ HN),
               (Researcher ↦ Omics)})}
```

The value of *crf* indicates that if a patient provides the CONSENTS1, only a user who has a NursingStaff role can access a patient's HN. As for the CONSENTS2, each role has access data fields differently. An oncologist can access a patient's information, e.g., HN, Name, and Age, but a researcher can access a patient's HN and Omics.

The variable value *userRoles* is:

```
{(AUTHORIZED_USER1 ↦ NursingStaff),
 (AUTHORIZED_USER1 ↦ LabStaff),
 (AUTHORIZED_USER2 ↦ Oncologist),
 (AUTHORIZED_USER2 ↦ Researcher),
 (AUTHORIZED_USER3 ↦ LabStaff)}
```



The value of *userRoles* indicates that AUTHORIZED_USERS1 obtains two roles, e.g., NursingStaff, and LabStaff. As for AUTHORIZED_USERS2 also has two roles, e.g., Oncologist, and Researcher. Lastly, AUTHORIZED_USERS3 obtain a role as LabStaff.

#### 6.1.1.1. The RP1 Test Case

According to Figure 35(A), AUTHORIZED_USERS3 login to the system as lab staff with SESSIONS2. Within the choice of events generated by ProB (Figure 35(B)), the Logout event is the only choice for AUTHORIZED_USERS3 to perform for the next event execution. It indicates that this user has no access to the AddPatient and AddConsent events because guard conditions are invalid for both events. Then, the state variable *sessions* has been updated with a new ordered pair (SESSIONS2 ↦ AUTHORIZED_USERS3), as shown in Figure 36.

Hence, simulation results point out that the RPSM model covered the RP1 test case.

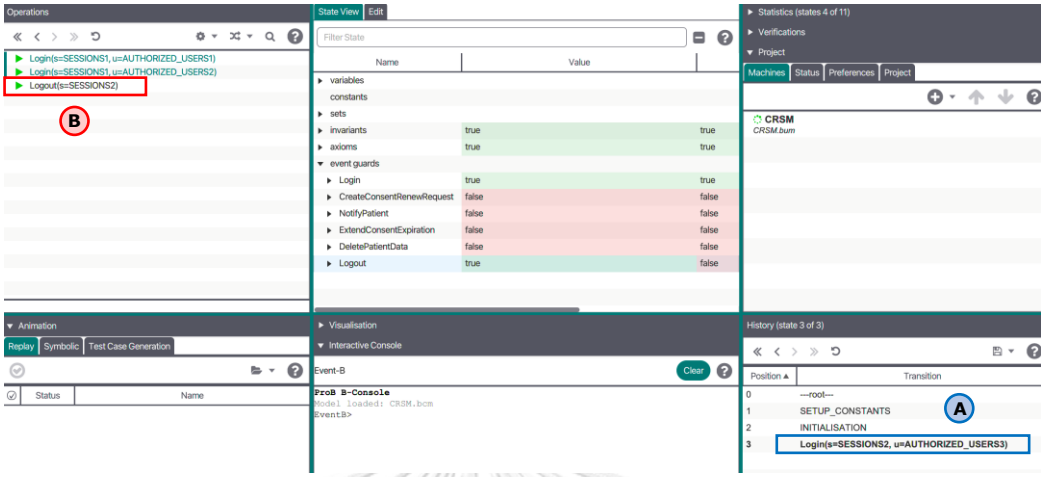



Figure 35: The simulation of the RP1 test case. (A) the Login event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS3 to perform for the next event execution.


Figure 36: The latest value of the variable *sessions* corresponds to event execution in the RP1 test case.



#### 6.1.1.2. The RP2 Test Case

In Figure 37(A), AUTHORIZED_USERS3 login to the system as lab staff with SESSIONS1 and creates an inquiry QUERIES1 to retrieve the personal data of PATIENTS1. Within the choice of events generated by ProB (Figure 37(B)), the Logout event is the only choice for AUTHORIZED_USERS3 to perform for the next event execution. It indicates that this user cannot access the ExecuteQuery event because guard conditions are invalid. Then, the state variable *queries* has been updated with a new ordered pair {(AUTHORIZED_USERS3 ↦ {(QUERIES1 ↦ PATIENTS1)})}, as shown in Figure 38.

Hence, simulation results point out that the RPSM model covered the RP2 test case.

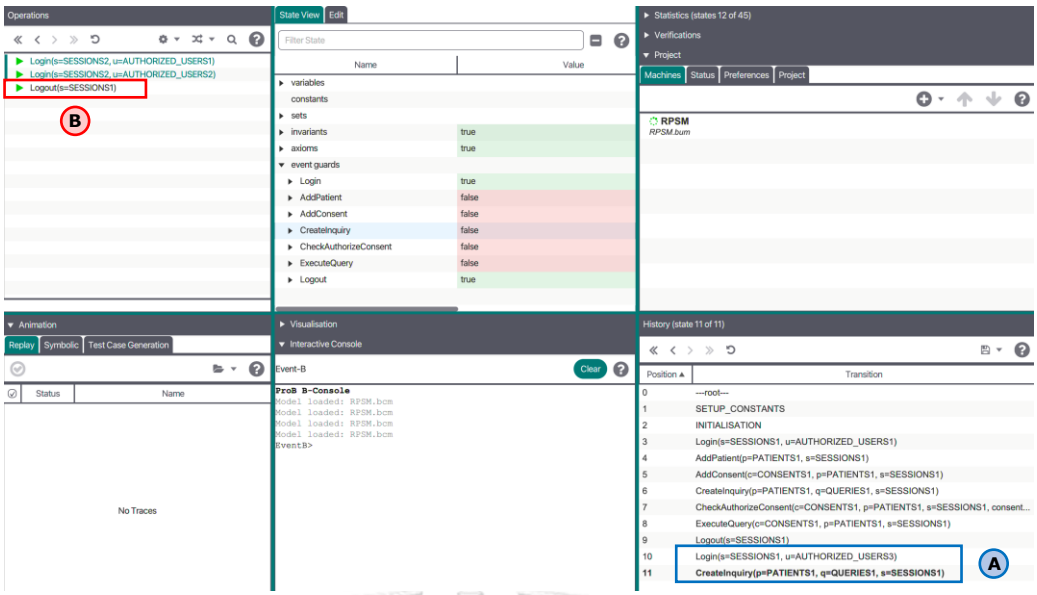



Figure 37: The simulation of the RP2 test case. (A) the CreateInquiry event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS3 to perform for the next event execution.

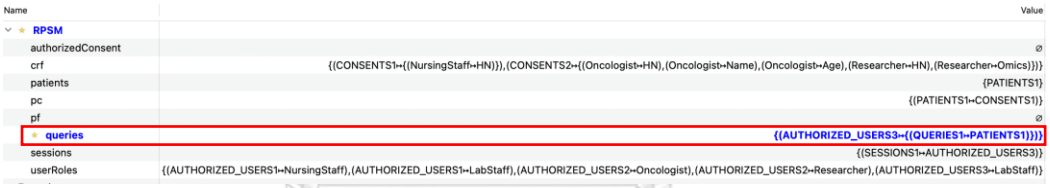



Figure 38: The latest value of the variable *queries* corresponds to event execution in the RP2 test case.

#### 6.1.1.3. The RP3 Test Case

Figure 39(A) demonstrates that AUTHORIZED_USERS1 login into the system as nursing staff creates an inquiry QUERIES1 to retrieve the personal data of PATIENTS1, and the query has been verified according to the patient's consent. Within the choice of events generated by ProB, two events are available for AUTHORIZED_USERS1 to perform for the next event execution, e.g., the ExecuteQuery event, and Logout events. It indicates that this user can access the ExecuteQuery event because guard conditions are valid (Figure 39(B)). Then, the state variable *authorizedConsent* has been updated with a new ordered pair {(AUTHORIZED_USERS1 ↦ {(PATIENTS1 ↦ CONSENTS1)})}, as shown in Figure 40.

Hence, simulation results point out that the RPSM model covered the RP3 test case.

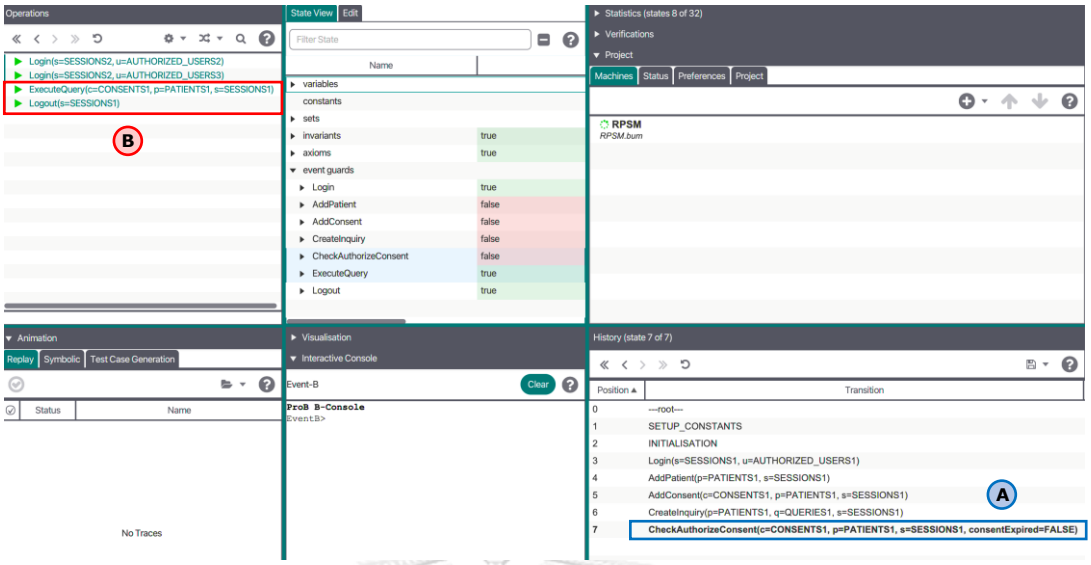



Figure 39: The simulation of the RP3 test case. (A) the CheckAuthorizeConsent event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS1 to perform for the next event execution.

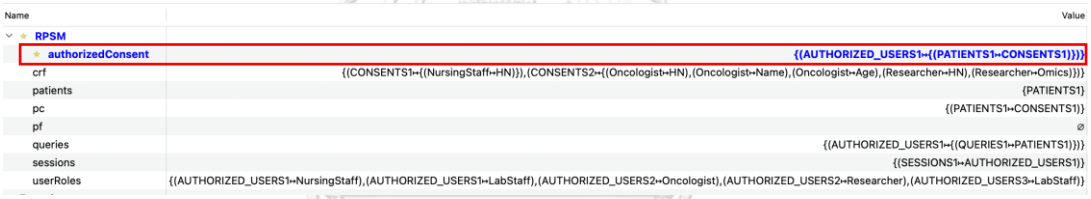



Figure 40: The latest value of the variable *authorizedConsent* corresponds to event execution in the RP3 test case.



#### 6.1.1.4. The RP4 Test Case

According to Figure 41(A), AUTHORIZED_USERS1 executes the query and receives the personal data of PATIENTS1 within CONSENTS1. Then, the state variable *pf* has been updated with a new ordered pair {(AUTHORIZED_UERS1 ↦ {(PATIENTS1 ↦ HN)})}. Based on the configuration of CONSENTS1, any user who obtains a nursing staff role has access to the patient's HN. So, the value of variable *pf* corresponds to the given consent, as shown in Figure 42.

Hence, simulation results point out that the RPSM model covered the RP4 test case.

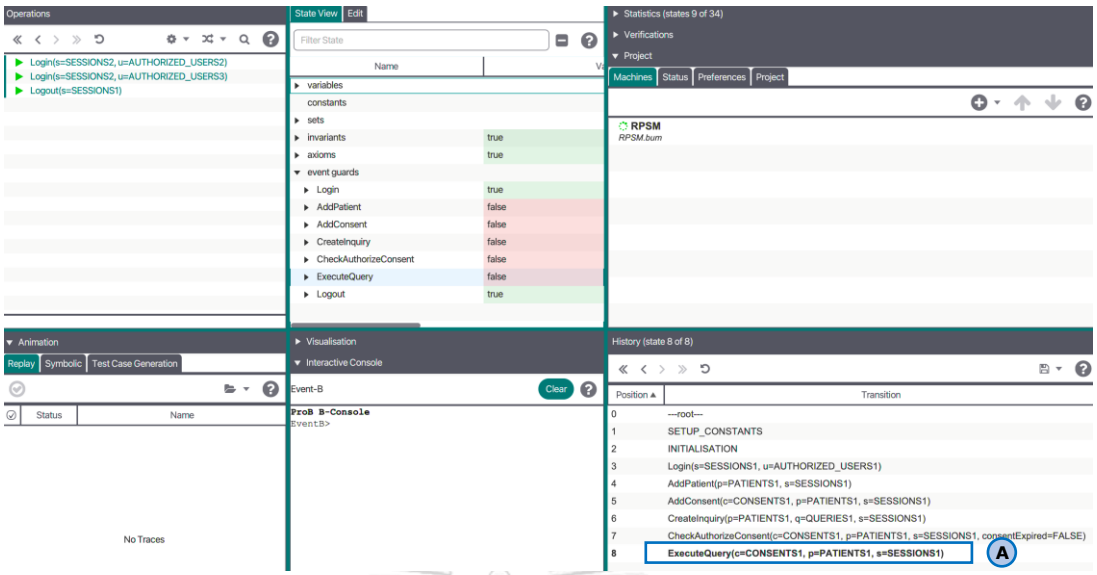



Figure 41: The simulation of the RP4 test case. (A) the ExecuteQuery event and its variables are produced by ProB, which has been executed in the history panel.

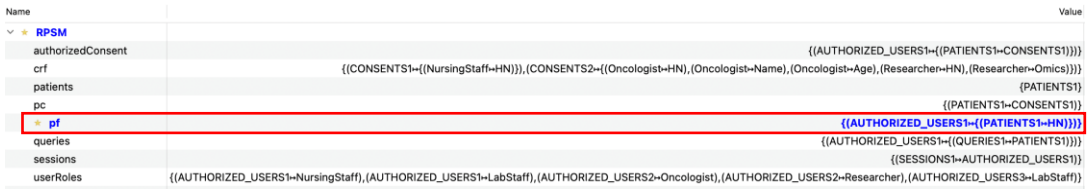



Figure 42: The latest value of the variable *pf* corresponds to event execution in the RP4 test case.

#### 6.1.1.5. Test RP5 Test Case

To begin with, AUTHORIZED_USERS1 adds PATIENTS1 and the patient's consent (PATIENTS1 ↦ CONSENTS2) into the system (Figure 43(A)). However, the configuration of CONSENTS2 states that a user with an oncologist role can access a patient's HN, Name, and Age; a user with a researcher role can access a patient's HN and Omics. So, a user who obtains these roles, e.g., oncologist and researcher, shall access a patient's HN, Name, Age, and Omics.

According to Figure 43(B), AUTHORIZED_USER2 login to the system, which obtains two roles, e.g., oncologist and researcher. Then, the user creates a query for accessing the personal data of PATIENTS1 under CONSENTS2. After verifying the consent validation, the system executes the query result. Hence, the value of variable *pf* corresponds to the expected result (Figure 44), which indicates that the RPSM model covered the RP5 test case.

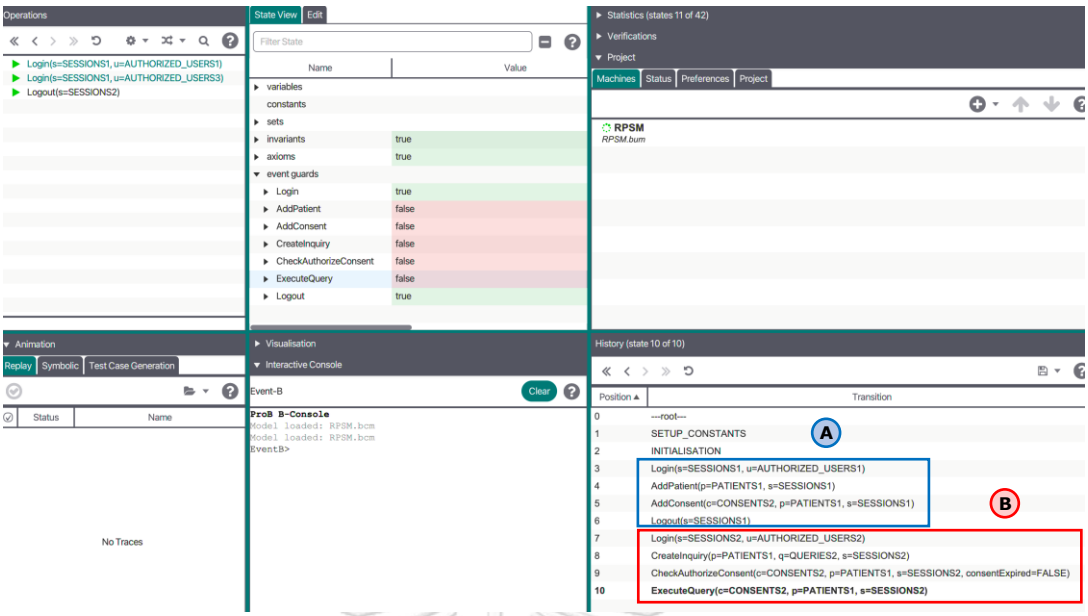



Figure 43: The simulation of the RP5 test case. (A) AUTHORIZED_USERS1 adds PATIENTS1 and his/her given consent. (B) AUTHORIZED_USERS2 creates query to access the information of PATIENTS1 under CONSENTS2.


Figure 44: The latest value of the variable *pf* corresponds to event execution in the RP5 test case.

### 6.1.2. Test Cases in the WASM Model

The WASM model describes the dynamic behavior of how the system manages the withdrawal approval process when patients request to withdraw their consent. The user's roles that are involved in this process are legal staff and legal approvers.

We then specify the test case objectives as follows:

- **WA1**: In the CreateWithdrawal, ApproveWithdrawal, and RejectWithdrawal events, a user who does not obtain the legal staff and legal approval roles shall not perform these events.
- **WA2**: In the CreateWithdrawal event, a user who has a legal staff role shall create the withdrawal request.
- **WA3**: In the ApproveWithdrawal event, a user who has a legal approver role shall permit to approve the withdrawal request on the condition that

*canWithdraw* is TRUE. After the withdrawal request has been approved, the withdrawal request's status shall be updated to approved, and the system shall add the patient's consent into the variable *markAsDeleted* to indicate that the patient's personal data shall be deleted from the system.

- **WA4**: In the RejectWithdrawal event, a user who has a legal approver role shall permit to reject the withdrawal request on the condition that *canWithdraw* is FALSE.

First, we determine the variable value of *userRoles* and *pc*, before running ProB.

The variable value *userRoles* is:

```
{(AUTHORIZED_USERS1 ↦ LegalStaff),
 (AUTHORIZED_USERS2 ↦ LegalApprover),
 (AUTHORIZED_USERS3 ↦ NursingStaff),
 (AUTHORIZED_USERS3 ↦ LabStaff)}
```

The value of *userRoles* indicates the AUTHORIZED_USERS1 and AUTHORIZED_USERS2, users obtain a role, i.e., LegalStaff, and LegalApprover, respectively. As for the AUTHORIZED_USERS3 obtains two roles, i.e., NursingStaff, and LabStaff.

The variable value *pc* is:

```
{(PATIENTS1 ↦ CONSENTS1),
 (PATIENTS2 ↦ CONSENTS1)}
```



The value of *pc* contains patients' consents, e.g., the PATIENTS1 has given the CONSENTS1, and the PATIENTS2 has given the CONSENTS1.

#### 6.1.2.1. The WA1 Test Case

According to Figure 45(A), AUTHORIZED_USERS3 login to the system with SESSIONS1. However, AUTHORIZED_USERS3 obtains two roles, i.e., nursing staff, and lab staff. Within the choice of events generated by ProB ( Figure 45(B)), the Logout event is the only choice for AUTHORIZED_USERS3 to perform for the next event execution. It indicates that this user cannot access the CreateWithdrawal, ApproveWithdrawal, and RejectWithdrawal events because guard conditions are invalid for all three events. Then, the state variable *sessions* has been updated with a new ordered pair (SESSIONS1 ↦ AUTHORIZED_USERS3), as shown in Figure 46.

Hence, simulation results point out that the WASM model covered the WA1 test case.

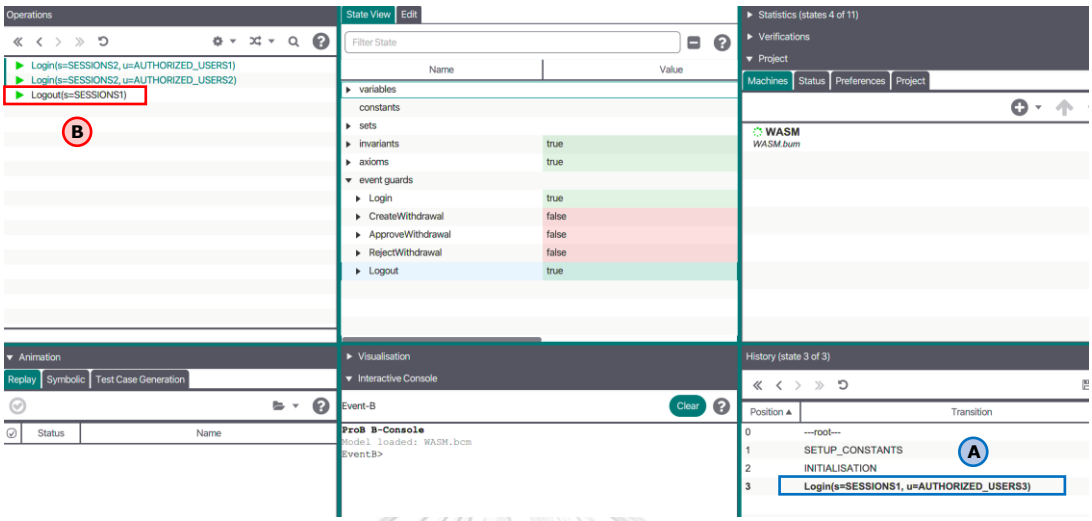



Figure 45: The simulation of the WA1 test case. (A) the Login event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS3 to perform for the next event execution.

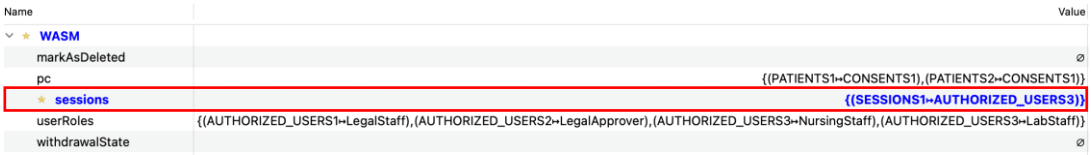



Figure 46: The latest value of the variable *sessions* corresponds to event execution in the WA1 test case.



### 6.1.2.2. The WA2 Test Case

In Figure 47(A), AUTHORIZED_USERS1 login to the system as legal staff and creates the withdrawal request for PATIENTS1 under CONSENTS1. Within the choice of events generated by ProB (Figure 47(B)), the Logout event is the only choice for AUTHORIZED_USERS1 to perform. It indicates that this user cannot access the ApproveWithdrawal and RejectWithdrawal events because guard conditions are invalid for both events. Then, the state variable *withdrawState* has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Void)}, as shown in Figure 48.

Hence, simulation results point out that the WASM model covered the WA2 test case.

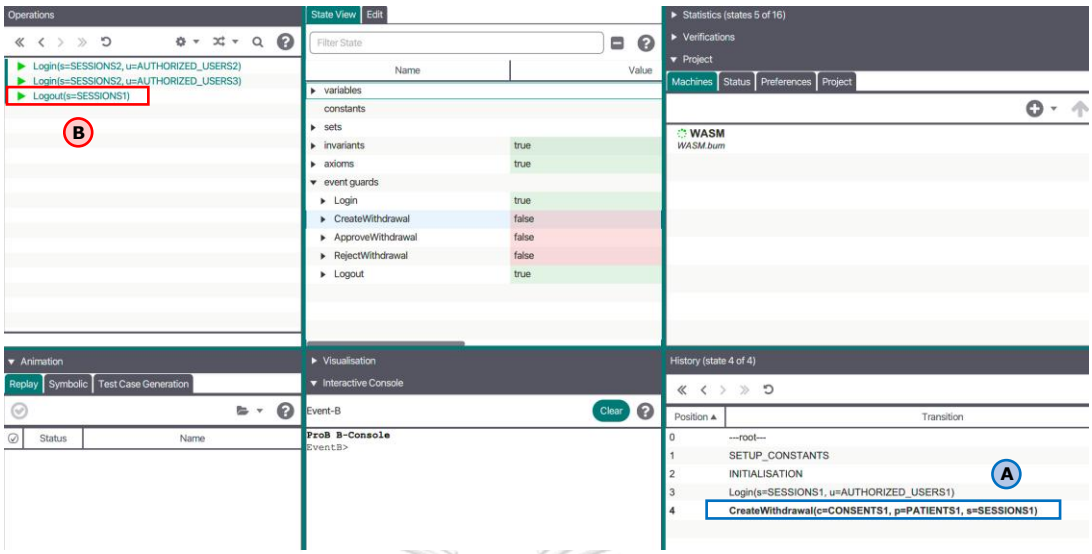



Figure 47: The simulation of the WA2 test case. (A) the CreateWithdrawal event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS1 to perform for the next event execution.

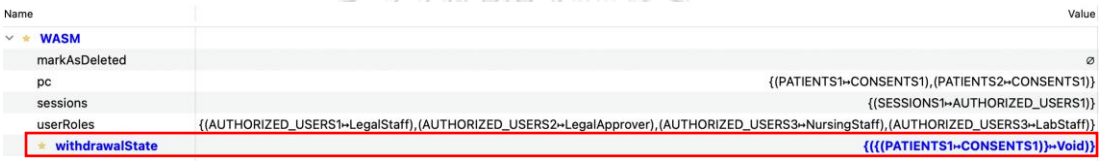



Figure 48: The latest value of the variable *withdrawState* corresponds to event execution in the WA2 test case.

#### 6.1.2.3. The WA3 Test Case

Figure 49(A) demonstrates that AUTHORIZED_USERS2 login to the system as legal approver and the local variable *canWithdraw* is TRUE (i.e., there is no conflict of interest on the consent revocation), then this user approves the withdrawal request. Hence, the *withdrawState* and *markAsDeleted* variables have been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Approved)} and {(PATIENT ↦ CONSENTS1)}, respectively, as shown in Figure 50.

Hence, simulation results point out that the WASM model covered the WA3 test case.

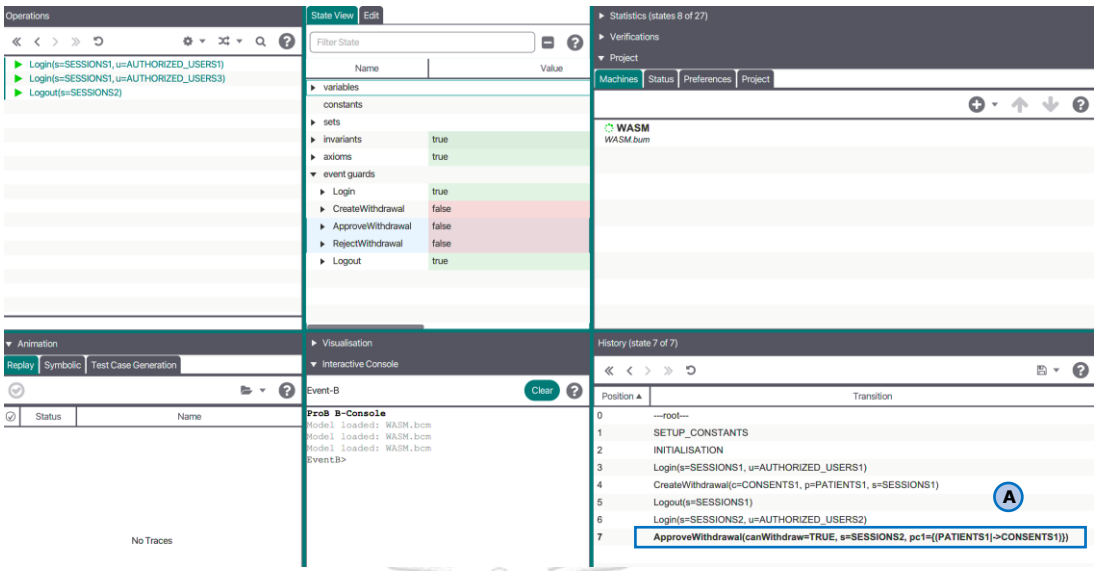



Figure 49: The simulation of the WA3 test case. (A) the ApproveWithdrawal event and its variables are produced by ProB, which has been executed in the history panel.

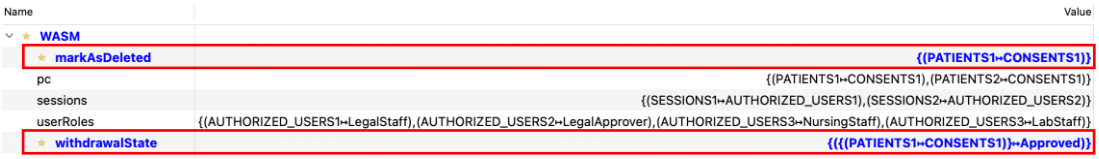



Figure 50: The latest values of *withdrawState and markAsDeleted* variables correspond to event execution in the WA3 test case.

#### 6.1.2.4. The WA4 Test Case

Figure 51(A) demonstrates that AUTHORIZED_USERS2 login to the system as legal approver and the local variable *canWithdraw* is FALSE (i.e., there exists a conflict of interest in the consent revocation), then this user rejects the withdrawal request. Hence, the *withdrawState* variable has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Rejected)}, as shown in Figure 52.

Hence, simulation results point out that the WASM model covered the WA4 test case.

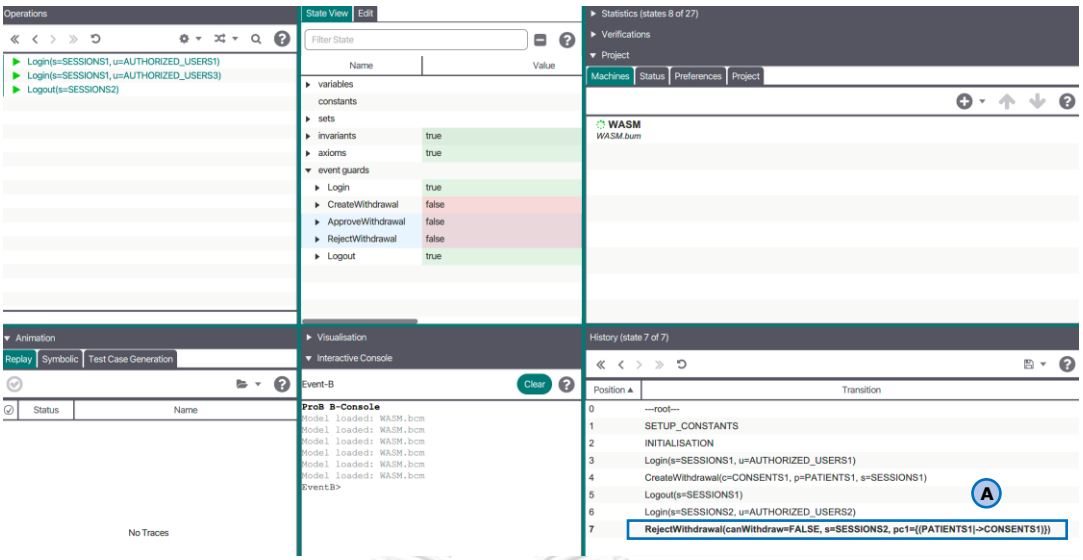



Figure 51: The simulation of the WA4 test case. (A) the RejectWithdrawal event and its variables are produced by ProB, which has been executed in the history panel.

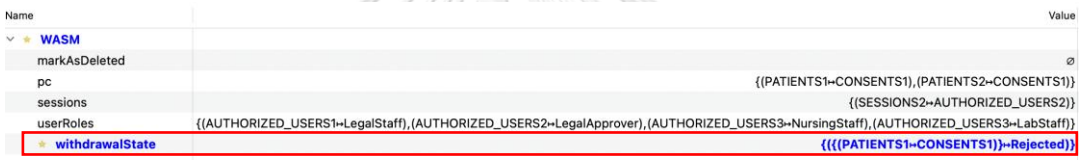



Figure 52: The latest value of the variable *withdrawState* corresponds to event execution in the WA4 test case.

### 6.1.3. Test Cases in the PASM Model

The PASM model describes the dynamic behavior of how the system manages the portable approval process when patients request to portable their personal. The user's roles that are involved in this process are legal staff and legal approver.

We then specify the test case objectives as follows:

- **PA1**: In the CreatePortable, ApprovePortable, and RejectPortable events, a user who does not obtain the legal staff and legal approver roles shall not perform these events.
- **PA2**: In the CreatePortable event, a user who has a legal staff role shall create the portable request.
- **PA3**: In the ApprovePortable event, a user who has a legal approver role shall permit to approve the portable request on the condition that *canPortable* is TRUE.

- **PA4**: In the RejectPortable event, a user who has a legal approver role shall permit to reject the portable request on the condition that *canPortable* is FALSE.

First, we determine the variable value of *userRoles* and *pc*, before running ProB.

The variable value *userRoles* is:

```
{(AUTHORIZED_USERS1 ↦ LegalStaff),
 (AUTHORIZED_USERS2 ↦ LegalApprover),
 (AUTHORIZED_USERS3 ↦ NursingStaff),
 (AUTHORIZED_USERS3 ↦ LabStaff)}
```

The value of *userRoles* indicates the AUTHORIZED_USERS1 and AUTHORIZED_USERS2, users obtain a role, i.e., LegalStaff, and LegalApprover, respectively. As for the AUTHORIZED_USERS3 obtains two roles, i.e., NursingStaff, and LabStaff.

The variable value *pc* is:

```
{(PATIENTS1 ↦ CONSENTS1),
 (PATIENTS2 ↦ CONSENTS1)}
```

The value of *pc* contains patients' consents, e.g., the PATIENTS1 has given the CONSENTS1, and the PATIENTS2 has given the CONSENTS1.

#### 6.1.3.1. The PA1 Test Case

According to Figure 53(A), AUTHORIZED_USERS3 login to the system with SESSIONS1. However, AUTHORIZED_USERS3 obtains two roles, e.g., nursing staff, and lab staff. Within the choice of events generated by ProB (Figure 53(B)), the Logout event is the only choice for AUTHORIZED_USERS3 to perform for the next event execution. It indicates that this user cannot access the CreatePortable, ApprovePortable, and RejectPortable events because guard conditions are invalid for all three events. Then, the state variable *sessions* has been updated with a new ordered pair (SESSIONS1 ↦ AUTHORIZED_USERS3), as shown in Figure 54.

Hence, simulation results point out that the PASM model covered the PA1 test case.

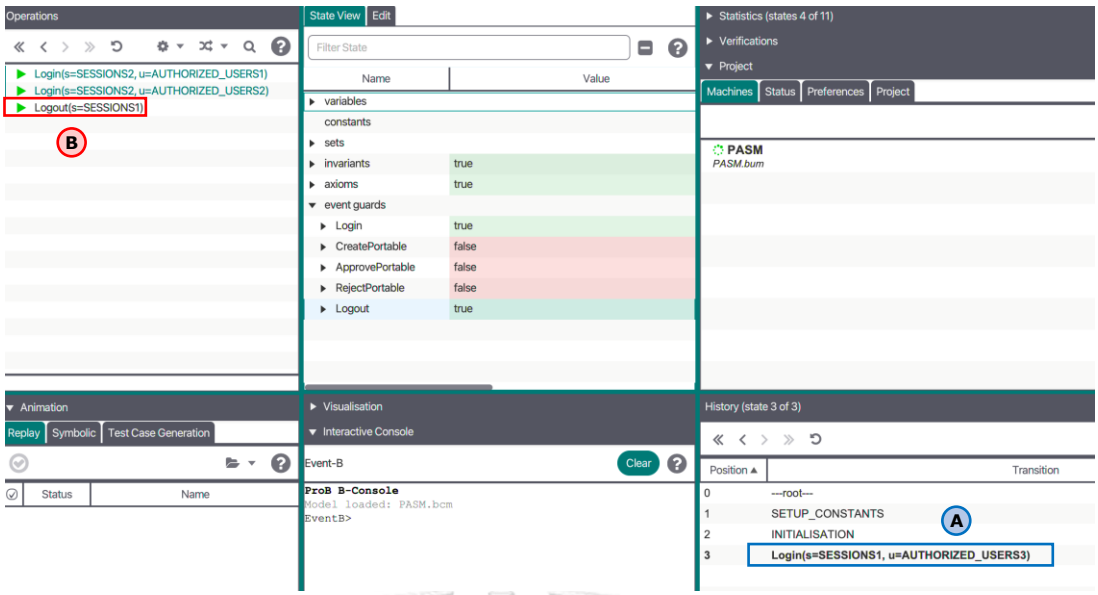



Figure 53:The simulation of the PA1 test case. (A) the Login event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS3 to perform for the next event execution.

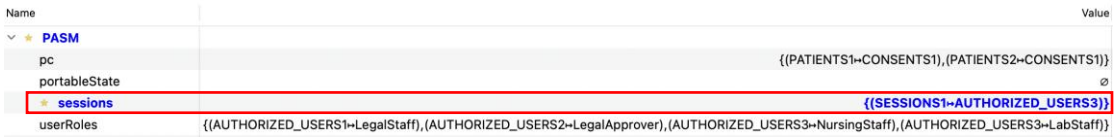



Figure 54: The latest value of the variable *sessions* corresponds to event execution in the PA1 test case.

#### 6.1.3.2. The PA2 Test Case

In Figure 55(A), AUTHORIZED_USERS1 login to the system as the legal staff and creates the portable request for PATIENTS1 under CONSENTS1. Within the choice of events generated by ProB Figure 55(B), the Logout event is the only choice for AUTHORIZED_USERS1 to perform for the next event execution. It indicates that this user cannot access the ApprovePortable and RejectPortable events because guard conditions are invalid for both events. Then, the state variable *withdrawState* has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Void)}, as shown in Figure 56.

Hence, simulation results point out that the PASM model covered the PA2 test case.

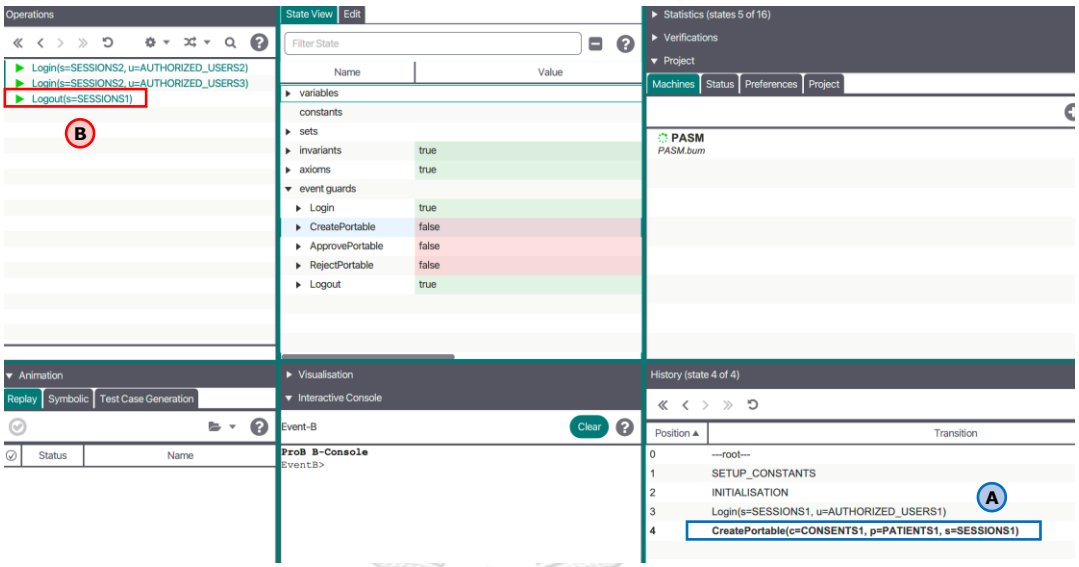



Figure 55: The simulation of the PA2 test case. (A) the CreatePortable event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS1 to perform for the next event execution.

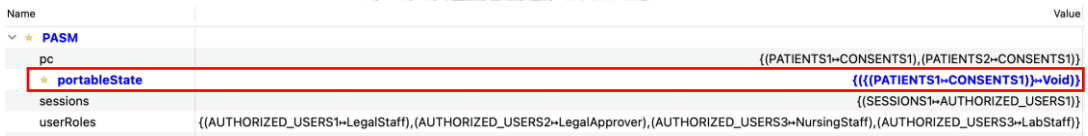



Figure 56: The latest value of the variable *portableState* corresponds to event execution in the PA2 test case.

#### 6.1.3.3. The PA3 Test Case

Figure 57(A) demonstrates that AUTHORIZED_USERS2 login to the system as legal approver and the local variable *canPortable* is TRUE (i.e., there might be a fee for exporting personal data, and if the patient accepts to pay, then this variable becomes TRUE), then this user approves the portable request. Hence, the *portableState* variable has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Approved)}, as shown in Figure 58.

Hence, simulation results point out that the PASM model covered the PA3 test case.

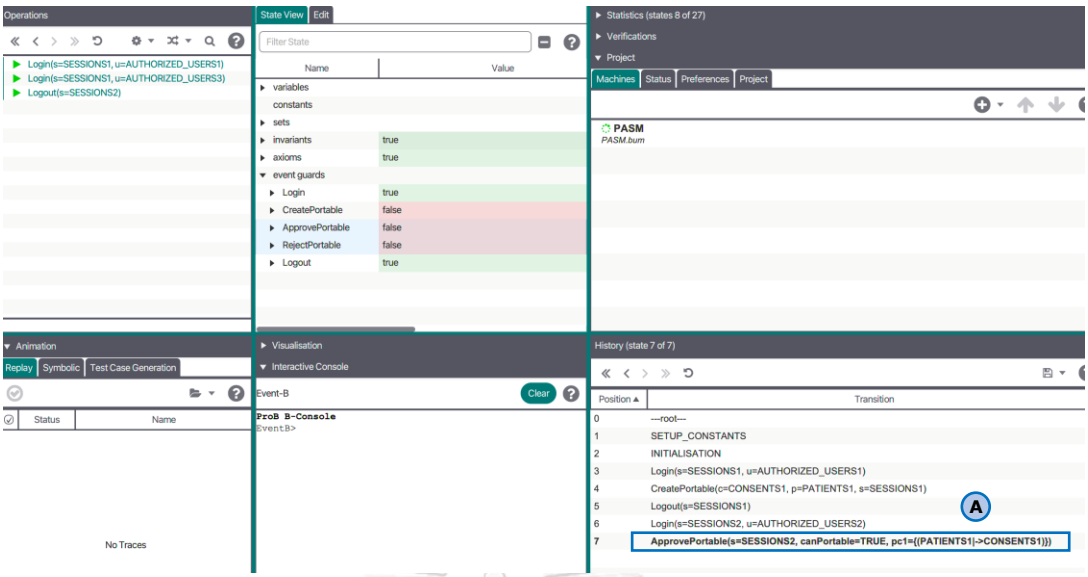



Figure 57: The simulation of the PA3 test case. (A) the ApprovePortable event and its variables are produced by ProB, which has been executed in the history panel.

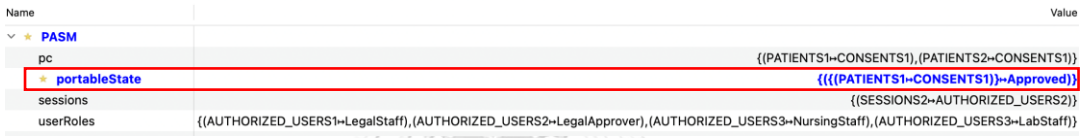



Figure 58: The latest value of the variable *portableState* corresponds to event execution in the PA3 test case.

### 6.1.3.4. The PA4 Test Case

Figure 59(A) demonstrates that AUTHORIZED_USERS2 login to the system as legal approver and the local variable *canPortable* is FALSE (i.e., there might be a fee for exporting personal data, and if the patient declines to pay, then this variable becomes FALSE), then this user rejects the portable request. Hence, the *portableState* variable has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Rejected)}, as shown in Figure 60.

Hence, simulation results point out that the PASM model covered the PA4 test case.

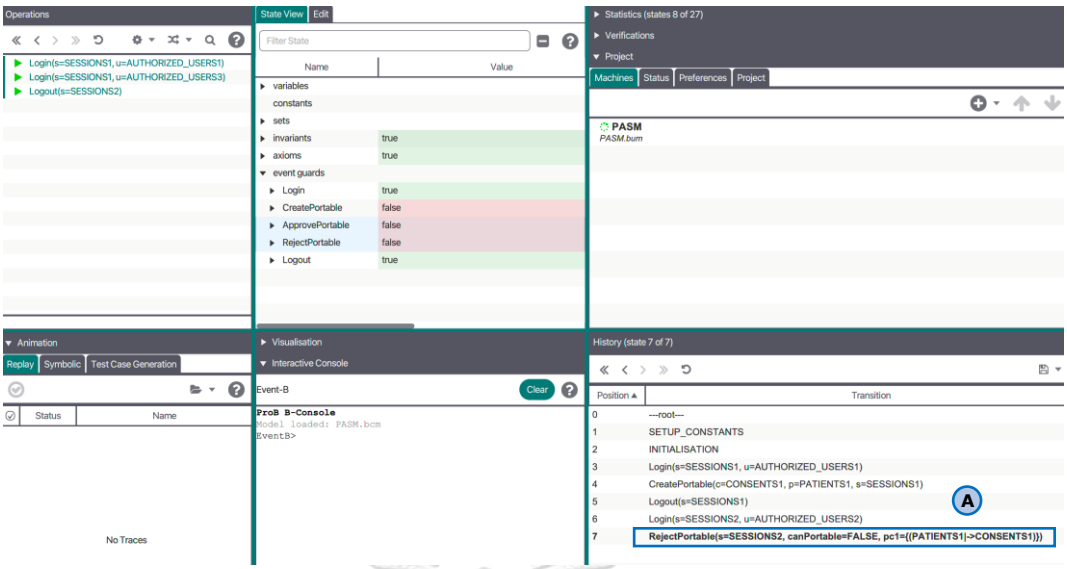



Figure 59: The simulation of the PA4 test case. (A) the RejectPortable event and its variables are produced by ProB, which has been executed in the history panel.

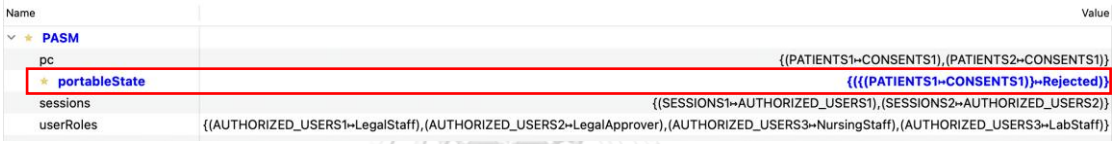



Figure 60: The latest value of the variable *portable* corresponds to event execution in the PA4 test case.

### 6.1.4. Test Cases in the CRSM Model

The CRSM model describes the dynamic behavior of how the system manages the consent renewal process when patients' consent expires. The user's role that is involved in this process is the legal staff.



We then specify the test case objectives as follows:

- **CR1**: In CreateConsentRenewRequest, NotifyPatient, ExtendConsentExpiration, and DeletePatientData events, a user who does not obtain the legal staff role shall not perform these events.
- **CR2**: In the CreateConsentRenewRequest event, only the legal staff shall create the consent renewal request under these conditions: the patient's consent is expired but is not withdrawn.
- **CR3**: As for the NotifyPatient event, the legal staff shall inform the patient about the consent renewal and receives the patient's response for approval or rejection on extending the data retention.

- **CR4**: If the patient approves the consent renewal, the legal staff shall update the consent to unexpired.
- **CR5**: If the patient rejects the consent renewal, the legal staff shall add the consent into the variable *markAsDeleted* to indicate that the patient's personal data shall be deleted from the system.

First, we determine the variable value of *userRoles* and *pc*, before running ProB.

The variable value *userRoles* is:

```
{(AUTHORIZED_USERS1 ↦ LegalStaff),
 (AUTHORIZED_USERS2 ↦ LegalApprover),
 (AUTHORIZED_USERS3 ↦ NursingStaff),
 (AUTHORIZED_USERS3 ↦ LabStaff)}
```

The value of *userRoles* indicates the AUTHORIZED_USERS1 and AUTHORIZED_USERS2, users obtain a role, i.e., LegalStaff, and LegalApprover, respectively. As for the AUTHORIZED_USERS3 obtains two roles, i.e., NursingStaff, and LabStaff.

The variable value *pc* is:

```
{(PATIENTS1 ↦ CONSENTS1),
 (PATIENTS2 ↦ CONSENTS1)}
```

The value of *pc* contains patients' consents, e.g., the PATIENTS1 has given the CONSENTS1, and the PATIENTS2 has given the CONSENTS1.



#### 6.1.4.1. The CR1 Test Case

According to Figure 61(A), AUTHORIZED_USERS3 login to the system with SESSIONS1. However, AUTHORIZED_USERS3 obtains two roles, i.e., nursing staff, and lab staff. Within the choice of events generated by ProB (Figure 61(B)), the Logout event is the only choice for AUTHORIZED_USERS3 to perform for the next event execution. It indicates that this user cannot access the CreateConsentRenewRequest, NotifyPatient, ExtendConsentExpiration, and DeletePatientData events because guard conditions are invalid for all four events. Then, the state variable *sessions* has been updated with a new ordered pair (SESSIONS1 ↦ AUTHORIZED_USERS3), as shown in Figure 62.

Hence, simulation results point out that the CRSM model covered the CR1 test case.

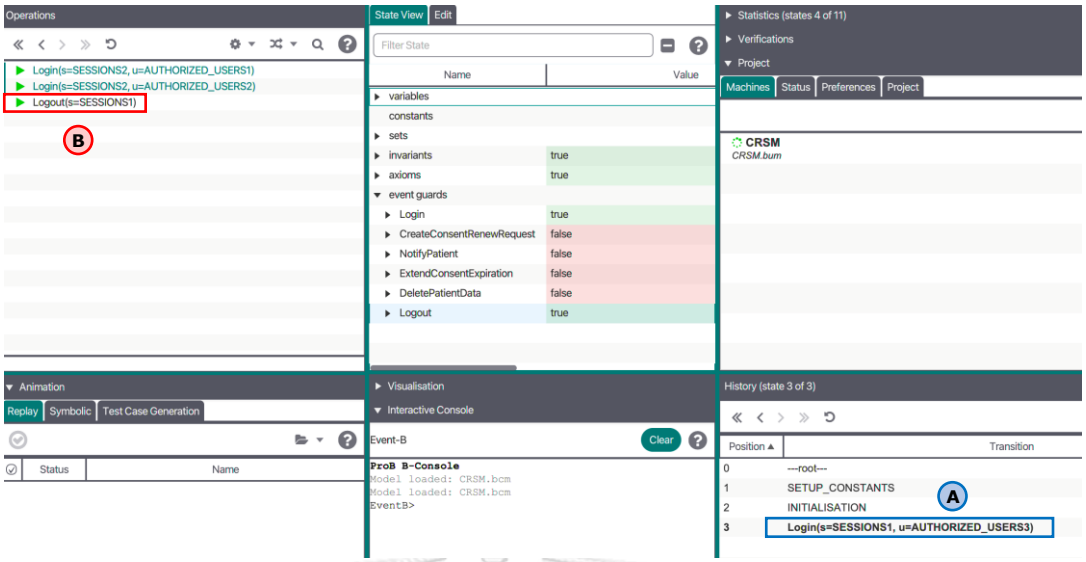



Figure 61: The simulation of the CR1 test case. (A) the Login event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS3 to perform for the next event execution.

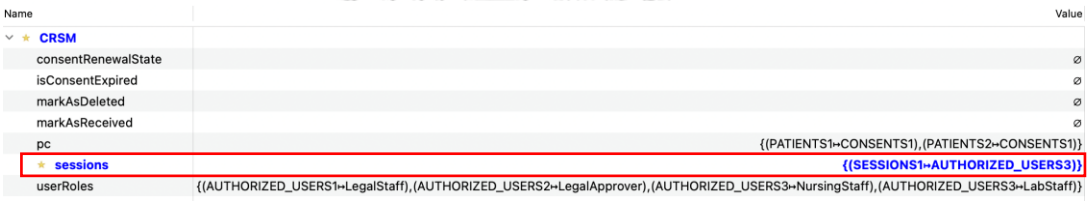



Figure 62: The latest value of the variable *sessions* corresponds to event execution in the CR1 test case.

### 6.1.4.2. The CR2 Test Case

Figure 63(A) demonstrates that AUTHORIZED_USERS1 login to the system as legal staff and the local variables of *expired* is TRUE (i.e., the patient's consent is expired) and *isWithdraw* is FALSE (i.e., the patient's consent is not withdrawn), then this user creates the consent renewal request to inform the patient about the data retention extension to allow the hospital to continue to process his/her personal. Within the choice of events generated by ProB (Figure 63(B)), there are two events available for AUTHORIZED_USERS1 to perform, e.g., the NotifyPatient, and Logout events. It indicates that this user can access the NotifyPatient event because guard conditions are valid. Then, the *consentRenewalState* and *isConsentExpired* variables have been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Void)} and {({(PATIENTS1 ↦ CONSENTS1)} ↦ TRUE)}, respectively, as shown in Figure 64.

Hence, simulation results point out that the CRSM model covered the CR2 test case.

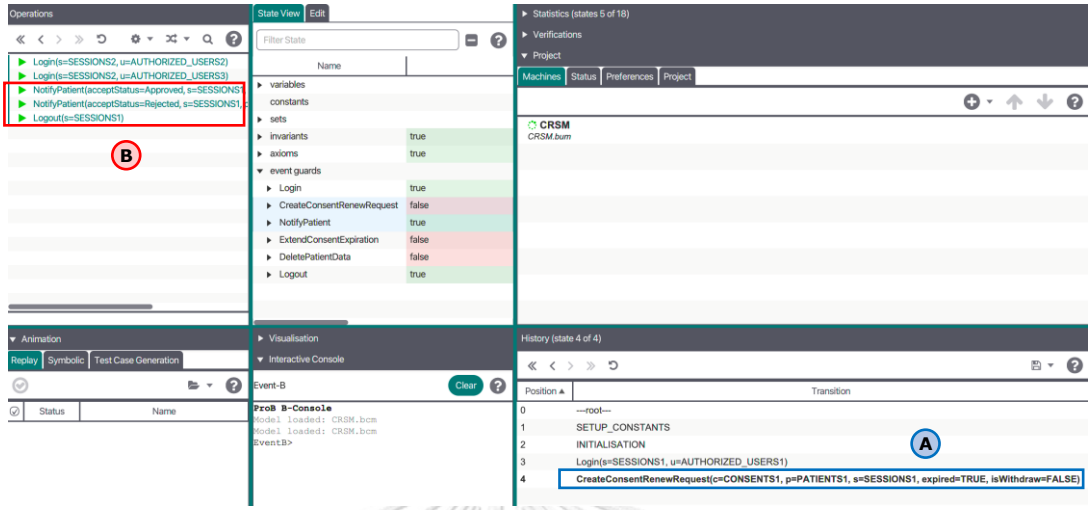



Figure 63: The simulation of the CR2 test case. (A) the CreateConsentRenewRequest event and its variables are produced by ProB, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS to perform for the next event execution.

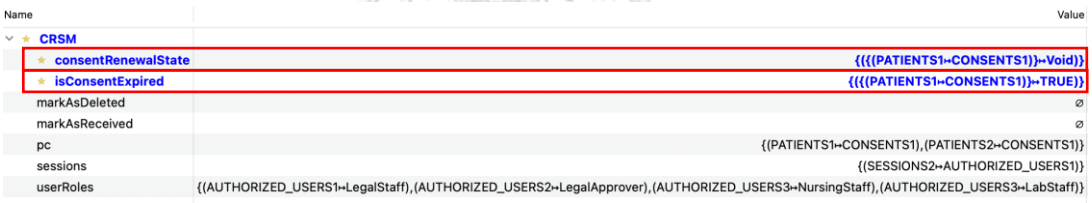



Figure 64: The latest values of *consentRenewalState and isConsentExpired* variables correspond to event execution in the CR2 test case.

### 6.1.4.3. The CR3 Test Case

As for the NotifyPatient event, AUTHORIZED_USERS1 informs the patient to extend data retention (Figure 65(A) and Figure 67(A)). After receiving the patient's answer (i.e., Approved, Rejected), then this user saves the answer into the system. If the patient approves extending data retention, then ProB generates the two possible events for AUTHORIZED_USERS1 to perform. i.e., the ExtendConsentExpiration, and Logout events (Figure 65(B)). The state variables *consentRenewalState* and *markAsReceived* have been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Approved)}, and {(PATIENTS1 ↦ CONSENTS1)}, respectively (Figure 66).

On the other hand, If the patient rejects to stop processing his/her personal data, then ProB generates the two events for AUTHOR-

IZED_USERS1 to perform. e.g., the DeletePatientData, and Logout events (Figure 67(B)). The state variables *consentRenewalState* and *markAsReceived* have been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ Rejected)}, and {(PATIENTS1 ↦ CONSENTS1)}, respectively (Figure 68).

Hence, simulation results point out that the CRSM model covered the CR3 test case.

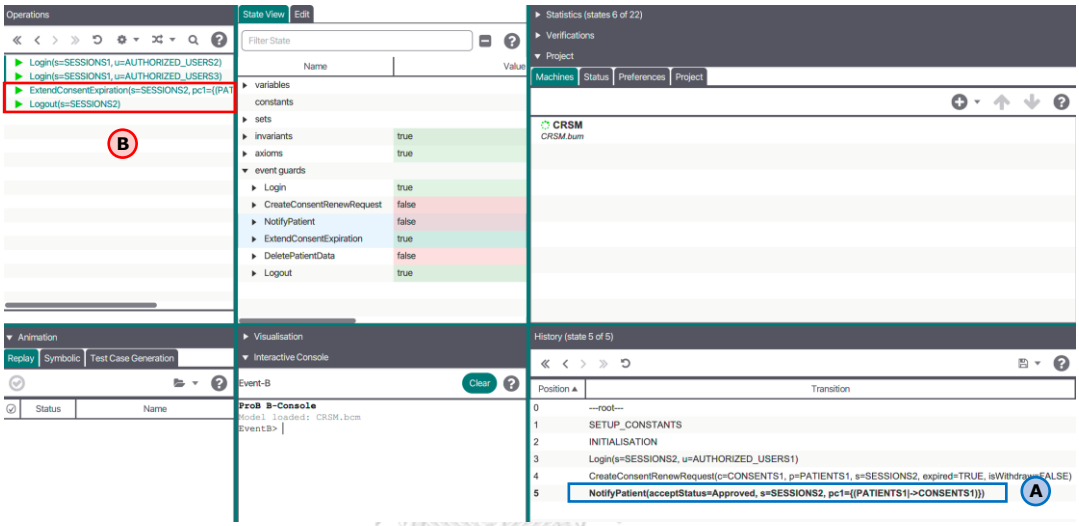



Figure 65: The simulation of the CR3 test case. (A) the NotifyPatient event with "Approved" status, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS1 to perform for the next event execution.

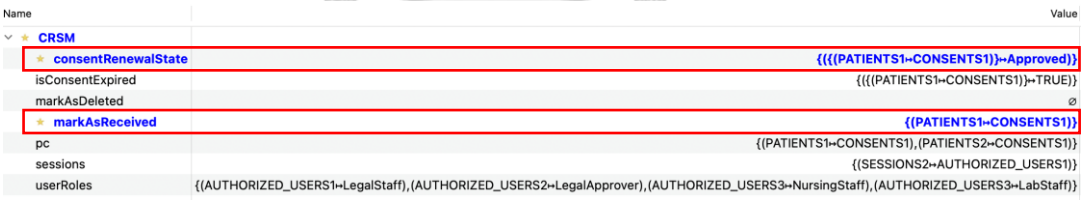



Figure 66: The latest values of *consentRenewalState and markAsReceived* variables correspond to event execution in the CR3 test case.

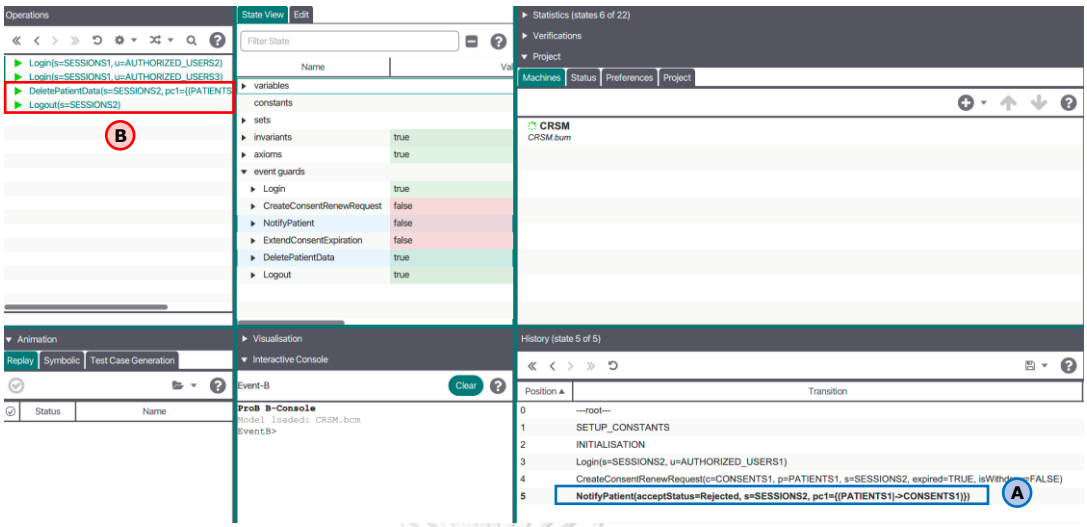



Figure 67: The simulation of the CR3 test case. (A) the NotifyPatient event with "Rejected" status, which has been executed in the history panel. (B) the choice of events allows AUTHORIZED_USERS1 to perform for the next event execution.

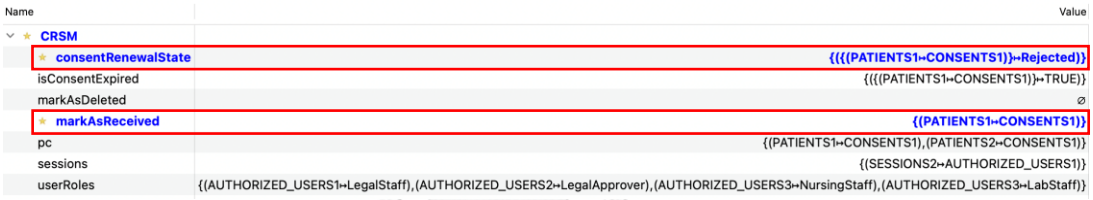



Figure 68: The latest values of *consentRenewalState and markAsReceived* variables correspond to event execution in the CR3 test case.

#### 6.1.4.4. The CR4 Test Case

After the patient approves the consent renewal request, then AUTHORIZED_USERS1 extends the data retention within the given consent (Figure 69(A)). Hence, the *isConsentExpired* variable has been updated with a new ordered pair {({(PATIENTS1 ↦ CONSENTS1)} ↦ FALSE)}, as shown in Figure 70.

Hence, simulation results point out that the CRSM model covered the CR4 test case.

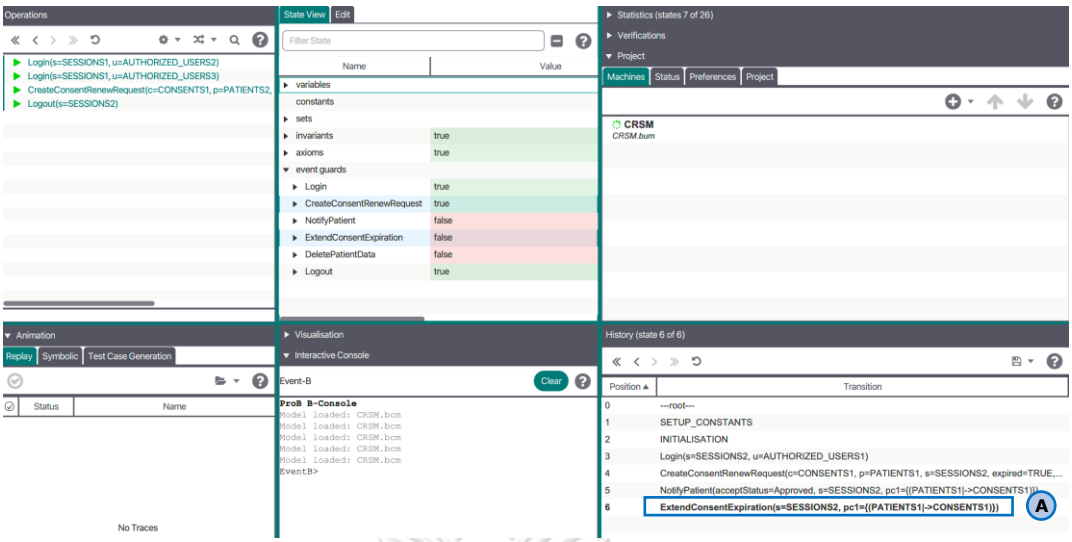



Figure 69: The simulation of the CR4 test case. (A) the ExtendConsentExpiration event and its variables are produced by ProB, which has been executed in the history panel.

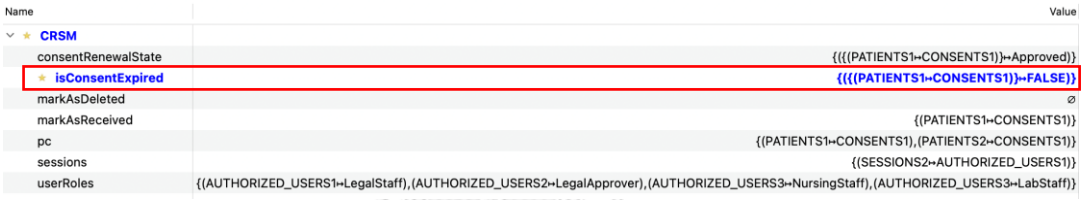



Figure 70: The latest value of the variable *isConsentExpired* corresponds to event execution in the CR4 test case.

#### 6.1.4.5. The CR5 Test Case

After the patient rejects the consent renewal request, then AUTHORIZED_USERS1 deletes the patient's personal data. Figure 71(A). Hence, the *markAsDeleted* variable has been updated with a new ordered pair {(PATIENTS1 ↦ CONSENTS1)}, as shown in Figure 72.

Hence, simulation results point out that the CRSM model covered the CR5 test case.

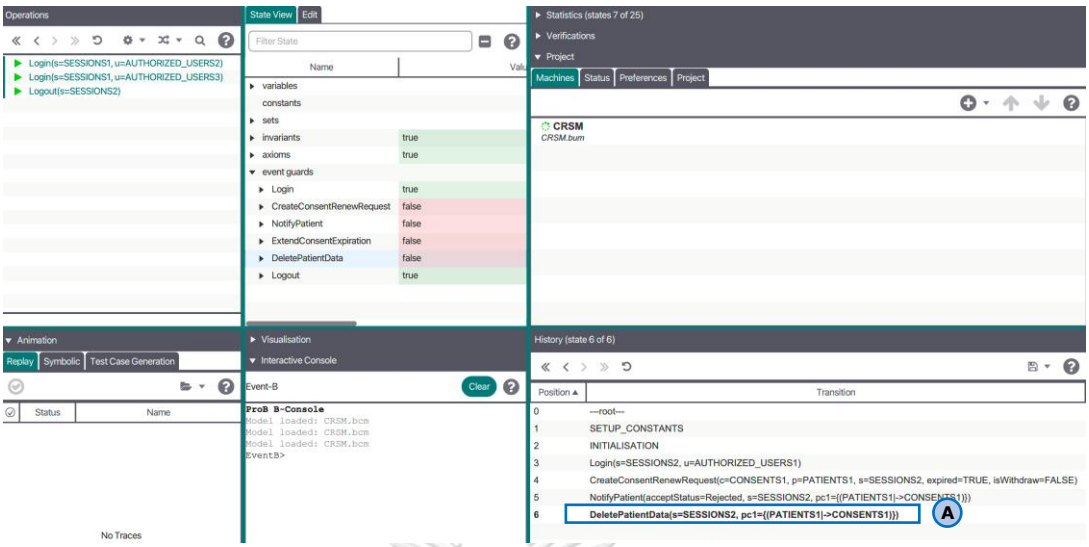



Figure 71: The simulation of the CR5 test case. (A) the DeletePatientData event and its variables are produced by ProB, which has been executed in the history panel.

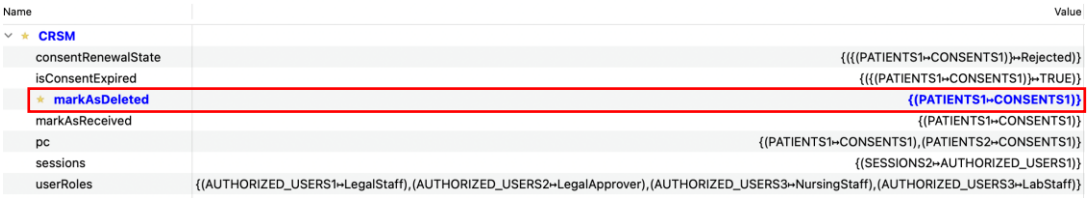



Figure 72: The latest value of the variable *markAsDeleted* corresponds to event execution in the CR5 test case.

## 6.2. Test Cases in CM for Distributed Systems in Data Sharing

### 6.2.1. Test Cases in the DSSM Model



The DSSM model describes the dynamic behavior of manipulating data subjects' consent and sharing personal data across multiple services through blockchain.

We then specify the test case objectives as follows:

- **DS1**: The model shall conduct consent and data subjects' consent.
- **DS2**: The model shall correctly manage the interaction between the requester and response services. As for the data transfer among services, it shall select data fields corresponding to consent configuration.
- **DS3**: For every step of the request-response services interaction, the model shall verify consent validity.
- **DS4**: The model shall manage one-time request per a patient's data.

- **DS5**: For every step of callback to request-response services, the model shall handle the blockchain oracle charge for API calls (i.e., an oraclize's fee) and the smart contract's insufficient balance.

Before running ProB, we first determined the constant *initialBalance* with 3 points representing the initial balance of the smart contract. Then, we assigned the *initialBalance* to the smart contract's address *this* as an ordered pair {(this ↦ 3)} in the variable *balanceOf,* indicating this smart contract's address has balance as 3 points.

#### 6.2.1.1. The DS1 Test Case

According to Figure 73(A), ConsentB has been added to the blockchain by ServiceB, and DataSubject1 provides permission to access personal data within ConsentB and its data fields. The relevant state variables which have been updated (Figure 74) are as follows: 1) the variable *consents* contains the collection of available consents updated with ConsentB, 2) the variable *dataFields* contains the collection of data fields under the specific consent updated with {(ConsentB ↦ {Name, BirthDate, BirthDefects})}, and 3) the variable *dataSubjectConsents* contains the valid data subject's consent within the specific responder service updated with {(ServiceA ↦ DataSubject1 ↦ ConsentB ↦ TRUE)}.

Hence, simulation results point out that the DSSM model covered the DS1 test case.



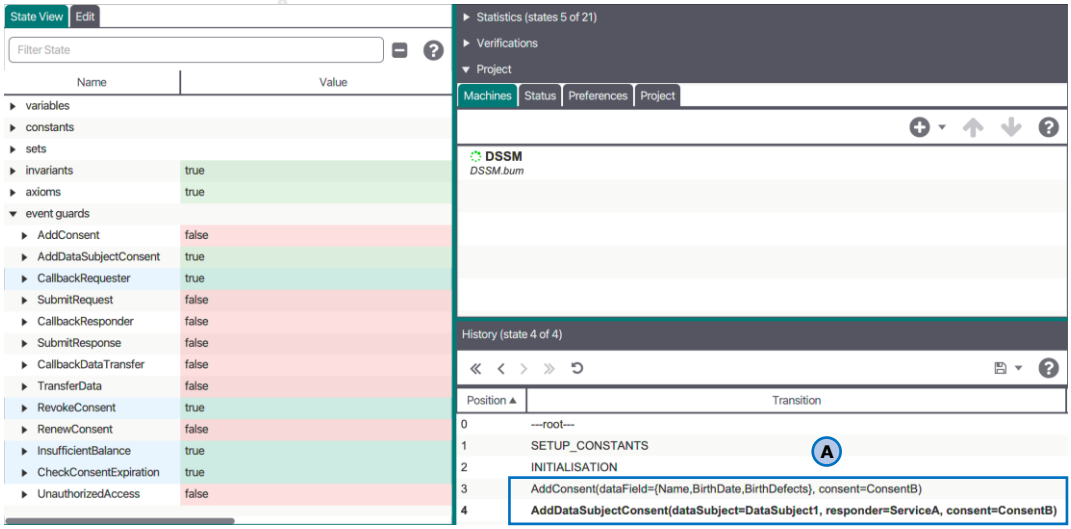



Figure 73: The simulation of the DS1 test case. (A) the AddConsent and AddDataSubject events and their variables are produced by ProB, which has been executed in the history panel.

| Name | Value |
|---|---|
| DSCX | |
| initialBalance | 3 |
| DSSM | |
| addresses | {this} |
| balanceOf | {(this↦3)} |
| callbackDataTransferStates | ∅ |
| callbackRequesterStates | ∅ |
| callbackResponderStates | ∅ |
| consents | {ConsentB} |
| dataAccessRequests | ∅ |
| dataAccessResponses | ∅ |
| dataFields | {(ConsentB↦{Name,BirthDate,BirthDefects})} |
| dataSubjectConsents | {(ServiceA↦DataSubject1↦ConsentB↦TRUE)} |
| dataTransferStates | ∅ |
| encryptedData | ∅ |

Figure 74: The latest values of *consents*, *dataFields*, and *dataSubjectConsents* variables correspond to event execution in the DS1 test case.

#### 6.2.1.2. The DS2 Test Case

Figure 75(A) demonstrates the request-response interaction between ServiceA and ServiceB on the blockchain. After ServiceA submits the data subject's consent, the blockchain then handles a callback URL to ServiceB. Besides, in every callback URL in the blockchain, the smart contract must pay a fee for the blockchain oracle to manage an API call with 1 point.

After ServiceB receives the API call, ServiceB submits the request back to the blockchain. The *balanceOf*, *callbackRequester*, and *dataAccessRequestes* variables have been updated with {(this ↦ 2)}, {(ServiceB ↦ DataSubject1 ↦ ConsentB)}, and {(Request1 ↦ (ServiceA ↦ DataSubject1 ↦ ConsentB))}, respectively (Figure 76).

The request submission of ServiceB triggers the blockchain to make the callback URL to ServiceA. Then, ServiceA submits the response back to the blockchain (Figure 75(B)). Then, the *balanceOf*, *callbackResponderStates*, and *dataAccessResponses* have been with {(this ↦ 1)}, Request1, and {(Response1 ↦ Request1)}, respectively (Figure 76).

After the blockchain receives the response from ServiceA, the blockchain makes the callback URL to ServiceA again to give the callback URL of ServiceB. Then, ServiceA encrypts the selected data fields based on the data subject's consent, and transfers encrypted personal data to ServiceB (Figure 75(C)). In doing so, the state variables *balanceOf*, *callbackDataTransferStates*, *dataTransferStates*, and *encryptedData* have been with {(this ↦ 0)}, Response1, {(Response1 ↦ TRUE)}, and {(Re-

sponse1 ↦ {(DataSubject1 ↦ Name), (DataSubject1 ↦ BirthDate), (DataSubject1 ↦ BirthDefects)})}, respectively (Figure 76).

Hence, simulation results point out that the DSSM model works correctly, and the change of state variables corresponds to the execution of the events, which covered the DS2 test case.

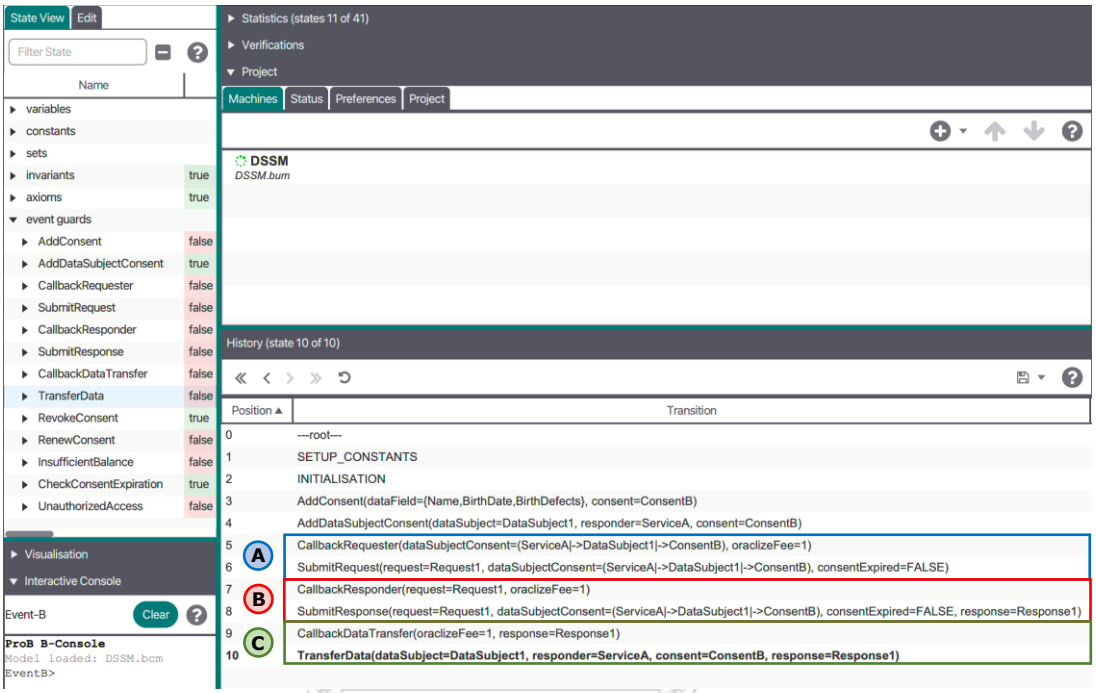



Figure 75: The simulation of the DS2 test case. (A) the CallbackRequester and SubmitRequest events, which have been executed in the history panel. (B) the CallbackResponder and SubmitResponse events, which have been executed in the history panel. (C) the CallbackDataTransfer and TransferData events, which have been executed in the history panel.



| Name | Value |
|---|---|
| DSCX | |
| initialBalance | 3 |
| DSSM | |
| addresses | {this} |
| balanceOf | {(this↦0)} |
| callbackDataTransferStates | {Response1} |
| callbackRequesterStates | {(ServiceA↦DataSubject1↦ConsentB)} |
| callbackResponderStates | {Request1} |
| consents | {ConsentB} |
| dataAccessRequests | {(Request1↦(ServiceA↦DataSubject1↦ConsentB))} |
| dataAccessResponses | {(Response1↦Request1)} |
| dataFields | {(ConsentB↦{Name,BirthDate,BirthDefects})} |
| dataSubjectConsents | {(ServiceA↦DataSubject1↦ConsentB↦TRUE)} |
| dataTransferStates | {(Response1↦TRUE)} |
| encryptedData | {(Response1↦{(DataSubject1↦Name),(DataSubject1↦BirthDate),(DataSubject1↦BirthDefects)})} |

Figure 76: The latest values of all state variables in the DSSM model correspond to event execution in the DS2 test case.

### 6.2.1.3. The DS3 Test Case

To verify the consent validation is working correctly, we then simulate the test case by firing the RevokeConsent event to make the consent invalid, before entering the following events: 1) the CallbackRequester event, 2) the SubmitRequest event, 3) the CallbackResponder event, 4) the SubmitResponse event, 5) the CallbackDataTransfer event, and 6) the TransferData event.

After firing the RevokeConsent event (Figure 77(A)), the guards of above events are invalid, as shown in Figure 77(B).

Hence, simulation results point out that the DSSM model covered the DS3 test case.

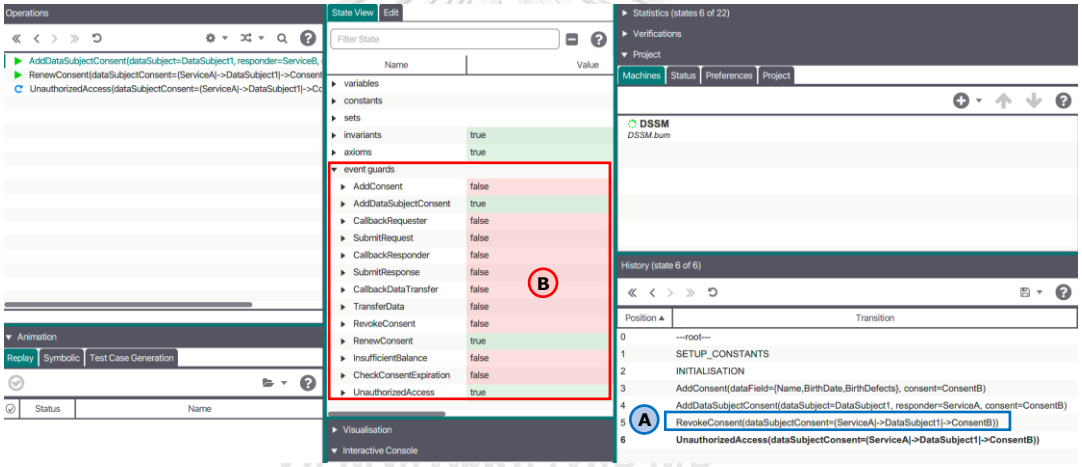



Figure 77: The simulation of the DS3 test case. (A) the RevokeConsent event and its variables produced by ProB, which have been executed in the history panel. (B) the list of unsatisfied and satisfied event guards corresponds to current state variables.

### 6.2.1.4. The DS4 Test Case

Firstly, we defined a fee for the blockchain oracle to manage an API call as 0 points, and the smart contract's balance currently remains at 3 points. Secondly, we simulated the different requests (e.g., Request1, Request2) for transferring personal data of the same data subject.

According to Figure 78, we executed events alternately between Request1 and Request2. The state variables during ProB simulation are correct, as shown in Figure 79.

The simulation results point out that the DSSM model covered the DS4 test case.

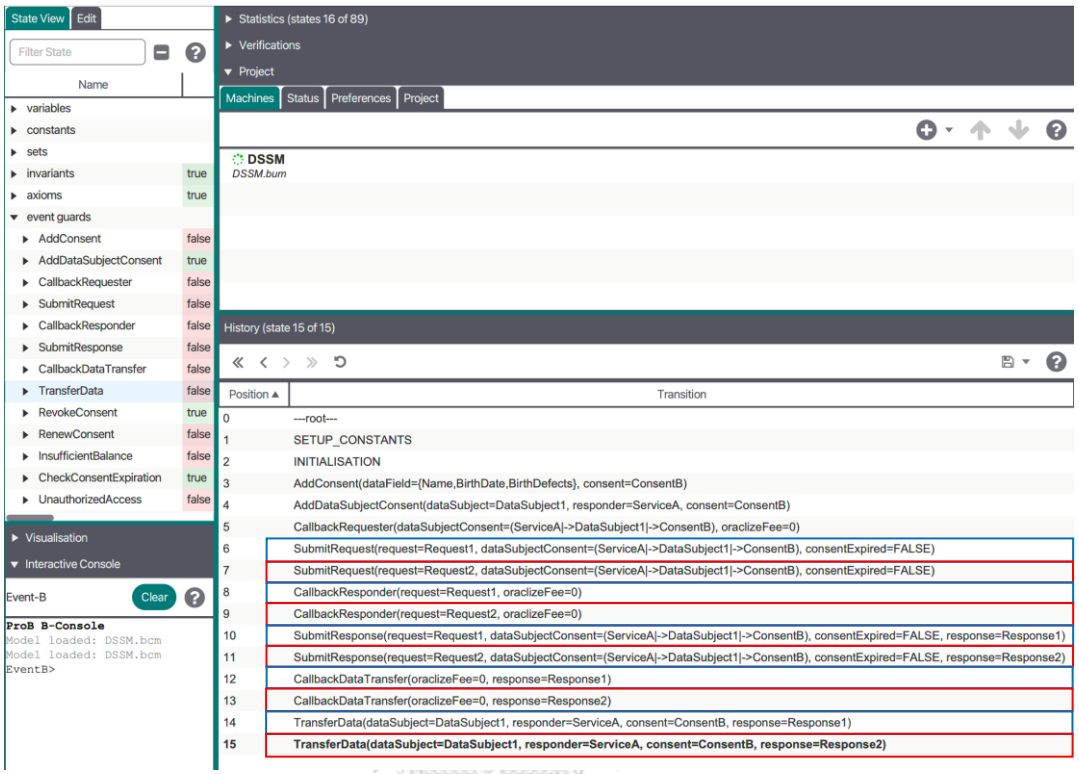



Figure 78: The simulation of the DS4 test case in the history panel.


Figure 79: The latest values of all state variables in the DSSM model correspond to event execution in the DS4 test case.

### 6.2.1.5. The DS5 Test Case

Firstly, we defined a fee for the blockchain oracle to manage an API call as 3 points, and the smart contract's balance remained 3 points.

The request-response interaction has begun after ServiceA submits the data subject's consent into the blockchain. After the CallbackRequester event firing, the smart contract's balance remains 0 points. ServiceB receives an API call and then submits the request to the blockchain, which

triggers the CallbackResponder event fires. It causes insufficient balance on the smart contract (Figure 80(A)).

Hence, simulation results point out that the DSSM model covered the DS5 test case.

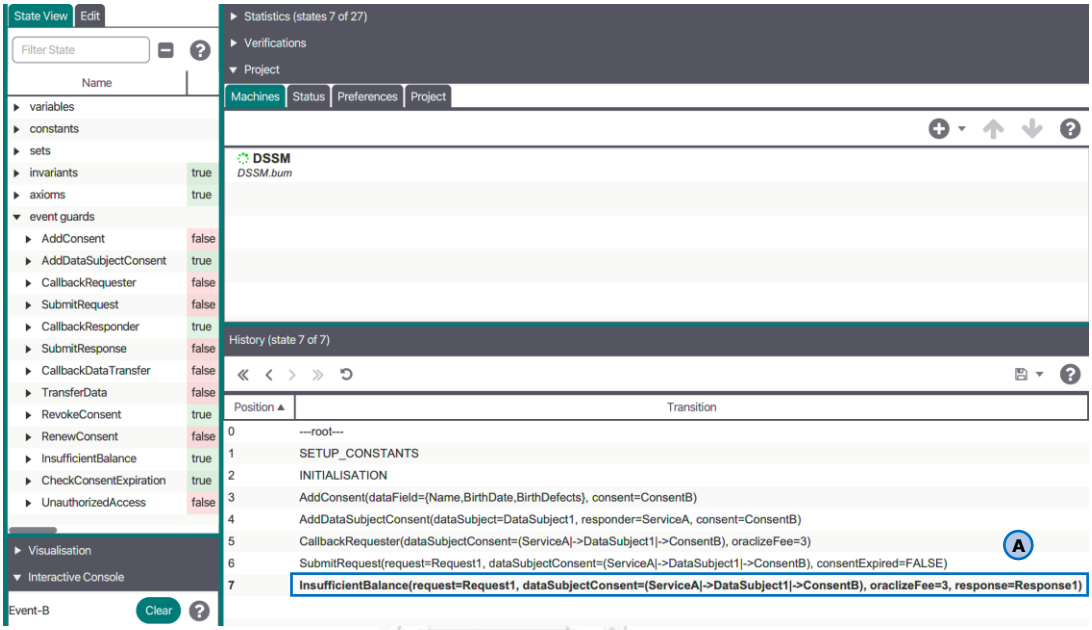



Figure 80: The simulation of the DS5 test case. (A) the InsufficientBalance event and its variables produced by ProB, which have been executed in the history panel.

Based on the above test cases, our proposed models covered five common functionalities outlined in the scope of work in CHAPTER I. Moreover, we constructed the mapping among competency questions and our study (Table 11), which comprises five state machines and covered the main aspects of consent management as follows: 1) Restricted Processing State Machine (RPSM), which explains the behavior of restriction for collecting and processing of individuals' data according to their given consent, 2) Withdrawal Approval State Machine (WASM), which explains the behavior of approval for revoking individuals' consent and removing their data, 3) Portable Approval State Machine (PASM), which explains the behavior of approval for requesting a portable copy of personal data, 4) Consent Renewal State Machine (CRSM), which explains the behavior of approval for renewing consent to extend the period of personal data usage, and 5) Data Sharing State Machine (DSSM), which explains the behavior of sharing personal data among requester and responder services through blockchain-based consent management, which allows automatic data sharing and open-access permanent audit logs.

Table 11: The mapping between competency questions for consent management and our study (cont'd).

| Formal model | | | | Class diagram | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **Event** | **Set** | **Local/state variable** | **Operation** | **Class** | **Attribute/Relation** | **GDPR article** |
| **Q1. Who is responsible for gathering consent agreements?** | | | | | | | |
| **RPSM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(7), Art. 6(1a), Art. 28 |
| **DSSM** | | PARTICIPANTS | responder | | DataSubjectConsent:struct | | |
| **Q2. For what purposes does a consent agreement cover?** | | | | | | | |
| **RPSM** | | CONSENTS | | | Consent | consentDetail, consentVersion, dataRetention | Art. 4(4). Art. 6(1a), Art. 7 |
| **DSSM** | AddConsent | CONSENTS | consents | addConsent | Consent:struct, ConsentContract | consentCode, consentVersion, dataRetention | |
| **Q3. How to revoke consent agreement?** | | | | | | | |
| **WASM** | revokeConsent | PATIENTS, CONSENTS, STATUSES | withdrawalState | revokeConsent | DataSubjectConsent | withdrawnFlag, withdrawnDate, | Art. 7 Rec. 63, Rec. 66 |
| **DSSM** | revokeConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | dataSubjectConsents | revokeConsent | DataSubjectConsent:struct, DataSubjectConsentContract | pseudonym, responderId, consentCode, consentVersion, withdrawnFlag, withdrawnTimestamp, dataSubjectConsentedActive | |
| **Q4. How long does a consent agreement last?** | | | | | | | |
| **RPSM** | | | consentExpired | expireConsent | Consent, DataSubjectConsent | dataRetention, createdDate | Art. 5(1) Rec. 32, Rec. 42 |
| **CRSM** | | PATIENTS, CONSENTS, BOOL | expired, isConsentExpired | expireConsent | Consent, DataSubjectConsent | dataRetention, createdDate | |

Table 11: The mapping between competency questions for consent management and our study (cont'd).

| Formal model | | | | Class diagram | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **Event** | **Set** | **Local/state variable** | **Operation** | **Class** | **Attribute/Relation** | **GDPR article** |
| **Q4. How long does a consent agreement last?** | | | | | | | |
| **DSSM** | | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | consentExpired, dataSubjectConsents | | DataSubjectConsent:struct, DataSubjectConsentContract | responderId, pseudonym, consentCode, consentVersion, dataRetention, createTimestamp, dataSubjectConsentedActive | Art. 5(1) Rec. 32, Rec. 42 |
| **Q5. When has consent been granted?** | | | | | | | |
| **RPSM** | AddConsent | PATIENTS, CONSENTS | pc | | Consent, DataSubjectConsent | consentDetail, consentVersion, dataRetention, acceptedFlag, createdDate | Art. 4(11), Art. 7, Art. 6(1a) |
| **DSSM** | AddDataSubjectConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | dataSubjectConsents | addDataSubjectConsent | Consent:struct, DataSubjectConsentContract | responderId, pseudonym, consentCode, consentVersion, dataRetention, acceptedFlag, createTimestamp | |
| **Q6. When has consent been withdrawn?** | | | | | | | |
| **WASM** | | BOOL | canWithdraw | canWithdraw | DataSubjectConsent | | Art. 17, Art. 19 |
| **Q7. When is consent permitted data to be portable?** | | | | | | | |
| **PASM** | | BOOL | canPortable | canPortable | DataSubjectConsent | | Art. 20 |
| **Q8. When has consent been renewed?** | | | | | | | |
| **CRSM** | | PATIENTS, CONSENTS, BOOL | expired, isConsentExpired | expireConsent | Consent, DataSubjectConsent | dataRetention, createdDate | Art. 4(11), Art. 7, Art. 6(1a) |

Table 11: The mapping between competency questions for consent management and our study (cont'd).

| Formal model | | | | Class diagam | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **Event** | **Set** | **Local/state variable** | **Operation** | **Class** | **Attribute/Relation** | **GDPR article** |
| **Q8. When has consent been renewed?** | | | | | | | |
| **DSSM** | RenewConsent | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL | consentExpired, dataSbujectConsents | isConsentValid | DataSubjectConsent:struct, DataSubjectConsentContract | dataRetention, createTimestamp, dataSubjectConsentedActive | Art. 4(11), Art. 7, Art. 6(1a) |
| **Q9. How has personal data been gathered?** | | | | | | | |
| **RPSM** | | FIELDS | | | DataField, DataSubject | fieldName, fieldType | Art. 4(1), Art. 9 |
| **DSSM** | | CONSENTS, FIELDS | dataFields | | DataField:struct, DataFieldContract | consentCode, consentVersion, fieldName | |
| **Q10. How has personal data been used?** | | | | | | | |
| **RPSM** | CheckAuthorizeConsent, CreateQuery, ExecuteQuery | PATIENTS, CONSENTS, ROLES, FIELDS, QUERIES | crf, queries, authorizedConsent, pf | checkAuthorizeConsent, createQuery, executeQuery | ConsentPolicyAccess, Query | | Art. 4(2) |
| **DSSM** | SubmitRequest, SubmitResponse | REQUESTS, PARTICIPANTS, DATA_SUBJECTS, RESPONSES | dataAccessRequests, dataAccessResponses | submitRequest, submitResponse | DataAccessRequest:struct, DataAccessRequestContract, DataAccessResponse:struct, DataAccessResponseContract | requestExists, requestId, pseudonym, consentCode, consentVersion, responseExists, responseId | |
| **Q11. How has personal data been gathered?** | | | | | | | |
| **RPSM** | AddPatient, AddConsent | PATIENTS, CONSENTS | pc, patients | | DataSubject, DataSubjectConsent, DataValue | | Art. 12, Art. 13, Art. 14, Rec. 39, Rec. 58, Rec. 62, Rec. 73 |
| **DSSM** | SubmitResponse | RESPONSES, REQUESTS | dataAccessResponses | submitResponse | DataAccessResponse:struct, DataAccessResponseContract | responseExists, responseId, pseudonym, consentCode, consentVersion | |

Table 11: The mapping between competency questions for consent management and our study (cont'd).

| Formal model | | | | Class diagram | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **Event** | **Set** | **Local/state variable** | **Operation** | **Class** | **Attribute/Relation** | **GDPR article** |
| **Q12. To whom personal data is disclosed?** | | | | | | | |
| **PASM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(7), Art. 6, Art. 28 |
| **DSSM** | | PARTICIPANTS | | | Consent:struct, DataSubjectConsent:struct | requesterId, requesterUrl, responderId, responderUrl | |
| **Q13. Who is in charge of personal data?** | | | | | | | |
| **PASM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 24, Rec. 74, Rec. 79 |
| **DSSM** | | PARTICIPANTS | | | Consent:struct, DataSubjectConsent:struct | requesterId, requesterUrl, responderId, responderUrl | |
| **Q14. How to minimize the data collection?** | | | | | | | |
| **RPSM** | | CONSENTS, ROLES, FIELDS | crf | | | | Art. 5(1c) |
| **DSSM** | | CONSENTS, FIELDS | dataFields | | DataField:struct, DataFieldContract | consentCode, consentVersion, fieldName | |
| **Q15. Where has personal data been obtained?** | | | | | | | |
| **RPSM** | | PATIENTS, CONSENTS | pc, patients | | DataSubject, DataSubjectConsent, DataValue | | Art. 5(1e) |
| **DSSM** | | RESPONSES, DATA_SUBJECTS, FIELDS, BOOL | encryptedData, dataTransferStates | | | | |
| **Q16. When should personal data be pseudonymized?** | | | | | | | |
| **DSSM** | | | | | DataSubjectConsent:struct | pseudonym | Art. 4(5), Rec. 26 |

Table 11: The mapping between competency questions for consent management and our study (cont'd).

| Formal model | | | | Class diagram | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **Event** | **Set** | **Local/state variable** | **Operation** | **Class** | **Attribute/Relation** | **GDPR article** |
| **Q17. Who has been identified as the data controller?** | | | | | | | |
| **RPSM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(7), Art. 28 |
| **DSSM** | | CONSENTS, FIELDS | dataFields | | DataField:struct, DataFieldContract | consentCode, consentVersion, fieldName | |
| **Q18. How to reach out to the data controller? (not applicable)** | | | | | | | |
| **Q19. What is the data controller in charge for?** | | | | | | | |
| **RPSM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(7), Art. 14, Art. 28, Art. 37 |
| **DSSM** | | CONSENTS, PARTICIPANTS | consent, responder | | Consent:struct, DataSubjectConsent:struct | requesterId, requesterUrl, responderId, responderUrl | |
| **Q20. How to embed data protection as a default setting for processing personal data?** | | | | | | | |
| **RPSM** | | CONSENTS, ROLES, FIELDS, PATIENTS, AUTHORIZED_USERS | pc, crf, userRoles | | ConsentPolicyAccess, ConsentRoleField, DataSubjcectConsent | userRoles | Art. 25 |
| **DSSM** | | CONSENTS, PARTICIPANTS | consent, responder | | Consent:struct, DataSubjectConsent:struct | requesterId, requesterUrl, responderId, responderUrl | |
| **Q21. Who has been identified as the data processor?** | | | | | | | |
| **RPSM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(8) |

Table 11: The mapping between competency questions for consent management and our study.

| Machine | Event | Set | Local/state variable | Operation | Class | Attribute/Relation | GDPR article |
|---|---|---|---|---|---|---|---|
| **Q21. Who has been identified as the data processor?** | | | | | | | |
| **DSSM** | | PARTICIPANTS, DATA_SUBJECTS, CONSENTS, BOOL, RESPONSES, REQUESTS | requestExists, requestId, pseudonym, consentCode, consentVersion, responseExists, responseId | isConsentValid, submitRequest, submitResponse | DataSubjectConsent:struct, DataSubjectConsentContract, DataAccessRequest:struct, DataAccessRequestContract, DataAccessResponse:struct, DataAccessResponseContract | consentExpired, dataSubjectConsents, dataAccessRequests, dataAccessResponses | Art. 4(8) |
| **Q22. Who has been identified as the data subject?** | | | | | | | |
| **RPSM** | | AUTHORIZED_USERS, ROLES | userRoles | | AuthorizedUser, Role | userRoles | Art. 4(1) |
| **DSSM** | | DATA_SUBJECTS | dataSubjects | | DataSubjectConsent:struct | | |
| **Q23. Whom to reach out to? (not applicable)** | | | | | | | |

# CHAPTER VII
# DISCUSSION AND CONCLUSION

## 7.1. Discussion

The objective of CM for centralized systems is to manage legal documents (i.e., consent) and data subjects' consent choices for collecting and processing personal data inside its system according to the role-based consent assignment, which consists of four state machines, including RPSM, WASM, PASM, and CRSM. The advantages of CM for centralized systems are that it provides great control of the personal data lifecycle and is easy to adopt into software systems. Moreover, the RPSM provides consent-based permission combined with RBAC to restrict stakeholders to process only specified data fields within the data subject's consent. Based on RBAC and a consent, all authorized users with the same roles can access data fields consented by the data subject. For example, all doctors can access a patient data even though that patient is not their case. We can adopt and formalize ABAC (Attribute-Based Access Control) to give more restrictions on data access in future work. As for the PASM, it only provides a portable approval workflow that permits data subjects to request a portable copy of their personal data. However, transferring personal data between organizations or services must proceed outside the system. To enable the automatic transferring of personal data across services, we then extended our research by designing CM for distributed systems in data sharing under the assumption that communications among systems are secured, which is described in the DSSM. Using blockchain technology in CM for distributed systems in data sharing helps enable secure, transparent, and traceable data sharing across services. The advantages of CM for distributed systems in data sharing are that it manages consent-authorized validation and request-response interaction among services as a middleware. Unfortunately, programs (i.e., smart contracts) that live on the blockchain are irreversible. Once they are deployed, it generates new addresses. With multiple times of deployments, it hardly maintains addresses and increases execution time. To bridge this gap, we designed reusable smart contracts which obtain only states of data subjects' consent and request-response interactions among services.

Choosing the right CMs for software systems depends on business objectives. For instance, the use of CM for centralized systems is proper for systems that have individuals' data to manage but do not provide disclosure of individuals' data automatically between organizations or services. In contrast, using CM for distributed systems in data sharing is proper only for systems that need to share individuals' data securely and enable irreversible audit trails among systems utilizing blockchain technology. By its nature, the blockchain's programs are not easy to alter once data has

persisted. Therefore, CM for distributed systems in data sharing shall use blockchain for keeping only the state of shared data subjects' data.

## 7.2. Conclusion

Privacy issues become a threat to individuals' lives. The GDPR then seeks to minimize the threat by outlining the data protection law to give individuals the power to control their personal data. According to the literature, the GDPR provisions are difficult to interpret and apply to software systems, leading to violating individuals' privacy. To bridge the gap, this thesis introduces CM for centralized systems and data sharing in distributed systems, which covers five common functionalities stated in the scope of work in CHAPTER I.

To begin with, CM in centralized systems handles the entire personal data lifecycle for a system with its own data subjects' data. On the other hand, CM in distributed systems is used to control the lifecycle of sharing personal data among multiple systems. The difference between these two approaches is that CM in centralized systems focuses on managing their data subjects' data based on role-based consent. In contrast, CM in distributed systems uses blockchain technology to enable open-access immutable audit logs and secure sharing of personal data among systems.

According to a modern software system, the system can conduct and disclose data subjects' data to other service providers, such as customer service management systems. To integrate data protection into the system, it simply adopts our proposed models and class diagrams as guidelines, which are proven correctness by the Event-B method.



As for further research, we will evaluate the operational performances of these two approaches against existing studies. Moreover, in the CM in distributed systems, we will assess data subjects' compensation costs for sharing their personal data to motivate their data contribution to healthcare research.

# APPENDIX A
# EVENT-B MODELS FOR CONSENT MANAGEMENT IN CENTRALIZED SYSTEMS

Event-B models were constructed based on four state machines: 1) RPSM, which covered conducting individuals' consent and limiting access to authorized personal data based on a given consent, 2) WASM, which provided a withdrawal approval process for allowing individuals to withdraw their consent at any time they wish to, 3) PASM, which provided a portable approval process for allowing individuals to request portable their personal data, and 4) CRSM, which provided a consent renewal process for enabling individuals to renew their consent for continued use of services and products offered by service providers. Besides, Event-B models are available for the public at https://github.com/cucpbioinfo/ConsentBasedPrivacy .

## 1. The RPSM Model

We modeled RPSM (Figure 81) to describe the dynamic behavior of how the system conducts data subjects' consent and how to restrict privileged permissions of stakeholders (e.g., doctors, nurses, researchers) for processing personal data within data subjects' consent. The RPSM model is divided into two parts, including the RPCX context, and the RPSM machine.

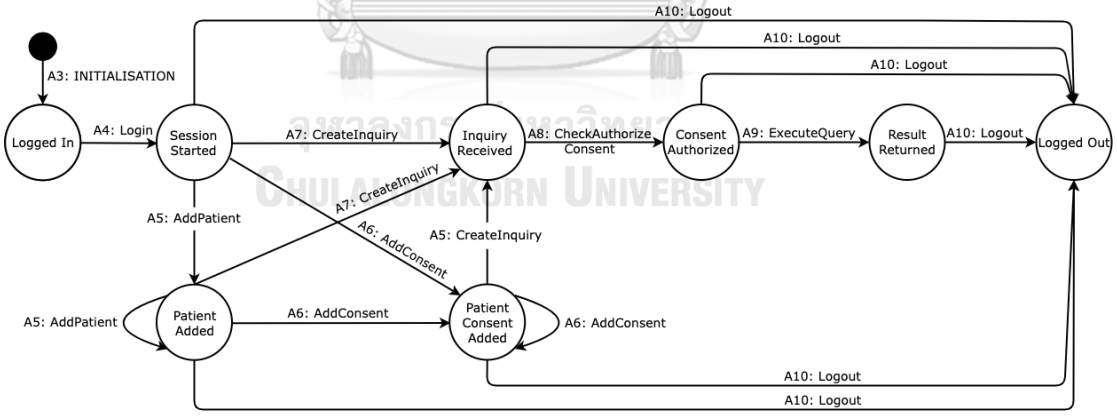



Figure 81: RPSM demonstrating how to restrict access to personal data according to data subjects' consent.

### 1.1. The RPCX Context

The RPCX context is the static part of the RPSM model containing the sets, constants, and axioms.

### 1.1.1. Sets in RPCX

Sets are a set of abstracts in the context of CM in health systems are comprises the following sets:

- PATIENTS is a set of individuals.
- CONSENTS is a set of consent agreements.
- FIELDS is a set of data fields that leads to specific personal characteristics.
- AUTHORIZED_USERS is a set of privileged users in the system.
- SESSIONS is a set of login sessions according to privileged users' requests to access the system.
- ROLES is a set of permissions that specify the users' area of responsibility and functionalities on the system.
- QUERIES is a set of queries to retrieve patients' information.

### 1.1.2. Constants in RPCX

Constants are elements of sets, which declare in the axiom section. There are two particular sets define in this section:

1. FIELDS contains the following constants: HN (i.e., hospital number), Name, Age, Weight, Height, Gender, and Race.
2. ROLES contains the following constants: NursingStaff, Oncologist, and LabStaff.



### 1.1.3. Axioms in RPCX

Axioms are used to determine known static relations written with predicate logic and assumed to be true. Moreover, they are also used to assign constants to pre-defined sets. According to Listing A1, the axm1 and axm2 are added to specify constants to pre-defined sets, e.g., ROLES and FIELDS, respectively. As for the four axioms (axm3 - axm6), they are added to deal with empty set assignments in variables restrained by partial functions, e.g., *sessions*, *queries*, *pf*, and *authorizedConsent*, respectively.

```
AXIOMS
  axm1 : partition(ROLES, {NursingStaff}, {Oncologist}, {LabStaff})
  axm2 : partition(FIELDS, {HN}, {Name}, {Age}, {Weight}, {Height},
```

```
        {Gender}, {Race})
axm3 : ø ∈ SESSIONS ⤖ AUTHORIZED_USERS
axm4 : ø ∈ AUTHORIZED_USERS ⇸ (QUERIES ⤖ PATIENTS)
axm5 : ø ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
axm6 : ø ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ CONSENTS)
```

Listing A1: The list of axioms in RPCX.

## 1.2. The RPSM Machine

The RPSM machine is the dynamic part of the RPSM model containing the invariants, variables, and events.

### 1.2.1. Invariants in RPSM

Invariants are constraints of state variables described by first-order logic expressions, as shown in Listing A2. In every event execution, actions change state variables' value, which must preserve all their invariants in the whole model.

```
INVARIANTS
  inv1 : sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
  inv2 : userRoles ∈ AUTHORIZED_USERS ↔ ROLES
  inv3 : pc ∈ PATIENTS ↔ CONSENTS
  inv4 : patients ∈ ℙ(PATIENTS)
  inv5 : crf ∈ CONSENTS ⇸ (ROLES ↔ FIELDS)
  inv6 : queries ∈ AUTHORIZED_USERS ⇸ (QUERIES ↔ PATIENTS)
  inv7 : pf ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
  inv8 : authorizedConsent ∈ AUTHORIZED_USERS ⇸
         (PATIENTS ↔ CONSENTS)
```

Listing A2: The list of invariants in RPSM.



The state variables are divided into eight variables:

- The variable *sessions* contains the one-to-one relationships between SESSIONS and AUTHORIZED_USERS.

  The example of the *sessions* value:

  ```
  {(SESSIONS1 ↦ AUTHORIZED_USER1)}
  ```

- The variable *userRoles* contains the relation between two given sets, e.g., AUTHORIZED_USER and ROLES for determining user activities and tasks based on user permissions that each system configures.

  The example of the *userRoles* value:

```
{(AUTHORIZED_USER1 ↦ NursingStaff),
 (AUTHORIZED_USER1 ↦ Oncologist),
 (AUTHORIZED_USER1 ↦ LabStaff),
 (AUTHORIZED_USER2 ↦ LabStaff)}
```

- The variable *patients* contains the PATIENTS set during the model refinement.

  The example of the *patients* value:

  ```
  {PATIENTS1}
  ```

- The variable *pc* contains the relation between two given sets, e.g., PATIENTS and CONSENTS, representing patients' consent agreements in which patients permit users who have been defined in consent agreements to process their personal data.

  The example of the *pc* value:

  ```
  {(PATIENTS1 ↦ CONSENTS1)}
  ```

- The variable *crf* contains the relation between three given sets, e.g., CONSENTS, ROLES, and FIELDS, representing consent-based permission in which only authorized users can access personal data according to a given consent.

  The example of the *crf* value:

  ```
  {(CONSENT1 ↦ {(NursingStaff ↦ HN)}),
   (CONSENT2 ↦ {(NursingStaff ↦ HN),
                (NursingStaff ↦ Name),
                (NursingStaff ↦ Age)})}
  ```



- The variable *queries* contains the relation between three given sets, e.g., AUTHORIZED_USERS, PATIENTS, and QUERIES, representing personal data queries.

  The example of the *queries* value:

  ```
  {(AUTHORIZED_USER1 ↦ {(QUERIES1 ↦ PATIENTS1)})}
  ```

- The variable *pf* contains the relation between three given sets, e.g., AUTHORIZED_USERS, PATIENTS, and FIELDS, representing query results. This variable holds query results of personal data in which selected only data fields that are apparent in the variable *crf*.

  The example of the *queries* value:

```
{(AUTHORIZED_USER1 ↦ {(PATIENTS1 ↦ HN)})}
```

- The variable *authorizedConsent* contains the relation between three given sets, e.g., AUTHORIZED_USERS, PATIENTS, and CONSENTS, representing consent validation results. This variable holds the result of consent validation which checks the validity before executing users' query to retrieve patients' data.

The example of the *authorizedConsent* value:

```
{(AUTHORIZED_USER1 ↦ {(PATIENTS1 ↦ CONSENTS1)})}
```

### 1.2.2. Events in RPSM

Events are the state transitions of the given model. In Event-B, the event will be executed when its guards meet conditions then state variables will be updated values.

The RPSM are partitioned into eight events:

#### 1.2.2.1. The INITIALISATION Event

This event is used to initiate all state variable values of the model. According to Listing A3, the six actions (act1 – act6) are assigned empty sets. As for act7 and act8, they are specified variables with first-order logic expressions using operation, called choice from set (i.e., :∈). In doing so, the *userRoles* and *crf* variables are automatically generated by the Rodin Platform.



```
INITIALISATION ≙
STATUS
  ordinary
BEGIN
  act1 : sessions ≔ ∅
  act2 : patients ≔ ∅
  act3 : pc ≔ ∅
  act4 : queries ≔ ∅
  act5 : pf ≔ ∅
  act6 : authorizedConsent ≔ ∅
  act7 : userRoles :∈ AUTHORIZED_USERS ↔ ROLES
  act8 : crf :∈ CONSENTS ⟶ ℙ1(ROLES × FIELDS)
END
```

Listing A3: The INITIALISATION event.

#### 1.2.2.2. The Login Event

This event describes the behavior of login (Listing A3). The event will be executed when the current user session does not exist, and this user is registered, then the user successfully login to the system.

```
Login ≙
STATUS
  ordinary
ANY
  s, u
WHERE
  grd1 : s ∈ SESSIONS ∧ s ∉ dom(sessions)
  grd2 : u ∈ AUTHORIZED_USERS ∧ s ∉ ran(sessions)
  grd3 : sessions ∪ {s ↦ u} ∈ SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ∪ {s ↦ u}
END
```

Listing A4: The Login event.

#### 1.2.2.3. The AddPatient Event

The event describes the behavior of creating a patient (Listing A5). The event will be executed when the authorized user has logged on with the nursing staff role, and this patient does not register to the system before, then the user adds the patient information successfully.



```
AddPatient ≙
STATUS
  ordinary
ANY
  s, p
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = NursingStaff
  grd3 : p ∈ PATIENTS ∧ p ∉ patients
  grd4 : sessions(s) ∉ dom(queries)
THEN
    act1 : patients ≔ patients ∪ {p}
END
```

Listing A5: The AddPatient event.

#### 1.2.2.4. The AddConsent Event

The event describes the behavior of adding consent (Listing A6). The event will be executed when the authorized user has logged on with the nursing staff role, and this patient's consent is not added to the system before, then the user adds the patient's consent successfully.

```
AddConsent ≙
STATUS
  ordinary
ANY
  s, p, c
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = NursingStaff
  grd3 : p ∈ patients ∧ c ∈ dom(crf)
  grd4 : p ↦ c ∉ pc
  grd5 : pc ∪ {p ↦ c} ∈ PATIENTS ↔ CONSENTS
  grd6 : sessions(s) ∉ dom(queries)
THEN
  act1 : pc ≔ pc ∪ {p ↦ c}
END
```

Listing A6: The AddConsent event.

#### 1.2.2.5. The CreateInquiry Event

This event describes the behavior of creating (Listing A7). The event will be executed when the authorized user has logged on, and this user wishes to retrieve a patient's information who has given their consent, then the user creates an inquiry successfully.

```
CreateInquiry ≙
STATUS
  ordinary
ANY
  s, p, q
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : q ∈ QUERIES ∧ p ∈ dom(pc) ∧ sessions(s) ∉ dom(queries)
  grd3 : queries ⩤ {sessions(s) ↦ {q ↦ p}} ∈
         AUTHORIZED_USERS ⇸ (QUERIES ↔ PATIENTS)
THEN
  act1 : queries(sessions(s)) ≔ {q ↦ p}
END
```

Listing A7: The CreateInquiry event.

#### 1.2.2.6. The CheckAuthorizeConsent Event

This event describes the behavior of checking authorized consent (Listing A8). The event will be executed when the created query is passed on the following conditions: 1) the patient's consent does not expire, and 2) the authorized user who created the query has consent permission to access the information of this patient. Then, the system captures the consent validation result.

```
CheckAuthorizeConsent ≙
STATUS
  ordinary
ANY
  s, p, c, consentExpired
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(queries)
  grd2 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
  grd3 : c ∈ pc[{p}] ∧ c ∈ dom(crf)
  grd4 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r ∈ dom(crf(c))
  grd5 : sessions(s) ∉ dom(authorizedConsent)
  grd6 : authorizedConsent ⩤ {sessions(s) ↦ {p ↦ c}} ∈
         AUTHORIZED_USERS ⇸ (PATIENTS ↔ CONSENTS)
THEN
  act1 : authorizedConsent(sessions(s)) ≔ {p ↦ c}
END
```

Listing A8: The CheckAuthorizeConsent event.

#### 1.2.2.7. The ExecuteQuery Event

This event describes the behavior of executing query (Listing A9). The event will be executed when the authorized consent has been verified, then the system returns the patient's data fields to the user.

```
ExecuteQuery ≙
STATUS
  ordinary
ANY
  s, p, c
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(queries)
  grd2 : p ∈ ran(queries(sessions(s))) ∧ c ∈ dom(crf)
  grd3 : sessions(s) ∈ dom(authorizedConsent) ∧ p ↦ c ∈
         authorizedConsent(sessions(s))
  grd4 : sessions(s) ∉ dom(pf)
  grd5 : pf ⩤ {sessions(s) ↦ {p} × ran(userRoles[sessions[{s}]] ◁
```

```
        crf(c))} ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
THEN
  act1 : pf(sessions(s)) ≔ {p} × ran(userRoles[sessions[{s}]] ◁
        crf(c))
END
```

Listing A9: The ExecuteQuery event.

#### 1.2.2.8. The Logout Event

This event describes the behavior of logout (Listing A10). The event will be executed when the current user session exists, then the system removes state variables values within the current user, including the *pf*, *queries*, *authorizedConsent*, and *sessions* variables.

```
Logout ≙
STATUS
  ordinary
ANY
  s
WHERE
  grd1 : s ∈ dom(sessions)
  grd2 : {sessions(s)} ◁ queries ∈ AUTHORIZED_USERS ⇸
        (QUERIES ↔ PATIENTS)
  grd3 : {sessions(s)} ◁ authorizedConsent ∈ AUTHORIZED_USERS ⇸
        (PATIENTS ↔ CONSENTS)
  grd4 : {sessions(s)} ◁ pf ∈ AUTHORIZED_USERS ⇸ (PATIENTS ↔ FIELDS)
  grd5 : sessions ⩥ {sessions(s)} ∈ SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : queries ≔ {sessions(s)} ◁ queries
  act2 : authorizedConsent ≔ {sessions(s)} ◁ authorizedConsent
  act3 : pf ≔ {sessions(s)} ◁ pf
  act4 : sessions ≔ sessions ⩥ {sessions(s)}
END
```

Listing A10: The Logout event.

### 2. The WASM Model

We modeled WASM (Figure 82) to describe the dynamic behavior of how the system manages the withdrawal approval process when patients request to withdraw their consent. The WASM model is divided into two parts, including the WACX context and the WASM machine.

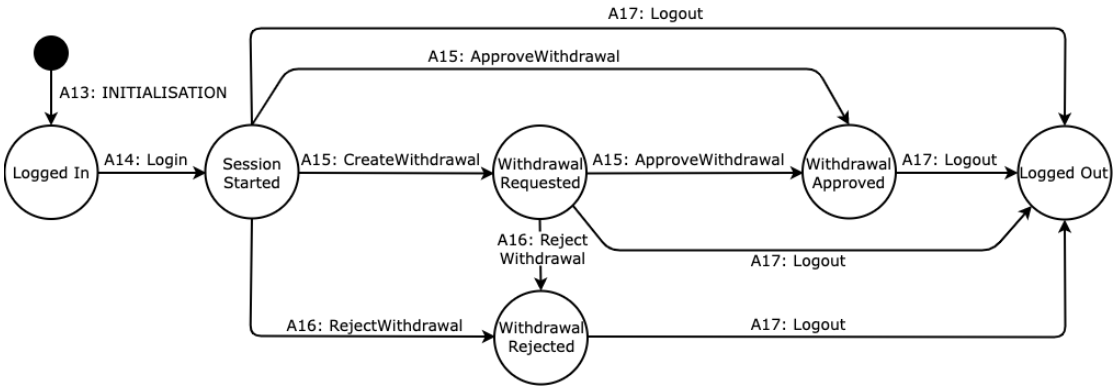



Figure 82: WASM demonstrating how to conduct the withdrawal approval process.

### 2.1. The WACX Context

The WACX context is the static part of the WASM model containing the sets, constants, and axioms.

#### 2.1.1. Sets in WACX

Sets are a set of abstracts in the context of CM in health systems are comprises the following sets:

- PATIENTS is a set of individuals.
- CONSENTS is a set of consent agreements.
- AUTHORIZED_USERS is a set of privileged users in the system.
- SESSIONS is a set of login sessions according to privileged users' requests to access the system.
- ROLES is a set of permissions that specify the users' area of responsibility and functionalities on the system.
- STATUSES is a set of withdrawal statuses.

#### 2.1.2. Constants in WACX

Constants are elements of sets, which declare in the axiom section. There are two particular sets define in this section:

1. ROLES is obtained with the following constants: LegalStaff, and LegalApprover.

2. STATUES is obtained with the following constants: Void, Approved, and Rejected.

### 2.1.3. Axioms in WACX

Axioms are used to determine known static relations written with predicate logic and assumed to be true. Moreover, they are also used to assign constants to pre-defined sets. According to Listing A11, the axm1 and axm2 are added to specify constants to pre-defined sets, e.g., ROLES, and STATUSES, respectively. As for the axm3 and axm4, they are added to deal with empty set assignments in variables restrained by partial functions, e.g., *sessions*, and *withdrawalState*, respectively.

```
AXIOMS
  axm1 : partition(ROLES, {LegalStaff}, {LegalApprover})
  axm2 : partition(STATUSES, {Void}, {Approved}, {Rejected})
  axm3 : ø ∈ SESSIONS ⤖ AUTHORIZED_USERS
  axm4 : ø ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

Listing A11: The list of axioms in WACX.

## 2.2. The WASM Machine

The WASM machine is the dynamic part of the WASM model containing the invariants, variables, and events.

### 2.2.1. Invariants in WASM

Invariants are constraints of state variables described by first-order logic expressions, as shown in Listing A12. In every event execution, actions change state variables' value, which must preserve all their invariants in the whole model.



```
INVARIANTS
  inv1 : sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
  inv2 : userRoles ∈ AUTHORIZED_USERS ↔ ROLES
  inv3 : pc ∈ PATIENTS ↔ CONSENTS
  inv4 : withdrawalState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  inv5 : markAsDeleted ∈ PATIENTS ↔ CONSENTS
```

Listing A12: The list of invariants in WASM.

The state variables are divided into five variables:

- The variable *sessions* contains the one-to-one relationships between SESSIONS and AUTHORIZED_USERS.

The example of the *sessions* value:

```
{(SESSIONS1 ↦ AUTHORIZED_USER2),
 (SESSIONS2 ↦ AUTHORIZED_USER1)}
```

- The variable *userRoles* contains the relation between two given sets, e.g., AUTHORIZED_USER, and ROLES for determining user activities and tasks based on user permissions that each system configures.

  The example of the *userRoles* value:

```
{(AUTHORIZED_USER1 ↦ LegalStaff),
 (AUTHORIZED_USER1 ↦ LegalApprover),
 (AUTHORIZED_USER2 ↦ LegalStaff)}
```

- The variable pc contains the relation between two given sets, e.g., PATIENTS and CONSENTS, representing patients' consent agreements in which patients permit users who have been defined in consent agreements to process their personal data.

  The example of the *pc* value:

```
{(PATIENTS1 ↦ CONSENTS1),
 (PATIENTS1 ↦ CONSENTS2),
 (PATIENTS2 ↦ CONSENTS1),
 (PATIENTS2 ↦ CONSENTS2)}
```

- The variable *withdrawalState* contains the relation between three given sets, e.g., PATIENTS, CONSENTS, and STATUSES, representing withdrawal requests.

  The example of the *withdrawalState* value:

```
{({(PATIENTS1 ↦ CONSENTS1)} ↦ Void),
 ({(PATIENTS1 ↦ CONSENTS2)} ↦ Approved),
 ({(PATIENTS2 ↦ CONSENTS2)} ↦ Rejected)}
```

- The variable *markAsDeleted* contains the relation between two given sets, e.g., PATIENTS, and CONSENTS, representing patient data has been deleted.

  The example of the *markAsDeleted* value:

```
{(PATIENTS1 ↦ CONSENTS2)}
```

### 2.2.2. Events in WASM

Events are the state transitions of the given model. In Event-B, the event will be executed when its guards meet conditions then state variables will be updated values.

The WASM are partitioned into six events:

#### 2.2.2.1. The INITIALISATION Event

This event is used to initiate all state variable values of the model. According to Listing A13, the three actions (act1 – act3) are assigned empty sets. As for act4 and act5, they are specified variables with first-order logic expressions using operation, called choice from set (i.e., :∈). In doing so, the *userRoles* and *pc* variables are automatically generated by the Rodin Platform.

```
INITIALISATION ≙
STATUS
  ordinary
BEGIN
  act1 : sessions ≔ ∅
  act2 : withdrawalState ≔ ∅
  act3 : markAsDeleted ≔ ∅
  act4 : userRoles :∈ AUTHORIZED_USERS ↔ ROLES
  act5 : pc :∈ ℙ1(PATIENTS × CONSENTS)
END
```

Listing A13: The INITIALISATION event.

#### 2.2.2.2. The Login Event

This event describes the behavior of login (Listing A14). The event will be executed when the current user session does not exist, and this user is registered, then the user successfully login to the system.



```
Login ≙
STATUS
  ordinary
ANY
  s, u
WHERE
  grd1 : s ∈ SESSIONS ∧ s ∉ dom(sessions)
  grd2 : u ∈ AUTHORIZED_USERS ∧ s ∉ ran(sessions)
  grd3 : sessions ∪ {s ↦ u} ∈ SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ∪ {s ↦ u}
END
```

Listing A14: The Login event.

#### 2.2.2.3. The CreateWithdrawal Event

The event describes the behavior of creating a withdrawal request (Listing A15). The event will be executed when the authorized user has logged on with the legal staff role, and this patient does not request to withdraw consent before, then the user creates the withdrawal request successfully.

```
CreateWithdrawal ≙
STATUS
  ordinary
ANY
  s, p, c
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧
         {p ↦ c} ∉ dom(withdrawalState)
  grd4 : withdrawalState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
THEN
  act1 : withdrawalState({p ↦ c}) ≔ Void
END
```

Listing A15: The CreateWithdrawal event.

#### 2.2.2.4. The ApproveWithdrawal Event

The event describes the behavior of approving a withdrawal request (Listing A16). The event will be executed when the authorized user has logged on with the legal approver role, the withdrawal request has the current status as Void, and there is no conflict exists the consent agreement, then the user approves the request successfully.

```
ApproveWithdrawal ≙
STATUS
  ordinary
ANY
  s, pc1, canWithdraw
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(withdrawalState) ∧
         withdrawalState(pc1) = Void
  grd4 : withdrawalState ⩤ {pc1 ↦ Approved} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canWithdraw ∈ BOOL ∧ canWithdraw = TRUE
```

```
  grd6 : markAsDeleted ⩤ pc1 ∈ PATIENTS ↔ CONSENTS
THEN
  act1 : withdrawalState(pc1) ≔ Approved
  act2 : markAsDeleted ≔ markAsDeleted ⩤ pc1
END
```

Listing A16: The ApproveWithdrawal event

#### 2.2.2.5. The RejectWithdrawal event

The event describes the behavior of rejecting a withdrawal request (Listing A17). The event will be executed when the authorized user has logged on with the legal approver role, the withdrawal request has the current status as Void, and there is conflict exists the consent agreement, then the user rejects the request successfully.

```
RejectWithdrawal ≙
STATUS
  ordinary
ANY
  s, pc1, canWithdraw
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(withdrawalState) ∧
         withdrawalState(pc1) = Void
  grd4 : withdrawalState ⩤ {pc1 ↦ Rejected} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canWithdraw ∈ BOOL ∧ canWithdraw = FALSE
THEN
  act1 : withdrawalState(pc1) ≔ Rejected
END
```

Listing A17: The RejectWithdrawal event.

#### 2.2.2.6. The Logout event

This event describes the behavior of logout (Listing A18). The event will be executed when the current user session exists, then the system removes the variable *sessions* values within the current user.

```
Logout ≙
STATUS
  ordinary
ANY
  s
```

```
WHERE
  grd1 : s ∈ dom(sessions)
  grd2 : sessions ⩥ {sessions(s)} ∈
         SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ⩥ {sessions(s)}
END
```

Listing A18: The Logout event.

## 3. The PASM Model

We modeled PASM (Figure 83) to describe the dynamic behavior of how the system manages the portable approval process when patients request portable their personal data. The PASM model is divided into two parts, including the PACX context and the PASM machine.

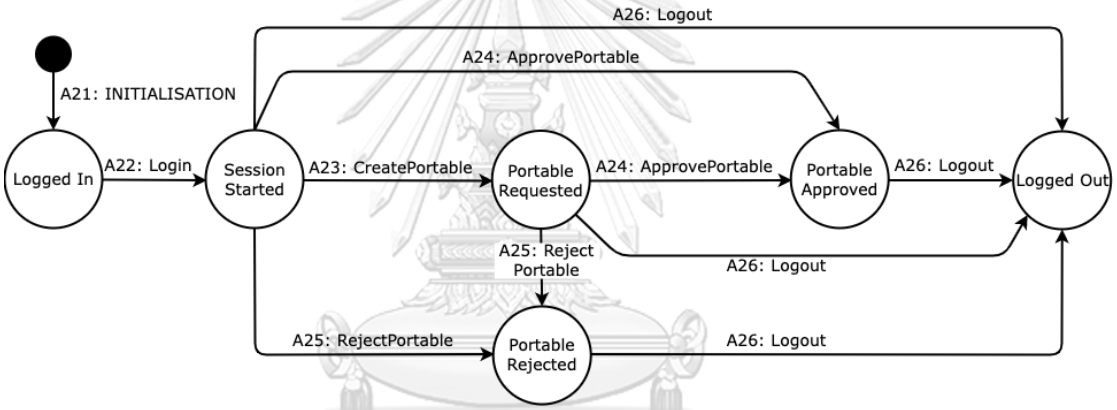



Figure 83: PASM demonstrating how to conduct the portable approval process.



### 3.1. The PACX Context

The PACX context is the static part of the PASM model containing the sets, constants, and axioms.

#### 3.1.1. Sets in PACX

Sets are a set of abstracts in the context of CM in health systems are comprises the following sets:

- PATIENTS is a set of individuals.
- CONSENTS is a set of consent agreements.
- AUTHORIZED_USERS is a set of privileged users in the system.

- SESSIONS is a set of login sessions according to privileged users' requests to access the system.
- ROLES is a set of permissions that specify the users' area of responsibility and functionalities on the system.
- STATUSES is a set of portable statuses.

### 3.1.2. Constants in PACX

Constants are elements of sets, which declare in the axiom section. There are two particular sets define in the section:

1. ROLES contains the following constants: LegalStaff, and LegalApprover.
2. STATUES contains the following constants: Void, Approved, and Rejected.

### 3.1.3. Axioms in PACX

Axioms are used to determine known static relations written with predicate logic and assumed to be true. Moreover, they are also used to assign constants to pre-defined sets. According to Listing A19, the axm1 and axm2 are added to specify constants to pre-defined sets, e.g., ROLES, and STATUSES, respectively. As for the axm3 and axm4, they are added to deal with empty set assignments in variables restrained by partial functions, e.g., *sessions*, and *portableState*, respectively.

```
AXIOMS
  axm1 : partition(ROLES, {LegalStaff}, {LegalApprover})
  axm2 : partition(STATUSES, {Void}, {Approved}, {Rejected})
  axm3 : ∅ ∈ SESSIONS ⤖ AUTHORIZED_USERS
  axm4 : ∅ ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

Listing A19: The list of axioms in PACX.

## 3.2. The PASM Machine

The PASM machine is the dynamic part of PASM model containing the invariants, variables, and events.

### 3.2.1. Invariants in PASM

Invariants are constraints of state variables described by first-order logic expressions, as shown in Listing A20. In every event execution, ac-

tions change state variables' value, which must preserve all their invariants in the whole model.

```
INVARIANTS
  inv1 : sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
  inv2 : userRoles ∈ AUTHORIZED_USERS ↔ ROLES
  inv3 : pc ∈ PATIENTS ↔ CONSENTS
  inv4 : portableState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

Listing A20: The list of invariants in PASM.

The state variables are divided into four variables:

- The variable *sessions* contains the one-to-one relationships between SESSIONS and AUTHORIZED_USERS.

  The example of the *sessions* value:

  ```
  {(SESSIONS1 ↦ AUTHORIZED_USER2),
   (SESSIONS2 ↦ AUTHORIZED_USER1)}
  ```

- The variable *userRoles* contains the relation between two given sets, e.g., AUTHORIZED_USER and ROLES for determining user activities and tasks based on user permissions that each system configures.

  The example of the *userRoles* value:

  ```
  {(AUTHORIZED_USER1 ↦ LegalStaff),
   (AUTHORIZED_USER1 ↦ LegalApprover),
   (AUTHORIZED_USER2 ↦ LegalStaff)}
  ```

- The variable *pc* contains the relation between two given sets, e.g., PATIENTS and CONSENTS, representing patients' consent agreements in which patients permit users who have been defined in consent agreements to process their personal data.

  The example of the *pc* value:

  ```
  {(PATIENTS1 ↦ CONSENTS1),
   (PATIENTS1 ↦ CONSENTS2)}
  ```

- The variable *portableState* contains the relation between two given sets, e.g., PATIENTS, CONSENTS, and STATUSES, representing portable requests.

  The example of the *portableState* value:

  ```
  {({(PATIENTS1 ↦ CONSENTS1)} ↦ Approved),
   ({(PATIENTS1 ↦ CONSENTS2)} ↦ Rejected)}
  ```

### 3.2.2. Events in PASM

Events are the state transitions of the given model. In Event-B, the event will be executed when its guards meet conditions then state variables will be updated values.

The PASM are partitioned into six events:

#### 3.2.2.1. The INITIALISATION Event

This event is used to initiate all state variable values of the model. According to Listing A21, the act1 and act2 actions are assigned empty sets. As for act3 and act4, they are specified variables with first-order logic expressions using operation, called choice from set (i.e., :∈). In doing so, the *userRoles* and *pc* variables are automatically generated by the Rodin Platform.

```
INITIALISATION ≙
STATUS
  ordinary
BEGIN
  act1 : sessions ≔ ∅
  act2 : portableState ≔ ∅
  act3 : userRoles :∈ AUTHORIZED_USERS ↔ ROLES
  act4 : pc :∈ ℙ1(PATIENTS × CONSENTS)
END
```

Listing A21: The INITIALISATION event.



#### 3.2.2.2. The Login Event

This event describes the behavior of login (Listing A22). The event will be executed when the current user session does not exist, and this user is registered, then the user successfully login to the system.

```
Login ≙
STATUS
  ordinary
ANY
  s, u
WHERE
  grd1 : s ∈ SESSIONS ∧ s ∉ dom(sessions)
  grd2 : u ∈ AUTHORIZED_USERS ∧ s ∉ ran(sessions)
  grd3 : sessions ∪ {s ↦ u} ∈ SESSIONS ⤖ AUTHORIZED_USERS
THEN
```

```
  act1 : sessions ≔ sessions ∪ {s ↦ u}
END
```

Listing A22: The Login event.

### 3.2.2.3. The CreatePortable Event

The event describes the behavior of creating a portable request (Listing A23). The event will be executed when the authorized user has logged on with the legal staff role, and this patient does not request portable personal data before, then the user creates the portable request successfully.

```
CreatePortable ≙
STATUS
  ordinary
ANY
  s, p, c
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧ {p ↦ c} ∉
         dom(portableState)
  grd4 : portableState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
THEN
  act1 : portableState({p ↦ c}) ≔ Void
END
```

Listing A23: The CreatePortable event.



### 3.2.2.4. The ApprovePortable Event

The event describes the behavior of approving a portable request (Listing A24). The event will be executed when the authorized user has logged on with the legal approver role, the portable request has the current status as Void, and the patient accept the prerequisite conditions (e.g., fee for data transferring), then the user approves the request successfully.

```
ApproveWithdrawal ≙
STATUS
  ordinary
ANY
  s, pc1, canPortable
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
```

```
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(portableState) ∧ portableState(pc1) = Void
  grd4 : portableState ⩤ {pc1 ↦ Approved} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canPortable ∈ BOOL ∧ canPortable = TRUE
THEN
  act1 : portableState(pc1) ≔ Approved
END
```

Listing A24: The ApprovePortable event.

#### 3.2.2.5. The RejectPortable Event

The event describes the behavior of rejecting a portable request (Listing A25). The event will be executed when the authorized user has logged on with the legal approver role, the portable request has the current status as Void, and there is conflict exists the consent agreement, then the user rejects the request successfully.

```
RejectPortable ≙
STATUS
  ordinary
ANY
  s, pc1, canPortable
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalApprover
  grd3 : pc1 ∈ dom(portableState) ∧ portableState(pc1) = Void
  grd4 : portableState ⩤ {pc1 ↦ Rejected} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd5 : canPortable ∈ BOOL ∧ canPortable = FALSE
THEN
  act1 : portableState(pc1) ≔ Rejected
END
```

Listing A25: The RejectPortable event.

#### 3.2.2.6. The Logout Event

This event describes the behavior of logout (Listing A26). The event will be executed when the current user session exists, then the system removes the variable *sessions* values within the current user.

```
Logout ≙
STATUS
  ordinary
ANY
  s
```

```
WHERE
  grd1 : s ∈ dom(sessions)
  grd2 : sessions ⩥ {sessions(s)} ∈
         SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ⩥ {sessions(s)}
END
```

Listing A26: The Logout event.

## 4. The CRSM Model

We modeled CRSM (Figure 84) to describe the dynamic behavior of how the system manages the consent renewal process when patients' consent expires. The CRSM model is divided into two parts, including the CRCX context and the CRSM machine.

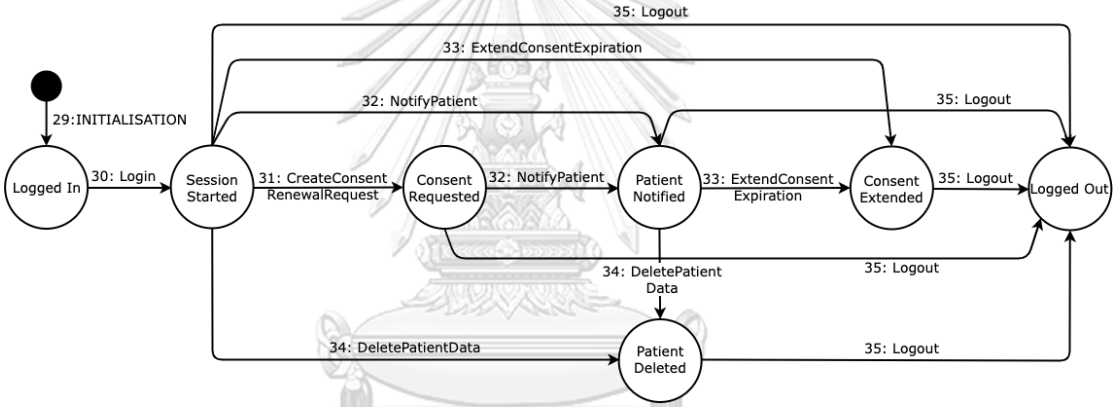



Figure 84: CRSM demonstrating how to conduct the consent renewal process.



### 4.1. The CRCX Context

The CRCX context is the static part of the CRSM model containing the sets, constants, and axioms.

#### 4.1.1. Sets in CRCX

Sets are a set of abstracts in the context of CM in health systems are comprises the following sets:

- PATIENTS is a set of individuals.
- CONSENTS is a set of consent agreements.
- AUTHORIZED_USERS is a set of privileged users in the system.

- SESSIONS is a set of login sessions according to privileged users' requests to access the system.
- ROLES is a set of permissions that specify the users' area of responsibility and functionalities on the system.
- STATUSES is a set of portable statuses.

### 4.1.2. Constants in CRCX

Constants are elements of sets, which declare in the axiom section. There are two particular sets define in this section:

1. ROLES contains the following constants: LegalStaff, and LegalApprover.
2. STATUES is obtained with the following constants: Void, Approved, and Rejected.

### 4.1.3. Axioms in CRCX

Axioms are used to determine known static relations written with predicate logic and assumed to be true. Moreover, they are also used to assign constants to pre-defined sets. According to Listing A27, the axm1 and axm2 are added to specify constants to pre-defined sets, e.g., ROLES, and STATUSES, respectively. As for the three axioms (axm3 - axm5), they are added to deal with empty set assignments in variables restrained by partial functions, e.g., *sessions*, *isConsentExpired*, and *consentRenewalState*, respectively.



```
AXIOMS
  axm1 : partition(ROLES, {LegalStaff}, {LegalApprover})
  axm2 : partition(STATUSES, {Void}, {Approved}, {Rejected})
  axm3 : ø ∈ SESSIONS ⤖ AUTHORIZED_USERS
  axm4 : ø ∈ (PATIENTS ↔ CONSENTS) ⤖ BOOL
  axm5 : ø ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
```

Listing A27: The list of axioms in CRCX.

## 4.2. The CRSM machine

The CRSM machine is the dynamic part of the CRSM model containing the invariants, variables, and events.

### 4.2.1. Invariants in CRSM

Invariants are constraints of state variables described by first-order logic expressions, as shown in Listing A28. In every event execution, actions change state variables' value, which must preserve all their invariants in the whole model.

```
INVARIANTS
  inv1 : sessions ∈ SESSIONS ⤖ AUTHORIZED_USERS
  inv2 : userRoles ∈ AUTHORIZED_USERS ↔ ROLES
  inv3 : pc ∈ PATIENTS ↔ CONSENTS
  inv4 : isConsentExpired ∈ (PATIENTS ↔ CONSENTS) ⤖ BOOL
  inv5 : consentRenewalState ∈ (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  inv6 : markAsDeleted ∈ PATIENTS ↔ CONSENTS
  inv7 : markAsReceived ∈ PATIENTS ↔ CONSENTS
```

Listing A28: The list of invariants in CRSM.

The state variables are divided into seven variables:

- The variable *sessions* contains the one-to-one relationships between SESSIONS and AUTHORIZED_USERS.

  The example of the *sessions* value:

  ```
  {(SESSIONS1 ↦ AUTHORIZED_USER2),
   (SESSIONS2 ↦ AUTHORIZED_USER1)}
  ```

- The variable *userRoles* contains the relation between two given sets, e.g., AUTHORIZED_USER and ROLES for determining user activities and tasks based on user permissions that each system configures.

  The example of the *userRoles* value:

  ```
  {(AUTHORIZED_USER1 ↦ LegalStaff),
   (AUTHORIZED_USER1 ↦ LegalApprover),
   (AUTHORIZED_USER2 ↦ LegalStaff)}
  ```

- The variable *pc* contains the relation between two given sets, e.g., PATIENTS and CONSENTS, representing patients' consent agreements in which patients permit users who have been defined in consent agreements to process their personal data.

  The example of the *pc* value:

  ```
  {(PATIENTS1 ↦ CONSENTS1),
   (PATIENTS1 ↦ CONSENTS2),
   (PATIENTS2 ↦ CONSENTS1),
   (PATIENTS2 ↦ CONSENTS2)}
  ```

- The variable *isConsentExpired* contains the relation between three given sets, e.g., PATIENTS, CONSENTS, and BOOL, representing the patient's consent is expired.

  The example of the *isConsentExpired* value:

```
{({(PATIENTS1 ↦ CONSENTS1)} ↦ FALSE),
 ({(PATIENTS2 ↦ CONSENTS2)} ↦ TRUE)}
```

- The variable *consentRenewalState* contains the relation between three given sets, e.g., PATIENTS, CONSENTS, and STATUSES, representing consent renewal requests.

  The example of the *consentRenewalState* value:

```
{({(PATIENTS1 ↦ CONSENTS1)} ↦ Approved),
 ({(PATIENTS2 ↦ CONSENTS2)} ↦ Rejected)}
```

- The variable *markAsDeleted* contains the relation between two given sets, e.g., PATIENTS, and CONSENTS, representing patient data has been deleted.

  The example of the *markAsDeleted* value:

```
{(PATIENTS2 ↦ CONSENTS2)}
```

- The variable *markAsReceived* contains the relation between two given sets, e.g., PATIENTS, and CONSENTS, representing the system has sent the notification to the patient for consent renewal.

  The example of the *markAsReceived* value:

```
{(PATIENTS1 ↦ CONSENTS1),
 (PATIENTS2 ↦ CONSENTS2)}
```



### 4.2.2. Events in CRSM

Events are the state transitions of the given model. In Event-B, the event will be executed when its guards meet conditions then state variables will be updated values.

The CRSM are partitioned into seven events:

#### 4.2.2.1. The INITIALISATION Event

This event is used to initiate all state variable values of the model. According to Listing A29, the five actions (act1 – act5) are assigned empty sets. As for act6 and act7, they are specified varia-

bles with first-order logic expressions using operation, called choice from set (i.e., :∈). In doing so, the *userRoles* and *pc* variables are automatically generated by the Rodin Platform.

```
INITIALISATION ≙
STATUS
  ordinary
BEGIN
  act1 : sessions ≔ ∅
  act2 : consentRenewalState ≔ ∅
  act3 : isConsentExpired ≔ ∅
  act4 : markAsDeleted ≔ ∅
  act5 : markAsReceived ≔ ∅
  act6 : userRoles :∈ AUTHORIZED_USERS ↔ ROLES
  act7 : pc :∈ ℙ1(PATIENTS × CONSENTS)
END
```

Listing A29: The INITIALISATION event.

#### 4.2.2.2. The Login Event

This event describes the behavior of login (Listing A30). The event will be executed when the current user session does not exist, and this user is registered, then the user successfully login to the system.

```
Login ≙
STATUS
  ordinary
ANY
  s, u
WHERE
  grd1 : s ∈ SESSIONS ∧ s ∉ dom(sessions)
  grd2 : u ∈ AUTHORIZED_USERS ∧ s ∉ ran(sessions)
  grd3 : sessions ∪ {s ↦ u} ∈ SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ∪ {s ↦ u}
END
```

Listing A30: The Login event.

#### 4.2.2.3. The CreateConsentRenewRequest Event

This event describes the behavior of creating a consent renewal request (Listing A31). The event will be executed when the authorized user has logged on with the legal staff role, and select a

patient whose consent is expired, then the user creates the renewal request successfully.

```
CreateConsentRenewRequest ≙
STATUS
  ordinary
ANY
  s, p, c, expired, isWithdraw
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : p ∈ dom(pc) ∧ c ∈ ran(pc) ∧ {p ↦ c} ∉
         dom(consentRenewalState)
  grd4 : expired ∈ BOOL ∧ expired = TRUE
  grd5 : isWithdraw ∈ BOOL ∧ isWithdraw = FALSE
  grd6 : consentRenewalState ⩤ {{p ↦ c} ↦ Void} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
  grd7 : isConsentExpired ⩤ {{p ↦ c} ↦ TRUE} ∈
         (PATIENTS ↔ CONSENTS) ⤖ BOOL
THEN
  act1 : consentRenewalState({p ↦ c}) ≔ Void
  act2 : isConsentExpired({p ↦ c}) ≔ TRUE
END
```

Listing A31: The CreateConsentRenewRequest event.

#### 4.2.2.4. The NotifyPatient Event

This event describes the behavior of notifying a consent renewal to the patient in which request for continuing the process of personal data (Listing A32). The event will be executed when the authorized user has logged on with the legal staff role, and the patient returns the answer to approve or reject a consent renewal request for permitting the process of his/her personal data, then the user saves the patient's answer into the system successfully.

```
NotifyPatient ≙
STATUS
  ordinary
ANY
  s, pc1, acceptStatus
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ∉ markAsReceived ∧ pc1 ∈
         dom(consentRenewalState) ∧
         consentRenewalState(pc1) = Void
```

```
  grd4 : acceptStatus ∈ STATUSES \ {Void}
  grd5 : consentRenewalState ⩤ {pc1 ↦ acceptStatus} ∈
         (PATIENTS ↔ CONSENTS) ⤖ STATUSES
THEN
  act1 : consentRenewalState(pc1) ≔ acceptStatus
  act2 : markAsReceived ≔ markAsReceived ∪ pc1
END
```

Listing A32: The NotifyPatient event.

#### 4.2.2.5. The ExtendConsentExpiration Event

This event describes the behavior of extending a consent's data retention after a patient approves the consent renewal request (Listing A33). The event will be executed when the authorized user has logged on with the legal staff role and has received approval from the patient, then the user extends the renewal period of consent.

```
ExtendConsentExpiration ≙
STATUS
  ordinary
ANY
  s, pc1
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ∈ dom(consentRenewalState) ∧
         consentRenewalState(pc1) = Approved
  grd4 : pc1 ⊆ markAsReceived ∧ pc1 ∈ dom(isConsentExpired) ∧
          isConsentExpired(pc1) = TRUE
  grd5 : isConsentExpired ⩤ {pc1 ↦ FALSE} ∈
          (PATIENTS ↔ CONSENTS) ⤖ BOOL
THEN
  act1 : isConsentExpired(pc1) ≔ FALSE
END
```

Listing A33: The ExtendConsentExpiration event

#### 4.2.2.6. The DeletePatientData Event

This event describes the behavior of deleting patient data after a patient rejects the consent renewal request (Listing A34). The event will be executed when the authorized user has logged on with the legal staff role and has received a rejective from the patient, then the user deletes the personal data.

```
ExtendConsentExpiration ≙
STATUS
  ordinary
ANY
  s, pc1
WHERE
  grd1 : s ∈ dom(sessions) ∧ sessions(s) ∈ dom(userRoles)
  grd2 : ∃r·r ∈ userRoles[sessions[{s}]] ∧ r = LegalStaff
  grd3 : pc1 ∈ dom(consentRenewalState) ∧
         consentRenewalState(pc1) = Rejected
  grd4 : pc1 ⊆ markAsReceived ∧ pc1 ∈ dom(isConsentExpired) ∧
         isConsentExpired(pc1) = TRUE
  grd5 : markAsDeleted ∩ pc1 = ∅
THEN
  act1 : markAsDeleted ≔ markAsDeleted ∪ pc1
END
```

Listing A34: The DeletePatientData event.

#### 4.2.2.7. The Logout Event

This event describes the behavior of logout (Listing A35). The event will be executed when the current user session exists, then the system removes the variable *sessions* values within the current user.

```
Logout ≙
STATUS
  ordinary
ANY
  s
WHERE
  grd1 : s ∈ dom(sessions)
  grd2 : sessions ⩤ {sessions(s)} ∈
         SESSIONS ⤖ AUTHORIZED_USERS
THEN
  act1 : sessions ≔ sessions ⩤ {sessions(s)}
END
```

Listing A35: The Logout event.

# APPENDIX B

# AN EVENT-B MODEL OF CONSENT MANAGEMENT FOR DISTRIBUTED SYSTEMS IN DATA SHARING

An Event-B model was constructed based on DSSM. The DSSM is a state machine that explains the dynamic behavior of how to conduct data subjects' consent and how to manage the interaction between the requester and response services for sharing personal data based on giving consent using blockchain technology without storing any personal data on-chain or off-chain storage servers. The Event-B model contains five functionalities: 1) conducting individuals' consent, 2) limiting access to authorized personal data based on the individual's consent, 3) allowing individuals to withdraw consents, 4) allowing individuals to request portable their personal data, and 5) enabling individuals to renew their consent for continued use of services and products offered by service providers. Besides, the Event-B model are available for the public at https://github.com/cucpbioinfo/BlockchainBasedDataSharing. Moreover, we developed a platform followed by the DSSM called SmartDataTrust. The source code is available at https://github.com/cucpbioinfo/SmartDataTrust.

## 1. The DSSM Model

We modeled DSSM (Figure 85) to describe the dynamic behavior of how to manage data subjects' consent and the sharing of personal data among services on blockchain. The DSSM model is divided into two parts, including the DSCX context and the DSSM machine.



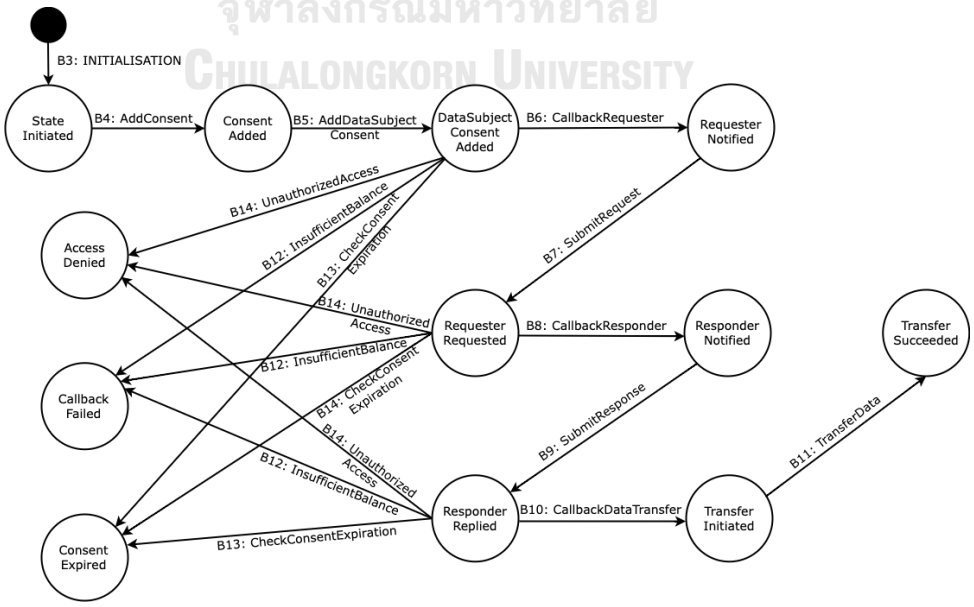



Figure 85: DSSM demonstrating blockchain-based consent management in data sharing

### 1.1. The DSCX Context

The DSCX context is the static part of the DSSM model containing the sets, constants, and axioms.

#### 1.1.1. Sets in DSCX

Sets are a set of abstracts in the context of CM in data sharing are comprises the following sets:

- DATA_SUBJECTS is a set of individuals.
- CONSENTS is a set of consent agreements.
- FIELDS is a set of data fields that leads to specific personal characteristics.
- PARTICIPANTS is a set of requester and responder services.
- REQUESTS is a set of data requests created by requester services for retrieving personal data.
- RESPONSES is a set of data responses created by responder services for replying to requester services.
- ADDRESSES is a set of smart contracts' addresses. The smart contract's address is a unique identifier that points to the collection of code and data on the blockchain.



#### 1.1.2. Axioms in DSCX

Axioms are used to determine known static relations written with predicate logic and assumed to be true. Moreover, they are also used to assign constants to pre-defined sets. According to Listing B*1*, the nine axioms (axm1 – axm9) are added to specify constants to pre-defined sets, e.g., PARTICIPANTS, CONSENTS, FIELDS, DATA_SUBJECTS, REQUESTS, RESPONSES, this (i.e., the smart contract's address), initialBalance and {this ↦ initialBalance} (i.e., defining the smart contract's balance), respectively. As for the six axioms (axm10 – axm15), they are added to deal with empty set assignments in variables restrained by partial functions, e.g., *dataFields*, *dataSubjectConsents*, *requests*, *responses*, *encryptedData* and *dataTransferStates*, respectively.

```
AXIOMS
  axm1 : partition(PARTICIPANTS, {ServiceA}, {ServiceB})
  axm2 : partition(CONSENTS,{ConsentB})
  axm3 : partition(FIELDS, {Name}, {BirthDate}, {BirthDefects})
  axm4 : partition(DATA_SUBJECTS, {DataSubject1})
  axm5 : partition(REQUESTS, {Request1})
  axm6 : partition(RESPONSES, {Response1})
  axm7 : this ∈ ADDRESSES
  axm8 : initialBalance ∈ ℕ
  axm9 : {this ↦ initialBalance} ∈ {this} ⟶ ℕ
  axm10 : ∅ ∈ CONSENTS ⇸ ℙ1(FIELDS)
  axm11 : ∅ ∈ PARTICIPANTS × DATA_SUBJECTS × CONSENTS ⇸ BOOL
  axm12 : ∅ ∈ REQUESTS ⇸ (PARTICIPANTS × DATA_SUBJECTS × CONSENTS)
  axm13 : ∅ ∈ RESPONSES ⤖ REQUESTS
  axm14 : ∅ ∈ RESPONSES ⇸ ℙ(DATA_SUBJECTS × FIELDS)
  axm15 : ∅ ∈ RESPONSES ⇸ BOOL
```

Listing B1: The list of axioms in DSCX.

## 1.2. The DSSM Machine

The DSSM machine is the dynamic part of the DSSM model containing the invariants, variables, and events.

### 1.2.1. Invariants in DSSM

Invariants constraints of state variables described by first-order logic expressions, as shown in Listing B2. In every event execution, actions change state variables' value, which must preserve all their invariants in the whole model.



```
INVARIANTS
  inv1 : consents ∈ ℙ(CONSENTS)
  inv2 : dataFields ∈ CONSENTS ⇸ ℙ1(FIELDS)
  inv3 : dataSubjectConsents ∈ PARTICIPANTS × DATA_SUBJECTS ×
         CONSENTS ⇸ BOOL
  inv4 : addresses ⊆ ADDRESSES
  inv5 : balanceOf ∈ addresses ⟶ ℕ
  inv6 : callbackRequesterStates ∈ ℙ(PARTICIPANTS ×
         DATA_SUBJECTS × CONSENTS)
  inv7 : dataAccessRequests ∈ REQUESTS ⇸ PARTICIPANTS ×
         DATA_SUBJECTS × CONSENTS
  inv8 : callbackResponderStates ∈ ℙ(REQUESTS)
  inv9 : dataAccessResponses ∈ RESPONSES ⤖ REQUESTS
  inv10 : callbackDataTransferStates ∈ ℙ(RESPONSES)
  inv11 : encryptedData ∈ RESPONSES ⇸ ℙ(DATA_SUBJECTS × FIELDS)
  inv12 : dataTransferStates ∈ RESPONSES ⇸ BOOL
```

Listing B2: The list of invariants in DSSM.

The state variables are divided into seven variables:

- The variable *consents* obtains the CONSENTS set during the model refinement.

  The example of the *consents* value:

  ```
  {ConsentB}
  ```

- The variable *dataFields* contains the relation between two given sets, e.g., CONSENTS, and FIELDS, representing the required data fields within each consent agreement.

  The example of the *dataFields* value:

  ```
  {(ConsentB ↦ {Name, BirthDate, BirthDefects})}
  ```

- The variable *dataSubjectConsents* contains the relation between four given sets, e.g., PARTICIPANTS, DATA_SUBJECTS, CONSENTS, and BOOL (i.e., TRUE or FALSE). This variable represents the valid data subject's consent within each responder service (i.e., the service which provides personal data for other services) for permitting the requester service to access personal data.

  The example of the *dataSubjectConsents* value:

  ```
  {(ServiceA ↦ DataSubject1 ↦ ConsentB ↦ FALSE),
   (ServiceB ↦ DataSubject1 ↦ ConsentB ↦ FALSE)}
  ```



- The variable *addresses* obtains the ADDRESSES, representing the smart contract's address.

  The example of the *addresses* value:

  ```
  {this}
  ```

- The variable *balanceOf* contains the relation between two given sets, e.g., ADDRESSES, and a natural number, representing the smart contract's balance.

  The example of the *balanceOf* value:

  ```
  {(this ↦ 2)}
  ```

- The variable *callbackRequesterStates* contains the relation between three given sets, e.g., PARTICIPANTS, DATA_ SUBJECTS, and CONSENTS, representing the blockchain in-

voking the callback URL to notify the requester service for requesting personal data from the responder service.

The example of the *callbackRequesterStates* value:

```
{(ServiceA ↦ DataSubject1 ↦ ConsentB),
 (ServiceB ↦ DataSubject1 ↦ ConsentB)}
```

- The variable *dataAccessRequests* contains the relation between four given sets, e.g., REQUESTS, PARTICIPANTS, DATA_SUBJECTS, and CONSENTS, representing the record of data request submitted by the requester service.

  The example of the *dataAccessRequests* value:

  ```
  {(Request1 ↦ (ServiceB ↦ DataSubject1 ↦ ConsentB))}
  ```

- The variable *callbackResponderStates* obtains the REQUESTS set, representing the blockchain invoking the callback URL to notify the responder service for replying to the requester service.

  The example of the *callbackResponderStates* value:

  ```
  {Request1}
  ```

- The variable *dataAccessResponses* contains the relation between two given sets, e.g., RESPONSES, and REQUESTS, representing the record of data response submitted by the responder service.

  The example of the *dataAccessResponses* value:

  ```
  {(Response ↦ Request)}
  ```



- The variable *callbackDataTransferStates* obtains the RESPONSES set, representing the blockchain invoking the callback URL to notify the responder service for transferring personal data between requester service.

  The example of the *callbackDataTransferStates* value:

  ```
  {Response1}
  ```

- The variable *encryptedData* contains the relation between three given sets, e.g., RESPONSES, DATA_SUBJECTS, and FIELDS, representing the personal data encryption in which data has been selected from the consent's data fields mapping.

  The example of the *encryptedData* value:

```
{(Response1 ↦ {(DataSubject1 ↦ Name),
               (DataSubject1 ↦ BirthDate)}})
```

- The variable *dataTransferStates* contains the relation between three given sets, e.g., RESPONSES, and BOOL, representing data transfer between the responder and requester services successful.

The example of the *dataTransferStates* value:

```
{(Response1 ↦ TRUE)}
```

### 1.2.2. Events in DSSM

Events are the state transitions of the given model. In Event-B, the event will be executed when its guards meet conditions then state variables will be updated values.

The DSSM are partitioned into thirteen events:

#### 1.2.2.1. The INITIALISATION Event

This event is used to initiate all state variable values of the model. According to Listing B3: The INITIALISATION event., the ten actions (act1 – act10) are assigned empty sets. As for act11 and act12, they are specified variables with first-order logic expressions using operation, called choice from set (i.e., :∈). In doing so, the *addresses* and *balanceOf* variables are automatically generated by the Rodin Platform.



```
INITIALISATION ≙
STATUS
  ordinary
BEGIN
  act1 : consents ≔ ∅
  act2 : dataFields ≔ ∅
  act3 : dataSubjectConsents ≔ ∅
  act4 : callbackRequesterStates ≔ ∅
  act5 : dataAccessRequests ≔ ∅
  act6 : callbackResponderStates ≔ ∅
  act7 : dataAccessResponses ≔ ∅
  act8 : callbackDataTransferStates ≔ ∅
  act9 : encryptedData ≔ ∅
  act10 : dataTransferStates ≔ ∅
  act11 : addresses ≔ {this}
```

```
  act12 : balanceOf ≔ {this ↦ initialBalance}
END
```

Listing B3: The INITIALISATION event.

#### 1.2.2.2. The AddConsent Event

This event describes the behavior of adding consent (Listing B4). The event will be executed when the consent does not exist, then the requester service adds a new consent into blockchain.

```
AddConsent ≙
STATUS
  ordinary
ANY
  consent, dataField
WHERE
  grd1 : consent ∈ CONSENTS ∧ consent ∉ consents
  grd2 : dataField ∈ ℙ1(FIELDS)
  grd3 : dataFields ⩤ {consent ↦ dataField} ∈
         CONSENTS ⇸ ℙ1(FIELDS)
THEN
  act1 : consents ≔ consents ∪ {consent}
  act2 : dataFields(consent) ≔ dataField
END
```

Listing B4: The AddConsent event.

#### 1.2.2.3. The AddDataSubjectConsent Event

This event describes the behavior of adding a data subject's consent (Listing B5). The event will be executed when the data subject's consent within the responder service does not exist in the blockchain (i.e., the data subject gives his/her consent under the responder service for the first time), then the blockchain saves the data subject's consent successfully.

```
AddDataSubjectConsent ≙
STATUS
  ordinary
ANY
  responder, consent, dataField
WHERE
  grd1 : responder ∈ PARTICIPANTS
  grd2 : dataSubject ∈ DATA_SUBJECTS
  grd3 : consent ∈ consents ∧ consent ∈ dom(dataFields)
  grd4 : responder ↦ dataSubject ↦ consent ∉
```

```
        dom(dataSubjectConsents)
  grd5 : dataSubjectConsents ⩤ {responder ↦ dataSubject ↦
        consent ↦ TRUE} ∈ (PARTICIPANTS × DATA_SUBJECTS ×
        CONSENTS) ⇸ BOOL
THEN
  act1 : dataSubjectConsents(responder ↦ dataSubject ↦
        consent) ≔ TRUE
END
```

Listing B5: The AddDataSubjectConsent event.

#### 1.2.2.4. The CallbackRequester Event

This event describes the behavior of making an API call to requester service by blockchain (Listing B6). The event will be executed when the smart contract's balance is enough to pay the oraclize's fee for the callback URL, the data subject's consent is valid, then the blockchain makes an API call to the requester service successfully.

```
CallbackRequester ≙
STATUS
  ordinary
ANY
  oraclizeFee, dataSubjectConsent
WHERE
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
        oraclizeFee ≤ balanceOf(this)
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
        dataSubjectConsent ∉ callbackRequesterStates ∧
        dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
        addresses ⟶ ℕ
THEN
  act1 : callbackRequesterStates ≔ callbackRequesterStates ∪
        {dataSubjectConsent}
  act2 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
        oraclizeFee}
END
```

Listing B6: The CallbackRequester event.

#### 1.2.2.5. The SubmitRequest Event

This event describes the behavior of submitting the data request to the blockchain by the requester service (Listing B7). The event will be executed when the data subject's consent is valid, and

the data request does not exist in the blockchain, then the blockchain saves the data request successfully.

```
SubmitRequest ≜
STATUS
  ordinary
ANY
  consentExpired, dataSubjectConsent, request
WHERE
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : dataSubjectConsent ∈ callbackRequesterStates
  grd4 : request ∈ REQUESTS ∧ request ∉ dom(dataAccessRequests)
  grd5 : dataAccessRequests ⩤ {request ↦ dataSubjectConsent} ∈
         REQUESTS ⇸ PARTICIPANTS × DATA_SUBJECTS × CONSENTS
THEN
  act1 : dataAccessRequests(request) ≔ dataSubjectConsent
END
```

Listing B7: The SubmitRequest event.

### 1.2.2.6. The CallbackResponder Event

This event describes the behavior of making an API call to responder service by blockchain (Listing B8). The event will be executed when the smart contract's balance is enough to pay the oraclize's fee for the callback URL, the data subject's consent is valid, then the blockchain makes an API call to the responder service successfully.



```
CallbackResponder ≜
STATUS
  ordinary
ANY
  oraclizeFee, request
WHERE
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee ≤ balanceOf(this)
  grd2 : request ∈ dom(dataAccessRequests) ∧
         request ∉ callbackResponderStates ∧
         dataAccessRequests(request) ∈
         dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataAccessRequests(request)) =
         TRUE
  grd3 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
         addresses ⟶ ℕ
THEN
```

```
  act1 : callbackResponderStates ≔ callbackResponderStates ∪
         {request}
  act2 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
         oraclizeFee}
END
```

Listing B8: The CallbackResponder event.

#### 1.2.2.7. The SubmitResponse Event

This event describes the behavior of submitting the data response to the blockchain by the responder service (Listing B9). The event will be executed when the data subject's consent is valid, and the data response does not exist in the blockchain, then the blockchain saves the data response successfully.

```
SubmitResponse ≙
STATUS
  ordinary
ANY
  consentExpired, dataSubjectConsent, request, response
WHERE
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = FALSE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : request ∈ callbackResponderStates
  grd4 : response ∈ RESPONSES ∧ response ∉
         dom(dataAccessResponses)
  grd5 : dataAccessResponses ⩤ {response ↦ request} ∈
         RESPONSES ⤖ REQUESTS
THEN
  act1 : dataAccessResponses ⩤ {response ↦ request} ∈
         RESPONSES ⤖ REQUESTS
END
```

Listing B9: The SubmitResponse event.

#### 1.2.2.8. The CallbackDataTransfer Event

This event describes the behavior of making an API call to responder service by blockchain (Listing B10). The event will be executed when the smart contract's balance is enough to pay the oraclize's fee for the callback URL, the data subject's consent is valid, then the blockchain makes an API call to the responder service successfully.

```
CallbackDataTransfer ≜
STATUS
  ordinary
ANY
  oraclizeFee, response
WHERE
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
         oraclizeFee ≤ balanceOf(this)
  grd2 : response ∈ dom(dataAccessResponses) ∧
         response ∉ callbackDataTransferStates
  grd3 : dataAccessResponses(response) ∈
         dom(dataAccessRequests) ∧
         dataAccessRequests(dataAccessResponses(response)) ∈
         dom(dataSubjectConsents) ∧ dataSubjectConsents(
         dataAccessRequests(dataAccessResponses(response))) =
         TRUE
  grd4 : balanceOf ⩤ {this ↦ balanceOf(this) − oraclizeFee} ∈
         addresses ⟶ ℕ
THEN
  act1 : callbackDataTransferStates ≔ callbackResponderStates ∪
         {response}
  act2 : balanceOf ≔ balanceOf ⩤ {this ↦ balanceOf(this) −
         oraclizeFee}
END
```

Listing B10: The CallbackDataTransfer event.

### 1.2.2.9. The TransferData Event

This event describes the behavior of transferring data between the responder and requester services (Listing B11). The event will be executed when the data subject's consent, data request, and data response are valid, then the responder service encrypts and transfers personal data to the requester service successfully.

```
TransferData ≜
STATUS
  ordinary
ANY
  responder, dataSubject, consent, response
WHERE
  grd1 : response ∈ callbackDataTransferStates ∧
         response ∈ dom(dataAccessResponses) ∧
         response ∉ dom(dataTransferStates)
  grd2 : consent ∈ dom(dataFields)
  grd3 : ∃x·x ∈
         dataAccessRequests[{dataAccessResponses(response)}] ∧
```

```
        x = responder ↦ dataSubject ↦ consent ∧
        responder ↦ dataSubject ↦ consent ∈
        dom(dataSubjectConsents)∧ dataSubjectConsents(x) =
        TRUE
  grd4 : encryptedData ⩤ {response ↦ {dataSubject} ×
        dataFields(consent)} ∈ RESPONSES ⇸ ℙ(DATA_SUBJECTS ×
        FIELDS)
  grd5 : dataTransferStates ⩤ {response ↦ TRUE} ∈ RESPONSES ⇸
        BOOL
THEN
  act1 : encryptedData(response) ≔ {dataSubject} ×
        dataFields(consent)
  act2 : dataTransferStates(response) ≔ TRUE
END
```

Listing B11: The TransferData event.

#### 1.2.2.10. The InsufficientBalance Event

This event describes the behavior of the smart contract having a balance insufficient to cover the oraclize's fee for making an API call outside the blockchain (Listing B12).

The event will be executed when the oraclize's fee is greater than the smart contract's balance which occurs in the callback URL events, then the blockchain handles the insufficient balance exception.

```
InsufficientBalance ≙
STATUS
  ordinary
ANY
  oraclizeFee, dataSubjectConsent, request, response
WHERE
  grd1 : this ∈ dom(balanceOf) ∧ oraclizeFee ∈ ℕ ∧
        oraclizeFee > balanceOf(this)
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
        dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : (dataSubjectConsent ∉ callbackRequesterStates) ∨
        (request ↦ dataSubjectConsent ∈ dataAccessRequests ∧
        request ∉ callbackResponderStates) ∨
        (response ↦ request ∈ dataAccessResponses ∧
        Response ∉ callbackDataTransferStates)
THEN
..skip
END
```

Listing B12: The InsufficientBalance event.

#### 1.2.2.11. The CheckConsentExpiration Event

This event describes the behavior of checking consent expiration (Listing B13). The event will be executed when the data subject's consent is expired, then the blockchain handles the expired exception.

```
CheckConsentExpiration ≙
STATUS
  ordinary
ANY
  consentExpired, dataSubjectConsent
WHERE
  grd1 : consentExpired ∈ BOOL ∧ consentExpired = TRUE
  grd2 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
  grd3 : dataSubjectConsents ⩤ {dataSubjectConsent ↦ FALSE} ∈
         PARTICIPANTS × DATA_SUBJECTS × CONSENTS ⇸ BOOL
THEN
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ FALSE
END
```

Listing B13: The CheckConsentExpiration event.

#### 1.2.2.12. The UnauthorizedAccess Event

This event describes the behavior of handling invalid data subject's consent during the interaction between the requester and responder services (Listing B14). The event will be executed when the data subject's consent is invalid, then the blockchain handles the invalid exception.



```
UnauthorizedAccess ≙
STATUS
  ordinary
ANY
  dataSubjectConsent
WHERE
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents)∧
         dataSubjectConsents(dataSubjectConsent) = FALSE
THEN
..skip
END
```

Listing B14: The UnauthorizedAccess event.

#### 1.2.2.13. The RevokeConsent Event

This event describes the behavior of withdrawing the data subject's consent via the responder service (Listing B15). The event will be executed when the data subject's consent is valid, then the blockchain updates the data subject's consent to invalid.

```
RevokeConsent ≙
STATUS
  ordinary
ANY
  dataSubjectConsent
WHERE
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
         dataSubjectConsents(dataSubjectConsent) = TRUE
    grd2 : dataSubjectConsents⩤ {dataSubjectConsent ↦ FALSE} ∈
         (PARTICIPANTS × DATA_SUBJECTS × CONSENTS) ⇸ BOOL
THEN
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ FALSE
END
```

Listing B15: The RevokeConsent event.

#### 1.2.2.14. The RenewConsent Event

This event describes the behavior of renewing the data subject's consent via the responder service (Listing B16). The event will be executed when the data subject's consent is invalid, then the blockchain updates the data subject's consent to valid.



```
RenewConsent ≙
STATUS
  ordinary
ANY
  dataSubjectConsent
WHERE
  grd1 : dataSubjectConsent ∈ dom(dataSubjectConsents) ∧
  grd2 : dataSubjectConsents(dataSubjectConsent) = FALSE
    grd3 : dataSubjectConsents ⩤ {dataSubjectConsent ↦ TRUE} ∈
         (PARTICIPANTS × DATA_SUBJECTS × CONSENTS) ⇸ BOOL
THEN
  act1 : dataSubjectConsents(dataSubjectConsent) ≔ TRUE
END
```

Listing B16: The RenewConsent event.

# REFERENCES


1. Commission, E., *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance)*. 2016, European Commission.
2. Daoudagh, S., et al. *How to Improve the GDPR Compliance through Consent Management and Access Control*. in *Proceedings of the 7th International Conference on Information Systems Security and Privacy - ICISSP*. 2021.
3. Agrafiotis, I., et al. *Applying Formal Methods to Detect and Resolve Ambiguities in Privacy Requirements*. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg.
4. de Carvalho, R.M., et al., *Protecting Citizens' Personal Data and Privacy: Joint Effort from GDPR EU Cluster Research Projects.* SN Computer Science, 2020. **1**(4): p. 217.
5. Alhazmi, A. and N.A.G. Arachchilage, *I'm all ears! Listening to software developers on putting GDPR principles into software development practice.* Personal and Ubiquitous Computing, 2021. **25**(5): p. 879-892.
6. Awanthika, S. and A. Nalin A. G., *Why developers cannot embed privacy into software systems? An empirical investigation*, in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. 2018: Christchurch, New Zealand. p. 211–216.
7. Bu, F., et al., *"Privacy by Design" implementation: Information system engineers' perspective.* International Journal of Information Management, 2020. **53**: p. 102124.
8. Hadar, I., et al., *Privacy by designers: software developers' privacy mindset.* Empirical Software Engineering, 2018. **23**(1): p. 259-289.
9. Spalevic, Z. and K. Vićentijević, *GDPR and challenges of personal data protection.* The European Journal of Applied Economics, 2022. **19(1)**: p. 55-65.
10. Cavoukian, A., *Privacy by Design: The 7 Foundational Principles*. 2011.
11. Cavoukian, A., *Understanding How to Implement Privacy by Design, One Step at a Time.* IEEE Consumer Electronics Magazine, 2020. **9**(2): p. 78-82.
12. Alkhariji, L., et al., *Synthesising Privacy by Design Knowledge Toward Explainable Internet of Things Application Designing in Healthcare.* ACM Transactions on Multimedia Computing, Communications, and Applications, 2021. **17**: p. 1-29.
13. Levin, A., *The Case Study of Ontario (January 1, 2018).* Canadian Journal of Comparative and Contemporary Law, 2018. **4 (1)**: p. 115-160.
14. al, K.S.e., *Deliverable D3.2: Cross Sectoral Cybersecurity Building Blocks.* 2020.
15. al, S.A.e., *Deliverable D3.11: Definition of Privacy by Design and Privacy Preserving Enablers.* 2020.
16. Gruschka, N. and M. Jensen, *Aligning user consent management and service process modeling.* Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI), 2014: p. 527-538.
17. Fatema, K., et al., *Compliance through Informed Consent: Semantic Based Consent Permission and Data Management Model*, in *Proceedings of the 5th*

*Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn2017) (PrivOn)*. 2017.
18. Bincoletto, G., *Data protection issues in cross-border interoperability of Electronic Health Record systems within the European Union.* Data & Policy, 2020. **2**: p. e3.
19. Koops, B.-J. and R.E. Leenes, *Privacy Regulation Cannot Be Hardcoded. A Critical Comment on the 'Privacy by Design' Provision in Data-Protection Law.* International Review of Law, Computers & Technology, 2014. **28(2)**: p. 159-171.
20. Kakarlapudi, P. and Q. Mahmoud, *A Systematic Review of Blockchain for Consent Management.* Healthcare, 2021. **9**: p. 137.
21. Stephen, C., et al., *Report on the NSF Workshop on Formal Methods for Security*. 2016, National Science Foundation, USA.
22. Tschantz, M.C. and J.M. Wing. *Formal Methods for Privacy*. 2009. Berlin, Heidelberg: Springer Berlin Heidelberg.
23. Abrial, J.-R., *Modeling in Event-B: system and software engineering*. 2010: Cambridge University Press.
24. Abrial, J.-R. *A System Development Process with Event-B and the Rodin Platform*. 2007. Berlin, Heidelberg: Springer Berlin Heidelberg.
25. Abrial, J.-R., et al., *Rodin: an open toolset for modelling and reasoning in Event-B.* International Journal on Software Tools for Technology Transfer, 2010. **12**(6): p. 447-466.
26. Michael, J.P.M., Butler, *Rodin User's Handbook: Covers Rodin v.2.8*. 2014: Publishing Platform, North Charleston, SC, USA.
27. Albanese, G., et al., *Dynamic consent management for clinical trials via private blockchain technology.* Journal of Ambient Intelligence and Humanized Computing, 2020. **11**(11): p. 4909-4926.
28. Dijana, P. *Guidelines for GDPR compliant consent and data management model in ICT businesses*. in *29th international conference of central European conference on information and intelligent systems*. 2018.
29. Steinbrook, R., *Personally Controlled Online Health Data — The Next Big Thing in Medical Care?* The New England journal of medicine, 2008. **358**: p. 1653-6.
30. Fatehi, F., et al., *General Data Protection Regulation (GDPR) in Healthcare: Hot Topics and Research Fronts.* Studies in health technology and informatics, 2020. **270**: p. 1118-1122.
31. Asghar, M.R., et al. *A Review of Privacy and Consent Management in Healthcare: A Focus on Emerging Data Sources*. in *2017 IEEE 13th International Conference on e-Science (e-Science)*. 2017.
32. Simone, F.-H., *IT-Security and Privacy: Design and Use of Privacy-Enhancing Security Mechanisms*. 2001: Springer-Verlag, Berlin.
33. Hert, P. and V. Papakonstantinou, *The proposed data protection Regulation replacing Directive 95/46/EC: A sound system for the protection of individuals.* The Computer Law & Security Review, 2012. **28**: p. 130–142.
34. Blume, P., *The myths pertaining to the proposed General Data Protection Regulation.* International Data Privacy Law, 2014. **4**: p. 269-273.
35. Gürses, S.F., C. Troncoso, and C. Díaz. *Engineering Privacy by Design*. in

*Computers, Privacy & Data Protection* 2011.
36. Blake, M.B. and S. Iman, *Formal Methods for Preserving Privacy for Big Data Extraction Software*, in *NSF Workshop on Big Data Security and Privacy*. 2014.
37. Kitchin, R., *Big data and human geography:Opportunities, challenges and risks.* Dialogues in Human Geography, 2013. **3**(3): p. 262-267.
38. Besik, S. and J.-C. Freytag, *Managing Consent in Workflows under GDPR*, in *ZEUS*. 2020.
39. Politou, E., E. Alepis, and C. Patsakis, *Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions.* Journal of Cybersecurity, 2018. **4**(1).
40. Voss, W.G., *Looking at European Union Data Protection Law Reform Through a Different Prism: The Proposed EU General Data Protection Regulation Two Years Later.* Journal of Internet Law, 2014. **17(9)**.
41. Wolters, P.T.J., *The Control by and Rights of the Data Subject Under the GDPR.* Journal of Internet Law, 2018. **22(1)**: p. 7-18.
42. Resnik, D., *Re-consenting human subjects: Ethical, legal and practical issues.* Journal of medical ethics, 2009. **35**: p. 656-7.
43. Jaiman, V. and V. Urovi, *A Consent Model for Blockchain-Based Health Data Sharing Platforms.* IEEE Access, 2020. **8**: p. 143734-143745.
44. Vargas, J.C. *Blockchain-based consent manager for GDPR compliance*. in *Open Identity Summit*. 2019.
45. Jung, H.-H. and F.J. Pfister, *Blockchain-enabled Clinical Study Consent Management.* Technology Innovation Management Review, 2020. **10**: p. 14-24.
46. Ameyed, D., et al. *Blockchain Based Model for Consent Management and Data Transparency Assurance*. in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2021.
47. Jung, S.-S., S.-J. Lee, and I.-C. Euom, *Delegation-Based Personal Data Processing Request Notarization Framework for GDPR Based on Private Blockchain.* Applied Sciences, 2021. **11**(22): p. 10574.
48. Finck, M., *Blockchains and Data Protection in the European Union*, M.P.I.f.I.a.C.U.o. Oxford, Editor. 2017.
49. Miltiadou, D., et al., *Leveraging Management of Customers' Consent Exploiting the Benefits of Blockchain Technology Towards Secure Data Sharing*, in *Big Data and Artificial Intelligence in Digital Finance: Increasing Personalization and Trust in Digital Finance using Big Data and AI*, J. Soldatos and D. Kyriazis, Editors. 2022, Springer International Publishing: Cham. p. 127-155.
50. Abedjan, Z., et al., *Data Science in Healthcare: Benefits, Challenges and Opportunities*, in *Data Science for Healthcare: Methodologies and Applications*, S. Consoli, D. Reforgiato Recupero, and M. Petković, Editors. 2019, Springer International Publishing: Cham. p. 3-38.
51. Stalla-Bourdillon, S., et al., *Data protection by design: Building the foundations of trustworthy data sharing.* Data & Policy, 2020. **2**: p. e4.
52. Schupp, S., *Tool Support of Formal Methods for Privacy by Design*. 2019.
53. Matwin, S., et al. *Privacy in Data Mining Using Formal Methods*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
54. Team, C.D. *The Coq Proof Assistant reference manual: Version 8.13.1*. 2021 26 September 2022]; Available from:

https://github.com/coq/coq/releases/download/V8.13.1/coq-8.13.1-reference-manual.pdf.
55. Zdravko, M. and R. Ingrid, *An introduction to the WEKA data mining system*, in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*. 2006, Association for Computing Machinery: Bologna, Italy. p. 367–368.
56. Stouppa, P. and T. Studer. *A Formal Model of Data Privacy*. 2007. Berlin, Heidelberg: Springer Berlin Heidelberg.
57. Ni, Q., et al., *Privacy-aware role-based access control.* ACM Trans. Inf. Syst. Secur., 2010. **13**(3): p. Article 24.
58. Ashley, P., et al., *Enterprise privacy authorization language (EPAL).* IBM Research, 2003. **30**: p. 31.
59. Abe, A. and A. Simpson. *Formal Models for Privacy*. in *EDBT/ICDT Workshops*. 2016.
60. *Data Protection Act 2018 Explanatory Notes*. 2018 [cited 26 September 2022]; Available from: http://www.legislation.gov.uk/ukpga/2018/12/notes
61. Alagar, V.S. and K. Periyasamy, *The Z Notation*, in *Specification of Software Systems*. 1998, Springer New York: New York, NY. p. 281-360.
62. Plagge, D. and M. Leuschel. *Validating Z Specifications Using the ProB Animator and Model Checker*. 2007. Berlin, Heidelberg: Springer Berlin Heidelberg.
63. Besik, S.I. and J.-C. Freytag, *A formal approach to build privacy-awareness into clinical workflows.* SICS Software-Intensive Cyber-Physical Systems, 2020. **35**(1): p. 141-152.
64. Tokas, S. and O. Owe. *A Formal Framework for Consent Management*. 2020. Cham: Springer International Publishing.
65. McCracken, D. and E. Reilly, *Backus-Naur form (BNF).* Encyclopedia of Computer Science, 2003: p. 129-131.
66. Hyysalo, J., et al., *Consent Management Architecture for Secure Data Transactions*. 2016.
67. Kuikkaniemi, K., A. Poikola, and H. Honko. *MyData A Nordic Model for human-centered personal data management and processing*. in *Ministry of Transport and Communications*. 2015.
68. Byström, N., et al. *MyData Architecture—The Stack, version 1.0.0*. 2015 [cited 26 September 2022]; Available from: https://hiit.github.io/mydata-stack/.
69. *Atomic Commit In SQLite*. 2017 [cited 26 September 2022]; Available from: http://www.sqlite.org/atomiccommit.html.
70. Marillonnet, P., et al., *An Efficient User-Centric Consent Management Design for Multiservices Platforms.* Security and Communication Networks, 2021. **2021**: p. 1-19.
71. *IBM Security Cost of a Data Breach Report*. 2022 [cited 26 September 2022]; Available from: https://www.ibm.com/security/data-breach.
72. Daudén-Esmel, C., et al. *Lightweight Blockchain-based Platform for GDPR-Compliant Personal Data Management*. in *2021 IEEE 5th International Conference on Cryptography, Security and Privacy (CSP)*. 2021.
73. Khan, S.N., et al., *Blockchain smart contracts: Applications, challenges, and future trends.* Peer-to-Peer Networking and Applications, 2021. **14**(5): p. 2901-

2925.
74. Merlec, M.M., et al., *A Smart Contract-Based Dynamic Consent Management System for Personal Data Usage under GDPR.* Sensors, 2021. **21**(23): p. 7994.
75. Zheng, G., et al., *Decentralized Application (DApp)*, in *Ethereum Smart Contract Development in Solidity*. 2021, Springer Singapore: Singapore. p. 253-280.
76. Chase, J.P.M. *A Permissioned Implementation of Ethereum*. 2018 26 September 2022]; Available from: https://github.com/jpmorganchase/quorum.
77. Vitalik, B. *Ethereum: A next-generation smart contract and decentralized application platform*. 2014 26 September 2022]; Available from: https://github.com/ethereum/wiki/ wiki/White-Paper.
78. Benet, J., *IPFS - Content Addressed, Versioned, P2P File System.* 2014.
79. Ezzat, S.K., Y.N.M. Saleh, and A.A. Abdel-Hamid, *Blockchain Oracles: State-of-the-Art and Research Directions.* IEEE Access, 2022. **10**: p. 67551-67572.
80. Chris, D., *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. 1st. ed. ed. 2017: Apress, USA.
81. *Istanbul BFT*. 26 September 2022]; Available from: https://github.com/ethereum/EIPs/issues/650.
82. *Raft-based consensus for Ethereum/Quorum*. 26 September 2022]; Available from: https://github.com/ jpmorganchase/quorum/blob/master/raft/doc.md.
83. Rantos, K., et al. *ADvoCATE: A Consent Management Platform for Personal Data Processing in the IoT Using Blockchain Technology*. 2019. Cham: Springer International Publishing.
84. Kosko, B., *Fuzzy cognitive maps.* International Journal of Man-Machine Studies, 1986. **24**(1): p. 65-75.
85. Nguyen, L.V., et al., *Cognitive Similarity-Based Collaborative Filtering Recommendation System.* Applied Sciences, 2020. **10**(12): p. 4183.
86. Azaria, A., et al. *MedRec: Using Blockchain for Medical Data Access and Permission Management*. in *2016 2nd International Conference on Open and Big Data (OBD)*. 2016.
87. Hu, C., et al., *CrowdMed-II: a blockchain-based framework for efficient consent management in health data sharing.* World Wide Web, 2022. **25**(3): p. 1489-1515.
88. Shah, M., et al. *CrowdMed: A Blockchain-Based Approach to Consent Management for Health Data Sharing*. 2019. Cham: Springer International Publishing.
89. Harbitter, A., *A critical look at centralized and distributed strategies for large scale justice sharing applications*. 2004, Washington, D.C: Integrated Justice Information Systems Institute.
90. van Steen, M. and A.S. Tanenbaum, *A brief introduction to distributed systems.* Computing, 2016. **98**(10): p. 967-1009.
91. *The proposed EU General Data Protection Regulation*. 2015, Hunton & Williams.
92. Kurteva, A., et al., *Consent Through the Lens of Semantics: State of the Art Survey and Best Practices.* Semantic Web, 2021: p. 1-27.
93. Pandit, H.J., et al. *GConsent - A Consent Ontology Based on the GDPR*. 2019. Cham: Springer International Publishing.

94. Kirrane, S., et al., *The SPECIAL-K Personal Data Processing Transparency and Compliance Platform.* ArXiv, 2020. **abs/2001.09461**.
95. Pandit, H.J., et al. *Creating a Vocabulary for Data Privacy*. 2019. Cham: Springer International Publishing.
96. Lioudakis, G.V., et al. *Facilitating GDPR Compliance: The H2020 BPR4GDPR Approach*. 2020. Cham: Springer International Publishing.
97. Palmirani, M., et al. *PrOnto: Privacy Ontology for Legal Reasoning*. 2018. Cham: Springer International Publishing.
98. Loukil, F., et al. *LIoPY: A Legal Compliant Ontology to Preserve Privacy for the Internet of Things*. in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. 2018.
99. Toumia, A., S. Szoniecky, and I. Saleh. *ColPri: Towards a Collaborative Privacy Knowledge Management Ontology for the Internet of Things*. in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2020.
100. Abrial, J.-R. and H. Stefan, *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B.* Fundam. Inf., 2007. **77**(1–2): p. 1–28.
101. Jarrar, A. and Y. Balouki, *Formal modeling of a complex adaptive air traffic control system.* Complex Adaptive Systems Modeling, 2018. **6**(1): p. 6.
102. Hallerstede, S., *On the purpose of Event-B proof obligations.* Formal Aspects of Computing, 2011. **23**(1): p. 133-150.
103. Ruíz Barradas, H., L. Burdy, and D. Déharbe. *Existence Proof Obligations for Constraints, Properties and Invariants in Atelier B*. 2020. Cham: Springer International Publishing.
104. Hoang, T.S., *An Introduction to the Event-B Modelling Method*, in *Industrial Deployment of System Engineering Methods*. 2013, Springer-Verlag. p. 211-236.
105. Dobrikov, I. and M. Leuschel. *Enabling Analysis for Event-B*. 2016. Cham: Springer International Publishing.
106. Sato, N. and F. Ishikawa, *Separation of considerations in event-B refinement toward industrial use.* CEUR Workshop Proceedings, 2015. **1385**: p. 43-50.
107. Méry, D. *Teaching programming methodology using Event B*. in *The B Method: from Research to Teaching*. 2008. Nantes, France: APCB.
108. Cansell, D. and D. Méry, *Tutorial on the event-based B method*. 2006, IFIP FORTE 2006 Paris.
109. Hoang, T.S. and J.-R. Abrial. *Reasoning about Liveness Properties in Event-B*. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg.
110. Yang, F. and J.-P. Jacquot. *An Event-B Plug-in for Creating Deadlock-Freeness Theorems*. in *14th Brazilian Symposium on Formal Methods*. 2011. São Paulo, Brazil.
111. Ligot, O., J. Bendisposto, and M. Leuschel. *Debugging Event-B Models using the ProB Disprover Plug-in !* in *AFADL'07*. 2007.
112. Leuschel, M. and M. Butler. *ProB: A Model Checker for B*. 2003. Berlin, Heidelberg: Springer Berlin Heidelberg.
113. Robinson, K. *A Concise Summary of the Event-B mathematical toolkit*. 2014 11.06.2022; Available from: https://wiki.event-b.org/images/EventB-Summary-refcard.pdf.
114. Suripeddi, M.K.S. and P. Purandare, *Blockchain and GDPR – A Study on*

*Compatibility Issues of the Distributed Ledger Technology with GDPR Data Processing.* Journal of Physics: Conference Series, 2021. **1964**(4): p. 042005.
115. Chinnasamy, P., et al., *Blockchain based Access Control and Data Sharing Systems for Smart Devices.* Journal of Physics: Conference Series, 2021. **1767**(1): p. 012056.
116. Sutton, A. and R. Samavi. *Blockchain Enabled Privacy Audit Logs*. 2017. Cham: Springer International Publishing.
117. Wang, X., *Design and Implementation of a Data Sharing Model for Improving Blockchain Technology.* Advances in Multimedia, 2022. **2022**: p. 4578525.
118. Agrawal, T.K., et al., *Demonstration of a blockchain-based framework using smart contracts for supply chain collaboration.* International Journal of Production Research, 2022: p. 1-20.
119. Monrat, A.A., O. Schelén, and K. Andersson, *A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities.* IEEE Access, 2019. **7**: p. 117134-117151.
120. Ramkumar, N., G. Sudhasadasivam, and K.G. Saranya. *A Survey on Different Consensus Mechanisms for the Blockchain Technology*. in *2020 International Conference on Communication and Signal Processing (ICCSP)*. 2020.
121. Sharma, D.K., et al., *Chapter 13 - Cryptocurrency Mechanisms for Blockchains: Models, Characteristics, Challenges, and Applications*, in *Handbook of Research on Blockchain Technology*, S. Krishnan, et al., Editors. 2020, Academic Press. p. 323-348.
122. Sharma, G., A. Kumar, and S.S. Gill, *Chapter 4 - Applications of blockchain in automated heavy vehicles: Yesterday, today, and tomorrow*, in *Autonomous and Connected Heavy Vehicle Technology*, R. Krishnamurthi, A. Kumar, and S.S. Gill, Editors. 2022, Academic Press. p. 81-93.
123. Aggarwal, S. and N. Kumar, *Chapter Ten - Core components of blockchain☆☆Introduction to blockchain*, in *Advances in Computers*, S. Aggarwal, N. Kumar, and P. Raj, Editors. 2021, Elsevier. p. 193-209.
124. Elli, A., et al. *Hyperledger fabric: a distributed operating system for permissioned blockchains*. in *Proceedings of the Thirteenth EuroSys Conference*. 2018. Porto, Portugal: Association for Computing Machinery.
125. Akcora, C.G., Y.R. Gel, and M. Kantarcioglu, *Blockchain networks: Data structures of Bitcoin, Monero, Zcash, Ethereum, Ripple, and Iota.* WIREs Data Mining and Knowledge Discovery, 2022. **12**(1): p. e1436.
126. Wang, S., et al. *An Overview of Smart Contract: Architecture, Applications, and Future Trends*. in *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018.
127. Bakri, A., S. Ellis, and A. Adel. *Blockchain-Based Applications in Higher Education: A Systematic Mapping Study*. in *The 5th International Conference on Information and Education Innovations (ICIEI '20)*. 2020. Association for Computing Machinery.
128. Somboun, T., *Survey of Smart Contract Technology and Application Based on Blockchain.* Open Journal of Applied Sciences, 2021. **11**: p. 1135-1148.
129. Abdeljalil, B., *A Study of Blockchain Oracles.* ArXiv, 2020.
130. Vanezi, E., et al. *GDPR Compliance in the Design of the INFORM e-Learning Platform: a Case Study*. in *2019 13th International Conference on Research Challenges in Information Science (RCIS)*. 2019.

131. Hoepman, J.-H. *Privacy Design Strategies*. 2014. Berlin, Heidelberg: Springer Berlin Heidelberg.
132. Rest, J.v., et al. *Designing privacy-by-design*. in *Annual Privacy Forum*. 2012. Springer.
133. Peyrone, N. and D. Wichadakul, *RUN-ONCO: A Highly Extensible Software Platform for Cancer Precision Medicine*, in *Proceedings of the 2019 6th International Conference on Biomedical and Bioinformatics Engineering*. 2019, Association for Computing Machinery: Shanghai, China. p. 142–147.
134. Merkel, D., *Docker: lightweight Linux containers for consistent development and deployment.* Linux J., 2014. **2014**(239): p. Article 2.
135. Dikaleh, S., O. Sheikh, and C. Felix, *Introduction to kubernetes*, in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. 2017, IBM Corp.: Markham, Ontario, Canada. p. 310.
136. Ramirez, A.O., *Three-Tier Architecture*. Linux J., 2000. **2000**(75es): p. 7–es.
137. Panda, S.K. and S.C. Satapathy. *An Investigation into Smart Contract Deployment on Ethereum Platform Using Web3.js and Solidity Using Blockchain*. 2021. Singapore: Springer Singapore.
138. Group, T.B. *Truffle*. 2020 26 April 2023]; Available from: https://www.trufflesuite.com/truffle.
139. Amine, M., B. Delahaye, and A. Lanoix, *Moving from Event-B to probabilistic Event-B*, in *Proceedings of the Symposium on Applied Computing*. 2017, Association for Computing Machinery: Marrakech, Morocco. p. 1348–1355.

# VITA

**NAME** Neda Peyrone