Context Specification Language for Formally Verifying Consent Properties on Models and Code

Myriam Clouet, Thibaud Antignac, Mathilde Arnaud, Julien Signoles

Context Specification Language for Formally Verifying Consent Properties on Models and Code

Myriam Clouet[1], Thibaud Antignac[⋆2], Mathilde Arnaud[1], and Julien Signoles[1]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
firstname.lastname@cea.fr
[2] CNIL (Commission nationale de l'informatique et des libertés), 3 place de Fontenoy, TSA 80715, 75334 Paris CEDEX 07, France
flastname@cnil.fr

Abstract. Recent privacy laws and regulations raise the stakes in verifying that software systems respect user consent. The current state of the art shows that privacy by design and formal methods can help. Still, ensuring the validity of privacy properties, in particular consent properties, at different stages of software development, is hard. This paper proposes a step towards solving this issue by introducing a new tool, named CASTT, that allows software engineers to verify consent properties at two different development stages: system modeling and code verification. To describe the system, this paper introduces a new formal context specification language named CSpeL, which allows to specify the key elements involved in consent and their relationships. The tool is evaluated on two use cases targeting different application domains: healthcare and website. We also evaluate the correctness and the efficiency of our tool.

Keywords: Privacy · Specification Language · Formal Verification

1 Introduction

Personal data processing occurs in various application domains : website services, voice assistants, or healthcare systems, to name but a few. Many laws and regulations have been established around the world to govern such processing, e.g. GDPR in Europe, the Privacy Act in Australia or the Act on The Protection of Personal Information in Japan. Failure to comply with these laws can be punished by substantial fines, which have been recently applied to Google[3] and WhatsApp[4]. Hence, verifying that a system respects expected privacy properties is crucial.

Formal methods provides a set of techniques based on logic, mathematics, and theoretical computer science used for specifying, developing and verifying software and hardware systems [22]. In particular, it can be used for privacy property verification [33]. Another way to provide privacy guarantees is to follow the *privacy by design* principle, which requires controllers to "both at the time of

⋆ The views, opinions, and positions expressed in this article are those of this author and not of the institution to which he belongs. This work was mostly done while the author was at CEA LIST.

[3] https://www.bbc.com/news/technology-46944696

[4] https://www.bbc.com/news/technology-58422465

the determination of the means for processing and at the time of the processing itself, implement appropriate technical and organizational measures" [15]. In this regard, the controllers have to integrate these measures at early stages of the development [2]. More generally, ensuring compliance of a software system with respect to privacy requires to verify the expected privacy properties expected hold during all the system lifecycle. It usually involves different abstraction levels (corresponding to the development steps in the lifecycle), which complicates the verification process.

Among all privacy properties, consent-related ones are of a particular kind as they relate to an agreement between interested parties concerning the processing of personal data [12]. However, these properties, even if disjointedly taken into account by legal departments, can be ignored at design time and are usually not checked at all at implementation and verification stages, which may lead to serious privacy issues regarding this legal basis.

This paper proposes an approach to verify consent properties at two different development stages, modeling and code verification, as illustrated in Fig. 1. First, a model specification language, named CSpeL, allows engineers to formally specify key system elements with regards to two specific consent properties. Second, this paper introduces a new tool, CASTT, that allows to verify the aforementioned properties on traces from a model (at Model Level), on traces from a program (at Program Level), or directly on a program.

This tool has been applied on use cases from two application domains at both model and program levels. Correctness and efficiency evaluations have been carried out to demonstrate the usefulness of our approach.

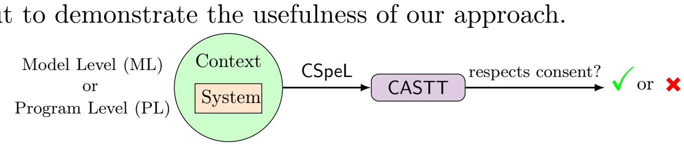

Fig. 1. High-level view of the contributions: CSpeL and CASTT.

More precisely, our contributions are the following:

- CSpeL, a **new formal context specification language** for specifying the key elements involved in two specific consent properties: *purpose compliance*, stating that personal data are only processed for granted purposes, and *necessity compliance*, stating that personal data are only processed when needed.
- CASTT, a **new verification tool** that includes:
  - a **method to verify** purpose and necessity compliance on traces from a model or from a program; and
  - a **translation mechanism** from CSpeL to the ACSL specification language [5] that allows the user to verify the purpose and necessity compliance on C source code;
- an **empirical evaluation** of CASTT on **use cases** from two different application domains, namely healthcare and website, that illustrates the usefulness of the overall approach.

The paper is organized as follows: Section 2 presents the related work while Section 3 introduces a running example used to illustrate our approach. Then, Section 4 details CSpeL, and shows how to use it to specify a system, while Section 5 presents CASTT and the associated verification process. Finally, Section 6 provides the results of the experimental evaluation of CASTT, and Section 7 concludes and discusses future works.

2 Related Work

Several existing approaches allow to verify formal properties at both model and program levels. Among them, the B method [1] allows engineers to specify behaviors of a system in B, and to refine this model iteratively down to a concrete executable model. Conversely, Greenaway et al. [18] propose a tool for abstracting the C semantics into higher-level specifications. However, these approaches target safety properties, expressing how the program is expected to behave. Regarding security properties, existing approaches consider either model or program level. For instance, Bernhard et al. [7] specify a new model of voting protocol satisfying specific formal properties, such as secrecy [29], while Dufay et al. [14] specify a JML-based language and use static analysis to verify a non interference property.

Table 1. Comparison of Consent-related Approaches.

Solution	Formal properties	ML	PL	Language	Tool
[3]	✓	✓	×	unnamed	×
[6]	✓	✓	×	CAPVerDE	CAPVerDE
[31]	✓	✓	×	unnamed	DataProVe
[24]	✓	✓	×	Prolog	Prolog-based
[25]	×	×	✓	unnamed	Poly
[32]	×	×	✓	unnamed	CASTOR
[19]	×	×	✓	OpenAPI	OpenAPI
[21]	×	×	✓	JIF	JIF
[this paper]	✓	✓	✓	CSpeL	CASTT & Frama-C

Table 1 compares works on verifying formal consent-related properties at model level (ML) or program level (PL). Consent properties target why personal data is processed and not who has access to the data and are thus complementary. Some approaches verify consent properties at model level: they rely on smart contracts for blockchains [3], a specific architecture design and verification based on second-order logic [6], a policy language and an architecture description language [31], or logs to verify actions scheduling [24]. Three of those approaches rely on tools: Bavendiek et al. [6], and Ta and Eiza [31] use their own dedicated tool, while de Montety et al. use Prolog [24]. However, none of these approaches check any implementation.

Other approaches propose solutions to ensure consent at Program Level, but they do not target verification at Model Level. Also, they do not formalize the verified consent property, but rather follow some privacy principles, typically the GDPR's *data protection by design and by default* principle [15]. These solutions rely on extending some permission model [25], on a dedicated specification language and static analysis [32], on extending an OpenAPI [19], or on information-flow control [21]. All of them rely on tools: Hayati and Abadi [21] use an existing tool, not initially designed for privacy. Similary, Nauman et al. [25], and Grünewald et al. [19] extend existing tools to tackle privacy concerns. Tokas et al. [32] prefer to implement a dedicated tool from scratch.

To sum up, our solution is the only one that targets both model and program levels for verifying consent-related properties. It is also the only one that allows to verify a formally-specified consent property at program level.

3 Running Example

This section presents our running example, which is adapted from an example of Petkovic et al. [26]. This example introduces a hospital information system that processes patients' data, named EPR (for **E**lectronic **P**atient **R**ecord). Each EPR contains some personal data (e.g., date of birth), and some non-personal data (e.g., drug dosages). The medical staff may use the hospital information system to process EPRs for two different purposes: providing treatment to patients (**Treatment**) or performing a clinical trial (**Research**). In both cases, doctors should ask for an access to patients' personal data at some point during the treatment or the clinical trial when they need them.

In the following, Section 3.1 introduces a model of this system, Section 3.2 the key implementation elements, and Section 3.3 the goals that we aim to achieve.

3.1 Model of the Hospital Information System

At model level, we use BPMN [10] to model the hospital information system: Fig. 2 defines a process **P1** corresponding to purpose **Treatment**, while Fig. 3 defines another process **P2** corresponding to purpose **Research**. Each BPMN process P_i contains a start element S_i, a final element E_i, and different tasks T_{ij}, executed sequentially one after the other. Some tasks use EPR when executed.

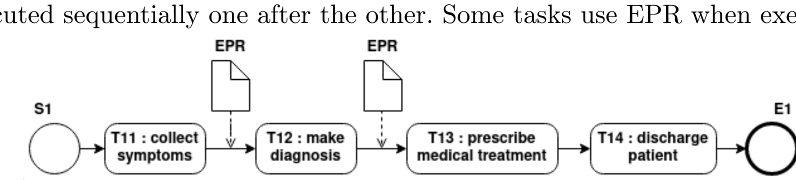

Fig. 2. Healthcare system at ML - Process P1: Treatment.

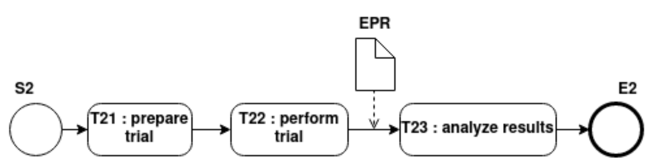

Fig. 3. Healthcare system at ML - Process P2: Research.

Process **P1** contains four tasks. Task **T11** collects the patient's symptoms, which are part of her **EPR**. Then, using this data, Task **T12** makes a diagnosis, which also uses the patient's **EPR**. Next, Task **T13** prescribes a medical treatment. Finally, Task **T14** corresponds to the patient's discharge.

Process **P2** only contains three tasks. Task **T21** prepares the trial, which is then performed by Task **T22**. This latter uses the patient's **EPR** for producing statistics. Finally, Task **T23** analyzes the results.

3.2 Implementation of the Information System

Fig. 4 introduces some key elements of a C implementation of the hospital information system, [5] while Table 3.2 shows the relationships between the BPMN models and the code, as explained below.

```
/*-- EPR Datatype --*/
typedef struct {char date[SIZE]; char medicine[SIZE];} Dos;
typedef struct { int id; char name[SIZE]; char birthdate[SIZE];
    Dos dosList[NB_D];...} Patient;
typedef struct { char birthdateList[NB][SIZE];
    char sexList[NB][SIZE]; Dos dosList[NB_D];
} TrialData ;

/*-- Processes' Tasks --*/
Patient makePrescr(Dos newd, Patient p) { ... }
TrialData getData(Patient patientList[NB]) { ... }
int computeStats(TrialData data) { ... }

/*-- Testing the system --*/
int main() {
    ...
    patient = makePrescr(newd, patient);
    TrialData d = getData(list1);
    int res = computeStats(d);
    return 0;
}
```

Fig. 4. Code snippet of the healthcare system.

Table 2. Mapping between the BPMN model and the code.

ML	PL
P1	`Patient makePrescr(Dos newd, Patient p)`
P2	`TrialData getData(Patient patientList[NB_PATIENT])` `int computeStats(TrialData data)`
EPR	`Patient`, `TrialData`, `Dos`

As seen in the previous section, BPMN processes **P1** and **P2** perform different tasks, possibly handling EPR. Some of these tasks and the associated EPR processing are manually executed (typically, by a doctor) and are thus not present in the code. For instance, preparing the clinical study (Task **T21**), or discharging the patient (Task **T14**) are manual unimplemented actions. The other tasks and associated EPR processing are implemented by C functions: task **T13** is implemented by the function `makePrescr` while task **T23** is implemented by the function `computeStats` and uses the necessary pieces of EPRs provided by the function `getData` (from **T22**). Therefore, we can consider that the whole process **P1** is represented at code level by the single function `makePrescr`, while the whole process **P2** is represented by the pair of functions `getData` and `computeStats`. Some tasks handle EPRs. At code level, an EPR is implemented by three C data structures, namely `Dos`, `Patient`, and `TrialData`. `Dos` represents the drug dosage. By itself, it does not contain any personal data`Patient` represents a patient. It contains personal data such as the name or the date of birth, and the list of prescribed treatments. `TrialData` represents data used for

[5] Note to the reviewers: https://julien-signoles.fr/castt/docker.html contains instructions for downloading and running a Docker image with the implementation of our tool CASTT and the examples presented in this paper.

statistical computations during the clinical trial. It also contains personal data such as collections of dates of birth and genders, and medical prescriptions.

We want to verify if a system execution respects the consent of the data subjects: to this end we will verify consent properties on traces representing processes executions.

3.3 Goal

In the following, we assume that the BPMN model can be simulated to generate execution traces. Whether the code is executable only impacts the usable code-level verification methods, as explained in Section 5.2 and it is not mandatory. In this paper, our goal consists in checking whether the hospital information system processes EPRs according to each patient's consent at both model and program levels. Checking consent properties at both levels is necessary. Indeed, some invalid actions may relate to manual tasks that are only represented in the model. On the other hand, invalid actions can also be introduced during the implementation stage.

Section 4 explains how to describe the hospital information system at model and implementation stages with the same language, named CSpeL. Then, Section 5 introduces a tool, named CASTT, that allows to verify a consent property on models execution traces (Section 5.1) and on a C implementation (Section 5.2).

4 Specifying a System with CSpeL

CSpeL is a language that allows to formally specify both a system in a privacy context and an execution trace, in order to verify consent properties. Section 4.1 presents how to formally model a system. and Section 4.2 how to represent a system execution. Next, Section 4.3 formalizes the consent properties verified in our approach. Finally, Section 4.4 introduces the CSpeL grammar used in practice.

4.1 Model of a System

In our setting, the model of a system is defined through the notion of context, formally defined as follows. In the following, we use $\top$ (resp. $\bot$) to denote the Boolean value "true" (resp. "false").

Definition 1 (Context). *A context $\mathcal{C}$ is a 6-tuple $(S, D, P, \gamma, \pi, \nu)$ where S is a set of processes, D a set of personal data, and P a set of purposes.*

The total function $\gamma \triangleq D \times P \rightarrow bool$ represents the data subject consent for the use of each piece of personal data, for each purpose. The total functions $\pi \triangleq S \rightarrow \mathcal{P}(P)$ and $\nu \triangleq S \rightarrow \mathcal{P}(D)$ return, for each process of the system, the purposes of the process, and the personal data needed respectively.

Example 1. The BPMN model introduced in Section 3.1 can be represented by the following context:

$$
\begin{array}{ll}
S \triangleq \{P1; P2\}; & \pi \triangleq P1 \mapsto \{Treatment\} \\
D \triangleq \{EPR\}; & \quad\; P2 \mapsto \{Research\}; \\
P \triangleq \{Treatment; Research\}; & \nu \triangleq P1 \mapsto \{EPR\} \\
\gamma \triangleq (EPR, Treatment) \mapsto \top & \quad\; P2 \mapsto \{EPR\}; \\
\quad\; (EPR, Research) \mapsto \bot; &
\end{array}
$$

In this instance, the user consented to the use of his personal data for Treatment purposes only. Similarly, we can define a context representing the system at code level introduced in Section 3.2:

$$
\begin{array}{ll}
S \triangleq \{\texttt{makePrescr}; \texttt{getData}; \texttt{computeStats}\}; & \pi \triangleq \texttt{makePrescr} \mapsto \{Treatment\} \\
D \triangleq \{Patient, TrialData\}; & \quad \texttt{getData} \mapsto \{Research\} \\
P \triangleq \{Treatment; Research\}; & \quad \texttt{computeStats} \mapsto \{Research\}; \\
\gamma \triangleq (Patient, Treatment) \mapsto \top & \nu \triangleq \texttt{makePrescr} \mapsto \{Patient\} \\
\quad (Patient, Research) \mapsto \bot; & \quad \texttt{getData} \mapsto \{Patient, TrialData\} \\
\quad (TrialData, Treatment) \mapsto \bot; & \quad \texttt{computeStats} \mapsto \{TrialData\}. \\
\quad (TrialData, Research) \mapsto \bot; &
\end{array}
$$

Our formalism allows to model the necessary elements for verifying the desired consent properties. In particular, modeling purposes apart from processes is important to accurately verify consent. Indeed, user consent is defined via purposes, but there is no one-to-one correspondence between processes and purposes. As processes are the entities handling personal data, they are the ones to be verified, thus the need for function π.

4.2 Execution Traces

We check system behavior w.r.t consent properties through trace analysis. This section introduces the notion of traces, while the consent properties will be defined in Section 4.3. Our traces are abstract enough to be generated either from a model (usually, by simulation) or from a program run, yet expressive enough to allow us to formally specify consent properties.

Definition 2 (Execution Trace). *Let $\mathcal{C} = (S, D, P, \gamma, \pi, \nu)$ be a context, $\{\sigma_i\}_{i\in\mathbb{N}} \subseteq S$, and $\{d_i\}_{i\in\mathbb{N}}$ be a set of (personal or non personal) data. The execution traces of $\mathcal{C}$ are defined by the following grammar:*

$$T ::= \epsilon \mid Handle(\sigma_i, d_i); T.$$

Trace ϵ is the empty trace, while event $Handle(\sigma_i, d_i); T$ indicates that σ_i processes data d_i.

It is worth noting that data in execution traces can be personal or non-personal. Also, only one piece of data at a time is processed in each event.

Example 2. Consider the BPMN model introduced in Section 3.1. The trace $Handle(P1, EPR); \epsilon$ (resp. $Handle(P2, EPR); \epsilon$) denotes the handling of EPR by process $P1$ (resp. $P2$), while the trace $Handle(P1, EPR); Handle(P2, EPR); \epsilon$ denotes the handling of EPR by process $P1$ followed by the handling of EPR by process $P2$. At code level, the trace

$$Handle(\texttt{makePrescr}, Patient); Handle(\texttt{makePrescr}, Dos); \epsilon$$

denotes the execution of function `makePrescr` on a patient for delivering some prescription.As function `makePrescr` processes two pieces of data, namely $Patient$ and Dos, the trace is composed with two events. Similarly,

$$Handle(\texttt{getData}, Patient); Handle(\texttt{getData}, TrialData);$$
$$Handle(\texttt{computeStats}, TrialData); \quad \epsilon$$

denotes the execution of function `getData`, with two distinct events as two pieces of data are processed, before computing statistics with `computeStats`.

4.3 Consent Properties

Our goal consists in verifying that an execution trace T respects a consent property $Prop$ in a context $\mathcal{C}$, noted $C, Prop \vdash T$. Consent refers to many different yet related notions [30]. In this paper, we focus on the notions of *purpose* and *data necessity*. More precisely, when some personal data is processed, we would like to check that the data subject agreed to at least one of the process's purposes and that the personal data is necessary to the process. We formally express these properties through the notions of *purpose compliance* and *necessity compliance*.

Definition 3 (Purpose Compliance). *An event of data processing $Handle(\sigma, d)$ is purpose-compliant with respect to a context $\mathcal{C} \triangleq (S, D, P, \gamma, \pi, \nu)$ if and only if consent was granted to at least one of the process' purposes, i.e.:*

$$\mathcal{C}, Purpose_{Comp} \vdash Handle(\sigma, d) \iff \begin{cases} d \notin D & ;\ or \\ \exists p \in \pi(\sigma), \gamma(d, p) = \top. \end{cases}$$

Definition 4 (Necessity Compliance). *An event of data processing $Handle(\sigma, d)$ is necessity-compliant with respect to a context $\mathcal{C} \triangleq (S, D, P, \gamma, \pi, \nu)$ if and only if the personal data is necessary for the processing, i.e.:*

$$\mathcal{C}, Necessity_{Comp} \vdash Handle(\sigma, d) \iff \begin{cases} d \notin D & ;\ or \\ d \in \nu(\sigma). \end{cases}$$

These notions of *purpose compliance* and *necessity compliance* are extended to execution traces thanks to the inference rules given in Fig. 5. The empty trace is *purpose-compliant* (resp. *necessity-compliant*) with respect to any context, while a non-empty trace is *purpose-compliant* (resp. *necessity-compliant*) if and only if all its events are *purpose-compliant* (resp. *necessity-compliant*).

$$\frac{}{\mathcal{C}, Prop \vdash \epsilon} \qquad \frac{\mathcal{C}, Prop \vdash Handle(\sigma, d) \quad \mathcal{C}, Prop \vdash T}{\mathcal{C}, Prop \vdash Handle(\sigma, d); T}$$

with $Prop \in \{Purpose_{Comp}; Necessity_{Comp}\}$

Fig. 5. Trace Consent Compliance.

Example 3. In our running example, at model level, consent was granted to process EPR for purpose $Treatment$ but not $Research$. The purpose of $P1$ is $Treatment$ and the purpose of $P2$ is $Research$. Thus $Handle(P1, EPR); \epsilon$ is *purpose-compliant* and $Handle(P2, EPR); \epsilon$ is not *purpose-compliant* [6]. As EPR is needed for $P1$ and for $P2$, both of these traces are *necessity-compliant*.

At program level, the trace

$$Handle(\texttt{makePrescr}, Patient); Handle(\texttt{makePrescr}, Dos); \epsilon$$

is *purpose-compliant*, because consent was granted for the processing of $Patient$ for purpose $Treatment$ associated with `makePrescr` and Dos is not a personal data. However the trace

$$Handle(\texttt{getData}, Patient); Handle(\texttt{getData}, TrialData);$$
$$Handle(\texttt{computeStats}, TrialData); \quad \epsilon$$

[6] Proof of this claim and the following ones are in Appendix B.

is not *purpose-compliant*, because the purpose of `getData` is *Research* and consent for the use of *Patient* (or *TrialData*) is not granted for this purpose.

As `makePrescr` needs *Patient*, the first trace is *necessity-compliant*. Similarly, `getData` needs *Patient*, *TrialData*, and `computeStats` needs *TrialData*, thus the second trace is also *necessity-compliant*.

4.4 Concrete Language

The formalism introduced so far allows to define the generic notions of *purpose compliance* and *necessity compliance* for any executable system. However, since these notions over the system are specified by a quite abstract notion of context, they are not convenient for working engineers. To circumvent this issue, this section introduces the practical language CSpeL linking these notions to executable systems. It can be used either at model level, or at program level.

Fig. 6 gives the formal syntax of CSpeL. Literals d, σ, and p are strings that respectively denote a data element, a process and a purpose. A CSpeL model M is a context C, possibly followed by a trace T.

$$
\begin{array}{rcl}
M & ::= & C \mid CT \\
C & ::= & \texttt{\textbackslash context}\{PR,PD,PU,G,HP,N,IS\} \\
  &   & \mid \texttt{\textbackslash context}\{PR,PD,PU,G,HP,N\} \\
PR & ::= & \texttt{\textbackslash process}\{\mathit{\Sigma Set}\} \\
PD & ::= & \texttt{\textbackslash personalData}\{\mathit{DSet}\} \\
PU & ::= & \texttt{\textbackslash purposes}\{\mathit{PSet}\} \\
G & ::= & \texttt{\textbackslash isGranted}\{\mathit{TSet}\} \\
HP & ::= & \texttt{\textbackslash hasPurposes}\{\mathit{APSet}\} \\
N & ::= & \texttt{\textbackslash needData}\{\mathit{ADSet}\} \\
IS & ::= & \texttt{\textbackslash init}\{\mathit{\Sigma Set}\} \\
T & ::= & \texttt{\textbackslash trace}\ \{\ \mathit{PCSet}\ \}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{\Sigma Set} & ::= & \sigma \mid \sigma,\mathit{\Sigma Set} \\
\mathit{DSet} & ::= & \mathrm{d} \mid \mathrm{d},\ \mathit{DSet} \\
\mathit{PSet} & ::= & \mathrm{p} \mid \mathrm{p},\ \mathit{PSet} \\
\mathit{TSet} & ::= & (\mathrm{d{:}p}) \mid (\mathrm{d{:}p}),\ \mathit{TSet} \\
\mathit{APSet} & ::= & (\sigma{:}\{\mathit{PSet}\}) \mid (\sigma{:}\{\mathit{PSet}\}),\ \mathit{APSet} \\
\mathit{ADSet} & ::= & (\sigma{:}\{\mathit{DSet}\}) \mid (\sigma{:}\{\mathit{DSet}\}),\ \mathit{ADSet} \\
\mathit{PCSet} & ::= & \texttt{\textbackslash handle}(\sigma,\ \mathrm{d});\ \mathit{PCSet} \mid \texttt{VOID}
\end{array}
$$

Fig. 6. CSpeL grammar.

A context (keyword `\context`), contains elements matching those in Definition 1: PR for S, PD for D, PU for P, G for γ, HP for π, and N for ν. It may also contain a process set to specify where the elements are initialized, IS.

The set S (resp. D, and P) of processes is introduced by the keyword `\process` (resp. `\personalData`, and `\purposes`). Similarly, the total function γ (resp. π, and ν) is introduced by the keyword `\isGranted` (resp. `\hasPurposes`, and `\needData`). Each keyword allows to map elements from the function's domain to elements from the function's co-domain. Currently, there is no check that the functions defined in this way are total.

A trace is introduced by the keyword `\trace`. It is a sequence *PCSet* of data processing. The empty sequence is `VOID`. An event of data processing in a trace is introduced by the keyword `\handle` and associates a process to a data.

Example 4. The model defined in Example 1 could be specified in CSpeL as:

```
\model{\context{\process      { P1; P2 },
                \personalData { EPR },
                \purposes     { Treatment; Research },
                \isGranted    { (EPR:Treatment) },
                \hasPurposes  { (P1: { Treatment }),(P2: { Research })},
                \needData     { (P1: { EPR }),(P2: { EPR }) }},
      \trace{\handle(P1,EPR); VOID }}
```

As we have seen, CSpeL allows to formally specify both a system in a privacy context and an execution trace, independently of its level of abstraction (Model or Program). Thanks to this, we can easily verify consent properties.

5 Verifying Consent with CASTT

We develop the CASTT tool as a Frama-C plug-in in order to verify consent properties from a specification written in CSpeL. Frama-C [4] is an open source extensible analysis platform for C code. It provides many plug-ins for analyzing C source code extended with formal annotations written in the ACSL specification language [5]. Its three main verification plug-ins are E-ACSL [28], Eva [9] and WP [8]. E-ACSL is a runtime assertion checker [11] that verifies ACSL properties during concrete program runs, Eva is a static tool based on abstract interpretation [27] that raises alarms on any potential undefined behavior and invalid ACSL property, and WP relies on deductive methods [20] for proving ACSL properties thanks to associated provers, such as Alt-Ergo [13]. As explained later, we use all of them on our case studies, together with CASTT.

CASTT can be used in two ways: to check either a trace written in CSpeL, or a C code w.r.t. a CSpeL file. The first usage targets offline runtime verification [16] of traces representing system executions at model or code level. The second one specifically targets verification of C code, either statically or dynamically.

First, Section 5.1 details CASTT's offline runtime verification. Then, Section 5.2 explains CASTT's translation to ACSL.

5.1 CSpeL Offline Runtime Verification

Offline runtime verification allows to verify properties on complete system executions (e.g. traces or logs). As shown in Fig.7, CASTT can verify that some specific trace of a system, described in CSpeL, satisfies the consent properties expressed in Definition 3 and in Section 4.3. Currently, the traces are manually written, but they could be automatically generated from a simulated model or from concrete program runs.

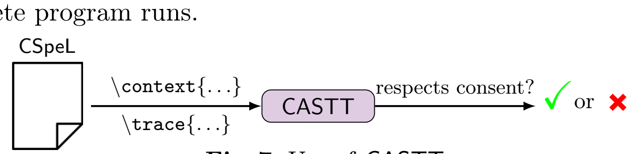

Fig. 7. Use of CASTT.

Algorithm 1 illustrates trace analysis in order to reach this goal. It is implemented in CASTT. For each event in the trace, it verifies whether the data being processed is in the personal data set. If so, the algorithm checks whether the data subject previously agreed to one of the purposes associated to the process: the trace is invalid if there is no such agreement. It also checks whether the data is necessary to the processing: the trace is invalid if the data is not necessary. The evaluation continues until all events have been checked. The trace is valid only if all events are valid. For completeness, we do not stop the algorithm at the first invalid event. The complexity of this algorithm is linear.

Algorithm 1: Trace Evaluation

```
Input: A context and a trace
Output: Evaluation result
is_valid = True ; // void trace is valid
while trace is not void do
    (process, data) ← trace.current event;
    trace ← trace.next();
    if data ∈ context.personal data then
        is_consented = False ; // purpose compliance
        foreach purpose ∈ context.hasPurposes(process) do
            if context.isGranted(data, purpose) then
                is_consented = True ;
        is_necessary = False ; // necessity compliance
        if data ∈ context.needData(process) then
            is_necessary = True ;
        is_valid = is_valid && is_consented && is_necessary
return is_valid;
```

5.2 Consent Verification on C Source Code

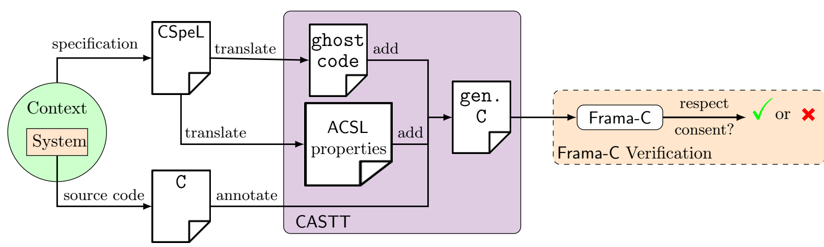

Fig. 8. Functional View of CASTT for Verifying Consent on C Code.

Fig. 8 shows the functional view of CASTT, together with Frama-C, for verifying our properties defined in Section 4.3 on a C source code. CASTT takes as inputs a consent specification written in CSpeL and a C source code in order to generate a new C source code extended with ACSL annotations that encode the CSpeL specification. Then, any Frama-C analyzer can be used to verify the ACSL annotations embedded in this generated code. Verifying all of them implies that the original properties are satisfied. In practice, the user can rely on E-ACSL, Eva, WP, or a combination of them, for verifying the generated code.

We do not detail how the code and the ACSL annotations are generated from a given CSpeL file and a C source code: we just give a few insights, in Table 3. The sets of personal data and purposes, respectively specified by `\personalData` and `\purposes`, are translated to enumeration types in the generated code. Based on these sets, CASTT generates a matrix `Consent` that specifies, for each personal data, for which purposes the data subject has granted consent. This matrix is declared as ghost code, which is a set of stateful ACSL annotations that do not interfere with the user's C code: ghost code cannot modify the state of the original program [17]. The command `\isGranted` is used to generate ghost statements that initialize this ghost matrix in the processes specified with `\init`. Similarly the matrix `Need` is initialized thanks to the command `\needData`. This matrix specifies which data are necessary for each process.

Table 3. Code Generation Snippets.

CSpeL Definition	Generated Code Snippet
\personalData	`enum _PersonalData {TRIALDATA = 0, PATIENT = 1}` `typedef enum _PersonalData PersonalData;`
\purposes	`enum _Purposes {RESEARCH = 0, TREATMENT = 1}` `typedef enum _Purposes Purposes;`
\personalData & \purposes	`/*@ ghost bool Consent[2][2];*/`
\process & \personalData	`/*@ ghost bool Need[3][2];*/`
\personalData & \purposes & \isGranted & \init	`/*@ ghost Consent[PATIENT][RESEARCH] = 0; */` `/*@ ghost Consent[PATIENT][TREATMENT] = 1; */` `/*@ ghost Consent[TRIALDATA][RESEARCH] = 0; */` `/*@ ghost Consent[TRIALDATA][TREATMENT] = 0; */`
\process & \personalData & \needData & \init	`/*@ ghost Need[MAKEPRESCR][PATIENT] = 1; */` `/*@ ghost Need[MAKEPRESCR][TRIALDATA] = 0; */` `/*@ ghost Need[GETDATA][PATIENT] = 1; */` `/*@ ghost Need[GETDATA][TRIALDATA] = 1; */`
\personalData & \hasPurposes	`/*@ assert Consent[PATIENT][TREATMENT] == 1; */`
\personalData	`/*@ assert Need[MAKEPRESCR][PATIENT] == 1; */`

For each statement in the program functions (i.e., process), if it corresponds to processing of a personal data, ACSL `assert` clauses, that correspond to our properties, are generated. For *purpose compliance*, the clause checks that consent was granted to process this data for at least one of the function purposes. For *necessity compliance*,the clause checks that the data is necessary to this function. These are defined by the user in the CSpeL file through \personalData and \hasPurposes. Thereafter, for any ACSL annotation `/*@ assert Consent[`d`][`p`]` $\equiv 1$`;*/` (resp. `/*@ assert Need[`σ`][`d`]` $\equiv 1$`;*/`) with d a personal data, p a purpose and σ a function name, the used Frama-C verification plug-in(s) will try to check that these properties are satisfied. This way, it ensures that the user agreed to the processing of data d for purpose p (resp. that the data d is necessary for the function σ).

6 Experimentation and Evaluation

This section presents our experimentation for evaluating our tool CASTT. First, Section 6.1 presents our use cases. Then, Section 6.2 (resp. 6.3) evaluates CASTT's trace analysis (resp. verification process at code level) on these use cases. The first use case is the healthcare running example already introduced in Section 3. The second use case is a website system, focusing on two purposes: keeping track of purchases and targeted advertising. For each use case and each abstraction level (model or code), a file specifying the CSpeL context is provided as input to CASTT. At code level, the use case's source code is also given as input. Even if our evaluation is still preliminary, it allows us to come to a few positive conclusions. Future work includes extending our evaluation to larger examples.

The evaluations were performed on a PC with a 2 GHz Intel Xeon CPU and 32 GB of RAM. We used CASTT with the public development version of Frama-C[7] as of 9/29/2022 (git commit 3c453a2b).

[7] https://git.frama-c.com/pub/frama-c

Our evaluation relies on a home-made shell script executing the necessary commands for running CASTT and the Frama-C verification tools, as well as a Python script for generating the graphics presented in this Section. We have also implemented a trace and a function call generators that allows us to evaluate our tool on examples containing traces with up to $1,000,000$ events, and programs with up to $3,000$ function calls. [8]

6.1 Examples used

We successfully executed our analysis using CASTT, both at model and program level, on the running example presented in Section 3. All the valid and invalid executions were detected.

Our running example Healthcare is a simplified version of the example of Petkovic et al. [26]. For our offline runtime verification evaluation we use the complete version, called **Purpose Control** in the following, that is more complex (with pools containing various tasks, events, conditional branches and message transfers), using the trace verification functionality of CASTT.

Since they do not provide a C implementation, we have implemented another use case for concerning a different application domain (website) for evaluating the CASTT's code verification functionality. In this use case, some functions have various purposes (and not just one as the previous example).

6.2 Offline Runtime Verification Evaluation

We evaluate CASTT's offline runtime verification with the following Research Questions in mind:

RQ1 *Can CASTT verify a consent property on a trace from a model?*
RQ2 *Can CASTT verify a consent property on a trace from a program?*
RQ3 *Can CASTT detect invalid traces*?
RQ4 *Is CASTT usable on large traces*?

For answering the questions **RQ1**, **RQ2**,**RQ3** we run a correctness script using CASTT. This script executes the following command on various CSpeL files, corresponding to various applicative domains and different levels (ML and PL): `frama-c -castt-verify-trace -castt-consent-file <file.cspel>` where `file.cspel` is the name of the test file.

Table 4. Experimental Results for CASTT's Trace Analysis.

Example	**LVL**	**NbTests**	**Size of traces**	**Valid traces detected**	**Invalid traces detected**
Healthcare	ML	6	1 to 3 events	✓	✓
	PL	6	2 to 6 events	✓	✓
Website	ML	9	1 to 5 events	✓	✓
	PL	9	5 to 24 events	✓	✓
Purpose Control	ML	10	1 to 20 events	✓	✓

Each experiment instantaneously (i.e., in less than a second) provides its results, which are summarized in Table 4. This table presents, for each use case and each abstraction level, the number of analyzed CSpeL files, the minimum

[8] All the resources to run our experimentation are available in CASTT repository.

and maximum number of events in the trace, and whether the validity statuses were correctly detected. Our experiments include as many valid traces as invalid traces. These results demonstrate that CASTT always provides the expected verdict on our examples, both at model and program level. Therefore, we can positively answer Research Questions **RQ1**, **RQ2**, and **RQ3**.

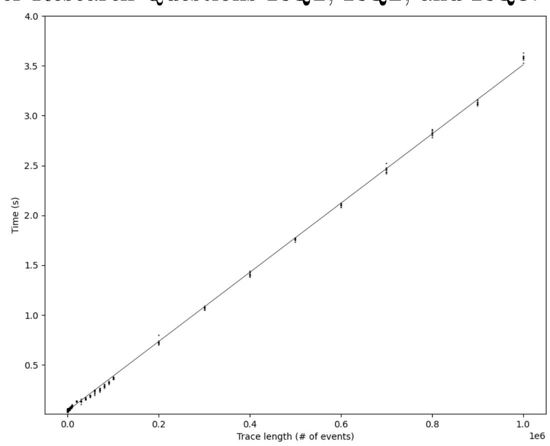

Fig. 9. Time for the trace verification by the size of the analyzed trace

To answer Question **RQ4**, we run a script measuring time efficiency. This script executes the previous command on various CSpeL files. These files correspond to one use case, but with various sizes of generated trace (containing from 10 to $1,000,000$ events). The script executes the verification 10 times for the same size and calculates the mean of the verification time. The results are shown on Fig. 9. It verifies that the trace verification algorithm is time linear. It also shows that large traces, i.e. with $1,000,000$ events, are verified by CASTT in less than 4 seconds. Therefore, we can positively answer **RQ4**.

6.3 Translation Mechanism Evaluation

We evaluate the translation mechanism of CASTT with the following Research Questions in mind:

RQ5 *Can CASTT translate a CSpeL file for a PL tool (ex: Frama-C)?*
RQ6 *Can CASTT be used to verify systems at PL?*
RQ7 *Can CASTT be used to detect invalid traces w.r.t. purpose compliance*?
RQ8 *Does CASTT reduce the number of hard-written specifications?*
RQ9 *Is CASTT usable on large code (i.e. with many lines and function calls)?*

To answer the questions **RQ5**, **RQ6**, **RQ7**, we run a correctness script. This script executes the following commands: `frama-c <source_file.c> -castt-annotate -castt-consent-file <context.cspel> -then-last -print -ocode <annotated_file.c>` and then `frama-c -<analyzer> <annotated_file.c>` on various use cases. Here, option `-analyzer` is either `-eva` for running Eva or `-wp` for running WP. We also monitor the code generated by CASTT with E-ACSL for dynamic verification. In this case, we run: `e-acsl-gcc.sh -c <annotated_file.c> -O <monitored_binary>` and then `./<monitored_binary>.e-acsl`

Table 5. Frama-C Code Verification Experimental Results.

Plug-in	Expected Result	Meaning	Evaluation
WP	all goals are proved	Valid	✓
Eva	all assertions are valid		✓
E-ACSL	no error is raised at runtime		✓
WP	some goals are not proved	Invalid	✓
Eva	some assertions are invalid		✓
E-ACSL	an error is raised at runtime		✓

All experiments instantaneously provide their results, which are summarized in Tables 5 and 6. The first table shows, for each Frama-C plug-in, its expected meaning of this result for our approach, whether this result is compliant with our expectations. For this evaluation with Eva and WP, we manually check the results in the Frama-C GUI. For E-ACSL, the raised error at runtime specifies which ACSL annotation is not satisfied at runtime. Our experimentation shows that CASTT, combined with any of the three main Frama-C's verification plug-ins, can successfully verify consent compliance of the provided code. Therefore, we can positively answer Research Questions **RQ4**, **RQ5**, **RQ6**, and **RQ7**. Table 6 presents, for each test case, the number of lines of code in the original source file, the number of lines generated by CASTT, the number of lines needed for WP, and the length of the CSpeL model. An example of CASTT generated file is given in Appendix A. As shown in the Table, in our examples, CSpeL files are used to generate files 3 to 5 times their size in lines. These generated lines amount to 60% to 75% of the source file. Thus, we can positively answer **RQ8**.

Table 6. CASTT's Code Generation Experimental Results.

Example	Healthcare			Website			
	T1	T2	T3	T1	T2	T3	
# **original lines of code**	53	54	50	121	122	121	60-75%
# **generated lines by CASTT**	33	35	33	92	94	90	
# **lines in CSpeL file**	10			18			× 3-5

To answer Question **RQ9**, we run a script measuring efficiency. This script executes the previous commands on a same C source file (except for the number of function calls) and a same CSpeL file. We use a function call generator to increase the number of function calls in the main function of the source file. Because the WP plug-in is not designed to managed this kind of test, we do not include it in our results (WP is modular and does not depend on the main).

Figure 10 shows our results obtained from files containing between 10 to 100, 000 function calls. For each size, we execute the test 10 times to calculate the time mean. We compute the time for the verification with and without the annotations generated by CASTT, to calculate the overhead generated by our approach. This evaluation shows that the time for CASTT annotation generation is negligible compared to the time of the verification plug-ins. It also shows that our generated annotations do not slow down too much the verification process (usually less than 10% for Eva and usually less than 5% for E-ACSL), and the bigger the size of the original program the smaller the overhead. In particular, our evaluation includes a code with 1, 000, 000 function calls. In this case, the overhead for Eva is about 1.6%, while it is 0.50% for E-ACSL. CASTT runs much faster than Eva or E-ACSL. In particular, it is always at least 5 times faster as soon as you exceed 5, 000 events. Therefore, we can positively answer **RQ9**.

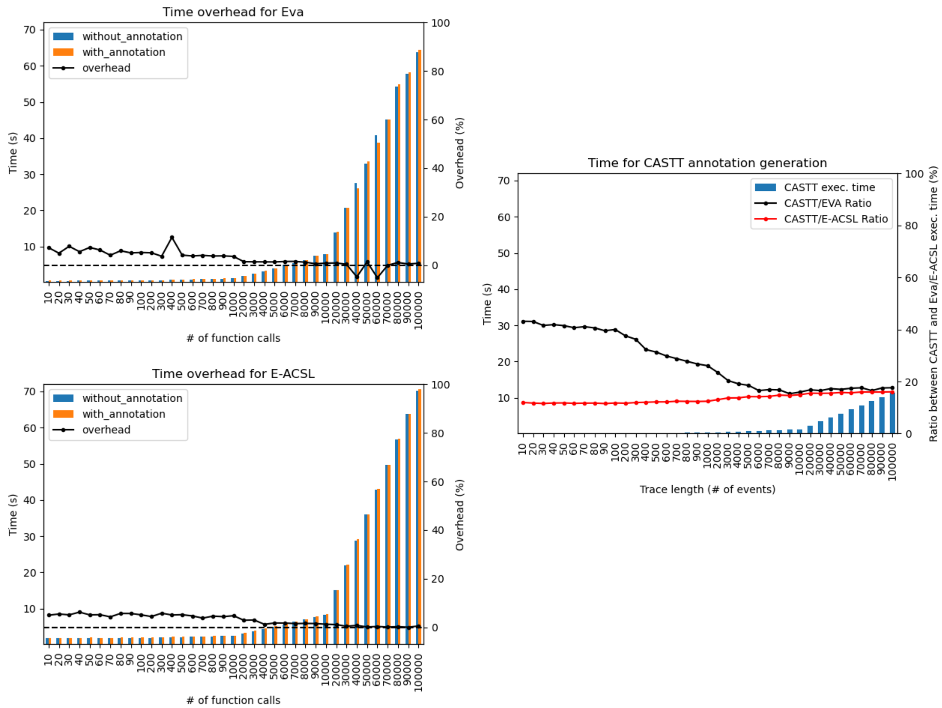

Fig. 10. Execution time by number of function calls

7 Conclusion and Perspectives

This paper presents two consent properties, *purpose compliance* and *necessity compliance*, and the CSpeL context specification language that formally describes systems targeting these properties. CSpeL is used by its companion tool, CASTT, in order to verify these properties at both model and program levels.

CASTT can be used to check these consent properties either for some given execution traces, with an ad-hoc offline runtime verification algorithm-based verifier, or for a C source code. Since CASTT is based on Frama-C, it benefits from existing Frama-C verification plug-ins, such as E-ACSL, Eva, and WP. We have evaluated our tool on two use cases, healthcare and website, at both model and code levels. CASTT is able to successfully verify the valid examples and detect the invalid ones. We have also evaluated our tool on large traces and large code. CASTT is able to handle traces with $1,000,000$ events in less than 4s, and adds a small overhead during the verification process using the Frama-C verification-based tool even on code with more than $100,000$ function calls. The current version of CASTT translates CSpeL to C code. A similar translation could be defined towards other mainstream programming languages for which an ACSL-like specification language exists, such as Java with JML annotations, or towards models (targetting for instance IAT [23], a tool for verifying executions of distributed systems). Another research direction consists in extending CSpeL and CASTT for specifying and verifying other consent and privacy properties such as consent evolution, or storage limitation. CSpeL could also be extended to allow users to define their own properties of interest depending on their particular use.

References

1. Abrial, J.R.: The B-Book, assigning programs to meaning. Cambridge University Press (1996)
2. Ahmadian, A.: Model-based privacy by design. Phd thesis, Universität Koblenz-Landau (2020)
3. Barati, M., Rana, O., Petri, I., Theodorakopoulos, G.: Gdpr compliance verification in internet of things. IEEE Access (2020)
4. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., et al.: The dogged pursuit of bug-free c programs: the frama-c software analysis platform. Communications of the ACM (2021)
5. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Tech. rep.
6. Bavendiek, K., Mueller, T., Wittner, F., Schwaneberg, T., Behrendt, C.A., Schulz, W., Federrath, H., Schupp, S.: Automatically proving purpose limitation in software architectures. In: IFIP Int. Conf. on ICT Systems Security and Privacy Protection. pp. 345–358. Springer (2019)
7. Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: Sok: A comprehensive analysis of game-based ballot privacy definitions. In: 2015 IEEE Symp. on Security and Privacy. pp. 499–516. IEEE (2015)
8. Blanchard, A.: Introduction to C program proof with Frama-C and its WP plugin. Tutorial (2020)
9. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Int. Conf. on Verification, Model Checking, and Abstract Interpretation. Springer (2017)
10. C., M., T., A.: Bpmn: An introduction to the standard. Computer Standards & Interfaces (2012)
11. Clarke, L., Rosenblum, D.: A historical perspective on runtime assertion checking in software development. SIGSOFT Softw. Eng. Notes (2006)
12. Clouet, M., Antignac, T., Arnaud, M., Pedroza, G., Signoles, J.: A new generic representation for modeling privacy. In: Int. Workshop on Privacy Engineering (IWPE'22) (2022)
13. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)
14. Dufay, G., Felty, A., Matwin, S.: Privacy-sensitive information flow with jml. In: Int. Conf. on Automated Deduction (2005)
15. European Commission: Regulation (EU) 2016/679 (General Data Protection Regulation). Tech. rep. (2016), https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679
16. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Engineering dependable software systems pp. 141–175 (2013)
17. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design (2016)
18. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of c (2012)
19. Grünewald, E., Wille, P., Pallas, F., Borges, M., Ulbricht, M.: Tira: An openapi extension and toolbox for gdpr transparency in restful architectures. arXiv preprint arXiv:2106.06001 (2021)

20. Hähnle, R., Huisman, M.: Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. Springer International Publishing (2019)
21. Hayati, K., Abadi, M.: Language-based enforcement of privacy policies. In: Int. Workshop on Privacy Enhancing Technologies (2004)
22. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press; 2nd edition (2004)
23. Mahe, E.: An operational semantics of interactions for verifying partially observed executions of distributed systems. Phd thesis, Université Paris-Saclay (2021)
24. de Montety, C., Antignac, T., Slim, C.: Gdpr modelling for log-based compliance checking. In: Meng, W., Cofta, P., Jensen, C., Grandison, T. (eds.) Trust Management XIII (2019)
25. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM symposium on information, computer and communications security (2010)
26. Petkovic, M., Prandi, D., Zannone, N.: Purpose control: Did you process the data for the intended purpose? In: Workshop on Secure Data Management (2011)
27. Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. MIT Press (2020)
28. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In: Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES) (2017)
29. Smyth, B., Bernhard, D.: Ballot secrecy and ballot independence coincide. In: Computer Security – ESORICS 2013. Springer Berlin Heidelberg (2013)
30. Solove, D.: A taxonomy of privacy (2005)
31. Ta, V., Eiza, M.: Dataprove: Fully automated conformance verification between data protection policies and system architectures. Proceedings on Privacy Enhancing Technologies (2022)
32. Tokas, S., Owe, O., Ramezanifarkhani, T.: Language-based mechanisms for privacy-by-design. In: IFIP Int. Summer School on Privacy and Identity Management (2019)
33. Tschantz, M.C., Wing, J.M.: Formal methods for privacy. In: International Symposium on Formal Methods (2009)

Appendix

This appendix provides optional additional materials for the reader. Section A displays a file generated by CASTT. Section B proves our claims in Example 3. Section C presents the CSpeL specification for the running example at code level.

A Example of generated file

Fig. 11 shows an example of a C file generated by CASTT, from a context specification written in CSpeL and a C source file. For simplicity, some pieces of code generated by Frama-C and not directly related to our translation have been replaced by "...".

B Proof of Properties of Example 3

This section proves purpose compliance for the traces of Example 3 (or purpose non-compliance, depending on traces). For each trace t, and according to Definition 3, we need to check that at least one of the purposes of some process p used in t is granted before a personal data is handled by p.

B.1 Traces at Model Level

We would like to prove that the trace $Handle(P_1, EPR); \epsilon$ is purpose compliant, while the trace $Handle(P_2, EPR); \epsilon$ is not purpose compliant for the first context $\mathcal{C} = (S, D, P, \gamma, \pi, \nu)$ of Example 1, defined by:

$$
\begin{aligned}
S &\triangleq \{P1; P2\};\\
D &\triangleq \{EPR\};\\
P &\triangleq \{Treatment; Research\};\\
\gamma &\triangleq \begin{cases} (EPR, Treatment) \mapsto \top \\ (EPR, Research) \quad \mapsto \bot; \end{cases}\\
\pi &\triangleq \begin{cases} P1 \mapsto \{Treatment\} \\ P2 \mapsto \{Research\}; \end{cases}\\
\nu &\triangleq \begin{cases} P1 \mapsto \{EPR\} \\ P2 \mapsto \{EPR\}. \end{cases}
\end{aligned}
$$

Let us prove that the first trace is purpose compliant, i.e.:

$$\mathcal{C} \vdash Handle(P_1, EPR); \epsilon.$$

1. According to the second inference rule of Fig. 5, this property holds if and only both $\mathcal{C} \vdash Handle(P_1, EPR)$ and $\mathcal{C} \vdash \epsilon$ holds.
2. The latter case (empty trace) is the axiom of the inference system, so it holds. Let us demonstrate the former.
3. By definition of purpose compliance, $\mathcal{C} \vdash Handle(P_1, EPR)$ holds if and only if either $EPR \notin D$, or $\gamma(EPR, p) = \top$ for some purpose p in $\pi(P_1)$. We prove the right part of the disjunction.

```
...
#include "stdio.h"
enum _Purposes {RESEARCH = 0,TREATMENT = 1};
typedef enum _Purposes Purposes;
enum _PersonalData {TRIALDATA = 0,PATIENT = 1};
typedef enum _PersonalData PersonalData;
/*@ ghost int \ghost Consent[2][2]; */

struct __anonstruct_Dos_1 {char date[20] ;char medicine[20] ;};
typedef struct __anonstruct_Dos_1 Dos;
struct __anonstruct_Patient_2 {
  int id ;char name[20] ;char lastname[20] ;char birthdate[20] ;
  char address[20] ;char sexe[20] ;Dos dosList[20] ;};
typedef struct __anonstruct_Patient_2 Patient;
struct __anonstruct_TrialData_3 {
  char birthdateList[20][20] ;
  char sexeList[20][20] ;Dos dosList[20] ;};
typedef struct __anonstruct_TrialData_3 TrialData;

Patient makePrescr(Dos newd, Patient p_makePrescr)
{
  /*@ assert Need[MAKEPRESCR][PATIENT] ≡1; */
  /*@ assert Consent[PATIENT][TREATMENT] ≡1; */
  return p_makePrescr;
}

TrialData getData(Patient patientList[3])
{
  /*@ assert Need[GETDATA][PATIENT] ≡1; */
  /*@ assert Consent[PATIENT][RESEARCH] ≡1; */
  Patient p_getData = *(patientList + 0);
  /*@ assert Need[GETDATA][TRIALDATA] ≡1; */
  /*@ assert Consent[TRIALDATA][RESEARCH] ≡1; */
  TrialData d = {.birthdateList = {...}, .genderList = {...},
                 .dosList = {...}};
  /*@ assert Need[GETDATA][TRIALDATA] ≡1; */
  /*@ assert Consent[TRIALDATA][RESEARCH] ≡1; */
  return d;
}

int computeStats(TrialData data)
{
  ...
  return __retres;
}

int main(void)
{
  int __retres;
  /*@ ghost Consent[PATIENT][RESEARCH] = 0; */
  /*@ ghost Consent[PATIENT][TREATMENT] = 1; */
  /*@ ghost Consent[TRIALDATA][RESEARCH] = 0; */
  /*@ ghost Consent[TRIALDATA][TREATMENT] = 1; */
  /*@ ghost Need[MAKEPRESCR][TRIALDATA] = 0; */
  /*@ ghost Need[MAKEPRESCR][PATIENT] = 1; */
  /*@ ghost Need[GETDATA][TRIALDATA] = 1; */
  /*@ ghost Need[GETDATA][PATIENT] = 1; */
  /*@ ghost Need[COMPUTESTATS][TRIALDATA] = 1; */
  /*@ ghost Need[COMPUTESTATS][PATIENT] = 0; */
    Patient patient =
    {.id = 0,
     .name = {(char)'J', (char)'o', (char)'h', (char)'n', (char)'\000'},
     .lastname = {(char)'D', (char)'o', (char)'e', (char)'\000'},
     .birthdate = {(char)0, ..., (char)0},
     .address = {(char)0, ..., (char)0},
     .sexe = {(char)0, ..., (char)0},
     .dosList = {...}};
  Dos newdos =
    {.date = {(char)'0',...,(char)'\000'},
     .medicine = {(char)'T',...,(char)'\000'}};
  patient = makePrescr(newdos,patient);
  __retres = 0;
  return __retres;
}
```

Fig. 11. Example of generated file.

4. Consider $p = Treatment$. Since, $Treatment \in \pi(P_1)$ and $\gamma(EPR, Treatment) = \top$, the property holds.

Therefore the first trace is purpose compliant in the context $\mathcal{C}$. Let us now prove that the second trace is not purpose compliant, i.e.:

$$\mathcal{C} \not\vdash Handle(P_2, EPR); \epsilon.$$

We prove this property by contradiction, so let us assume that this trace is purpose compliant, i.e.:

$$\mathcal{C} \vdash Handle(P_2, EPR); \epsilon.$$

1. From this property and according to the second inference rule of Fig. 5, $\mathcal{C} \vdash Handle(P2, EPR)$ holds.
2. Therefore, by definition of purpose compliance, either $EPR \notin D$ or $\gamma(EPR, p) = \top$ for some purpose p in $\pi(P_2)$.
3. The former case contradicts the definition of D: EPR is a personal data in $\mathcal{C}$.
4. Consider the latter case. By definition of π, $Research$ is the only purpose of P_2. However, $\gamma(EPR, Research) = \bot$, which contradicts $\gamma(EPR, Research) = \top$.
5. Each case leads to a contradiction, so the initial hypothesis. $\mathcal{C} \vdash Handle(P_2, EPR); \epsilon$ does not hold.

Therefore the second trace is not purpose compliant in the context $\mathcal{C}$.

B.2 Traces at Program Level

We would like to prove that the trace $Handle(P_1, EPR); \epsilon$ is purpose compliant, while the trace $Handle(P_2, EPR); \epsilon$ is not purpose compliant for the second context $\mathcal{C} = (S, D, P, \gamma, \pi, \nu)$ of Example 1, defined by:

$$S \triangleq \{\texttt{makePrescr}; \texttt{getData}; \texttt{computeStats}\};$$

$$D \triangleq \{Patient, TrialData\};$$

$$P \triangleq \{Treatment, Research\};$$

$$\gamma \triangleq \begin{cases} (Patient, Treatment) & \mapsto \top \\ (Patient, Research) & \mapsto \bot \\ (TrialData, Treatment) & \mapsto \bot; \\ (TrialData, Research) & \mapsto \bot; \end{cases}$$

$$\pi \triangleq \begin{cases} \texttt{makePrescr} & \mapsto \{Treatment\} \\ \texttt{getData} & \mapsto \{Research\} \\ \texttt{computeStats} & \mapsto \{Research\}; \end{cases}$$

$$\nu \triangleq \begin{cases} \texttt{makePrescr} & \mapsto \{Patient\} \\ \texttt{getData} & \mapsto \{Patient, TrialData\} \\ \texttt{computeStats} & \mapsto \{TrialData\}. \end{cases}$$

Let us prove that the first trace is purpose compliant, i.e.:

$$\mathcal{C} \vdash Handle(\texttt{makePrescr}, Patient); Handle(\texttt{makePrescr}, Dos); \epsilon.$$

1. According to the second inference rule of Fig. 5, this property holds if and only both $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Patient)$ and $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Dos); \epsilon$ holds.
   Let us prove first the left-hand side of this conjunction.
2. By definition of purpose compliance, $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Patient)$ holds if and only if either $Patient \notin D$, or $\gamma(Patient, p) = \top$ for some purpose p in $\pi(\texttt{makePrescr})$. We prove the right part of the disjunction.
3. Consider $p = Treatment$. Since, $Treatment \in \pi(\texttt{makePrescr})$ and $\gamma(Patient, Treatment) = \top$, the property holds.
4. Let us now prove the right-hand side of the conjunction at item 1, which is $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Dos); \epsilon$ holds. According to the second inference rule of Fig. 5, this property holds if and only both $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Dos)$ and $\mathcal{C} \vdash \epsilon$ holds.
5. The latter case (empty trace) is the axiom of the inference system, so it holds. Let us demonstrate the former.
6. By definition of purpose compliance, $\mathcal{C} \vdash Handle(\texttt{makePrescr}, Dos)$ holds if and only if either $Dos) \notin D$, or $\gamma(Dos), p) = \top$ for some purpose p in $\pi(\texttt{makePrescr})$. The former case corresponds to the definition of D: Dos is not a personal data in $\mathcal{C}$.

Therefore the first trace is purpose compliant in the context $\mathcal{C}$.

Let us now prove that the second trace is not purpose compliant, i.e.:

$$\mathcal{C} \nvdash Handle(\texttt{getData}, Patient); Handle(\texttt{getData}, TrialData); \\ Handle(\texttt{computeStats}, TrialData); \epsilon.$$

We prove this property by contradiction, so let us assume that this trace is purpose compliant, i.e.:

$$\mathcal{C} \vdash Handle(\texttt{getData}, Patient); Handle(\texttt{getData}, TrialData); \\ Handle(\texttt{computeStats}, TrialData); \epsilon.$$

1. From this property and according to the second inference rule of Fig. 5, $\mathcal{C} \vdash Handle(\texttt{getData}, Patient)$ holds.
2. Therefore, by definition of purpose compliance, either $Patient \notin D$ or $\gamma(Patient, p) = \top$ for some purpose p in $\pi(\texttt{getData})$.
3. The former case contradicts the definition of D: $Patient$ is a personal data in $\mathcal{C}$.
4. Consider the latter case. By definition of π, $Research$ is the only purpose of $\texttt{getData}$. However, $\gamma(Patient, Research) = \bot$, which contradicts $\gamma(Patient, Research) = \top$.

5. Each case leads to a contradiction, so the initial hypothesis
$$\mathcal{C} \vdash Handle(\texttt{getData}, Patient); Handle(\texttt{getData}, TrialData);$$
$$Handle(\texttt{computeStats}, TrialData); \epsilon$$
does not hold.

Therefore the second trace is not purpose compliant in the context $\mathcal{C}$.

C Instantiations using CSpeL

This section presents the CSpeL's specification for the running example at code level. The corresponding context, which is the second context of Example 1, can be specified as follows.

```
\context {
  \process      { makePrescr; getData; computeStats },
  \personalData { Patient, TrialData },
  \purposes     { Treatment, Research },
  \isGranted    { (Patient: Treatment) },
  \hasPurposes  { (makePrescr: { Treatment }),
                  (getData: { Research }),
                  (computeStats: { Research })
                },
  \needData     { (makePrescr: { Patient }),
                  (getData: {Patient, TrialData }),
                  (computeStats: {TrialData })
                },
}
```

Additionnally, we can write in CSpeL the code level's traces of Example 2 as follows.

– For the first trace:

```
\trace {
  \handle(makePrescr, Patient);
  \handle(makePrescr, Dos);
  VOID
}
```

– For the second trace:

```
\trace {
  \handle(getData, Patient);
  \handle(getData, TrialData);
  \handle(computeStats, TrialData);
  VOID
}
```