Refining Privacy-Aware Data Flow Diagrams

Downloaded from: https://research.chalmers.se, 2026-04-06 17:47 UTC

Citation for the original published paper (version of record):

Alshareef, H., Stucki, S., Schneider, G. (2021). Refining Privacy-Aware Data Flow Diagrams. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 13085: 121-140. http://dx.doi.org/10.1007/978-3-030-92124-8_8

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Refining Privacy-Aware Data Flow Diagrams*

Hanaa Alshareef[1], Sandro Stucki[1][0000−0001−5608−8273], and
Gerardo Schneider[2][0000−0003−0629−6853]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] University of Gothenburg, Gothenburg, Sweden
{hanaa,sandros,gersch}@chalmers.se

Abstract. Privacy, like security, is a non-functional property, yet most software design tools are focused on functional aspects, using for instance Data Flow Diagrams (DFDs). In previous work, a conceptual model was introduced where DFDs were extended into so-called Privacy-Aware Data Flow Diagrams (PA-DFDs) with the aim of adding specific privacy checks to existing DFDs. An implementation to add such automatic checks has also been developed. In this paper, we define the notion of refinement for both DFDs and PA-DFDs as a special type of structure-preserving map (or graph homomorphism). We also provide three algorithms to find, check and transform refinements, and we show that the standard diagram "transform→refine / refine→transform" commutes. We have implemented our algorithms in a proof-of-concept tool called *DFD Refinery*, and have applied it to realistic scenarios.

Keywords: Privacy by design · DFDs · GDPR · Refinement.

1 Introduction

Privacy compliance has become a primary concern for most companies since the enactment of strong and demanding regulations on personal data protection, such as the *European General Data Protection Regulation* (GDPR) introduced few years ago [16]. Enforcing privacy compliance, however, is not easy. Indeed, *privacy* refers to a whole family of properties, including confidentiality, secrecy, data minimization (DM), privacy impact assessment (PIA), user consent, the right to be forgotten, purpose limitation, and more. Furthermore, even for specific properties, privacy compliance is in general undecidable [30, 28].

A good practice to handle the "privacy problem" is to follow the *Privacy by Design* (PbD) principle [11], where privacy is taken into account from the very beginning of the software development process. This approach has been shown to make the problem of privacy compliance more tractable [13].

* This is a postprint version of a paper presented at the 19th International Conference on Software Engineering and Formal Methods (SEFM 2021). The original publication is available from Springer at https://doi.org/10.1007/978-3-030-92124-8_8. This research has been partially supported by the Cultural Office of the Saudi Embassy in Berlin, Germany and by the Swedish Research Council (Vetenskapsrådet) under Grant 2018-04230 "Perspex".

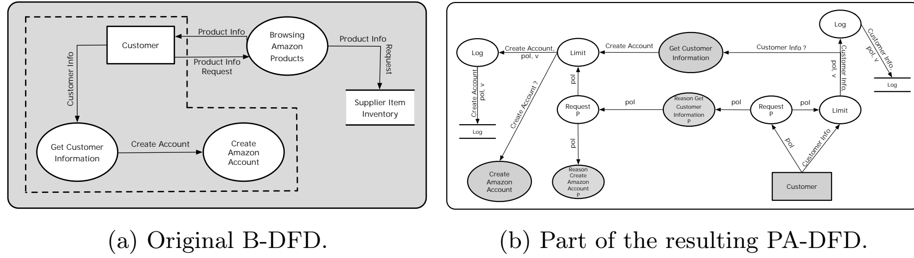
One such PbD approach was introduced by Antignac et al. [6, 7], who proposed a technique based on model transformation for automatically adding privacy checks to *Data Flow Diagrams* (DFDs). They considered an extension of DFDs called *Business-oriented Data Flow Diagrams* (B-DFDs) and further extended them with checks for specific privacy concepts, namely retention time and purpose limitation. These checks are automatically added for each operation on sensitive (personal) data (storage, forwarding, and processing of data). The enhanced diagram is called a *Privacy-Aware Data Flow Diagram* (PA-DFD). In that proposal, the software engineer designs a B-DFD, pushes a button to obtain a PA-DFD, inspects it manually, with the aim to generate a program template from the PA-DFD to guide the programmer in the concrete implementation of the privacy checks. Antignac et al. outlined their transformation from B-DFDs to PA-DFDs through set of high-level graphical "rules". A full algorithm and reference implementation were later provided by Alshareef et al. [5].

B-DFDs have been shown to be useful for software engineers when designing functional properties, and the privacy-enhanced PA-DFDs are a step towards adding specific non-functional aspects to such designs.

One issue with B-DFDs (and PA-DFDs) is that they may become big when modeling real-life systems. The traditional solution to pragmatically circumvent this problem is to either compose smaller processes, following a bottom-up approach, or to start from a high-level design consisting of composite processes that are later refined into more detailed processes, following a top-down approach. In both cases, there is a need to relate different levels of abstraction. To do so we should have a precise definition of refinement, and a rigorous methodology to check and obtain suitable refinements preserving relevant properties.

In this paper, we are concerned with the formal refinement of both B-DFDs and PA-DFDs. Concretely, we make the following contributions:

1. We propose a notion of refinement for both B-DFDs and PA-DFDs, formalizing the comparison of different levels of abstractions of such diagrams. Our notion of refinement is declarative and applies both to top-down and bottom-up refinement; for PA-DFDs, it preserves privacy and types. Furthermore, our notion of refinement has the property that it commutes with transformation (from B-DFDs to PA-DFDs). Although many informal rules and conditions for DFD refinement have been proposed and discussed in the software engineering literature, ours is, to the best of our knowledge, the first formal definition of DFD refinement. Though we are primarily interested in its applications to B-DFDs and PA-DFDs, we think that it is flexible enough to be extended to many other flavors of DFDs. (Section 3.)
2. We provide three algorithms:
   (a) *Refinement Checking.* Given abstract and concrete B-DFDs, and a candidate mapping, our refinement checking algorithm assesses whether the mapping is a refinement according to our formal definition. (Section 3.2.)
   (b) *Refinement Search.* This algorithm takes partial (incomplete) mappings and proposes possible extensions to produce a complete refinement. When



(a) Original B-DFD.

(b) Part of the resulting PA-DFD.

Fig. 1: Excerpts from DFDs modeling an e-store ordering system.

starting from an empty mapping, it produces all possible refinements between the abstract and concrete B-DFDs. (Section 3.2.)

(c) *Refinement Transformation.* It is essential that privacy checks are preserved by refinements of PA-DFDs. Our third algorithm takes a (correct) mapping between abstract and concrete B-DFDs, and transforms it into a refinement between the corresponding PA-DFDs obtained by transforming the abstract and concrete B-DFDs. The resulting refinement ensures that all privacy checks between the abstract and concrete PA-DFDs are preserved. (Section 3.3.)

3. We have implemented the above algorithms in Python as part of a proof-of-concept tool called *DFD Refinery* (Section 4), and have applied it to a case study on an automated payment system. (Section 5.)

2 Preliminaries

GDPR The European *General Data Protection Regulation* (GDPR) contains 99 articles regulating *personal data* processing. It is organized around a number of key concepts, most notably its seven *principles* relating to personal data processing, the *rights* of data subjects and six *lawful grounds* for data processing operations. Relevant to this paper are the principles of *purpose limitation* (data may only be used for purposes to which the data subject consented) and *accountability*, as well as the *right to be forgotten* and the lawful ground of *consent*. See [16] and [23] for more details on the GDPR.

Data Flow Diagrams (DFDs) A *data flow diagram* (DFD) is a graphical representation of how data flows among software components. As shown in Fig. 1, DFDs are composed of *activators* and *flows*. Activators can be *external entities* (rectangles), *processes* (ellipses) and *data stores* (double horizontal lines). Processes may represent detailed low-level operations or complex high-level functionality that could be refined into sub-processes (the latter are drawn as double-lined ellipses). Data *flow* is represented by arrows.

Antignac et al. [6, 7] extended DFDs with a *data deletion* type of flow and a data structure to specify personal data: (i) the *owner* of personal data, (ii) the

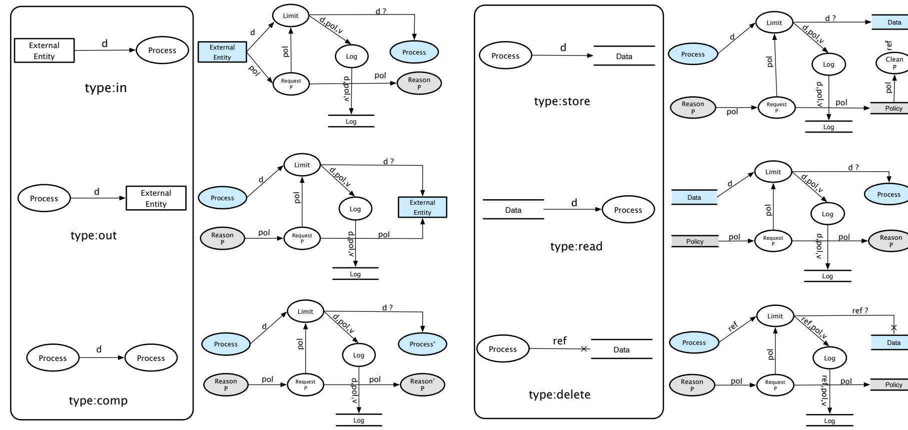


Fig. 2: Selection of B-DFD flow types and corresponding transformation rules [5].

purpose for which the data can be used as consented by the data subject, and (iii) the *retention* time for the data. This extension is referred to as *Business-oriented DFD* (B-DFD). Note that the data structure associated with B-DFDs is not relevant here.

Adding privacy checks to DFDs Antignac et al. [6, 7] further extended B-DFDs with privacy checks for purpose limitation and retention time, as well as privacy mechanisms to ensure accountability and policy management. The resulting diagrams are called *Privacy-Aware Data Flow Diagrams* (PA-DFDs). Building on that work, we defined and implemented an algorithm for transforming B-DFDs into PA-DFDs [5]. The transformation is rule-based, with one rule for every type of B-DFD flow.

Fig. 2 shows a subset of basic B-DFD flow types and the corresponding transformation rules. (The remaining rules, which cover composite activators, are given in Appendix A.) The right-hand side of each rule shows the PA-DFD corresponding to the original B-DFD flow; it extends the original flow with new activators and flows implementing the necessary privacy mechanisms.

To represent these mechanisms, the set of activator types in PA-DFDs is augmented with five novel "Process" subtypes: "Limit", "Reason", "Request", "Log" and "Clean". "Limit" activators implement the principle of purpose limitation: they inspect whether the consent given by the data subject is compatible with the action of a downstream process and discard data values for which this is not the case. The corresponding policy is supplied by a "Request" activator. "Log" activators store the decisions of "Limit" activators (and the associated data) in a dedicated data store, ensuring the principle of accountability. The "Reason" activator is used to get an updated policy "pol" corresponding to a newly computed data value. Finally, "Clean" ensures that personal data is eliminated from the data store upon expiry, guaranteeing data retention policy.

To illustrate our transformation, consider the B-DFD shown in Fig. 1a and (part of) its corresponding PA-DFD in Fig. 1b. The two rules for the flow types in and comp have been applied to a subset of the B-DFD in Fig. 1a (the part inside the dashed line). Consider the in flow labeled "Customer Information", and the corresponding PA-DFD elements shown in the right half of Fig. 1b. In addition to the original "Customer Info" data, the external entity "Customer" in the PA-DFD now also provides an associated privacy policy information "pol". The data flows to the "Limit" process which verifies that the data subject has consented to the use of "Customer Info" for downstream processing. The consent is specified in the policy "pol", received via the "Request" process. The data value, its policy and the verdict ("v") of the "Limit" process are all logged by the "Log" process in the "Log" store. If the verdict is positive, the data and policy are forwarded to the process "Get Customer Information" and its associated "Reason" process, respectively. The latter computes the updated privacy policy information associated with the output flow "Create Account".

For details about PA-DFDs and our transformation, see Alshareef et al. [5].

Hierarchical modeling (refinement of DFDs) Refinement is a method used to relate the abstract model of a software system to another more concrete model while maintaining the abstract model's properties [1]. It is applicable to system artifacts ranging from modeling and design levels to implementation and programming levels. It is typical to specify invariants that define the properties of the system being modeled at the most abstract level. These invariants must be preserved by all the refined versions of the model.

Concerning refinement in DFDs, several works discuss *leveling* (hierarchical modeling) and informal *consistency rules* [14, 29, 33]. The highest level of DFD shows all external entities and the primary data flows between the external entities and a system, represented as one composite process. This level is called *Context Diagram*. It is typically decomposed into a lower-level diagram, called the *Level 0* DFD, which can be further decomposed into a *Level 1* DFD, and so on. There are two standard rules for ensuring consistency. First, every process, data store and external entity on an abstract level is shown on a refined level (*balancing rule*). Second, the input and output data flows specified in an abstract level must hold on its refined version (*preservation of connectivity*). We formalize these rules (and others) in the next section.

3 Refining B-DFDs and PA-DFDs

3.1 Refinement of Attributed Multigraphs

Following our previous work on PA-DFDs [5], we formally represent DFDs as *attributed multigraphs* with activators as nodes and flows as edges.

Definition 1. *An* attributed multigraph *(or simply* graph*) G is a tuple $G = (\mathcal{N}, \mathcal{F}, \mathcal{A}, \mathcal{V}, s, t, \ell_{\mathcal{N}}, \ell_{\mathcal{F}})$ where $\mathcal{N}$, $\mathcal{F}$, $\mathcal{A}$ and $\mathcal{V}$ are sets of nodes, edges, attributes*

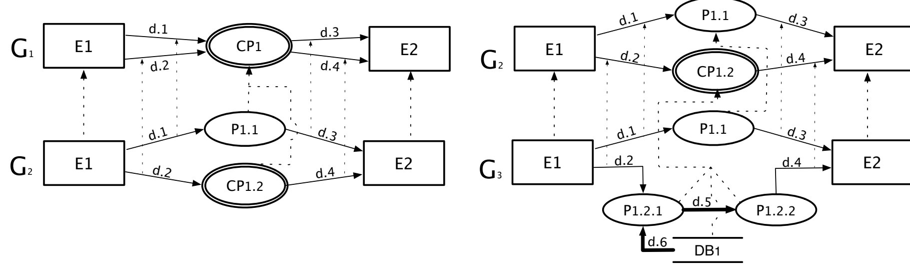


Fig. 3: Example of B-DFDs levels

and attribute values, respectively; $s, t\colon \mathcal{F} \to \mathcal{N}$ *are the source and target maps;* $\ell_{\mathcal{N}}\colon \mathcal{N} \to (\mathcal{A} \rightharpoonup \mathcal{V})$ *and* $\ell_{\mathcal{F}}\colon \mathcal{F} \to (\mathcal{A} \rightharpoonup \mathcal{V})$ *are attribute maps that assign values for the different attributes to nodes and flows, respectively.*

Examples of attributed multigraphs are shown in Fig. 3. The graph G_1 has nodes $\mathcal{N} = \{\text{E1}, \text{E2}, \text{CP1}\}$ and edges $\mathcal{F} = \{\text{d.1}, \ldots, \text{d.4}\}$. G_1 is a multigraph since both edges d.1 and d.2 connect the same source and target nodes: $s(\text{d.1}) = s(\text{d.2}) = \text{E1}$ and $t(\text{d.1}) = t(\text{d.2}) = \text{CP1}$. Attributes allow us to specify properties of activators and flows, such as their type or associated privacy information. For example, the graph G_1 has two kinds of nodes, external entities and composite processes. We formalize this by defining its attribute and value sets as $\mathcal{A} = \{\mathsf{type}\}$ and $\mathcal{V} = \{\mathsf{ext}, \mathsf{cproc}\}$, and its node attribute map as $\ell_{\mathcal{N}}(\text{E1})(\mathsf{type}) = \ell_{\mathcal{N}}(\text{E2})(\mathsf{type}) = \mathsf{ext}$ and $\ell_{\mathcal{N}}(\text{CP1})(\mathsf{type}) = \mathsf{cproc}$. Note that the attribute maps are partial, i.e., nodes and edges may lack values for certain attributes. If we extend the value set $\mathcal{V}$ with types for processes (proc) and data stores (db), we can encode the graphs G_2 and G_3 shown in Fig. 3 similarly.

Henceforth, we use the letters n, m to denote nodes and e, f to denote edges. We write $e\colon n \rightsquigarrow m$ to indicate that e has source $s(e) = n$ and target $t(e) = m$. For example, we have $\text{d.1}\colon \text{E1} \rightsquigarrow \text{CP1}$ in G_1. We use "." to select attributes, writing $n.a$ for $\ell_{\mathcal{N}}(n)(a)$ and $f.a$ for $\ell_{\mathcal{F}}(f)(a)$. For example, $\text{E1}.\mathsf{type} = \mathsf{ext}$ in G_1. The set $S(G) \subseteq \mathcal{N}$ of *source nodes* in G is defined as $S(G) = \{n \mid \exists e. s(e) = n\}$; similarly, $T(G)$ denotes the set of *target nodes* in G.

The characteristic property of a refinement is that it preserves the essential structure of some abstract object in a more concrete (or refined) object. Since we model DFDs as graphs, it is therefore natural to represent refinements of DFDs as structure preserving maps, so-called *graph homomorphism*, between concrete and abstract graphs.

Definition 2. *Let* G *and* H *be attributed multigraphs with the same sets of attributes* $\mathcal{A}_G = \mathcal{A}_H$ *and values* $\mathcal{V}_G = \mathcal{V}_H$. *A* homomorphism $h\colon G \to H$ *from* G *to* H *is a pair of maps* $h_{\mathcal{N}}\colon \mathcal{N}_G \to \mathcal{N}_H$ *and* $h_{\mathcal{F}}\colon \mathcal{F}_G \to \mathcal{F}_H$, *such that, for all nodes* n, *edges* e *and attributes* a,

$$h_{\mathcal{N}}(s_G(e)) = s_H(h_{\mathcal{F}}(e)) \qquad h_{\mathcal{N}}(t_G(e)) = t_H(h_{\mathcal{F}}(e)) \tag{1}$$

$$n.a = h_{\mathcal{N}}(n).a \qquad e.a = h_{\mathcal{F}}(e).a \tag{2}$$

(a) $\preceq$ on node types

(b) $\preceq$ on flow types

Fig. 4: The subtyping preorder $\preceq$

Condition (1) says that h preserves the connective structure of G in H; condition (2) says that h preserves attributes.

Conditions (1) and (2) are a bit too restrictive to represent DFD refinements. To see this, consider again the graphs G_1, G_2 and G_3 shown in Fig. 3. Intuitively, G_2 refines G_1 and G_3 refines G_2, so we would like to show that there are homomorphisms $g\colon G_3 \to G_2$ and $h\colon G_2 \to G_1$. But this is not the case. There are obvious candidate maps $h_{\mathcal{N}}$ and $h_{\mathcal{F}}$, represented using dotted arrows in Fig. 3, but the map $h_{\mathcal{N}}$ violates condition (2) because it does not *strictly* preserve the type of P1.1. Concretely, we have CP1.type = cproc $\neq$ proc = $h_{\mathcal{N}}$(P1.1).type. There is a similar problem if we try to map the lower processes and data store in G_3 to the composite process in G_2 as shown in the figure. In addition, there is no way to map the bold flows (d.5 and d.6) in G_3 to flows in G_2 without violating condition (1). It may be tempting to define $g_{\mathcal{N}}(\text{P1.2.1}) = g_{\mathcal{N}}(\text{P1.2.2}) = \text{CP1.2}$, as shown via the dotted arrows, but by (1) this would require $h_{\mathcal{F}}(\text{d.5})\colon \text{CP1.2} \rightsquigarrow \text{CP1.2}$, which is impossible since the node CP1.2 has no *loops*, i.e., edges connecting CP1.2 to itself. (Indeed loops are forbidden in DFDs).

We address these two issues separately. First, we relax condition (2). We want to allow refinements to *weakly* preserve attributes (such as types). Concretely, we wish to treat processes and databases as *subtypes* of composite processes. To this end, we define a preorder $\preceq \subseteq \mathcal{V} \times \mathcal{V}$ on the set of types, according to the Hasse diagram shown in Fig. 4a, and weaken condition (2) in Definition 2 to

$$n.a \preceq h_{\mathcal{N}}(n).a \qquad\qquad e.a \preceq h_{\mathcal{F}}(e).a \tag{3}$$

for all nodes n, edges e and attributes a. Note that this condition extends to attributes other than types, provided the preorder $\preceq$ is extended appropriately. A homomorphism $h\colon G \to H$ is called *lax* if it fulfills (3) and *strong* if it fulfills (2). Clearly, every (strong) homomorphism is also a lax homomorphism.

To allow "internal" edges like d.5 and d.6 in refinements without adding explicit loops in DFDs, we further adjust the definition of homomorphisms by introducing *implicit loops.*

Definition 3. *Let G and H be attributed multigraphs with the same sets of attributes $\mathcal{A}_G = \mathcal{A}_H$ and values $\mathcal{V}_G = \mathcal{V}_H$, and let $\preceq \subseteq \mathcal{V}_G \times \mathcal{V}_G$ be a preorder on values. An* abstraction $\alpha\colon G \to H$ *from G to H consists of a total map*

$\alpha_\mathcal{N}\colon \mathcal{N}_G \to \mathcal{N}_H$ and a partial map $\alpha_\mathcal{F}\colon \mathcal{F}_G \rightharpoonup \mathcal{F}_H$, such that, for all n, e and a,

$$\alpha_\mathcal{N}(s_G(e)) = s_H(\alpha_\mathcal{F}(e)) \text{ and } \alpha_\mathcal{N}(t_G(e)) = t_H(\alpha_\mathcal{F}(e)) \quad \text{if } e \in \mathrm{dom}(\alpha_\mathcal{F}),$$
$$\alpha_\mathcal{N}(s_G(e)) = \alpha_\mathcal{N}(t_G(e)) \quad \text{otherwise.} \tag{4}$$
$$n.a \preceq \alpha_\mathcal{N}(n).a \quad \text{and} \quad e.a \preceq \alpha_\mathcal{F}(e).a \quad \text{if } e \in \mathrm{dom}(\alpha_\mathcal{F}). \tag{5}$$

An abstraction is balanced *if $\alpha_\mathcal{N}$ and $\alpha_\mathcal{F}$ are surjective.*

Given an abstraction $\alpha\colon G \to H$, we call G the *concrete* graph and H the *abstract* graph of α, and we say that G *refines* H or that G is a *refinement* of H. Unless otherwise noted, we assume that all abstractions are balanced. If $\alpha_\mathcal{F}$ is undefined for an edge e, i.e., $e \notin \mathrm{dom}(\alpha_\mathcal{F})$, we say that e is *internal*. Intuitively, an internal edge $e\colon n_1 \rightsquigarrow n_2$ in G is mapped to an abstract edge "inside" the node $m = \alpha_\mathcal{N}(n_1) = \alpha_\mathcal{N}(n_2)$ in H. For example, in Fig. 3, the internal edge d.5 in G_3 is mapped to an abstract edge hidden inside the composite process CP1.2.

It is easy to verify that every graph G refines itself via the identity abstraction $\mathrm{id}_G = (\mathrm{id}_{\mathcal{N}_G}, \mathrm{id}_{\mathcal{F}_G})$, and that the composition of the maps underlying two abstractions $\alpha\colon G \to H$ and $\beta\colon H \to I$ induces an abstraction $(\beta \circ \alpha)\colon G \to I$.

A note on terminology. We deliberately chose the term *abstraction* rather than *refinement* for $\alpha\colon G \to H$ to avoid confusion. Although every abstraction corresponds to a refinement, some readers may find it more intuitive to think of a "refinement from G to H" as a process that takes an abstract G and produces a concrete refinement H of G with a corresponding abstraction $\alpha\colon H \to G$. In other words, abstractions go in the opposite direction of refinements. We continue to use the term "refinement" informally when there is no risk of confusion (e.g., to say that G is a refinement of H) but avoid its use in formal statements.

3.2 B-DFD Refinement

A B-DFD is an attributed multigraph with a fixed choice of attributes $\mathcal{A} = \{\mathsf{type}\}$ and values $\mathcal{V} = \mathcal{T}_\mathsf{dn} \uplus \mathcal{T}_\mathsf{df}$. The set of *data node types* $\mathcal{T}_\mathsf{dn}$, the set of *data flow types* $\mathcal{T}_\mathsf{df}$ and the associated *subtyping* order $\preceq$ are shown in Fig. 4. Since the type attribute plays an important role in B-DFDs (and PA-DFDs), we introduce shorthands for typing activators and flows. We write $n\colon t$ to abbreviate $n.\mathsf{type} = t$, and $f\colon n \rightsquigarrow_t m$ to indicate that $f\colon n \rightsquigarrow m$ and $f.\mathsf{type} = t$.

We require that B-DFDs be *well-formed*. First, the type t of a flow $f\colon n \rightsquigarrow_t m$ determines the types $n.\mathsf{type}$ and $m.\mathsf{type}$ of its source and target activators. The valid combinations of source, target and flow types are shown on the left-hand side of Figs. 2, 8 and 9. In addition to these flow typing constraints, we adopt the standard rules from the DFD literature for well-formed B-DFDs: diagrams should not contain loops (flows with identical source and target activators), activators cannot be isolated (disconnected from all other activators), and processes must have at least one incoming and outgoing flow (see e.g., [18, 15]).

Definition 4. *A* well-formed B-DFD *is an attributed multigraph G, where $\mathcal{A}_G = \{\mathsf{type}\}$ and $\mathcal{V}_G = \mathcal{T}_\mathsf{dn} \uplus \mathcal{T}_\mathsf{df}$. In addition, for all flows f and activators n, m,*

– n.type $\in \mathcal{T}_{\mathsf{dn}}$ *and* f.type $\in \mathcal{T}_{\mathsf{df}}$*;*
– *if* $f : n \rightsquigarrow_{\mathsf{in}} m$ *then* n: ext *and* m: proc*;*
– *if* $f : n \rightsquigarrow_{\mathsf{out}} m$ *then* n: proc *and* m: ext*;*
⋮ *(12 more flow typing conditions, as shown in the LHS of Figs. 2, 8 and 9)*
– *if* $f : n \rightsquigarrow_{\mathsf{comp}} m$ *or* $f : n \rightsquigarrow_{\mathsf{ccompc}} m$ *then* $n \neq m$*;*
– *if* n: cproc *or* n: proc *then* $n \in S(G)$ *and* $n \in T(G)$
– *if* n: ext *or* n: db *then* $n \in S(G)$ *or* $n \in T(G)$

An abstraction $\alpha : G \to H$ between B-DFDs G and H is just an abstraction of the underlying attributed multigraphs with the additional condition that the source (and target) of internal edges need to be composite processes,

$$\alpha_{\mathcal{N}}(s(f)).\mathsf{type} = \mathsf{cproc} \quad \text{and} \quad \alpha_{\mathcal{N}}(t(f)).\mathsf{type} = \mathsf{cproc} \qquad \text{if } f \notin \mathrm{dom}(\alpha_{\mathcal{F}}).$$

In earlier work, we described a *Type-inference* algorithm for checking the well-formedness of B-DFDs [5]. Here we introduce algorithms for checking the validity of a given abstraction map between abstract and concrete B-DFDs and for finding all possible abstraction maps between a pair of B-DFDs.

Checking refinements Assume we are given a pair of well-formed B-DFDs G and H, and we wish to establish that G refines H. How might we proceed? We may start by defining a pair $\alpha_{\mathcal{N}}, \alpha_{\mathcal{F}}$ of maps relating the concrete B-DFD G to the abstract B-DFD H. To guarantee the preservation of the connective structure and types, we need to check that the given maps form an abstraction. This is the purpose of the *Refinement Checking* algorithm (Alg. 1).

Our tool detects and reports any violations of the abstraction conditions (4) and (5). In addition, *DFD Refinery* can suggest corrections for broken abstraction maps based on the given abstract and concrete B-DFDs.

Algorithm 1: Refinement Checking

input : B-DFDs G, H and maps $\alpha_{\mathcal{N}} : \mathcal{N}_G \to \mathcal{N}_H$, $\alpha_{\mathcal{F}} : \mathcal{F}_G \rightharpoonup \mathcal{F}_H$.
output : An error message in case of failure.

```
1  foreach f : m ⇝ n ∈ F_G do
2      m′ ← α_N(m); n′ ← α_N(n);
3      if f ∉ dom(α_F) then
4          if n′ ≠ m′ ∨ n′.type ≠ cproc then
5              Error: "mapping of internal f is not internal";
6      else
7          f′ ← α_F(f);
8          if s_H(f′) ≠ m′ ∨ t_H(f′) ≠ n′ then
9              Error: "mapping f to α_F(f) does not preserve connections";
10         else if f.type ⊀ f′.type ∨ m.type ⊀ m′.type ∨ n.type ⊀ n′.type then
11             Error: "mapping f to α_F(f) does not preserve types";
```

Function ExtendPartial($G, H, \alpha_{\mathcal{N}}, \alpha_{\mathcal{F}}, U$) – extend partial abstractions.

input : B-DFDs G and H, partial maps $\alpha_{\mathcal{N}}\colon \mathcal{N}_G \rightharpoonup \mathcal{N}_H$, $\alpha_{\mathcal{F}}\colon \mathcal{F}_G \rightharpoonup \mathcal{F}_H$, and a set of unmapped flows $U \subseteq \mathcal{F}_G$.

output : A set of abstractions from G to H.

1 **if** $U = \emptyset$ **then** — *have all flows been mapped?*
2   **return** $\{(\alpha_{\mathcal{N}}, \alpha_{\mathcal{F}})\}$
3 **else**
4     $f\colon m \rightsquigarrow n \leftarrow$ an arbitrary flow in U;
5     $U' \leftarrow U \setminus \{f\}$;
6     $L \leftarrow \emptyset$; — *initialize result set*
7   **foreach** $m' \in \mathcal{N}_H$ **do** — *find extension where f is internal*
        — *check if the candidate conflicts with existing mappings of m and n*
8   **if** $(m \in \mathrm{dom}(\alpha_{\mathcal{N}}) \wedge \alpha_{\mathcal{N}}(m) \neq m') \vee (n \in \mathrm{dom}(\alpha_{\mathcal{N}}) \wedge \alpha_{\mathcal{N}}(n) \neq m')$ **then**
9   **continue**;
        — *check types*
10   **if** m'.type $\neq$ cproc $\vee$ m.type $\not\preceq$ cproc $\vee$ n.type $\not\preceq$ cproc **then**
11   **continue**;
12         $\alpha'_{\mathcal{N}} \leftarrow \alpha_{\mathcal{N}} \cup \{m \mapsto m', n \mapsto m'\}$; — *compute updated node map*
13         $L' \leftarrow$ `ExtendPartial(`$G, H, \alpha'_{\mathcal{N}}, \alpha_{\mathcal{F}}, U'$`)`; — *extend the new mapping*
14         $L \leftarrow L \cup L'$;
15   **foreach** $f'\colon m' \rightsquigarrow n' \in \mathcal{F}_H$ **do** — *find candidates in H for mapping f*
        — *check if the candidate conflicts with existing mappings of m and n*
16   **if** $(m \in \mathrm{dom}(\alpha_{\mathcal{N}}) \wedge \alpha_{\mathcal{N}}(m) \neq m') \vee (n \in \mathrm{dom}(\alpha_{\mathcal{N}}) \wedge \alpha_{\mathcal{N}}(n) \neq n')$ **then**
17   **continue**;
        — *check types*
18   **if** f.type $\not\preceq$ f'.type $\vee$ m.type $\not\preceq$ m'.type $\vee$ n.type $\not\preceq$ n'.type **then**
19   **continue**;
20         $\alpha'_{\mathcal{N}} \leftarrow \alpha_{\mathcal{N}} \cup \{m \mapsto m', n \mapsto n'\}$; — *compute updated maps*
21         $\alpha'_{\mathcal{F}} \leftarrow \alpha_{\mathcal{F}} \cup \{f \mapsto f'\}$;
22         $L' \leftarrow$ `ExtendPartial(`$G, H, \alpha'_{\mathcal{N}}, \alpha'_{\mathcal{F}}, U'$`)`; — *extend the new mapping*
23         $L \leftarrow L \cup L'$;
24   **return** L;

Finding refinements The *Refinement Checking* algorithm works when abstraction maps are already available. However, defining such maps manually is a tedious and error-prone task, especially for large systems. Hence, rather than leaving it to software designers, we automate it. In general, there are several ways to relate an abstract model to a concrete model while maintaining the abstract model's properties. An refinement search algorithm should thus report all possible refinements and allow the designer to select the right one.

The *Refinement Search* algorithm takes a pair of abstract and concrete B-DFDs and computes the full set of abstractions between them. We first define a helper function (ExtendPartial) to extend *partial* abstractions. The function takes a pair of B-DFDs G, H, a pair of partial abstraction maps $\alpha_{\mathcal{N}}$, $\alpha_{\mathcal{F}}$ and a set

$U \subseteq \mathcal{F}_G$ of unmapped flows, i.e., those flows for which we wish to find candidate mappings. The function returns the set of all possible abstractions from G to H that extend $(\alpha_\mathcal{N}, \alpha_\mathcal{F})$. It does so using a naive, depth-first branch-and-bound strategy: it picks an unmapped flow $f \in U$, finds all candidate mappings for f, adds them to $\alpha_\mathcal{N}$, $\alpha_\mathcal{F}$, and recursively extends them. The *Refinement Search* algorithm then consists of a single call to ExtendPartial$(G, H, \emptyset, \emptyset, \mathcal{F}_G)$,

Note that the ExtendPartial function does not check whether the resulting abstractions are balanced (i.e., that all the maps involved are surjective). Hence, there is no guarantee that all nodes and flows in the abstract B-DFD actually have a refinement in the concrete B-DFD – the concrete diagram could just be a refinement of a subset of the abstract diagram. However, a balance check can easily be added via a post-processing phase that removes non-balanced abstractions (and we have implemented such a check in *DFD Refinery*).

3.3 PA-DFD Refinement

The primary difference between B-DFDs and PA-DFDs is that the latter contain additional activators and flows that implement privacy checks. We distinguish between three kinds of PA-DFD activators and flows: those that were already present in B-DFDs, called *data* flows and nodes (e.g., processes and data stores); those that handle and carry policy information, called *policy* flows and nodes (e.g., limit and reason processes); and those that track and manage system events, called *admin* flows and nodes (e.g., log processes and data stores). Some types of activators play multiple roles, e.g., a limit process is both a data and a policy node since it handles both data and policy information.

As with B-DFDs, we use attributed graphs to represent PA-DFDs formally.

Definition 5. *Define the set* $\mathcal{T}_{\mathsf{pn}} = \{\mathsf{limit}, \mathsf{request}, \mathsf{reason}, \mathsf{policy\text{-}db}\}$ *of* policy node types *and the set* $\mathcal{T}_{\mathsf{an}} = \{\mathsf{log}, \mathsf{log\text{-}db}, \mathsf{clean}\}$ *of* admin node types. *A* PA-DFD *is an attributed graph G, where* $\mathcal{A} = \{\mathsf{type}, \mathsf{partner}\}$ *and* $\mathcal{V} = \mathcal{T}_{\mathsf{dn}} \uplus \mathcal{T}_{\mathsf{pn}} \uplus \mathcal{T}_{\mathsf{an}} \uplus \{\mathsf{pf}, \mathsf{df}\} \uplus \mathcal{N}$. *In addition, the following must hold:*

- $n.\mathsf{type} \in \mathcal{T}_{\mathsf{dn}} \uplus \mathcal{T}_{\mathsf{pn}} \uplus \mathcal{T}_{\mathsf{an}}$ *and* $f.\mathsf{type} \in \{\mathsf{pf}, \mathsf{df}\}$*;*
- *if* $n.\mathsf{partner}$ *is defined, then* $n.\mathsf{partner} \in \mathcal{N}$*.*

The partner attribute is used by the transformation algorithm and can be ignored for the purposes of this paper (cf. [5]). In principle, the flows of PA-DFDs ought to be subject to similar typing conditions as those for well-formed B-DFDs. Following the principle used for well-formed B-DFDs, we could type each flow based on the types of its source and target. For example, the flows connecting request to limit activators could be given type reqlim. This would result in twenty-two new flow types. To simplify presentation, we instead use just two flow types for PA-DFDs: *plain* flows (pf) and *deletion* flows (df).

All the new node types are special kinds of processes or data stores and, as such, are considered subtypes of composite processes. To reflect this, we extend the subtyping relation as follows:

$$n \preceq n \qquad n \preceq \mathsf{cproc} \qquad \text{for all } n \in \mathcal{T}_{\mathsf{pn}} \uplus \mathcal{T}_{\mathsf{an}}.$$

An abstraction $\alpha\colon G \to H$ between PA-DFDs G and H is just an abstraction on the underlying attributed multigraphs with the extra condition that, if $f\colon m \rightsquigarrow n$ in G is internal, then $\alpha_{\mathcal{N}}(m) = \alpha_{\mathcal{N}}(n)\colon \mathsf{cproc}$.

Transforming refinements Having defined algorithms to check and find refinements between B-DFDs in the previous section, we could now do the same for refinements of PA-DFDs. However, the changes to the algorithms would be minimal and largely uninteresting. After all, PA-DFDs are still just a special type of attributed graph, and the definition of abstractions is robust against changes in the choice of attributes and values. Furthermore, we neither expect nor intend software engineers to manipulate PA-DFDs manually: they should be automatically generated from B-DFDs. The same principle should apply to refinements of PA-DFDs: rather than manually refining an automatically generated PA-DFD, we expect software engineers to refine an abstract B-DFD H into a concrete B-DFD G and then automatically transform the latter into a (concrete) PA-DFD G'. For this process to make sense, we require that the resulting PA-DFD G' be a refinement of the PA-DFD H' obtained by transforming the original abstract B-DFD H. Diagrammatically,

$$\begin{array}{rccc}
 & \text{(B-DFD)} & & \text{(PA-DFD)} \\
\text{(abstract)} & H & \xmapsto{\text{transform}} & H' \\
 & \forall \uparrow \alpha & & \exists \uparrow \alpha' \\
\text{(concrete)} & G & \xmapsto{\text{transform}} & G'
\end{array}$$

In fact, the process of finding a PA-DFD abstraction $\alpha'\colon G' \to H'$ corresponding to a B-DFD abstraction $\alpha\colon G \to H$ is itself a transformation (of abstractions) that can be automated. We have defined and implemented an algorithm for this transformation. Space constraints prevent us from reproducing the full algorithm here, so we give instead a high-level outline and illustrate the main ideas.

To track the relationship between the nodes and flows of the original B-DFDs G, H and the resulting PA-DFDs G', H', the refinement transformation takes, as additional inputs, four maps:

$$\begin{aligned}
o_{\mathcal{N}_G} &: \mathcal{N}_{G'} \to \mathcal{N}_G, & o_{\mathcal{F}_G} &: \mathcal{N}_{G'} \uplus \mathcal{F}_{G'} \to \mathcal{F}_G, \\
o_{\mathcal{N}_H} &: \mathcal{N}_{H'} \to \mathcal{N}_H, & o_{\mathcal{F}_H} &: \mathcal{N}_{H'} \uplus \mathcal{F}_{H'} \to \mathcal{F}_H.
\end{aligned}$$

The maps keep track of which B-DFD nodes and flows resulted in the creation of a given PA-DFD node or flow. For instance, $o_{\mathcal{N}_G}$ maps every proc or reason node n in the concrete PA-DFD G' to the corresponding original proc node $o_{\mathcal{N}_H}(n)$ in the concrete B-DFD G. Conversely, the inverse image $o_{\mathcal{N}_G}^{-1}(m)$ of a node $m\colon \mathsf{proc}$ in G is a set $\{m_1, m_2\}$ containing the pair of nodes $m_1\colon \mathsf{proc}$ and $m_2\colon \mathsf{reason}$ created during the transformation of n. The four maps can easily be generated as an output of the DFD transformation algorithm.

In much the same way that the transformation algorithm on DFDs first transforms nodes and then flows, the refinement transformation first transforms node mappings $(n \mapsto m) \in \alpha_{\mathcal{N}}$ and then the flow mappings $(e \mapsto f) \in \alpha_{\mathcal{F}}$.

1. The transformation of the node mappings proceeds by case analysis on the types $n.\mathsf{type}$ and $m.\mathsf{type}$. Only a few combinations are valid. For example, if $n\colon \mathsf{proc}$ then either $m\colon \mathsf{proc}$ or $m\colon \mathsf{cproc}$. If $m\colon \mathsf{proc}$, we must have $o^{-1}_{\mathcal{N}_G}(n) = \{n_1\colon \mathsf{proc}, n_2\colon \mathsf{reason}\}$ and $o^{-1}_{\mathcal{N}_H}(m) = \{m_1\colon \mathsf{proc}, m_2\colon \mathsf{reason}\}$. This results in two new PA-DFD node mappings $n_1 \mapsto m_1$ and $n_2 \mapsto m_2$. If $m\colon \mathsf{cproc}$ instead, then $o^{-1}_{\mathcal{N}_H}(m) = \{m'\colon \mathsf{cproc}\}$, resulting in the two new mappings $n_1 \mapsto m'$ and $n_2 \mapsto m'$. The other cases are similar.
2. In a second step, we transform all the flow mappings. Because the number of (combinations of) flow types is larger, the process is more tedious but equally straightforward. We iterate over all $e\colon n_1 \rightsquigarrow n_2 \in \mathcal{F}_G$ and check whether e is internal. If so, e has no counterpart in the abstract B-DFD H and hence there is no edge there to transform. Intuitively, the edge is "hidden" inside a composite process in the abstract B-DFD H, and all the new edges and privacy checks from the concrete PA-DFD G' will also be "hidden" inside a new composite process in the abstract PA-DFD H'. Concretely, we know that there is a node $m = \alpha_{\mathcal{N}}(n_1) = \alpha_{\mathcal{N}}(n_2)$ with $m\colon \mathsf{cproc}$ in G. Hence, $o^{-1}_{\mathcal{N}_H}(m) = \{m'\colon \mathsf{cproc}\}$ in H, and we add mappings $n' \mapsto m'$ for all nodes $n' \in o^{-1}_{\mathcal{F}_H}(e)$ while leaving all edges $e' \in o^{-1}_{\mathcal{F}_H}(e)$ unmapped (they are internal).
   If e is not internal, the transformation of the associated edge mapping $e \mapsto f$ resembles that for a node mapping. By case analysis on the types $e.\mathsf{type}$ and $f.\mathsf{type}$, we determine how the edges e and f were transformed. For example, if $e.\mathsf{type} = \mathsf{store}$ then we must have $f.\mathsf{type} \in \{\mathsf{store}, \mathsf{compc}, \mathsf{cstore}, \mathsf{ccompc}\}$. In each sub-case, $o^{-1}_{\mathcal{F}_G}(e)$ is the set of nodes and flows produced by the store transformation rule, while $o^{-1}_{\mathcal{F}_H}(f)$ is a similar set of nodes and flows associated with the transformation rule for the type $f.\mathsf{type}$. In each case, there is a straightforward mapping between the corresponding nodes and flows, based on their type.

Our tool *DFD Refinery* implements the above algorithms.[3]

4 *DFD Refinery*

We have proposed a refinement framework comprising three algorithms, implemented in our *DFD Refinery* tool: *Refinement Checking*, *Refinement Search* and *Refinement Transformation*. *DFD Refinery* also includes an updated version of our previous tool for transforming B-DFDs into PA-DFDs [5, 4].

DFD Refinery uses `draw.io`, a user-friendly, easy-to-use, cross-platform and open source third-party application for drawing DFDs. We use Henriksen's open source library [22] to provide additional support for manipulating DFDs. Since it is easy to import and export diagrams from/to XML format in `draw.io`, our tool processes DFD diagrams represented in an XML format and generates PA-DFD diagrams in the same format.

The abstraction maps for B-DFDs and PA-DFD produce CSV/Text files. Our tool is implemented in Python and has been tested on a MacBook Pro.[3]

[3] Source code available at https://github.com/alshareef-hanaa/Refining_PA-DFD.

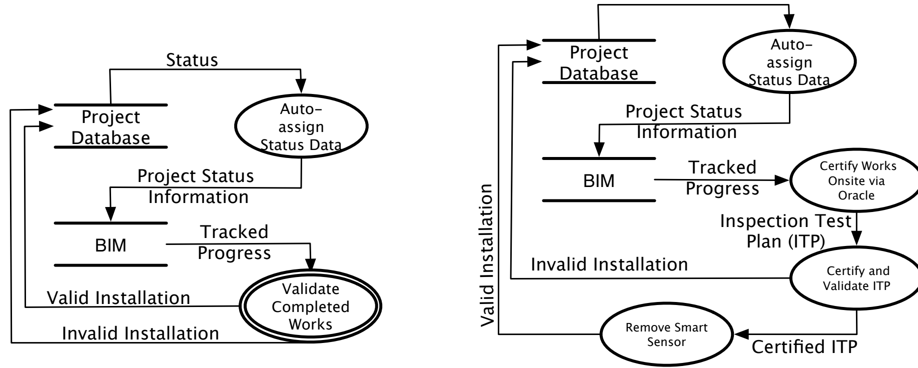


Fig. 5: Part of an Automated Payment System DFD: Level 0 (left) and Level 1 (right).

5 Case Study

To validate our algorithms, we have applied *DFD Refinery* to a realistic application: an automated payment system. The DFD (context diagram, Level 0 and Level 1) for this system is due to Chong and Diamantopoulos [12]; it has been reviewed by domain experts and models a system for making automatic payments to subcontractors in a construction project. Here we consider Levels 0 and 1 of the DFD.

We start our evaluation by applying the *Type-inference* and *Transformation* algorithms from our previous work [5] to check that the input B-DFD is well-formed and to obtain the corresponding PA-DFD. Had the designers of the Automated Payment System provided CSV files specifying abstractions for the three B-DFDs, we could have directly applied our *Refinement Checking* algorithm to verify their correctness. In the absence of such files, we instead apply our *Refinement Search* algorithm to the B-DFDs for Levels 0 and 1 (see Fig. 5).

The algorithm returns only one (balanced) abstraction since there is only one proper way to map the activators and flows of Level 1 to cover those on Level 0 according to our refinement framework. The resulting abstraction maps the processes ("Certify Works Onsite via Oracle", "Certify and Validate ITP", "Remove Smart Sensor") on Level 1 to the composite process "Validate Completed Works" on Level 1. Then, we apply our Transformation algorithm to each level of B-DFDs to obtain the corresponding PA-DFDs (see Figs. 6 and 7). During the transformation, we (automatically) create auxiliary maps to track the relationship between B-DFDs' activators and flows and the resulting PA-DFDs. For instance, the process "Auto-assign Status Data" is transformed into a process and its partner, "Reason Auto-assign Status Data". Likewise, the flows in the B-DFDs have their targets in the corresponding PA-DFDs. For example, "Tracked progress flow", which is typed as `read`, has seven target flows (e.g., "Tracked

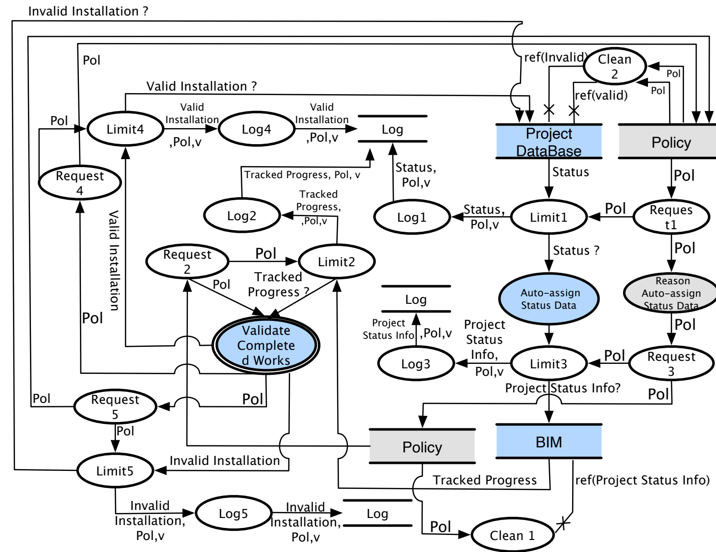


Fig. 6: Part of Automated Payment System PA-DFD level 0

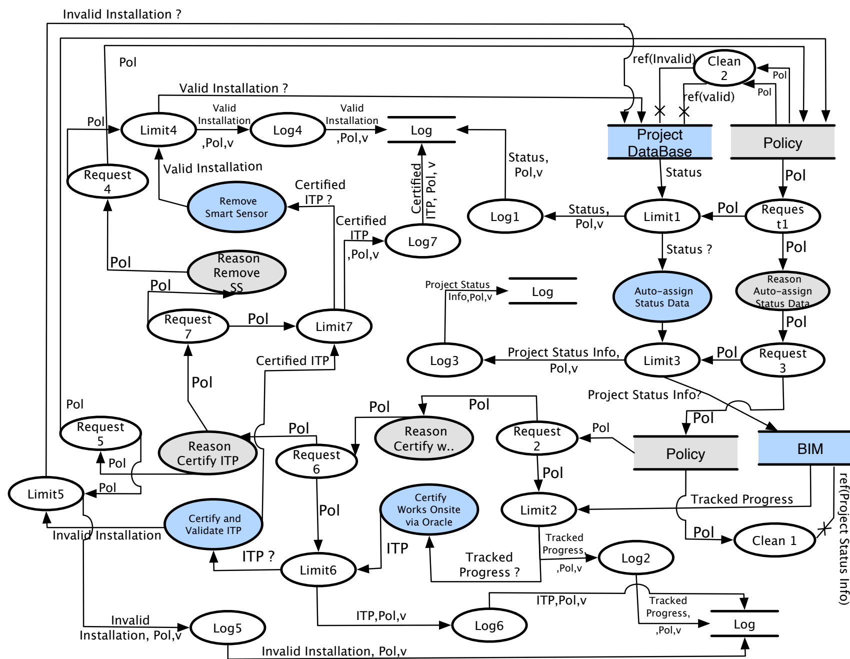


Fig. 7: Part of Automated Payment System PA-DFD level 1

progress ?" and "Tracked progress,pol,v") and four activators (e.g., "Limit2", "Request2"). These tracking maps and the abstraction between the B-DFDs are used by the *Refinement Transformation* algorithm to construct a valid abstraction between the PA-DFDs at Level 1 and Level 0. For instance, the B-DFD abstraction shows that the internal flow "Certified ITP" on Level 1 is mapped to (a hidden flow inside) the composite process "Validate Completed Works".

6 Related Work

The notion of refining abstract specifications into more concrete models, and even to executable code, is not new. Refinement has been advocated for the B method and variants like Event-B [1, 2, 3], for the Z method (e.g., [32]) and VDM (e.g., [25]), as well as for many other formal specification languages. In many such languages, notably B and Z and the refinement calculus [8], the support for refinement is considered a very important feature of the language and its design methodology. Refinement has also been introduced for other "diagrammatic" modeling languages, including class and use-case diagrams in UML [17].

There have been earlier attempts to formalize DFDs to reduce ambiguity and detect inconsistency and incompleteness (e.g., [27, 21, 19, 9, 20, 26]), and some works provide formal techniques to support the definition of hierarchical DFDs (e.g., [31, 10, 27, 24]). Representing DFDs in different levels of abstraction does not automatically guarantee consistency between the different abstract models. Lee and Tan [27] model DFDs using Petri Nets, and thus are able to check consistency of the DFDs by enforcing constraints on their Petri Net model. Though theoretically interesting, we believe the approach is not of practical use for software engineers as Petri Nets are more complicated to handle and understand than DFDs.

The only work we are aware of that defines a notion of refinement for DFDs is that by Ibrahim et al. [24]. Indeed, they have formalized some of the standard structured DFD rules to check the consistency of different models but only between the context and Level 0 DFDs. Our refining framework has a simple set of rules, including all the standard structured DFD rules, for checking if a concrete B-DFD is consistent with its abstraction. Ours is a rule-based approach built on the rigorous mathematical theory of graph homomorphisms, and can be applied to any two B-DFDs at different levels of abstraction.

To the best of our knowledge, no previous work has provided a formal definition of refinement for DFDs for arbitrary number of levels. Also, the notion of refinement for PA-DFDs is completely original, preserving not only structural and functional properties but also the underlying privacy concepts.

7 Conclusions

We have introduced *abstractions* as a new, formal notion of refinement for both DFDs and PA-DFDs and showed that the standard diagram relating transfor-

mations and refinements commute. We have provided three different algorithms for checking, finding and transforming refinements.

The *Refinement Checking* algorithm evaluates whether a pair of maps between an abstract and concrete B-DFD form an abstraction. The second algorithm takes a partial (or empty) abstraction between two B-DFDs and produces all possible extensions that form valid abstractions. Finally, the *Refinement Transformation* algorithm takes an abstraction witnessing that a concrete B-DFD refines an abstract one as its input and transforms it into an abstraction between the corresponding PA-DFDs obtained by transforming the abstract and concrete B-DFDs. The resulting PA-DFD abstraction witnesses that all privacy checks between the abstract and concrete PA-DFDs are preserved.

We have implemented the refinement algorithms and evaluated them on a case study. As future work, we intend to further extend our transformation (and refinement) so that it also covers accountability and policy management.

A Additional transformation rules

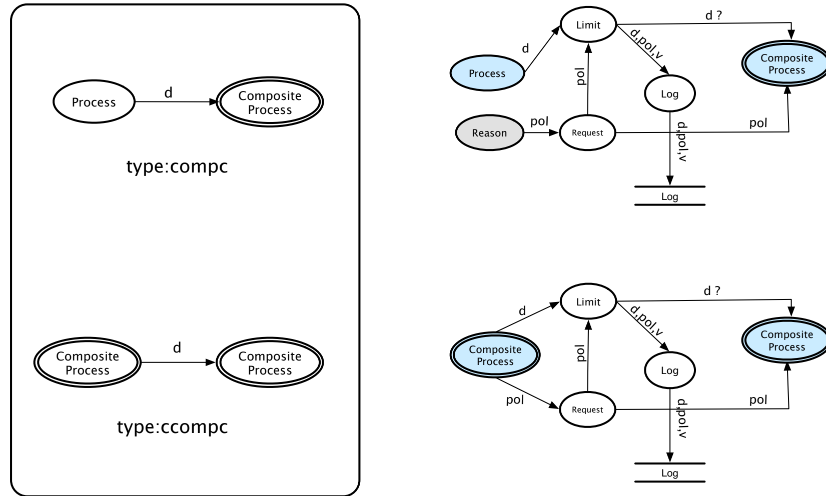


Fig. 8: B-DFD flow types and corresponding transformation rules – Part 2.

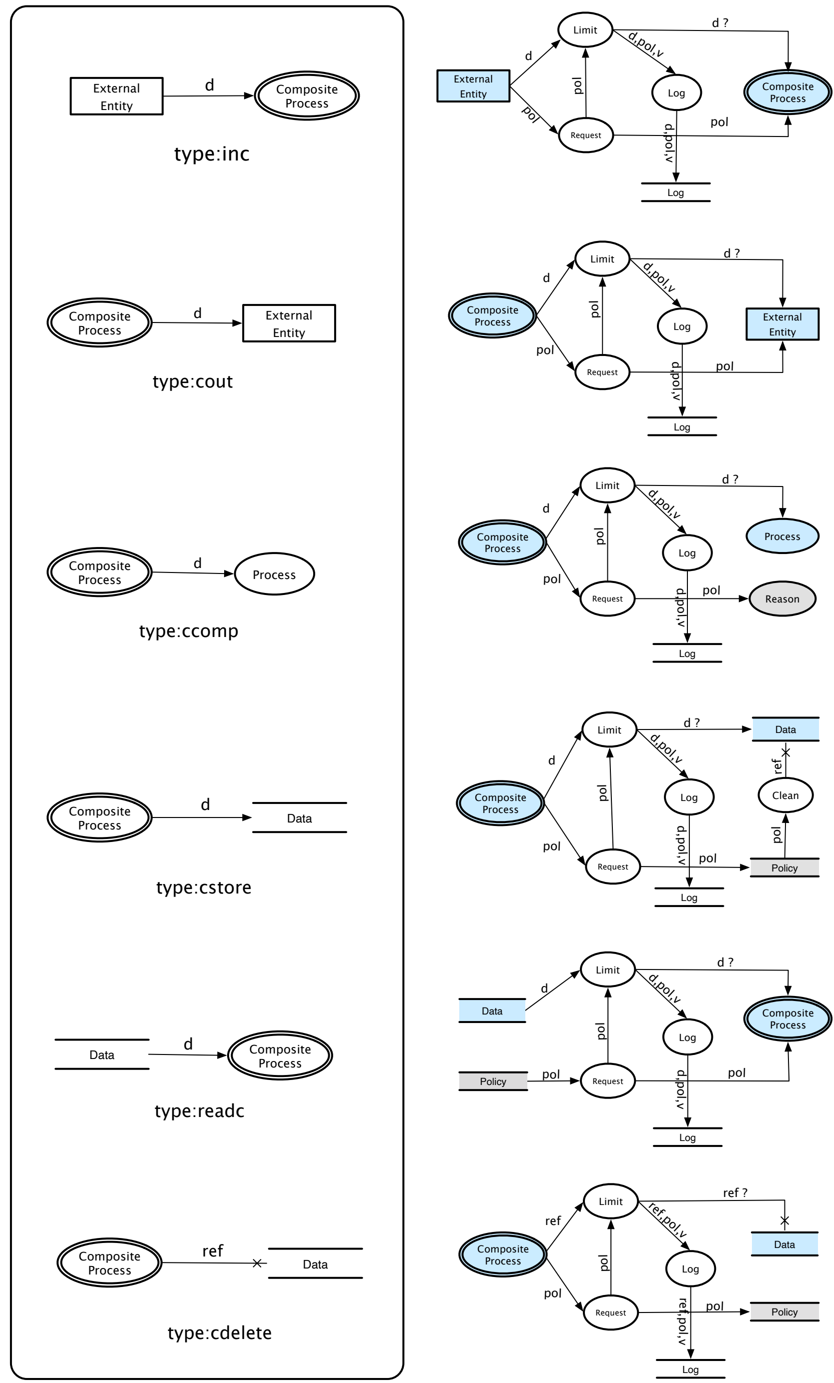


Fig. 9: B-DFD flow types and corresponding transformation rules – Part 3.

Bibliography

[1] Abrial, J.: The B tool (abstract). In: VDM'88. LNCS, vol. 328, pp. 86–87. Springer (1988)

[2] Abrial, J.R., Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge university press (2005)

[3] Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundamenta Informaticae pp. 1–28 (2007)

[4] Alshareef, H., Stucki, S., Schneider, G.: Transforming data flow diagrams for privacy compliance (long version). CoRR abs/2011.12028 (2020)

[5] Alshareef, H., Stucki, S., Schneider, G.: Transforming data flow diagrams for privacy compliance. In: MODELSWARD'21. pp. 207–215. SCITEPRESS (2021)

[6] Antignac, T., Scandariato, R., Schneider, G.: A privacy-aware conceptual model for handling personal data. In: ISoLA'16. pp. 942–957 (2016)

[7] Antignac, T., Scandariato, R., Schneider, G.: Privacy compliance via model transformations. In: IWPE'18. pp. 120–126. IEEE (2018)

[8] Back, R., von Wright, J.: Refinement calculus, part I: sequential nondeterministic programs. In: REX Workshop. LNCS, vol. 430, pp. 42–66. Springer (1989)

[9] Bruza, P.D., Van der Weide, T.: The semantics of data flow diagrams. Univ. of Nijmegen, Dept. of Informatics (1989)

[10] Butler, G., Grogono, P., Shinghal, R., Tjandra, I.: Analyzing the logical structure of data flow diagrams in software documents. In: Proceedings of the 3rd International Conference on Document Analysis and Recognition. vol. 2, pp. 575–578. IEEE (1995)

[11] Cavoukian, A.: Privacy by design: origins, meaning, and prospects for assuring privacy and trust in the information era. In: Privacy Protection Measures and Tech. in Business Org., pp. 170–208. IGI Global (2012)

[12] Chong, H.Y., Diamantopoulos, A.: Integrating advanced technologies to uphold security of payment: Data flow diagram. Automation in Construction 114, 103–158 (2020)

[13] Danezis, G., Domingo-Ferrer, J., Hansen, M., Hoepman, J.H., Le Métayer, D., Tirtea, R., Schiffner, S.: Privacy and data protection by design. ENISA Report (2015)

[14] DeMarco, T.: Structure analysis and system specification. In: Pioneers and Their Contributions to Software Engineering, pp. 255–288. Springer (1979)

[15] Dennis, A., Wixom, B.H., Roth, R.M.: Systems analysis and design. John wiley & sons (2018)

[16] European Commission: General data protection regulation (GDPR). Regulation 2016/679, European Commission (2016)

[17] Faitelson, D., Tyszberowicz, S.: Uml diagram refinement (focusing on class- and use case diagrams). In: ICSE'17. pp. 735–745. IEEE / ACM (2017)

[18] Falkenberg, E., Pols, R.V.D., Weide, T.V.D.: Understanding process structure diagrams. Information Systems 16(4), 417 – 428 (1991)
[19] France, R.B.: Semantically extended data flow diagrams: A formal specification tool. IEEE Transactions on Software Engineering 18(4), 329 (1992)
[20] Fraser, M.D., Kumar, K., Vaishnavi, V.K.: Informal and formal requirements specification languages: bridging the gap. IEEE transactions on Software Engineering p. 454 (1991)
[21] Gao, X.l., Miao, H.k., Liu, L.: Functionality semantics of predicate data flow diagram. Journal of Shanghai University (English Edition) pp. 309–316 (2004)
[22] Henriksen, M.: Draw.io libraries for threat modeling diagrams (2018), https://github.com/michenriksen/drawio-threatmodeling
[23] Hert, P.D., Papakonstantinou, V.: The new general data protection regulation: Still a sound system for the protection of individuals? Computer Law & Security Review 32(2), 179–194 (2016)
[24] Ibrahim, R., et al.: Formalization of the data flow diagram rules for consistency check. arXiv preprint arXiv:1011.0278 (2010)
[25] Jones, C.B.: Systematic software development using vdm. Prentice Hall International Series in Computer Science (1990)
[26] de Lara, J., Vangheluwe, H.: Using AToM^3 as a meta-CASE tool. In: Proceedings of the 4st International Conference on Enterprise Information Systems (ICEIS 2002). pp. 642–649 (2002)
[27] Lee, P.T., Tan, K.: Modelling of visualised data-flow diagrams using petri net model. Software engineering journal pp. 4–12 (1992)
[28] Schneider, G.: Is privacy by construction possible? In: ISoLA'18. pp. 471–485. Springer (2018)
[29] Tao, Y., Kung, C.: Formal definition and verification of data flow diagrams. Journal of Systems and Software pp. 29–36 (1991)
[30] Tsormpatzoudi, P., Berendt, B., Coudert, F.: Privacy by design: From research and policy to practice - the challenge of multi-disciplinarity. In: APF'15. pp. 199–212. Springer (2015)
[31] Wing, J.M., Zaremski, A.M.: Unintrusive ways to integrate formal specifications in practice. In: International Symposium of VDM Europe. pp. 545–569. Springer (1991)
[32] Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996)
[33] Woodman, M.: Yourdon dataflow diagrams: a tool for disciplined requirements analysis. Information and Software Technology 30(9), 515–533 (1988)