A Logic of Efficient and Optimal Designs

Giuseppe Primiero
Department of Philosophy
University of Milan

Abstract

Logics of design have been formulated until recently to offer systematic treatments of the way systems express the relation between resources, processes and their outputs. We present a logic of systems design which explicitly formalizes this relation as a decidable checking problem on resource access and define computable efficiency and optimality properties.

1 Introduction

The problem of rationalising system design dates back to the 1960s, with attempts falling largely in the category of problem-solving processes and logics. The formulation of designs as logical processes is meant to offer systematic treatments of the way systems express the relation between processes and their outputs. During the successive decades, many methodological approaches to design have been formulated and the need for a generalization of the different domains led to the need for a logic of design.

Going back to Peirce's work on abduction, March (1976) presents a logic of production including inductive and deductive processes. In order to better qualify the types of knowledge and inference found in design, Fawcett (1987) qualifies the laws relating design and relevant information, with deduction being used to infer such information from a given design and design rules: logic programming is then explored to perform such tasks automatically. In Zeng and Cheng (1991) the logic of design is characterised as a general scheme to represent the reasoning form generating a given design, and identifying the separation between the formal, the scientific and the engineering aspects of the process. The logical formulation offered in this context amounts to the identification of a recursive process mimicking the designer's presumption that a certain design exists satisfying the intended system: such recursive process starts from the intended result to infer some base case and a possibly partial rule of inference. In Galle (1997), extensions to standard predicate logic are proposed for a logical study of patterns of inference in design reasoning. A modal logic of design is formally defined by reduction to a quantificational fragment over different perspectives on a given design universe.

A recent attempt to introduce a novel logic of design is made in Floridi (2017) as a conceptual logic of information. The latter formulates the definition of the blueprint of an intended system. Here, again, we find the tension between logic and information and the need to systematise it. In particular, this model relies on the system requiring a blueprint and its implementation, according to the following steps:[1]

- *origin*: the system is supposed to do something new;
- *focus*: the system is constructed to satisfy some well-defined requirements;
- *design*: the system is developed according to a blueprint;
- *build*: the system is implemented;
- *occupy*: the system is functional and used.

Focus and *design* consist in formulating functional and non-functional requirements, i.e. what is usually called requirements elicitation:[2] functional requirements describe *what the system is supposed to do*, non-functional ones describe *how to do it, and under which conditions is the system supposed to do it.* Hence, focus and design describe the purpose for which the system is designed, the resources (information) needed for the system to be considered functional, and the level of abstraction at which these are taken. In Floridi (2017), the relation between the two phases is presented as the implementation of requirements by a given system. A major distinction of this approach from previous logical descriptions of design consists in arguing that the logical relation of satisfaction

$$S \models \{R_1, \ldots, R_n\}$$

where S is the system of interest and each R_i is one of the intended requirements of the system to be designed, is too strong as it expresses a relation of necessitation. Instead, the intended meaning for designs would be that a given system is a *sufficient* solution to implement all the desired functional and non-functional requirements:

$$\{R_1, \ldots, R_n\} \dashv S$$

The proposal in Floridi (2017) explores the initial steps towards a more appropriate conceptual, albeit not yet formal, characterization. A crucial limitation of this approach consists precisely in its informal, conceptual nature. There is no precise qualification of what the blueprint of a system is; the elements constituting the system are not specified; and the requirements are left to their intuitive meaning. The ability to provide a formal and algorithmic account of the underlying notion of design and of its properties resides in completing

[1] See `http://howdesignworks.aia.org/fivephases.asp`.

[2] See Pohl (2010); Zowghi and Coulin (2005). For an extensive analysis considering several theoretical aspects of interest to our work, see Sifakis (2013).

this formal translation. This is one main aim of the present work: we do not propose a different conceptual logic of design, rather a formal account of design developed from within a computational setting. A second crucial aim is to identify useful properties for designs so defined, and to offer algorithmic procedures for their definition.

A first task for a formal logic of design is to clarify the domain over which any forcing relation $\models$ (in either direction) ranges: we take this relation to be one between sets of sentences expressing properties of systems (functionalities, requirements and so on as appropriate). Models of evaluations correspond to instances of the ontological counterparts, i.e. the real systems. Correspondingly, a semantic forcing relation $\models$, or its syntactic derivability counterpart $\vdash$, expresses correctness of a given system formulated in a language. First, we will provide the linguistic interpretation of the design of a system as a tuple that contains its basic elements: we will distinguish the static resources that the system uses from the dynamic processes using those resources;[3] the tuple is completed by the declarative description of the system's intended function or output. If process and required resources are to be defined correct with respect to the intended output, then the relation between these terms is one which can be reduced back to one of necessitation:

$$[Resources - Process] \models Output$$

where the output should be understood as the intended one. This formulation has two advantages. First, it identifies the three terms needed to define the informal notion of blueprint:

Definition 1 (Blueprint of a Design (informal))**.** *The blueprint of a system is given by the combination of process, resources and output required to instantiate that system.*

Second, it allows to distinguish logically equivalent but distinct pairs of process and resources for the same output. This means that the distinction between a system necessary for the intended output and one only sufficient for it can be expressed in terms of properties of the $[Resource - Process]$ pair.[4] We formally express such properties of designs.

To clarify this issue further, let us start by reconsidering the problem of *output correctness*. From a meta-theoretical viewpoint, one can ask whether it is possible to establish the correctness of a given logical relation with respect to the intended output.[5] For the logic of design, this corresponds to asking

[3] This distinction is especially common in computer science, as the distinction between code and data. As it will be further explained below, the present treatment of a logic of design is especially fit for computational artefacts, but is not constrained to software systems.

[4] In general, the notion of process must be intended as an algorithmic formulation of an instruction set (i.e. finite, well-defined at each step) and this has been traditionally seen instantiated either in the form of a proof, or as a program.

[5] This can be called the Output Correctness Problem, and it reflects the best fit for the notion of logical validity in a computational setting, see Primiero (2015). The correctness problem as it is formulated in Computer Science since the 60s replies to the question whether,

whether the relation between $[Resources - Process]$ and $Output$ is decidable:[6]

Definition 2 (Design Checking Problem)**.** *Given a set of resources, a process and an intended output, can one establish whether there exists a logical relation among them such that they constitute a valid blueprint?*

In other words, is it formally possible to establish whether a certain triple $\{Resources, Process, Output\}$ instantiate a valid blueprint for a system? This is notoriously a decidable problem: it consists in checking whether the output obtained by the execution of a given process with a certain set of resources is the intended one. This, in turn, requires to have not only a complete formulation of the required resources, but also a fully functional description of the process and of the output. Provided that in realistic situations there is usually an intended output and a given set of available resources, the problem can be more realistically offered as one of constructing the process which can actually complete the intended blueprint:[7]

Definition 3 (Design Reconstruction Problem)**.** *Given some set of resources and an intended output, is it possible to identify a process such that it completes an intended valid blueprint?*

This is, in general, an undecidable problem as there is no effective procedure that can guarantee the existence of such a process. Moreover, the existence of the sought procedure can be formulated under a normalization constraint: all processes satisfying a given output, if found, should be eventually proven logically equivalent.

As far as the Design Checking Problem is concerned, two further assumptions should be clarified:

1. the process of requirements elicitation is considered completed and satisfied; and

2. the process of requirements satisfaction is considered solved, in the sense that the process present in the blueprint of interest is considered correct with respect to the intended output.

Given these assumptions, the present work considers a further weakening of the Design Checking Problem. The peculiarity of system design is that, even in the decidable context considered, issues of resource accessibility and limited executability should be taken into account. Hence, one typically considers different versions of the $[Resources - Process]$ pair that approximates (hopefully better and better) the required output. This is, in concrete and formalizable terms, the distinction between a system which is sufficient for an intended set

given a program, is it possible to check that it does what it is supposed to do, i.e. it satisfies its intended output.

[6] The Design Correctness Problem is a counterpart to the Type Checking Problem, see Pierce (2002).

[7] The Design Reconstruction Problem is a counterpart to the Type Reconstruction Problem, see Pierce (2002). For the normalization constraint mentioned below, see Nederpelt (1994).

of requirements, and one which is necessary. To formulate with some precision the idea of better approximations of the requirements, one might proceed in different ways by maximizing different parameters. On the one hand, e.g. in the presence of scarce available data or memory, the designer might want to develop the process which minimizes the resources in use for the satisfaction of the requirements. On the other hand, e.g. in the presence of safety critical systems which rely essentially on redundancy, the designer might wish to maximize the use of resources available, while making sure the system still satisfies its intended output. These are two distinct, non-exclusive ways in which a process can be qualified as satisfying the same intended output in different contexts, under different priorities and for different applications. If the intention of the logic of design is then to express sufficient conditions for the satisfaction of a set of requirements, the designer should be able to distinguish between:

1. the minimal design which satisfies all and only the intended requirements,
2. and a maximal one which satisfies any number of additional requirements as long as it preserves the intended ones.

We call the former an *efficient design*, the latter an *optimal design*. This distinction is especially relevant in the context of software systems, where it is closely related to the problem of malfunctioning: a misfunctioning software system is one which has non-intended side effects, hence technically satisfying an optimal design; a disfunctioning software system is one which does not or cannot satisfy some of the intended requirements, hence amounting to a non-efficient design.[8]

Accordingly, one might reformulate the Design Checking Problem in view of the two properties mentioned above: efficiency and optimality. Efficiency reflects the idea of a blueprint in which the process correctly satisfies the output with a minimal use of resources compared to other processes with the same output. The Design Checking Problem under efficiency constraints asks for the formulation of a design in which the process uses a minimal subset of the available resources sufficient for the satisfaction of the intended output. This interpretation in turn implies our ability to identify the minimal set of resources for output satisfaction:

Definition 4 (Efficient Design Checking Problem)**.** *Given resources* Γ*, process* t *and output* α *satisfying a given blueprint, is it possible to establish the minimal set* $\Gamma' \subseteq \Gamma$ *required for* t *using* Γ' *to satisfy* α*?*

Optimality reflects the idea of a design in which the process still correctly satisfies the same output with a maximal extension of the currently used resources.[9] The optimization problem describes the selection of a process optimal among all the sufficiently similar versions for the given blueprint, including those that use more resources than required, but always preserve the intended functionalities (albeit maybe along with other ones). Optimality is here intended as corresponding to maximization of functionalities:

[8] See Floridi et al. (2015).

[9] In the present context, resources for a design are always constrained to finite sets.

Definition 5 (Optimal Design Checking Problem)**.** *Given resources* Γ*, process* t *and output* α*, is it possible to establish a maximally consistent set* $\Gamma' \supseteq \Gamma$ *such that* t *using* Γ' *still satisfies* α*?*

Provided our formal translation of the blueprint of a system as the set composed by the output, the available resources and the process which using the latter should lead to the former, a natural way to approach the two problems of efficiency and optimality is to analyse them in terms of access operations on the resources. This is useful to identify some meta-theoretical problems of the informational logic of design, to investigate its formal dynamics and eventually to offer some observations on its impact on the issues of malfunctioning. To obtain this aim, we will consider the satisfiability of a formal relation between locations where the resources required to satisfy a blueprint are held.[10]

The rest of this paper is organised as follows. In Section 2, we introduce a language to express the construction of simple and complex designs through the control of resources access and operations on their locations. In Section 3, we provide results on the efficiency and optimality of both simple and complex designs. In Section 4, we illustrate through the definition of algorithms how these properties can be computed. We conclude with an overview of current and future problems in the formal approach to the logic of design.

2 Access on Resources

The aim of the present section is to offer a language for design construction based on accessing resources at locations. We understand the *blueprint* of a system as the set composed by resources needed, executable process and output for the system under construction. Under this understanding, a blueprint is not a fully operational description of the intended system, as it does not allow to read where resources are accessible, nor in which order they should be accessed, for process execution. Because of such limitation, a blueprint does not suffice for distinguishing among instances of the same system that differ for resource usage and functionalities implementation. On the other hand, we understand the *design* of a system as the operationalization between the elements of the blueprint, i.e. the actual access rules to resources which allow a given process to satisfy the intended output. In this sense, an *efficient design* corresponds to the

[10] This strategy is similar to (and partly inspired by) ludics and the geometry of cognition, see in particular Girard (2001, 2003). In the following, our terminology also follows partly that of ludics. Despite the acknowledgment of this inspiration, the motivation behind our task is very different and it is not constrained to some specific logical calculus. Another relevant influence for our work is represented by access control models, extensively used for example in security policies, and resources calculi, see e.g. Fernández and Siafakas (2014). The common route of these two research areas lies notoriously in the substructural nature of linear logic. The use of designs as objects of Ludics corresponding to the skeleton of proofs where formulas are not manipulated, but rather their location do, is extended in terms of interactions called *disputes* in Faggian and Hyland (2002). In our approach, we look at some basic combinatorial operations on designs, required to obtain more complex structures for the satisfaction of requirements. We are interested in a computational approach to efficiency and optimality properties for any given design.

least resourceful implementation of a given blueprint; an *optimal design* corresponds to a maximally resourceful implementation of a blueprint still preserving intended functionalities.

We start by offering the syntax of our language.

Definition 6 (Syntax)**.**

$$\begin{aligned}
&\mathtt{T} := \{t, u, \ldots, z\}\\
&\mathtt{L} := \{A, B, \ldots, N\}\\
&\mathtt{O} := \{\alpha, \ldots, \tau | \bot\}\\
&\mathtt{R} := \Gamma^A, \cdot | \Gamma^A, (u : \tau)^A
\end{aligned}$$

In this language we consider: a finite set $\mathtt{T}$ of process terms; a finite set of locations $\mathtt{L}$ where resources are available; output $\mathtt{O}$ denote the semantics of terms, i.e. their functional description: for a term t, we declare such description α by the formula $t : \alpha$, which means that t will be a (possibly complex) term with output of type α; we abstract here from the internal semantics of the term t, which could be defined in view of any desired semantics by logical connectives; for completeness we include the contradictory output with the symbol $\bot$, as this would be used to describe output $\alpha \to \bot$, but we do not investigate this case explicitly here. The formula $(t : \alpha)^A$ further specifies the location A at which term t is considered to produce output α. Resources $\mathtt{R}$ are expressed by finite sets of formulas Γ^A, each of them of the form $(t : \alpha)^A$, required to correctly obtain a term satisfying the intended output.[11]

Definition 7 (Resource Access Operation)**.** *A formula* $\Gamma^A \vdash (t : \alpha)^B$ *has the following informal reading: under resources* Γ *issued at location* A*, process* t *satisfies output* α *at location* B*.*

We investigate the Efficient and Optimal Design Checking Problems by considering the derivability of a formula of the form $\Gamma^A \vdash (t : \alpha)^B$. For the present purposes, the qualification of the derivability relation $\vdash$ is irrelevant, and it can be characterized according to the context of application. Instead, we first want to define the blueprint which such a formula satisfies:

Definition 8 (Blueprint)**.** *A blueprint is a triple* $\mathtt{B} := \{\Gamma, t, \alpha\}$ *composed by a set of resources, a process and an intended output.*

A blueprint is an abstract object and it comes unqualified with respect to locations. We are interested in checking its satisfiability, by considering the relation between locations where the resources required to satisfy an access operation are held. Note, therefore, that a blueprint does not guarantee a

[11] As briefly mentioned above, while the sufficientisation relation from Floridi (2017) only identifies a System and the Requirements it is supposed to satisfy, our resource access operation is meant to separate the specification as intended output (Floridi's Requirements) from a pair process and resources (Floridi's System). In doing so, we can distinguish between a process t that uses a minimal set of resources in Γ to satisfy output α, and one that uses a maximal set of resources in Γ still satisfying output α.

working design, as it can include non-accessible resources, or even a non well-formed pair process-output.

In the formulation of designs, we define notions and rules appropriate for regulating accesses between locations at which resources are valid either dependently or independently of other previously accessed resources. In turn, efficiency and optimality refer respectively to sufficiency (minimality condition) and precision (maximality condition) of the resource set. In order to make our process more realistic, we want to constrain the accessibility from resource to resource (and hence from location to location) in view of a validity condition expressed by an authorization protocol. For example, the protocol regulating access to the resource set might require that one can access location B only after access to location A has been granted. This might be due to the logical or conceptual or legal priority of one requirement over another, each requirement identified by the appropriate location where it can be accessed. To express a *policy* of valid accesses between locations we refer to a partial order over locations $\mathtt{L}$. Imposing such a policy over a (sub)set of locations gives us the notion of network:

Definition 9 (Networks)**.** *A network* $\mathcal{N} := \{A \leqslant \ldots \leqslant N\}$ *is a finite subset of locations* $\mathcal{N} \subseteq \mathtt{L}$ *equipped with an authorization policy* $\sim := \{<, =, >\}$.

Each location can now be identified by the paths and the related authorization policy through which it can be accessed within a network. To this aim, we introduce the notion of Address Pattern.

Definition 10 (Address Pattern)**.** *An address pattern is a pair* $\mathcal{A} := \{\mathcal{P}, r\}$ *composed by a finite subset of a Network* $\mathcal{P} \subseteq \mathcal{N}$ *called path and a dominance relation* $r = \{(n, m), \sim\}$ *with a policy* $\sim$ *over integers* $(n, m) \in Z$*, each assigned to a location in* $\mathcal{P}$.

Given a blueprint $\mathtt{B} := \{\Gamma, t, \alpha\}$, a valid Address Pattern $\mathcal{A}$ for $\mathtt{B}$ is constructed from the paths $\mathcal{P}$ across set of locations connecting Γ^A and $(t : \alpha)^B$ according to the related order on integers from the policy $\sim$. While the notion of Address Pattern and the use of integers assigned to locations might be considered redundant in view of the previously defined notion of Network, it represents a way to denote the policy existing between any two locations directly, i.e. without referring back to their positioning in the order of the network. In this way, any two locations can be assessed from the design they occur in, even when they originate from different networks with some common location.

2.1 Simple Designs

Consider, as an initial example, the simple resource access operation for blueprint $\mathtt{B} := \{\Gamma, t, \alpha\}$:

$$\frac{}{\Gamma^A \vdash (t{:}\alpha)^B} \tag{1}$$

We extract the address pattern to construct a protocol-based access authoriza-

tion. Our network is (A, B), i.e. the locations relevant to the access of a process t for the output α. Moreover, we need to refer to their dominance relation $\sim$, given by the relative positioning of those addresses in the authorization policy, say $(A < B)$, to refer to the fact that process t for output α located at B can only be accessed *after* access to resources Γ at A has been granted. As these are the only two locations in the network, we simply assign them positive integers $r = (1, 2)$. The address pattern can now be used as an access rule for the related blueprint:

$$\frac{}{\vdash \{\Gamma, t, \alpha\}} (A, B), (1 < 2) \tag{2}$$

Here $\mathtt{B} := \{\Gamma, t, \alpha\}$ is described as obtained by the Address Pattern $((A, B), (1 < 2))$. In other words, the Address Pattern makes $\mathtt{B}$ a valid blueprint in that it provides all the required instructions to access the process t and the resources Γ to satisfy the output α.

An *abstract design* is now intended as a proof that removes any reference to resources, in order to focus on access policy. The access policy abstracts away from the Resource Access Operation, but it allows its reconstruction. The content of the blueprint can be subsumed as external to the calculus as in the present case, or it can be added in the form of a side-notation: this will be essential in the real-case scenario where multiple resources are available at the same location and therefore a given Address Pattern is valid for several blueprints, but we will abstract from this specific aspect in the following and assume the simple case where we always have the blueprint available.

Definition 11 (Abstract Design Formula)**.** *A formula $A \vdash \{\mathcal{P}, r\}$ has the following informal reading: from location A, the path $\mathcal{P}$ with access policy expressed by a dominance relation r across its locations is accessible.*

Note that a path accessed by a location, or from which a location is accessed, can be a singleton. Similarly, in the above definition of Abstract Design Formula, the location A can be replaced by a full path $\mathcal{P}'$. Assume, as in the above example, that resources Γ are available at location A in position 1 of the policy and process term t is available at location B in position 2 of the policy, then we rewrite the access operation as follows:

$$\frac{}{A \vdash B, (1 < 2)} \tag{3}$$

In this case, from location A the path which extends to location B is accessible according to the dominance relation $1 < 2$. The process of deriving the path A to B is called a *simple design*:

Definition 12 (Simple Design)**.** *A simple design is a construction of a direct address pattern for a blueprint.*

Both well-foundedness and finiteness of the design are given by the definition of path and the order policy. When needed, we will refer to a sub-design has a part of the design, i.e. as a subset of its Address Pattern. A simple design

as defined above can be obtained in terms of rules to construct sequences of access patterns for blueprints. Such rules are constrained in the following by three basic steps:

1. the start of a path of locations;
2. the access of a location within a given path;
3. the extension of a completed path of locations by a new location.

These are reflected by the following formal rules:[12]

Definition 13 (Authorizations Rules for Simple Designs)**.** *Each access authorization for the design of a blueprint is obtained by one of the following rules:*

Root*: an axiomatic declaration of location access authorization*

$$\frac{}{\vdash A}\ \mathit{Root}$$

Dominance*: given a non-empty path $\mathcal{P}$, a location $A \in \mathcal{P}$ and a dominance relation $(n \sim m)$ for some $n, m \in Z$, where n, m are associated to elements of the path, an inference of a dominance relation can be formulated:*

$$\frac{\{\mathcal{P}, r\} \vdash A}{\vdash A, \{\mathcal{P}, r\}}\ \mathit{Dominance}$$

Access*: given a dominance relation r for a path $\mathcal{P}$ and the set $\{\mathcal{P}, (n \sim m)\}$ for each $n, m \in Z$ associated with elements in the path, simply denoted $\{\mathcal{P}, r\}$, an inference of the full dominance relation for some $A \in \mathcal{P}$ can be formulated:*

$$\frac{\vdash A, \{\mathcal{P}, r\}}{A, r \vdash \mathcal{P}}\ \mathit{Access}$$

The *Root* rule allows direct access to a location as the starting point of a given path. Unless a location is a Root, every path comes decorated with a dominance relation in the form $(n \sim m)$. The *Dominance* rule declares that given the accessibility of a location A from a path $\mathcal{P}$ according to a dominance relation r, A is accessible in $\mathcal{P}$. The right-hand side of an abstract design formula

[12] The rules in the following have a basic correspondence with operations common to Ludics, see Girard (2003), in particular: Root corresponds to Daimon; Dominance corresponds to the positive rule; Access corresponds to the negative rule. The Rule for Shared Access in the next section is a form of Cut. The crucial difference here is the addition of the accessibility operation for resources based on the Address Pattern. The rules for combining designs in Section 2.3 are inspired by Access Control Systems and not present in the standard tradition from Ludics.

is also called the Dominance side. The *Access* rule states that given a location A accessible in $\mathcal{P}$ according to dominance r, A is an access point to $\mathcal{P}$ according to r; this conclusion brings the dominance relation on the left-hand side of the turnstile. The left-hand side of an abstract design formula is also called the Access side. Note that the conclusion of the Access rule reduces to the premise of the Dominance rule with $A \equiv \mathcal{P}$, i.e. the latter rule works as elimination for the former rule. An example of the conclusion of the Access rule presenting a path $\mathcal{P}$ as a single location is offered in the the following extension of equation 3:

$$\dfrac{\dfrac{\dfrac{}{\vdash A}\,\text{Root} \qquad A, (1 < 2) \vdash B}{\vdash (A, B), (1 < 2)}\,\text{Dominance}}{A, (1 < 2) \vdash B}\,\text{Access} \tag{4}$$

2.2 Shared Access

Once basic operations for the construction of a simple design have been offered, a further rule is required for the extension of a path by legal sharing of resources among locations on possibly different paths. This reflects the idea that a design can borrow a path from a distinct design, as long as the relevant access authorizations are respected. A most interesting aspect, e.g. in the context of security applications, is the behaviour of a resource access operation in view of a transitive share of resources between locations. In the following, we will denote with $sup(r)$ the greatest element in the order r, and with $inf(r)$ its least element. Then a construction based on the coincidence of location and order is obtained by an application of the following rule:[13]:

Definition 14 (Shared Design)**.** *The access authorization for a blueprint sharing resources from two distinct designs is obtained by the following rule:*

> ***Share****: given a valid design obtained by an Access rule, and a valid design obtained by a Dominance Rule, with a shared address B, a valid shared design is constructed as follows:*
>
> $$\dfrac{\{\mathcal{P}, r\} \vdash B \qquad \vdash B, \{\mathcal{P}', r'\}}{\{\mathcal{P}, r''\} \vdash \mathcal{P}'}\ \textit{Share, with } r'' \subseteq \{inf(r) \leqslant sup(r')\}$$

Note that the first premise of this rule can be seen as the conclusion of an Access Rule with $\mathcal{P} \equiv A$ on the left-hand side of the turnstile, and $B \equiv \mathcal{P}$ on the right-hand side. The present rule is more general, as the Access side in the first premise does not need to be constrained to a single location, but it can be generalised to a path; at the same time, it respects the coincidence of a location on the Dominance side of the first and the second premises.

[13] Formally, this has the general form of a cut rule in a proof system.

Example. Consider a Resource Access Operation for a given blueprint such that a functionality ψ is executable at location C provided a resource ϕ is accessible at B. Also, resource ϕ has an access policy which requires a set of resources Γ to be available at location A for it to be executed. Then a full Resource Access Operation valid for ψ is implemented by the following tree:

$$\dfrac{\dfrac{}{\Gamma \vdash \phi}\,(A,B),(1<2) \qquad \dfrac{}{\phi \vdash \psi}\,(B,C),(2<3)}{\Gamma;\phi \vdash \psi}\,(A,B,C),(1,2<3) \tag{5}$$

The set of Access Operations above can be interpreted by designs as follows:

$$\dfrac{\dfrac{\vdash (A,B),(1<2)}{A,(1<2)\vdash B}\,\text{Access} \qquad \dfrac{B,(2<3)\vdash C}{\vdash (B,C),(2<3)}\,\text{Dominance}}{(A,B),(1<3)\vdash C}\,\text{Share} \tag{6}$$

with $A \vdash B$ and $B \vdash C$ being the operations involved by a shared access. The informal reading of the above derivation is as follows: given an access from location A to location B at positions $1,2$ of the policy, and accessibility of location C from location B at positions $2,3$ of the policy, one builds an access from location A to B,C at positions $1,2$ and finally an access from location A,B to C at positions $1,3$ of the policy. In this shared access, from location A one has access to resource at location B with $A < B$; a similar relation holds between locations $B < C$; sharing access to resources implies downward authorization for location A with respect to resources available from location C.

Definition 15 (Shared resource access)**.** *A shared resource access is a coincidence of a location and a dominance value $n \in r$ in some Access and Dominance operations.*

Intuitively, a shared access should be equivalent to a direct access (normalization) if the operation is performed according to the dominance relation: the previous shared access would be invalid when $\vdash (B,C),(2>1)$ (although in this specific case C might be accessible directly from A at 1). Hence, reducing computational steps of resource access has to happen under ordered resources: given access from location A to resource at location B, and an access from location B to resources at location C, it is possible to use A to access C *if and only if* allowed by the access policy reflected in the dominance relation between locations A,B,C. A normalization theorem across designs has to account for this requirement.[14] While in our interpretation the finite nature of locations lists makes it impossible to have diverging normalisations by an infinite series of conversions,[15] we are faced with a different kind of problem: undesired shares are those where the network has an access policy that should not allow transitive accesses across designs. To avoid these, we formulate below a normalisation theorem with an explicit reference to policy, reflected in the proof.

[14] Notoriously, in the standard construction of the normalization proof in ludics, conversion actually induces a connected design by allowing weakenings.

[15] This possibility requires the additional formulation of **Faith** in the original Girard (2001).

Theorem 1 (Normalisation on Dominance for Simple Designs)**.** *Any design D_1 containing a shared access where a location occurs twice, once in the access side (as premise of the Dominance rule) and once in the dominance side (as conclusion of the Access Rule), can be replaced by a design D_2 that contains no such repetition and ends with the final authorized location on the dominance side iff the policy associated with the design reflects the dominance relation of the location.*

Proof. The inductive cases are generated by the last application of a rule in the design D_1:

- the last step in the design D_1 is the result of *Root*: D_1 normalises to a design D_2 with an empty root.
- the last step in the design D_1 is the combined result of Dominance from a sub-design $D_{1.1}$ and Access in a sub-design $D_{1.2}$. Then a Share rule is applied to some path $\mathcal{P}$ such that in $D_{1.2}$, a location $A \in \mathcal{P}$ is in the right-hand side and in $D_{1.1}$ the same location $A \in \mathcal{P}$ is in the left-hand side (both by construction of the corresponding rules):
  - if the dominance r in $D_{1.1}$ is not contained in the set $\{(\mathcal{P}, r)\}$ of the design $D_{1.2}$, then normalization fails;
  - else, for each $(n \sim m) \in \mathcal{P}$, consider the sub-design $D_{1.1.x}$ terminating in $(\mathcal{P}, (n \sim m)) \vdash A$ and the sub-design $D_{1.2.y}$ induced by $\vdash A, \{(\mathcal{P}, r)\}$; replacing $D_{1.1}$ with $D_{1.1.x}$ and $D_{1.2}$ with $D_{1.2.y}$, if $(n \sim m)$ reflects the policy order wrt $A \in \mathcal{P}$, then the design is connected and thus in normal form. This last case for $D1$ is shown by the tree in example 6.

□

The requirement related to the dominance relation 'reflecting the policy order' is explicitly left vague, because such policy might be different from context to context and depending on applications: the generally valid assumption is that the share should be valid for $\mathcal{P}, (m < n) \vdash A$ and $\vdash (A, \mathcal{P}'), (n < o)$. If one wants to model access according to a different order relation, this can be changed by design and according to specific requirements for distinct access modes (e.g. upwards in the dominance relation, with a cyclic order and so on).

Definition 16 (Orthogonal access)**.** *Two designs D_1, D_2 are orthogonal when normalisation on dominance is obtained. The resulting design is of the form given by a declaration of shared access.*

The existence of designs with shared access and their normalised versions with only simple designs induce a notion of equivalence class defined over them:

Definition 17 (Behaviour)**.** *The behaviour $\mathbb{B}$ is the equivalence class of designs $\{D_1, \ldots, D_n\}$ for a blueprint* `B` *closed under shares.*

The behaviour of a blueprint contains all the possible paths for the resources required to correctly implement B. Normalisation is the reduction function to the shortest of such paths.

2.3 Combining Designs

A simple design for a blueprint is constructed according to the dominance relation of locations. But a blueprint can also be obtained by the modular combination of designs for distinct blueprints. Such modular combination typically requires to cut across locations that might be not in the same order relation. For this reason, we need to generalise simple designs.

Definition 18 (Combined Design)**.** *A combined design* $\mathbf{D}$ *is a modular construction of address patterns of distinct behaviours* $\{\mathbb{B}_1, \ldots, \mathbb{B}_n\}$.

In other words, we consider a design $D_1 \in \mathbb{B}_1$ and a design $D_2 \in \mathbb{B}_2$ to be combined in a new design $\mathbf{D} = \{D_1; D_2\}$. We need to define such composition by new rules and this will also require a new principle for the normalisation of shares of combined behaviours. The kinds of composition that most easily occur in a combined design are induced by:

1. some look up rule on the accesses of another design;
2. the import of such an access, when its addresses are not available directly to the current design;
3. and copying of the writing protocols of such access in the current path of a given design.

In the following rules, with slight abuse of notation, we use $\mathbb{B}_1, \mathbb{B}_2$ to denote any two designs $D_1 \in \mathbb{B}_1$, $D_2 \in \mathbb{B}_2$ respectively. Along with the already introduced supremum and infimum for the policy, we will use respectively $sup(\mathcal{P})$ and $inf(\mathcal{P})$ for the least and last accessible addresses in a given path.

Definition 19 (Authorization Rules for Combined Designs)**.** *Each access authorization for the combined design* $\mathbf{D}$ *of distinct behaviours* $\mathbb{B}_1; \mathbb{B}_2$ *is obtained by one of the following rules:*

> ***Reading****: given behaviours* $\mathbb{B}_1, \mathbb{B}_2$*, the behaviour* $\mathbb{B}_1 \circledR \mathbb{B}_2$ *is an instance of a Dominance rule obtained by taking the terminating location of a design* $D_1 \in \mathbb{B}_1$ *and the initial location of a design* $D_2 \in \mathbb{B}_2$ *according to a valid shared dominance relation:*
>
> $$\frac{\{\mathcal{P}, r\} \vdash \mathbb{B}_1 \qquad \{\mathcal{P}', r'\} \vdash \mathbb{B}_2}{\vdash (\mathbb{B}_1, \mathbb{B}_2), \{\mathcal{P}'', r''\}} \circledR$$
>
> *with* $r'' \subseteq \{inf(r) \leqslant sup(r')\}$ *and with* $\mathcal{P}'' \subseteq \{\mathcal{P} \cup \mathcal{P}'\}$

***Importing**: given behaviours $\mathbb{B}_1, \mathbb{B}_2$, the behaviour $\mathbb{B}_1 \textcircled{1} \mathbb{B}_2$ is an instance of an Access rule from the initial location of a design $D_1 \in \mathbb{B}_1$ dominating the terminating location of a design $D_2 \in \mathbb{B}_2$:*

$$\frac{\vdash \mathbb{B}_1, \{\mathcal{P}, r\} \qquad \vdash \mathbb{B}_2, \{\mathcal{P}', r'\} \qquad \mathbb{B}_1, \{\mathcal{P}, r\} \vdash \mathbb{B}_2, \{\mathcal{P}', r'\}}{\mathbb{B}_1, \{\mathcal{P}'', r''\} \vdash \mathbb{B}_2} \textcircled{1}$$

with $r'' \subseteq \{inf(r) \leqslant sup(r')\}$ and with $\mathcal{P}'' \subseteq \{\mathcal{P} \cup \mathcal{P}'\}$

***Copying**: given behaviours $\mathbb{B}_1, \mathbb{B}_2$, the behaviour $\mathbb{B}_1 \textcircled{C} \mathbb{B}_2$ is the result of accessing $\mathbb{B}_2$ from a dominating $\mathbb{B}_1$*

$$\frac{\vdash (\mathbb{B}_1, \mathbb{B}_2), \{\mathcal{P}, r\} \qquad \mathbb{B}_1, \{\mathcal{P}', r'\} \vdash \mathbb{B}_2}{\mathbb{B}_1 \vdash \mathbb{B}_2, \{\mathcal{P}'', r''\}} \textcircled{C}$$

with $r'' \subseteq \{inf(r) \leqslant sup(r')\}$ and with $\mathcal{P}'' \subseteq \{\mathcal{P} \cup \mathcal{P}'\}$

The third premise in ① is costly, as it requires to produce a novel path between designs to be possible.

Examples. The following are applications of the Authorization rules, where locations (A, B) and (B, C) must be intended as (posssibly sub-)design of distinct behaviours.

$$\frac{A, (1 < 2) \vdash B \qquad B, (2 < 3) \vdash C}{\vdash ((A, B), (B, C)), (1 < 3)} \textcircled{R} \tag{7}$$

$$\frac{\vdash (A, B), (1 < 2) \qquad \vdash (B, C), (2 < 3) \qquad (A, B), (1 < 2) \vdash (B, C), (2 < 3)}{(A, B), (1 < 3) \vdash (B, C)} \textcircled{1} \tag{8}$$

$$\frac{\vdash ((A, B), (B, C)), (1 < 3) \qquad (A, B), (1 < 3) \vdash (B, C)}{(A, B) \vdash (B, C), (1 < 3)} \textcircled{C} \tag{9}$$

2.4 Shared Access for Combined Designs

A further rule is required for the extension of a path by a sharing of resources among locations for a combined design. This rule generalizes shares to combined designs.

Definition 20 (Combined Shared Design)**.** *The access authorization for a blueprint sharing resources from two distinct designs is obtained by the following rule:*

***Combined Share**: given a valid design $\mathbb{B}_1 \textcircled{C} \mathbb{B}_2$ and a valid design $\mathbb{B}_2 \textcircled{C} \mathbb{B}_3$, a combined share of designs $\mathbb{B}_1$,$\mathbb{B}_3$ is constructed as follows:*

$$\frac{\mathbb{B}_1 \vdash \mathbb{B}_2, (\mathcal{P}, r) \qquad \mathbb{B}_2 \vdash \mathbb{B}_3, (\mathcal{P}', r')}{\mathbb{B}_1, (\mathcal{P}'', r'') \vdash \mathbb{B}_3}\ \mathit{CShare}$$

with $r'' \subseteq \{inf(r) \leqslant sup(r')\}$ *and* $P'' \subseteq \{inf(\mathcal{P}) \cup sup(\mathcal{P}')\}$.

Note that the conclusion of *CShare* has the same logical form of the conclusion of an instance of the Import rule ⓘ, but: its premises relate distinct designs through a shared one (possibly a single location), and the path in the conclusion is constructed joining only the least and last elements of the originating paths.

Example. Consider behaviours: $\mathbb{A} = (A, C), (1 < 3)$, obtained by a tree with an instance of ®as in Equation 7, an instance of ⓘ as in Equation 8; and constructing an instance of a ©rule as in Equation 9.

$$\dfrac{\dfrac{}{\vdash ((A,B),(B,C)),(1<3)}\ ® \qquad \dfrac{}{(A,B),(1<3) \vdash (B,C)}\ ⓘ}{(A,B) \vdash (B,C),(1<3)}\ ©$$

Now consider the behaviour $\mathbb{B} = (C, F), (3 < 4)$ obtained by a specular tree as follows:

$$\dfrac{\dfrac{}{\vdash ((C,D),(E,F)),(2<3)}\ ® \qquad \dfrac{}{(C,D),(2<4) \vdash (E,F)}\ ⓘ}{(C,D) \vdash (E,F),(2<4)}\ ©$$

Now the shared locations across designs allow to formulate a new access from (A, B) to (E, F), making use of the existence of the designs $(B, C), (C, D)$:

$$\frac{(A,B) \vdash (B,C),(1<3) \qquad (C,D) \vdash (E,F),(2<4)}{(A,B),(1<4) \vdash (E,F)}\ \text{CShare}$$

Definition 21 (Shared resource access for combined designs)**.** *A shared resource access for combined designs is a coincidence of a location in Copying operations for distinct design.*

A complex behaviour is the equivalence class of composed designs under CShares:

Definition 22 (Complex Behaviour)**.** *A complex behaviour* $\mathfrak{B}$ *is the equivalence class of combined designs* $\{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$ *for a complex blueprint* `B` *closed under CShares.*

The behaviour of a complex design contains all the possible paths for the resources required to produce `B`, when the latter is the result of multiple designs $A; B$. We can now present normalization on behaviours that possibly include combined shares. Intuitively, a shared access should be equivalent to a direct access (normalization). Undesired shares are those for which a common location and position in the dominance cannot be identified. For this reason we formulate below a normalisation theorem with an explicit reference to both conditions.

Theorem 2 (Normalisation with Import for Combined Designs)**.** *A combined share is a coincidence of a location within the appropriate dominance relation in two combined design* $\mathbb{B}_1, \mathbb{B}_2$ *and* $\mathbb{B}_3, \mathbb{B}_4$ *for some such that the base of any given design in the first path ends in* $\mathbb{B}_1 \copyright \mathbb{B}_2$*, the other path ends with* $\mathbb{B}_3 \copyright \mathbb{B}_4$*.*

Proof. The shared location A in the path of the combined designs will occur on the left-hand side of one design of behaviour $\mathbb{B}_2$ in the branch $D_{1.2}$; and on the right-hand side of another design of behaviour $\mathbb{B}_3$ in the branch $D_{1.1}$:

- For the former, consider the sub-design $D_{1.1.x}$ terminating in $(\mathcal{P}, (n \sim m)) \vdash A$, i.e. by an instance of an Import rule: this does not necessarily preserve the position of the location in the dominance r of the ©rule.
- For the latter, consider the sub-design $D_{1.2.y}$ induced by $\vdash A, \{(\mathcal{P}, \sim)\}$, i.e. where the location is obtained by a Reading rule; and the position of the location in the dominance r of the ©rule is preserved.

Then, if the dominance constructed by the *CShare* rule coincides in the preserved location of A with respect to the other behaviours, the design is necessarily connected, by assumption for the location, and by construction for the dominance relation, and thus in normal form. □

3 Ordering Simple and Combined Designs

All designs of a given behaviour will have at least the same resources and locations (i.e. those essential to satisfy the intended specification), but possibly more by covering different paths connecting such locations (hence inducing additional functionalities). The most efficient design in a behaviour will be the most direct one to a resource access operation $A \vdash B$, assuming dominance is respected: this can be shown to correspond to the path with the least number of shares. The optimal design in a behaviour will be the one that includes the most resources for which the intended blueprint is still satisfied: this corresponds to the selection paths with the largest number of imports.

3.1 Efficient and Optimal Simple Designs

Recall that a blueprint might have different designs corresponding to it, with longer and shorter paths and distinct dominance relations. We denote with $\mathcal{A}(D_i)$ an address pattern $\mathcal{A}$ of a design D_i. The length of the address pattern of a simple design, which we now denote by $\overline{\mathcal{A}(D_i)}$, is directly proportional to the number of instances of the Share rule, which we denote by $|Share(D_i)|$. In other words, a simple design normalised with respect to the Share rule has a shorter address pattern than a non-normal one in the same behaviour. We can therefore order any $D_1, D_2 \in \mathbb{B}$ as follows:

Theorem 3. *For all* $D_1, D_2 \in \mathbb{B}$*,* $\overline{\mathcal{A}(D_1)} < \overline{\mathcal{A}(D_2)}$ *iff* $|Share(D_1)| < |Share(D_2)|$*.*

Proof. In the two directions:

→ For any D_i of $\mathtt{B} = \{\Gamma, t, \alpha\}$, $\underline{\mathcal{A}(D_1)} = \underline{(\mathcal{N}, r)}$, where $\mathcal{N} = \{A, \ldots, N\} \in \Gamma$ and $r = (n \sim \cdots \sim o)$. If $\overline{\mathcal{A}(D_1)} < \overline{\mathcal{A}(D_2)}$, then there is $\Gamma' \subseteq \Gamma$ such that $\Gamma' \in \mathcal{A}(D_1)$ and $\Gamma \in \mathcal{A}(D_2)$. Because $D_1, D_2 \in \mathbb{B}$ by assumption, Γ' must satisfy at least as much as Γ but be necessarily shorter, this means having a common path in $\mathcal{N}$ and hence there is at least one sequence $\{A < B < C\} \in \Gamma$ which can be reduced to $\{A < C\} \in \Gamma'$ and Share is the only rule which allows such reduction. Hence, $|Share(D_1)| < |Share(D_2)|$.

← If $|Share(D_1)| < |Share(D_2)|$ then $\mathcal{A}(D_1) \supseteq \{A < C\}$ and $\mathcal{A}(D_2) \supseteq \{A < B < C\}$ for at least one pair $\underline{\{A, C\} \in \Gamma}$ and a blueprint $\mathtt{B} = \{\Gamma, t, \alpha\}$. By construction of $\mathcal{A}$, then $\overline{\mathcal{A}(D_1)} < \overline{\mathcal{A}(D_2)}$.

□

Lemma 1 (Efficiency on Simple Designs)**.** *For all $D_1, D_2 \in \mathbb{B}$ for a blueprint $\mathtt{B} := \{\Gamma, t, \alpha\}$, if D_1 is the normalisation on dominance of D_2, then $\mathcal{A}(D_1)$ contains the minimal set $\Gamma' \subseteq \Gamma$ valid for* B*.*

The Efficiency Lemma follows directly from Definition 9 and 10 and Theorems 1 and 3. The Efficiency Lemma answers to the Efficient Design Checking Problem, in the sense that given any two designs implementing the same blueprint, the one involving the shortest path across resources is the most efficient one.

Lemma 2 (Optimality on Simple Designs)**.** *For all $D_1, D_2 \in \mathbb{B}$ for a blueprint $\mathtt{B} := \{\Gamma, t, \alpha\}$, if D_1 is the normalisation on dominance of D_2, then given the set Γ' of resources in $\mathcal{A}(D_1)$, we denote with $\Delta = \Gamma \backslash \Gamma'$ the largest set of resources non-essential for* B*. Then $Cn(\Gamma') \cap \Delta$ is the optimal set of resources for* B*.*

In the above we denote with $Cn(\Gamma')$ the deductive closure of Γ', according to some consequence relation of interest. The Optimality Lemma follows from the finiteness of Definition 6 and Lemma 1. The Optimality Lemma answers to the Optimal Design Checking Problem in an indirect way: given our ability to compute the most efficient path through resources that satisfy a blueprint, it computes all the resources non required by such path (provided our set of total resources is finite by definition); hence, one can add any available resource consistent with the essential ones required by the blueprint, starting from the empty set (the minimal subset of the most efficient resource set) and up to a maximally consistent set.

3.2 Efficient and Optimal Complex Designs

To compute the length of the address patterns of composed designs, we will assume that their simple components have been normalised (hence they have minimal address pattern lengths). The length of the address pattern of the complex behaviour of normalised composed designs, which we now denote by $\overline{\mathcal{A}(\mathbf{D})}$, is directly proportional to the number of instances of the CShare rule, which we denote by $|CShare(\mathbf{D})|$. In other words, a behaviour normalised with

respect to CShare instances has a shorter address pattern than an equivalent non-normal one. We can therefore order any $\mathbf{D}_1, \mathbf{D}_2 \in \mathfrak{B}$ as follows:

Theorem 4. *For all* $\mathbf{D}_1, \mathbf{D}_2 \in \mathfrak{B}$, $\overline{\mathcal{A}(\mathbf{D}_1)} < \overline{\mathcal{A}(\mathbf{D}_2)}$ *iff* $|CShare(\mathbf{D}_1)| < |CShare(\mathbf{D}_2)|$.

Proof. Let us assume $\mathbf{D}_1 = \{D_1, D_2\}$, where D_1 is of blueprint $\mathtt{B}_1$ and D_2 is of blueprint $\mathtt{B}_2$. Similarly, $\mathbf{D}_2 = \{D_3, D_4\}$, where D_3 is of blueprint $\mathtt{B}_3$ and D_4 is of blueprint $\mathtt{B}_4$. The complex design $(D_1; D_2)$ is of blueprint $(\mathtt{B}_1; \mathtt{B}_2)$; and the complex design $(D_3; D_4)$ is of blueprint $(\mathtt{B}_3; \mathtt{B}_4)$; because $(D_1; D_2)$ and $(D_3; D_4)$ are both in $\mathfrak{B}$ by assumption, they satisfy the same outputs and have at least equivalent resource sets, although with possibly different address patterns.

$\rightarrow$ $\overline{\mathcal{A}(\mathbf{D}_1)} = \overline{\mathcal{A}(D_1)} + \overline{\mathcal{A}(D_2)}$ and $\overline{\mathcal{A}(\mathbf{D}_2)} = \overline{\mathcal{A}(D_3)} + \overline{\mathcal{A}(D_4)}$. As all simple designs are assumed to be normalised, there is no $\Gamma' \subset \Gamma$ such that $\Gamma \in \mathcal{A}(\mathbf{D}_1)$ and $\mathtt{B}_1 = \{\Gamma', t, \alpha\}$ and there is no $\Delta' \subset \Delta$ such that $\Delta \in \mathcal{A}(\mathbf{D}_2)$ and $\mathtt{B}_2 = \{\Delta', t, \alpha\}$. If $\overline{\mathcal{A}(\mathbf{D}_1)} < \overline{\mathcal{A}(\mathbf{D}_2)}$ then $\overline{\mathcal{A}(\mathbf{D}_1)} < \overline{\mathcal{A}(D_i)} + \overline{\mathcal{A}(D_j)}$ for any D_i, D_j of $\mathtt{B}_2$. Because $\mathbf{D}_1, \mathbf{D}_2 \in \mathfrak{B}$ by assumption, Γ must satisfy at least as much requirements as Δ, but be necessarily shorter, i.e. they must share a common path in $\mathcal{N}$ and hence there is at least one sequence $\{A < B < C\} \in \Delta$ which can be reduced to $\{A < C\} \in \Gamma$ and CShare is the only rule to which such reduction applies. Hence, $|CShare(\mathbf{D}_1)| < |CShare(\mathbf{D}_2)|$.

$\leftarrow$ If $|CShare(\mathbf{D}_1)| < |CShare(\mathbf{D}_2)|$ then $\exists\{A < C\} \subseteq \mathcal{A}(\mathbf{D}_1)$ and $\exists\{A < B < C\} \subseteq \mathcal{A}(\mathbf{D}_2)$ for at least one pair $\{A, C\} \in \Gamma$, where $\Gamma \in \mathcal{A}(\mathbf{D}_1)$ and a triple $\{A, B, C\} \in \Delta$, where $\Delta \in \mathcal{A}(\mathbf{D}_2)$, with $\mathtt{B} = \{(\Gamma \equiv \Delta), t, \alpha\}$. By construction of $\mathcal{A}$, then $\overline{\mathcal{A}(\mathbf{D}_1)} < \overline{\mathcal{A}(\mathbf{D}_2)}$.

□

Lemma 3 (Efficiency on Complex Designs)**.** *For all* $\mathbf{D}_1, \mathbf{D}_2 \in \mathfrak{B}$ *for a complex blueprint* $\mathtt{B} := \{\Gamma, t, \alpha\}$*, if* $\mathbf{D}_1$ *is the normalisation with Import of* $\mathbf{D}_2$*, then* $\mathcal{A}(\mathbf{D}_1)$ *contains the smallest set* $\Gamma' \subseteq \Gamma$ *valid for* $\mathtt{B}$*.*

The Complex Efficiency Lemma follows directly from Definition 9 and 10 and Theorems 2 and 4. This Lemma brings Lemma 1 to an higher level of abstraction, allowing efficiency for composed designs.

Lemma 4 (Optimality on Complex Designs)**.** *For all* $\mathbf{D}_1, \mathbf{D}_2 \in \mathfrak{B}$ *for a complex blueprint* $\mathtt{B} := \{\Gamma, t, \alpha\}$*, if* $\mathbf{D}_1$ *is the normalisation with Import of* $\mathbf{D}_2$*, then given the set* Γ' *of resources in* $\mathcal{A}(\mathbf{D}_1)$*, we denote with* $\Delta = \Gamma \backslash \Gamma'$ *the largest set of resources non essential for* $\mathtt{B}$*. Then* $Cn(\Gamma') \cap \Delta$ *is the optimal set of resources for* $\mathtt{B}$*.*

The Optimality Lemma follows from the finiteness of Definition 6 and Lemma 3. This Lemma brings Lemma 2 to an higher level of abstraction, allowing optimality for composed designs.

4 Computing Efficiency and Optimality

The Lemmas 1,2, 3 and 4 can now be shown to refer to computable processes through algorithms in pseudo-code.

Figure 1 offers an algorithm to compute the Efficient Simple Design, given as input a set of resources Γ and output α (line 1). It requires locations and a subset Γ' of the resources in the blueprint (lines 3-5). It then proceeds as follows: construct the Network by returning in an order every location satisfying a resource (lines 7-12); construct the Path and keep count of the Shares: start from the empty path and the null counter (lines 14-17); going through each location in the Network: add to the path the minimal element in the order (lines 20-23); for every successive element in the Network, add it to the Path preserving the order (lines 25-28); add every next location in the order that satisfies the next required resource (lines 30-33); if there are two of the previous cases sharing a location, cut across the location and add one to the counter (lines 35-40); proceed until there are locations available in the Network and stop when one reaches the resource satifying α, otherwise add a previously unreached resource and return (lines 40-44). This last step is essential if the current selection of resources has not turned out to be sufficient to satisfy α. Now construct the ordered equivalence class of such Paths, by selecting first the path with the lower counter (lines 46-53). Finally, return as output the minimal element in such order.

Figure 2 offers an algorithm to compute the Optimal Simple Design, given as input a set of resources Γ and output α (line 1). It requires locations and a superset of the resources in the blueprint, i.e. all initially available resources (lines 3-5). It then proceeds as follows: construct the set of non-required resources by selecting the set difference between the set of available resources and the set of resources required for an efficient design of the given blueprint (lines 7-9, hence it calls the execution of the previous algorithm); construct a maximally consistent union starting from the empty set and adding to it any element which is in the set of non-required resources but consistent with admissible requirements of the efficient design, until there are available resources to explore. Note that the possibility to obtain a maximally consistent union set is given by two properties: fineteness and linearity. We start from an empty set, we proceed by adding to the required resources step by step those non-needed resources that are linearly accessible from the current ones, and proceed as long as new resources can be added preserving consistency. The set might not be unique: two resources might be available at the same next address, and each be consistent with the current union set, but inclusion of both might make the set inconsistent. Terminate by returning as output the result of such selection.

Figure 3 offers an algorithm to compute the Efficient Complex Design, given as input a set of resources Γ and a combination of output $\alpha_1; \alpha_2$ (line 1). It requires two efficient simple design (lines 8-10). It then proceeds as follows: construct the Network by returning in an order every location satisfying a resource either in the first or in the second design (lines 12-18); construct the Path and keep count of the CShares: start from the empty path and the null counter (line 21); going through each location in the Network (line 24): add to the path the minimal element in the order; for every element in the Network (independently from the order), add it to the Path (lines 25-28); add every next location in the order that satisfies the next required resource (lines 30-33); if there are two of

```
1  PROCEDURE Efficient_SimpleDesign(Γ, α)
2
3  -- given available locations and resources
4  L := {A < ... < N};
5  Γ' ⊂ Γ := {u_i, ... u_n};
6
7  -- construct Network
8  N := {}
9  FOR each u_i ∈ Γ'
10   IF ∃A ∈ L ⊢ u_i
11     THEN N := N ∪ {A, <};
12   ENDIF
13 ENDFOR
14
15 -- construct Path with policy and keep count of Shares
16 i := 0; P := P_i; SET Counter:=0;
17 DO
18   P_i = ∅;
19   FOR EACH {A, <} ∈ N
20
21       (ROOT)
22       IF A is a minimal element of N with respect to <
23           THEN P_i := P_i ∪ {A};
24       ENDIF
25
26       (DOMINANCE)
27       IF A < B ∈ L with respect to <
28          THEN P_i := P_i ∪ {A, B};
29       ENDIF
30
31       (ACCESS)
32       IF ∃u_b ∈ B such that P_i ∪ {A} ⊢ u_b
33          THEN P_i := P_i ∪ {A, B};
34       ENDIF
35
36       (SHARE)
37       IF P_i ∪ {A, B}
38          AND P_i ∪ {B, C}
39          THEN Counter(P_i):=Counter+1 AND P_i := P_i ∪ {A, C};
40       ENDIF
41
42       (BRANCHING)
43       IF ∃u_b ∈ B such that P_i ∪ {A} ⊢ u_b AND
44          ∃u_c ∈ C such that P_i ∪ {A} ⊢ u_c
45          THEN P_i := P_i ∪ {A, B} AND P_j := P_i ∪ {A, C};
46       ENDIF
47
48       (TERMINATION)
49       IF P_i ⊢ u_m : α
50          THEN HALT
51       ELSE P_i ∪ {u_{n+1} ∈ Γ\Γ'} AND
52            RETURN;
53       ENDIF
54 ENDFOR
55
56 -- construct ordered Behaviour
57 FOR
58 i > 1 to max(Counter(P));
59   DO B := {P_i}
60   WHILE Counter(P_j) > Counter(P_i);
61     DO B := {P_i < P_j}
62 WHILE P ≠ ∅;
63 ENDFOR
64
65 RETURN P_i ∈ B such that is minimal with respect to <.
66
67 ENDPROCEDURE
```

Figure 1: Algorithm for Efficient Simple Design

```
1  PROCEDURE Optimal_SimpleDesign(Γ, α)
2
3  -- given available locations and resources
4  L := {A < ... < N};
5  Γ ⊂ Γ' := {u_i, ... u_n};
6
7  -- construct the set of non-required resources
8  Efficient_SimpleDesign(Γ'', α);
9  Δ := Γ'\Γ'';
10
11 -- construct maximal union Efficient_SimpleDesign
12 i := 0;
13 WHILE Δ ∪ Γ' ≠ ∅
14 DO
15   Γ = ∅;
16   FOR EACH u_i ∈ Δ ∩ Cn(Γ'')
17       Γ ∪ {u_i};
18   ENDFOR
19 RETURN (Γ, α).
20
21 ENDPROCEDURE
```

Figure 2: Algorithm for Optimal Simple Design

the previous cases sharing a location, cut across the location and add one to the counter (lines 35-38); and proceed as previously for branching an termination; proceed until there are locations available in the Network and stop when one reaches the set of resources satifying α. Now construct the ordered equivalence class of such Paths, by selecting first the path with the lower counter (lines 61-66). Finally, return as output the minimal element in such order (line 69).

Figure 4 offers an algorithm to compute the Optimal Complex Design, given as input a set of resources Γ and a composed output $\alpha_1; \alpha_2$ (line 1). It requires locations and a superset of the resources in the blueprints, i.e. all initially available resources (lines 3-5). It then proceeds as follows: construct the set of non-required resources by selecting the set difference between the set of available resources and the set of resources required for efficient designs composed for the given blueprints (lines 7-9, hence it calls the execution of the previous algorithm); construct the maximal union starting from the empty set and adding to it any element which is in the set of non-required resources but consistent with admissible requirements of the composed design, until there are available resources to explore in either set. Terminate by returning as output the result of such selection.

5 Conclusions

In this paper we have considered a logic of design formulated as resource access control. In this formulation it is possible to clarify the associated decidability properties: the problem of determining whether a set of resources and a process satisfy a given blueprint is decidable; on the other hand, to establish whether for a given set of requirements there exists a process that satisfies a given blueprint

```
1  PROCEDURE Efficient_ComplexDesign(Γ, α1; α2)
2
3  -- given available locations and resources
4  L := {A < ... < N};
5  Γ'1 ⊂ Γ := {ui, ... un};
6  Γ'2 ⊂ Γ := {ui, ... uo};
7
8  -- given two Efficient Simple designs
9  Efficient_SimpleDesign(Γ'1, α1);
10 Efficient_SimpleDesign(Γ'2, α2);
11
12 -- construct Complex Network
13 N := {}
14 FOR each ui ∈ Γ'1 ∨ Γ'2
15   IF ∃A ∈ L ⊢ ui
16     THEN N := N ∪ {A, <};
17   ENDIF
18 ENDFOR
19
20 -- construct Path and keep count of CShares
21 i := 0; P := Pi; SET Counter:=0;
22 DO
23   Pi = ∅;
24   FOR EACH {A, B, <} ∈ N
25       (READING)
26       IF Pi := Pi ∪ {A, B}
27            AND Pj := Pi ∪ {B, C}
28            THEN Pi := Pi ∪ Pj;
29       ENDIF
30
31       (IMPORTING)
32       IF Pi := Pi ∪ {A, B}
33         AND Pj := Pi ∪ {B, C}
34         AND Pi := Pi ∪ Pj
35         THEN Pi := Pi ∪ {A, C};
36       ENDIF
37
38       (COPYING)
39       IF Pi := Pi ∪ {A, C}
40          THEN ∃ub ∈ B such that Pi ∪ {A} ⊢ ub AND ub ⊢ C;
41       ENDIF
42
43       (CSHARE)
44       IF Pi ∪ {A, B} AND Pi ∪ {B, C}
45          THEN Counter(Pi)=Counter+1 AND Pi := Pi ∪ {A, C};
46       ENDIF
47
48       (BRANCHING)
49       IF ∃ub ∈ B such that Pi ∪ {A} ⊢ ub
50          AND ∃uc ∈ C such that Pi ∪ {A} ⊢ uc
51          THEN Pi := Pi ∪ {A, B} AND Pj := Pi ∪ {A, C};
52       ENDIF
53
54       (TERMINATION)
55       IF Pi ⊢ um : α1; α2
56          THEN HALT
57          ELSE Pi ∪ {un+1 ∈ Γ\Γ'1 ∨ Γ'2} AND
58               RETURN;
59       ENDIF
60   ENDFOR
61
62 -- construct ordered Complex Behaviour
63 FOR
64 i > 1 to max(Counter(P));
65   DO B := {Pi}
66   WHILE Counter(Pj)>Counter(Pi);
67     DO B := {Pi < Pj}
68 WHILE P ≠ ∅;
69 ENDFOR
70
71 RETURN Pi ∈ B such that is minimal with respect to <.
72
73 ENDPROCEDURE
```


Figure 3: Algorithm for Efficient Complex Design

```
1  PROCEDURE Optimal_ComplexDesign(Γ, α; α_2)
2
3  -- given available locations and resources
4  L := {A < ... < N};
5  Γ ⊂ Γ' := {u_i, ... u_n};
6
7  -- construct the set of non-required resources
8  Efficient_SimpleDesign(Γ''_1, α_1);
9  Efficient_SimpleDesign(Γ''_2, α_2);
10 Δ := Γ\Γ''_{1,2};
11
12 -- construct maximal union Efficient_SimpleDesigns
13 i := 0;
14 WHILE Δ ∪ Γ' ≠ ∅
15 DO
16   Γ = ∅;
17   FOR EACH u_i ∉ Δ ∩ Cn((Γ''_1, α_1) ∪ (Γ''_2, α_2))
18        Γ ∪ {u_i};
19   ENDFOR
20 RETURN (Γ, α_1; α_2)
21
22 ENDPROCEDURE
```

Figure 4: Algorithm for Optimal Complex Design

is an undecidable problem.

Limiting oneself to the decidable aspect of the meta-theory of the logic of design, there are still important issues to be answered. In particular, resource accessibility and correctness are essential ones to qualify good designs. Questions to be answered concern how efficiently can a system satisfy a given set of requirements, i.e. what is the minimal amount of resources required to satisfy the requirements; and whether a given way of satisfying intended requirements is compatible with additional, non essential ones. We have here considered these two problems, by describing a language for designs and investigating efficiency (minimality) and optimality (maximality) as structural properties on the associated derivability relation (which needs not to be intended as the only derivability relation privileged by the logic of design). We have provided results on their normal forms, lenghts and computability.

The problems that can be treated within this framework are various: determining that there exists neither a more efficient nor a more optimal design for the intended blueprint; determining that there exists no alternative algorithmic process for the intended blueprint; determining that a design is efficient with respect to a given output; determining that the output of a design is as desired and as intended by the given blueprint; determining the allowed variants of a given blueprint in terms of its designs, and the design variants which are not allowed with respect to the intended blueprint. Most of these problems reformulate conceptual and formal questions from the literature of computational logic within the logic of designs. Their formulation is left for future research.

A major philosophical import of this analysis is reflected in the distinction between disfunctioning systems (systems with a design that permanently or temporarily does not satisfy some or all of the intended functionalities of their

blueprint) and misfunctioning systems (systems with a design that allows for functionalities not intended by their blueprint, while possibly preserving the intended one). A computational approach to these system design processes can sensibly improve our abilities to establish, prevent and control malfunctioning.

Acknowledgments

The author wishes to thank three anonymous reviewers for extensive and helpful comments on previous versions of this work.

References

Faggian, C. and Hyland, M. (2002). Designs, disputes and strategies. In Bradfield, J. C., editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 442–457. Springer.

Fawcett, W. (1987). A note on the logic of design. *Design Studies*, 8(2):82 – 87.

Fernández, M. and Siafakas, N. (2014). Labelled calculi of resources. *J. Log. Comput.*, 24(3):591–613.

Floridi, L. (2017). The logic of design as a conceptual logic of information. *Minds and Machines*, 27(3):495–519.

Floridi, L., Fresco, N., and Primiero, G. (2015). On malfunctioning software. *Synthese*, 192(4):1199–1220.

Galle, P. (1997). Towards a formal logic of design rationalization. *Design Studies*, 18(2):195 – 219.

Girard, J. (2001). Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506.

Girard, J. (2003). From foundations to ludics. *Bulletin of Symbolic Logic*, 9(2):131–168.

March, L. (1976). *The logic of design and the question of value*. Cambridge University Press.

Nederpelt, R. (1994). Strong normalization in a typed lambda calculus with lambda structured types. *Studies in Logic and the Foundations of Mathematics*, 133:389 – 468. Selected Papers on Automath.

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition.

Primiero, G. (2015). Realist consequence, epistemic inference, computational correctness. In *The Road to Universal Logic*, volume II, pages 573–588. Springer.

Sifakis, J. (2013). Rigorous system design. *Foundations and Trends in Electronic Design Automation*, 6(4):293–362.

Zeng, Y. and Cheng, G. (1991). On the logic of design. *Design Studies*, 12(3):137 – 141.

Zowghi, D. and Coulin, C. (2005). *Requirements Elicitation: A Survey of Techniques, Approaches, and Tools*, pages 19–46. Springer Berlin Heidelberg, Berlin, Heidelberg.