A Taxonomy of Errors for Information Systems

Giuseppe Primiero
Centre for Logic and Philosophy of Science
Ghent University

Preprint. In *Minds & Machines*. DOI: 10.1007/s11023-013-9307-5.

Abstract

We provide a full characterization of computational error states for information systems. The class of errors considered is general enough to include human rational processes, logical reasoning, scientific progress and data processing in some functional programming languages. The aim is to reach a full taxonomy of error states by analysing the recovery and processing of data. We conclude by presenting machine-readable checking and resolve algorithms.

1 Introduction

The topic of error detection and resolution has been of crucial importance in epistemology at least since the Popperian doctrine of science as a process of conjectures formation, hypothesis refutation and theory change, see [21]. Neither Popperian falsificationism nor the large debate that followed, including Lakatos and Kuhn, were interested in establishing sources and typologies of errors and to design appropriate resolution strategies. Since then, the identification of errors with hypothesis refutation in the context of scientific progress has undergone a significant shift in background and application domain. A great deal of such research is devoted nowadays to the treatment of uncertainty by statistical or Bayesian methods, see e.g. [17]. We shall in the following be concerned with a more general approach to uncertainty and error and will not be engaged with a statistical, rather with a purely logical approach. If the present investigation is different in methodology, we share the view that correctness in rational processes can be approached as a system of probing, manipulating and simulating errors.

In the past few decades, logical approaches to agent-based, defeasible and bounded knowledge have identified errors against a principle of correctness as idealized rationality. This approach is evident for semantic theories that define truth on *evidence* and establish relevant criteria for *correctness*. We shall in the following define a computational semantics that appeals largely to the understanding of true contents in a procedural, quasi-verificationist way. But our task is also dictated by a further generalization: the massive presence of – and interaction with – mechanical computation in data and processes analysis suggests that natural and artificial reasoning have similar methodologies, including the case of error-handling. A conceptual and formal identity existing between proofs and programs, falling under the so-called Curry-Howard isomorphism for verificationist semantics,[1] allows us to use a computational approach typical of programming for human knowledge processing. In a similar vein, the understanding and control of errors is crucial in both human and mechanical systems to establish correctness. Using this analogy, we shall formulate a common semantic analysis for human and mechanical systems to study information processing under uncertainty and to define correctness as a process of error reduction. This requires:

- a full characterization of error states for informational systems;
- a formal model of logical processes with error states.

We focus here on the first of these problems. Starting from an informational semantics, we formulate a taxonomy of errors: these will include phenomena typically treated by literature in psychology (see e.g. [24]); fallacies considered in epistemic logic and in real-life cognitive agency ([31]); inexact knowledge and its limits ([28], [29], [30], [8]); presupposition failures ([7]). The semantics is also meant to cover error cases as represented in data compiling for some functional programming languages, and in particular we will show this feature by an implementation.

In the following we proceed as follows. In section 2 we characterize the systems of interest in view of an informational semantics that uses a quasi-verificational, procedural basis to couple instructional and semantic information. In section 3 we define the scope of errors for the epistemology of such systems as the space between knowledge and ignorance, where uncertainty occurs. Here a first categorization of errors is provided. In section 4 the proper taxonomy is given in view of two main levels of analysis and three grouping categories. In section 5 we provide a probing method for the families of errors defined by designing error check and resolve algorithms in a type-checked language.

2 Characterizing Information Systems

In this section we formally introduce an *information system*. By this term we intend a system that performs either an epistemic or a computational process based on an informational semantics.[2] A reason to formulate our system in view of such a semantics is that it offers a neutral interpretation broad enough to include both an operational and a denotational interpretation of validity. On the

[1]The isomorphism originates in observations by Curry [10], [9] and Howard [14]. For a systematic formal treatment of the issue, see [25].

[2]For a full treatment of the notion of informational semantics and of the operations it defines, see [4].

one hand, the system is structured around states at which a given informational content is declared to hold, thus offering a denotational interpretation of validity. On the other hand, it sets this static approach within a procedural setting (typical e.g. of Abstract State Machines), by allowing two main operations on such informational contents: *access* and *use*. In this way, the local validity of contents is explained back in terms of the instructions needed to reach a given state; global validity is given by the set of states the system goes through to reach a given goal. Moving to reach a goal is explained by: *accessing* the information at a given starting state; *using* that information by performing syntactic transformations on it; obtaining the next state. The structural and the functional aspects of the system offer together a descriptive and normative structure. In the following we shall consider a language with: countably many types; terms as objects in types; a formula of the language is then the expression that an object is of a given type; a finite number of operations is included that work on terms to define non-atomic formulas; we shall allow sets of formulas; finally, states corresponds to models of the language where types are declared valid.

Definition 1 (Syntax) *The syntax is defined by the following alphabet:*

$$\Sigma(Types/Props) := \{A, B, \ldots\}$$

$$\Pi(Processes) := \{a_1, a_2, a_3, \ldots\}$$

$$\mathcal{C}(Data\ Stacks) := \{\Gamma_i, \Gamma_j, \ldots\}$$

$$\mathtt{S}(States) := \{S, S', \ldots\}$$

$$\Phi(Operations) := \{r_i, \ldots, r_n\}$$

Types of our system express propositions true by terms (proofs): for example, the construction of type $\mathbb{N}$ will be generated by the proposition that term 0 is in $\mathbb{N}$ and the operation that says that $S(0)$ is a term in $\mathbb{N}$. Similarly, When the term is defined in the form of a program accessed and executed at some state (e.g. the program that generates natural numbers according to the successor operation), its type corresponds to the specification declaring validity of a content by that program (i.e., the list of elements in the set $\mathbb{N}$). Under this view, an information system includes computational systems as a proper subclass, by offering a reading of valid types as *specifications* and of terms as *programs*. In this setting, a specification is not based on a stipulative but rather on a functional interpretation of the system, see [27].

Definition 2 (Specification α) *A specification $\alpha = (A\ valid)$ is the content made valid by accessing and using a program $a_i \in \Pi$ of type $A \in \Sigma$ at state $S \in \mathtt{S}$.*

When a sequence of processes or programs is accessed and executed at states S_1 up to state S_i to reach a specification α at state S_{i+1}, this (possibly empty) collection of processes is referred to as data stack (or context) Γ_i. To every non-terminal state S in Γ_i, a finite set of rules applies to reach content valid at S':

Definition 3 (Operational Model) *In a state $S = (\Gamma_i, \alpha)$ a process a_i is executed instantiating a specification α under a (possibly empty) set of other states collected in Γ_i. A model for our system evaluates every S by transition to some S'. A transition system is a triple $\langle \mathcal{S}, \phi, \mapsto \rangle$ with $\mathcal{S} \subseteq \mathtt{S}$, $\phi \subseteq \Phi$ and $\mapsto$ a ternary relation over states $(\mathcal{S} \times \phi \times \mathcal{S})$. If $S, S' \in \mathcal{S}$, then $\phi(S \mapsto S')$ means that there is a transition $\mapsto$ from state S to state S' according to the set of rules in ϕ.*

Example for such commands in Φ would be: execution, local compound execution, functional composition, mobility rules, etc. We abstract from making them epxlicit as our system is intended at a very high level of abstraction.

Definition 4 (Goal) *We say that a state $S = (\Gamma_i, \alpha)$ is the goal $\mathcal{G}$ for the system if α is its intended specification, i.e. the processes in Γ_i are supposed to make the type A in α valid.*

Definition 5 (Strategy) *We call strategy $\phi \subseteq \Phi$ the collection of rules or instructions that, given a certain initial state, are used by the system to reach a goal $\mathcal{G}$ by accessing and using the information valid at intermediate states. When applied to a goal, a strategy replaces this goal with its sub-goal(s).*

Definition 6 (Procedure) *We call procedure the pair $\mathcal{P} = \langle \phi, \mathcal{S} \rangle$ defined by a set of states and a strategy by which a given goal $\mathcal{G}$ is correctly reached.*

A procedure offers instructions on how to use the information at a given state to access information contained at the next state.[3] At each (sub-)goal, $\mathcal{G} = (S_i \in \mathtt{S})$, accessed data Π are attributed a semantic value; the system is in an epistemic state with respect to each such content in that it uses it to access data at the next state. The rule that allows to reach a final state is usually a deductive rule, of the form: "If information A_1 is accessed at S_1, up to information A_n accessed at S_n, then information A is validly accessed at S'. To reach a goal corresponds to be informed about the content of the goal-expression, $I_S(\alpha)$:[4]

$$\frac{\mathcal{P} \text{ is a procedure for } A}{A \ valid}$$

The internal structure of a procedure $\mathcal{P}$ offers by definition the set of states and their computational contents $\{a_1, \dots, a_n\}$ to obtain the final content A:

[3]For the logic of becoming informed that applies at the instructional level of states, see [22] and [23].

[4]For the logic of being informed that applies at the level of goals, see [11].

$$\frac{a_1 \dots a_n \text{ are processes for } A_1, \dots, A_n}{A \textit{ valid}}$$

By this rule-based construction, the system instantiates the required state-transition from functional to semantic information[5]:

$$\frac{\text{Information } A_1 \text{ holds} \qquad \text{Use } A_1 \text{ to access } A_2, \dots \text{ Use } A_{n-1} \text{ to access } A_n}{\text{Information } A \text{ holds}}$$

When the previous inference schema is fixed at states $1, \dots, n$, it means that there is no further process that can falsify A, which therefore becomes a knowledge content:[6]

$$\frac{\text{Information } A_1 \text{ holds} \qquad \text{Use } A_1 \text{ to access } A_2 \dots \text{ Use } A_{n-1} \text{ to access } A_n}{A \text{ is known to be valid at states } 1, \dots, n}$$

Correspondingly, information *inaccessibility* generates a state of ignorance:

$$\frac{\text{Information } A_1, \dots, A_{n-1} \text{ holds} \qquad \text{Information } A \text{ cannot be accessed at } n}{A \text{ is not known to be valid at states } 1, \dots, n}$$

In the next section, we will explore the epistemic and semantic space included between knowledge and ignorance and characterize it as the scope where errors occur.

3 Scope and Categorization of Errors

A procedure $\mathcal{P}$ formulating validity of content A is also said to resolve *uncertainty* with respect to A. There is a range of epistemic states that goes from knowing A to being ignorant about A:

1. information access and use induce correctness of content A and knowledge of its validity at all accessed states;

2. inaccessibility induces full uncertainty about A and corresponds to ignorance;

[5]The two notions of semantic and functional information are complementary and non-exclusive. See respectively [12] and [23].

[6]In [23] the upgrade from functional information to knowledge is explained by interpreting information use in terms of a verification function to make data semantically qualified. Knowledge requires the network in which those contents are accessed to be no greater than the set of states where such information cannot be turned into misinformation (a localized consistency requirement). A content of semantic information becomes a knowledge content if it is accessible and usable from *every other state* of the same network without consistency being lost.

3. between knowledge and ignorance graded uncertainty is possible.

Establishing the semantic value of contents is to be understood as moving from an uncertainty state to a knowledge state: for every correctly executed process a_i of content A_i, the informational system is said to transit from state (A_i *valid*) *unknown* to state (A_i *valid*) *known*. An explanation of the notion of error is given therefore in terms of wrong resolution of uncertainty.[7] This task is commonly understood in view of Peircean qualitative and quantitative induction, which provides the theoretical model for Bayesian statistical error detection and correction, with the addition of errors of first and second kind in the correction of hypothesis procedure formation.[8] Notoriously, by inductive methods one refers usually to both numerical statistical approximation and deductive (qualitative) inferences. In the present context, we are not considering statistical errors and their resolution forms. We are instead looking for a qualitative analysis to formulate a descriptive taxonomy of *error types*. Our model is non-probabilistic and uncertainty does not correspond to a variable degree of trustworthiness attached to the method, nor is bound to its degree of approximation to truth. Our understanding of uncertainty refers instead to possible restrictions or limitations on the execution of valid and correct processes, due to limited accessibility or incompleteness of data required for the problem resolution. This notion of uncertainty fits with the informational semantics endorsed in the previous section. Validity of a procedure $\mathcal{P}$, on which access to and validation of content A is based, requires:

1. full data availability;
2. full reconstruction of dependency relations internal to $\mathcal{P}$;
3. explicit competence in their use.

In [26] an analysis of errors for a proof-theoretical semantics provides a first basic distinction between *errors* and *mistakes*. On the one hand, a mistake is understood as a wrong act which is immediately recognized and can be easily fixed within the *paradigm* in which it is generated, as the latter establishes the conceptual and technical tools to define what is right and what is wrong. On the other hand, an error is considered as a more basic and ground failure, something that prevents knowledge to be attained, without an evident and clearly recognizable cause. Our task is to offer a more precise and detailed description. We start by referring to an error as a *non-realizable* process a_i to access an information content A, based on the pair $\langle \mathcal{P}, \mathcal{G} \rangle$. In the following, given the procedural understanding of our semantics, correctness corresponds to correct execution of a process; validity corresponds instead to the selection of the appropriate process for the selected goal. This allows us to design two cases of things going wrong:[9]

[7] For the relation here explored between error and negative knowledge, see also [2], [3].

[8] See [20], [17], [18].

[9] The pair $\langle \mathcal{P}, \mathcal{G} \rangle$ by definition corresponds to a triple $\langle \phi, \mathcal{S}_{1\to i}, \alpha \rangle$, where ϕ contains the set of required strategies and $\mathcal{S}_{1\to i}$ the set of states/processes the system goes through from 1 to i to reach the goal $\alpha = (A\ \textit{valid})$ at state S_i.

- *wrong informational coupling*: an error in building the pair $\langle \mathcal{P}, \mathcal{G} \rangle$, where $\mathcal{P}$ is inappropriate, though possibly well-executed and therefore correct, procedure to validate content A in $\mathcal{G}$;
- *informational malfunctioning*:[10] an execution error which makes $\mathcal{P}$ an incorrect procedure for $\mathcal{G}$, but when executed correctly, $\mathcal{P}$ is indeed a valid procedure for content A in $\mathcal{G}$.

In both cases, uncertainty with respect to goal obtains:

- a missing procedure $\mathcal{P}$ for A corresponds to total uncertainty with respect to $\mathcal{G}$;
- a malfunctioning procedure $\mathcal{P}$ for A corresponds to partial uncertainty with respect to $\mathcal{G}$;
- a wrong informational coupling of $\mathcal{P}$ with A corresponds to wrong certainty with respect to $\mathcal{G}$.

An error generated from a wrong coupling of process and goal corresponds to the selection of a (possibly correct) but inappropriate procedure to validate a content which is thus inaccessible:

Definition 7 (Error by inappropriate procedure) *A procedure $\mathcal{P}$ is executed to access $\mathcal{G}$, but $\mathcal{G}$ requires a different procedure $\mathcal{P}'$.*

This case has multiple instances. In the most relevant case, it refers to the selection of an old process for a new content, which in its strongest form suggests a *categorical or paradigmatic change*. This typically happens when one tries to solve a (conceptually novel) problem using known methods, whereas resolution is possible only by moving to a novel conceptual schema. We shall refer to these cases below as *mistakes*, the family of errors happening at the conceptual level.[11] This family will also include errors in strategy or rule design. Typically, such cases have no correction procedure immediately available: if one designs a specification or the corresponding algorithm wrongly, it will not be enough to reinitialize the procedure to realize where one goes wrong, as the error occurs at the level of definition and requires a major redesign.

In a more simple case, an error by inappropriate procedure refers to the *selection of bad rules*: to access content A, one applies rule r_i which allows to reach a state S, whereas one should have used rule r_j to reach state S'. In turn the accessed content is wrongly declared valid. This error involves therefore the procedural aspect of transit from state to state, rather than the static relation

[10]This notion lies at the very basis of the recent philosophical analysis of logical and technical malfunction for engineering and semantic systems. See e.g. [6], [13], [16], [15].

[11]Discussion with B.G. Sundholm has clarified the relation with paradigm changes. Notice, however, that we depart here from the terminology used in [26], where mistakes are explained as simple acts gone wrong. This is due to our more general use of the term 'error', and it also agrees with a similar use of the term 'mistake' from literature in psychology, see [24].

between a given process and the related content. For this reason, the selection of bad rules will be identified as a form of *failure*, a family that collects errors at the procedural level. Typical examples would be errors performed by *insufficient data encoding*: the way one (wrongly) specifies the use for the (right) rule; and strategy selection: the (faulty) selection of data on which the (right) rule is applied. Also in this case, a correction procedure is not necessarily easily available: if one applies correctly designed rules in the context of wrong data, or with a wrong interpretation, or equipped with correct but inaccessible data (e.g. in the case of large distributed databases), a major structural correction will be required. Failures as conceptual errors of a procedural kind fall under a second typology, referring to the invalid formulation of a process:[12]

Definition 8 (Error by invalid procedure) *An invalid procedure $\mathcal{P}$ is used to access $\mathcal{G}$ with specification $\alpha = (A\ valid)$, where in fact a valid procedure $\mathcal{P}'$ for content $(\neg A\ valid)$ holds.*

Finally, the third main typology of error is one of pure analytic malfunctioning due to the faulty execution of an appropriate procedure:[13]

Definition 9 (Error by appropriate but incorrectly executed procedure) *An appropriate procedure $\mathcal{P}$ is used to access $\mathcal{G}$, but $\mathcal{P}$ is not correctly executed.*

It is based on an *incorrectly executed process* which has a correction mechanism available within a given paradigm. In fact, error correction in this case is seen as a reinforcing process for the given conceptual schema in which the process occurs. Usually, this kind of errors are due to wrong execution of rules appropriate to the required goal and they include cases of failures (namely those falling under the flag of *inaccurate data encoding*) and the family of *slips*.

Notice that none of the cases above should be confused with the state obtained by a correct process for a negative content (as already shown by the correctness of the procedure referred to in Definition 9):

Definition 10 (Negative Content) *A procedure $\mathcal{P}$ allows to access $(\neg A\ valid)$ (alternatively, $\mathcal{P}$ is a procedure to establish that A is misinformation).*

Hence contents might be wrongly accessed in two senses: either because the correct procedure is wrongly formulated, or because the wrong procedure is selected (though possibly correctly executed). This indicates that access alone is not a guarantee for an error-free system and error detection proceeds on the selection and execution of functional information. Information access and use *with* an error probe method guarantees an error-free information system.

[12]A counterpart of this case in simple propositional terms is an 'incorrectly justified, false claim'.

[13]The content A in this case is propositionally treated as a 'faulty justified true claim'.

4 The basic taxonomy

Despite the semantics of sentences of the form '$\mathcal{P}$ is a procedure for (A *valid*)' offers a clear distinction between conceptual and material errors, some overlapping in the previous section can been noticed: when considering errors caused by an *inappropriate procedure*, we refer to both mistakes and conceptual failures of process execution; material failures fall under the flag of errors by *invalid procedure*; finally, errors by a *malfunctioning procedure* include a specific group of material failures and slips. To provide a better explanation of system correctness, we enhance the analysis of errors by formulating three general categories:[14]

1. *Conceptual Validity*: it is related to the conceptual description and design of the specification or goal;
2. *Procedural Correctness*: it is related to the purely functional or procedural aspect of the specification or goal;
3. *Contextual Admissibility*: it is related to both conceptual and procedural aspects of the environment in which the specification or goal is designed and executed.

The order in which *correctness* of the procedural/contextual level and *validity* of the conceptual level are considered is relevant. We shall consider both the definitional order according to which a valid content is accessed by a correct procedure, and the reverse relation that considers a procedure correct if it allows to access a valid content. To further analyse the two aspects, we apply some additional structure:

	Conceptual	**Procedural**	**Contextual**
Internal Level	Specification Description	Process Construction	Dependency Recursion
External Level	Problem design	Data retrieval	Dependency accessibility

[14]A basic taxonomy for human reasoning given in [24] categorizes errors primarily according to a specular threefold structure of conceptual, behavioural, and contextual levels. Whereas certain sorts of behavioural errors will be excluded from the present taxonomy as they do not fall under the level of abstraction we are considering, we aim at maintaining our taxonomy as general as possible and claim that our notion of procedural error levels for information systems include various cases of epistemic errors common in the behavioural family for human reasoning. Typical cases of such errors that we will not consider are those induced by attention problems, memory problems, voluntary and involuntary deceptions, and the like. Nonetheless, some further reductions might be possible, as for example of memory problems in terms of the later introduced family of *storage errors*.

At the internal level, one starts with the *specification* or *problem description*, consisting in the choice of the goal $\mathcal{G} = (A\ valid)$ for which a procedure is to be formulated. This consists in the purely semantic task of laying down conditions and definitions required for A to be a valid specification with a corresponding procedure in terms of correct processes and strategies. An error in the target formulation will inevitably induce an error state in terms of a conceptual mistake. The following step consists in the *procedure* or *process construction* $\mathcal{P}$ for the content A. Depending on the kind of error, this case generates either an incorrect procedure or an informational malfunctioning. The additional category offered by the contextual formulation of procedures refers to the *definition of dependency recursion relations* in $\mathcal{P}$ among processes $a_1, \dots, a_n$, which have to be called upon in the specification and procedure design in order a to be a correct process for $A\ valid$.

Where the internal level looks at the semantics of the system from the point of view of *resources description*, the external level will look at *resources access and execution*, i.e. from a practical perspective. At the conceptual level, one requires the most appropriate specification *design* for the implementation to be obtained. At the procedural level, procedure $\mathcal{P}$ invokes data retrieval for the main process a as required for the chosen design of A. Finally, the contextual category refers to the effective accessibility (at location, in terms of computability problems) for routines $a_1, \dots, a_n$ required by the selected data a.

There is a formal correspondence of this schema with the steps that a type-checker would execute in verifying the correctness of a formal system: the internal cases refer to decidability from the viewpoint of type-reconstruction or type-inhabitation; correctness refers to the source of error with respect to sub-calls and recursion processes. Errors of data retrieval address the correctness problem referring to the *contents* of the computational process, while errors affecting the dependency relations refer to the *structure* of the computational system. This basic categorization can be generalized by organizing types of errors in view of two ground categories (conceptual vs. material) and three main families:[15]

[15]This classification agrees with the one for errors in science from [2]. In the following, we shall not consider the *observational* and *discursive* kinds that are included in that analysis.

Type of Error	**Conceptual**	**Material**
Mistakes	Problem Description: Categorization	Problem design: Category Structuring
Failures	Procedure Definition: Form of main process	Procedure Construction: Accessibility of dependent processes
Slips	Algorithm Design: Efficiency	Algorithm execution: Performance

This enhanced schema provides now a more detailed description of the problems occurring in the validation procedure, distinguishing clearly between conceptual errors and their material occurrence. For the purposes of the inter-disciplinary use of our taxonomy, it is appropriate to stress the correspondence of this categorization with the usual terminology in software engineering (SE) and the related error-handling methodology. We have chosen to use the term *error* as a general category, while this term is used in SE typically to refer to inappropriate decisions at the development stage. An error occurs then at system requirements analysis or at system design, resulting from a gap in requirements which then propagates to the coding stage as an error in the program code. These 'errors' are more precisely defined by our family of *mistakes* as categorization errors. An error of this form is usually bound to induce a *defect*. In SE a defect refers to the difference between expected and actual result in the context of software testing. It corresponds typically to any flaw possibly induced by a mistake, e.g. in scalability, occurring at program execution (runtime), which is why we consider it as the occurrence of a faulty procedure and have included it in our family of *failures*. An example of defect induced by a designer/developer mistake is a piece of code leaving an undefined procedure generating an infinite loop, which in turn induces consumption of all available memory. If the latter further produces a system crash, we are moving down to a *failure* in SE terminology. In SE a *failure* is any unacceptable behaviour of the system, with the main case of system crash. This terminological practice in SE is slightly confusing, as it takes a failure as an event (i.e. the non-expected behaviour) rather than as its cause (in this case, exhausted memory). Nonetheless, we can still consider failures in SE as the results of any case of errors from our taxonomy, including *slips*, when e.g. efficiency is so reduced that it causes system halting.[16] To sum up: though the terminology in software engineering is somewhat less precise, our categorization remains largely valid for it as well:

[16]For a treatment of error-handling in software engineering see e.g. [1].

- breach of validity conditions at system requirements design and algorithm design will fall naturally in our class of conceptual errors, thus being classified as mistakes;
- breach of correctness conditions at algorithm implementation and execution will fall in our class of material errors, thus being classified as (procedural) failures;
- any further case of error, e.g. at user interaction level will occupy the remaining class of slips.

In the following we analyse in detail each of the error kinds introduced in this section.

4.1 Mistakes

Mistakes are errors involving the description and the design of the problem to be solved, or the specification to be implemented. We refer to them as *categorization* or *planning errors.* Mistakes involve the norm of correctness that the task at hand requires, resulting from a faulty or incorrect explanation and presentation of the object for which a validation procedure is required. To list cases of mistakes or planning errors, we consider in which form the invalidity of a content A for which a term a is sought can occur:

1. *conceptual mistake*: the analysed pair $\langle \mathcal{P}, \mathcal{G} \rangle$ contains or refers to a ill-defined category A in the environment. The conceptual aspect of mistakes refers explicitly to the categorization and semantics of concepts needed to formulate correctly the problem that one wishes to solve, or the function one wishes to execute. This corresponds formally to wrong categorization, in the form of presuppositions failure required by the target A *valid.* As an example, consider the formulation of a specification that requires as its subroutine an algorithm for quotient by 3, including the post-condition that no reminder be allowed. To define the domain of such function as containing anything besides multiples of 3 would represent a conceptual mistake. A more extreme case would be that of a goal including contradictory or circular definitions;[17]
2. *material mistake*: it refers to the structural design of the strategy, where a pair $\langle \mathcal{P}, \mathcal{G} \rangle$ is given that includes elements $r_i \in \Phi, a_i \in \Pi$ that do not constitute a strategical (sub-)goal to $\mathcal{G}$. The material aspect of mistakes refers therefore to the design of the specification or problem, in terms of the choice of elements, functions and definitions selected to resolve a problem. The example above can be easily adapted to its material counterpart when considering the design of an algorithm implemented by the following pseudo-code: `var isMultipleOfThree = function (x)  return x % 2 === 0;`. Notice how in software engineering jargon this would be a

[17]Notice, however, that mutual or co-recursive definitions do not need to be circular.

logical error, as code syntax is correct and the algorithm would execute correctly, so that only at run-time it would be discovered that the output does not match the required specification. This means that at the design level an error has occurred in implementing the system requirements specification, hence we categorize this as a material case of mistake.

4.2 Failures

Failures are errors explicitly referring to the rules used in the evaluation and resolution of the problem (respectively, the validation of the specification) or related to the resources these rules have to access. Hence failures occur internally in $\mathcal{P}$ and they are typical of real implementations, where bounded resources and incomplete information are the standard. Also in this case, we analyse both the conceptual and the material levels.

1. *Conceptual failures* reflect problems in the selection and formulation of rules or strategies, therefore we refer to them also as *execution errors*:

    - *selection of bad rules*: it corresponds to the (possibly correct) execution of the *wrong* process a within the given procedure for the problem/specification A at hand; it generates an illegal application such that the wrong rule $r_i \in \phi$ is selected for the current (a_i, A) pair. A simple example is the selection (and correct execution) of a *conjunction elimination* rule for the resolution of $A \vee B$;
    - *mis-formulation of good rules*: it corresponds to the *faulty* formulation of a validly selected rule in a process; it happens at the level of form-designing of a rule, where the rule or strategy is correctly chosen but is later wrongly defined.[18] A simple example is inferring $(A \wedge B) \vee C$ by application of *conjunction introduction* from $A \wedge B$;
    - *insufficient data encoding*: failures of this kind are particularly important for the study of reasoning processes with bounded resources. They can be listed as follows:
        - selection of wrong goals;
        - selection of rule or procedure with insufficient computational depth;
        - selection of construction or context with wrong sub-typing;
        - selection of strategy or language with insufficient rules-set.

    The first case (wrong goal) refers to a limited perception of possible tasks to be performed: this might induce selective and possibly excessive exploration of some goals and exclusion of other relevant ones. The second case (rule or strategy selection) reflects the wrong selection of a solving model (deductive, inductive, analogical) or rules (connectives and quantifiers introduction and elimination; validity of

[18]For the corresponding case of faulty algorithm *execution* see below the category of *slips*.

structural rules such as premise exchange) based on the problem presentation. For natural reasoning, it might refer to a cognitive strain that induces a certain inappropriate choice; in the case of mechanical reasoning it can be induced by data presentation suggesting the selection of a wrong solving strategy. The third case (construction selection) has also different possible applications: task selection might go wrong so that relevant information (either in the process or in its context) is not formulated; it might refer to the structure of contexts itself, by a wrong ordering of relevant data or by applying a confirmation bias that dismisses the contextual relevance in following uses of the same process. Finally, a ground form of failure in this group is the selection of an incomplete set of data or strategy procedures to solve the task at hand.

2. *Material failures* reflect problems related to the *accessibility* of the resources required for the correct execution of a procedure for the problem or specification at hand. We refer to them also as *storage errors*:

    - *misaddressed resources*: the required resources, possibly available in the current environment, are addressed by incorrect or insufficient instructions; typical examples are wrong indexing preservation on contexts for logical expressions or wrong addressing of databases;
    - *non-reachable resources*: the resources are well-defined but beyond the scope of the procedure, i.e. not available in the current environment; typical examples are the use of constants or variables that miss appropriate declaration, or databases protected by user-privileges;
    - *inaccurate data encoding*: processes are wrongly categorized or dependencies are wrongly formulated. The first reduces to a case of mistake, inducing a failure in the context of a dependent process (possibly well-categorized); the second refers to the order of execution of the routines required by the process, their locations, or nested requirements that are not made explicit. Inaccurate data encoding happens in the following forms:
        - *by inattention*: omitting checks, including selection of the wrong path of a branching tree, under-use of rule (e.g. missing to go through any branch of a disjunctive rule), missing search for (sub-)goals space; missing novel variable declaration and wrong (sub-)typing by accident;
        - *by over-attention*: inappropriate checks, including unnecessarily repeated novel variable declaration, establish a wrong level of abstraction and the overuse of rule (e.g. acting on both branches of a disjunctive rule).

4.3 Slips

Slips are material errors generated by the applications of rules that are appropriate to the given goal, but that do not match some formal criteria. Though it still refers to the internal structure of the procedure $\mathcal{P}$, this group of errors is different from the previous one because it does not necessarily induce a non-evaluated state of system or does not falsify a transition rule from state to state. Slips include also use of inadvisable rules (which might also lead to a valid – though non-required state) or inelegant applications thereof. The distinction between the conceptual and the material levels is blurred, as slips occur by rule application by an agent performing a task, but at the same time they implicitly refer to the rule as it was given at design level (in these cases they will most of the time induce a corresponding mistake). However, also in this case a ground distinction can be formulated:

1. *conceptual slips*: these are practical errors related to algorithm design, i.e. where the selection of range and domain, the order of rules applied and sub-recursion definitions are chosen for an *efficient* algorithm; a typical example is that of an algorithm designed for testing properties on inputs (e.g. the property of being even on integers) and submitting some inappropriate (though not invalid) input (e.g. the whole domain of the naturals, instead of the output of the function $n \times 2 - 1$ thus elegantly producing the new required range by concatenation with an identity function); in the case of invalid input, a mistake occurs;

2. *material slips*: these refer instead to a performance problem, in terms of redundant steps or rule strength. These are some typical cases:

    - *Exceptions*: the rule is applied within a category that accommodates it, but with respect to a construction/individual that represents an exception;[19] also in this case, if the exception is not accounted for because of a bad description or understanding of the problem then, most likely, this is a case of mistake;
    - *Rule strength*: the rule is applied in a strong setting, admitting its global validity, whereas the current context allows only a local validation of the rule; an example of such error case is the formulation of restricted versions of structural rules like weakening, whose validity might depend from structure or complexity issues;
    - *Rigidity*: a fixed set of data or rules is selected for different tasks that can be more easily resolved by different kinds of strategies. As an example, consider proving all theorems by contradiction, while a proof by induction might generate a more elegant presentation of the valid theorem.

[19]Exceptions are largely used in knowledge representation problems by means of description logics, where default rules are used to state and infer relations that are true only in 'normal cases'. See e.g. [5].

Notice that the procedural basis of our information system allows for a first-person based perspective, what one could call here the designer-compiler perspective. Under this reading, rule and knowledge-based errors that subsequently induce attempts to find solutions can be further categorized as *failure in expertise*. This would be the case of slips. On the other hand, mistakes and failures indicate cases of *lack of expertise*, ground errors that do not necessarily have immediate solutions.

4.4 Triggering Errors from Errors

An error can also be triggered by the occurrence of another error. Some relevant cases are explored in the following:

- *from mistakes to failures*: a specification design error induces an error in rule execution; as a very trivial example, consider the case of designing a division function for a programming language, where the designer does not throw an appropriate exception on the value `zero` as divisor; the algorithm execution will not be constrained in the appropriate way and will eventually search for a valid output value on `division(n,zero)`, which mis-formulates the rule of division;

- *from failures to mistakes*: a resource based failure, e.g. in devising as appropriate a wrong goal or strategy, induces a design error in re-assessing the specification design; as an example, consider the process of designing a logical proof, where rules selection determines at each stage the appropriate sub-goals; the use of a wrongly selected rule can lead the prover to a wrong sub-goal and in turn to a wrong design of the strategy for the final goal state;

- *from slips to mistakes*: a processing error induces an error in the intended (sub-)goal state; as an example, consider the case of an user that is required to input a password of a certain complexity strength; by a slip (either of attention or of mechanical processing on the keyboard), the user inputs a no-digits, no-symbol six-letters password; the algorithm will *correctly* reject this input, outputting an error state; but while the algorithm execution is correct, hence no failure occurs, the resulting final state of the human-machine interaction does not correspond with the intended goal (i.e. validate a password of a certain complexity), hence presenting a typical case of error by inappropriate procedure.

5 Error probing for information systems

We design now a method of error detection and resolution. The given taxonomy of errors for information systems induces a hypothesis testing strategy that amounts to an error probing method. Testing data happens in the construction of a new system, whose procedure consists of the following steps: first, establish the goal formula (specification); second, select the strategies within the procedure according to which the formula can be validated (process); third, formulate the appropriate context of resources for validating the specification (environment). At each of these steps an error might occur.

A severe testing method has to satisfy two conditions:[20]

- the test procedure must validate processes on a large account of the environment, i.e. the environment has to be sufficiently large for the validity conditions to be considered robust;
- the test procedure must be well-defined to establish valid processes; moreover, the test procedure must be itself independent from resources or conditions of the environment it checks.

The method that follows offers at once both the required properties: it validates a large environment by the formal, content-independent language used; and it is tested independently from the taxonomy that it defines, by mechanical type-checking of the algorithms involved in the error-probing method.

Our error probing method consists of two combined strategies. The first relies on the translation of the notion of information system and its components into a machine-readable syntax, which can be checked by a program. For this first step, we choose to formalize our method through the Coq proof assistant, based on the calculus of inductive constructions.[21] The task of a proof-assistant is typically to check proofs, in order to testify their correctness. By the formal identity underlying proofs and programs, one can use a proof assistant to test the correctness of a program that has the same logical structure of a given derivation. In Coq, elements of the language are types, inhabited by either propositions (with proofs as terms) or specifications (with programs as terms). The underlying logic for terms is the intuitionistic fragment $\{\wedge, \rightarrow, \vee\}$, extended to quantifiers and equality. Goals are reached by derivation of appropriate subgoals by applying tactics that use assumptions and provide rules to introduce or eliminate auxiliary propositions (different for each logical form available). Standard libraries include basic logical notations and properties, basic data types (boolean and natural numbers), operations $(+, \times, \min)$ and relations $(<, \leq)$. The logic can be axiomatically extended to a classical setting by introducing excluded middle. Additional libraries include e.g. the rules for algebraic laws or properties of orders, lists, basic functions and properties of lists. Programs use the definition of inductive types, predicates and families, structurally recursive programs, pattern matching.

The second step of our method consists in going beyond the pure proof of program-correctness by enhancing the type-checking algorithm with additional algorithms for detecting and resolving the error cases defined by our taxonomy. These algorithms are checked for formal correctness by the Coq proof-assistant. Besides plainly invalid statements, we consider here three distinct kinds of data that are treated as hypotheses and can be run for an error test in this process:

[20]These conditions present a strict analogy to the self-correcting thesis in Peirce.

[21]`coq.inria.fr`

1. assumption of a term for a new sub-goal generating no direct conflict in the current environment;

2. formulation of a valid process:

   of the same type of an already given process (redundant data);

   of the same type of an already given process, but generating a new sub-goal by rules (qualitatively different data);

3. formulation of a valid process generating an environment that is quantitatively stronger than the one without it and which allows to characterize how much of the older environment is falsified by the new data.

The error probing method consists in analysing the value of the (possibly newly generated) data, according to the typology given above. Once an error is detected, one defines an error type and returns an appropriate error object.[22]

For computational systems, a type-checking apparatus is appropriate to find errors at compile-time, but entirely insufficient to discover errors at run-time. Our task is therefore not to provide an error-probing method by type-checking; we rather present an algorithmic error-testing method that is verified correct by type-checking. While the `check` and `resolve` algorithms run over states of the system to discover and resolve any of the categories of errors introduce above, the type-checking algorithm is used to check their correctness by defining them as Coq-valid expressions.

We provide here few explanations needed for understanding the preamble of the code in light of the formal apparatus introduced in Section 2. Corresponding to elements of Σ (specifications), we introduce the expression `type`, a category of terms; as its element we define the category of expressions `Prop`, explicitly declared for the category of valid propositions, with metavariables $A, B, \dots$. In the following code, elements of `Prop` are contents valid at corresponding states of the system (the set $\mathcal{S}$ is not transparently translated into the syntax). Elements in Π correspond to `tm` in `Prop`: terms are the domain of functions with boolean values as range, depending on a term being or not being an element of the given type. The expression `value` refers to a normalized object and presupposes standard inductive definition on boolean values given by `bvalue`. A full evaluation function `full_eval` is defined as reduction from terms to terms in `Prop`. Notice also that the machinery *underlying* our proof-checking method (tactics and rules) is hidden at a lower level of abstraction (machine language): hence, the set Φ of strategies to derive contents is not transparent in the following definitions, but it is interpreted by the rules and derivation methods allowed by the proof-checker.

```
Coq <
Coq < Check Prop : Type.
```

[22]This strategy is extracted from the typing of errors in functional programming. See e.g. [19, ch.5]. Notice, however, that the analysis is here fully expanded in view of the error types generated by the previously given taxonomy.

```
Prop:Type
     : Type

Coq < Variable A B C : Prop.
A is assumed
Warning: A is declared as a parameter because it is at a global level
B is assumed
Warning: B is declared as a parameter because it is at a global level
C is assumed
Warning: C is declared as a parameter because it is at a global level

Coq <
Coq < Inductive tm : Prop :=
Coq < | tm_true : tm
Coq < | tm_false : tm
Coq < | tm_if : tm -> tm -> tm -> tm.
tm is defined
tm_ind is defined

Coq <
Coq <
Coq < Inductive bvalue : tm -> Prop :=
Coq < | b_true : bvalue tm_true
Coq < | b_false : bvalue tm_false.
bvalue is defined
bvalue_ind is defined

Coq <
Coq < Definition value (t : tm) : Prop :=
Coq <   bvalue t.
value is defined

Coq <
Coq < Inductive full_eval : tm -> tm -> Prop :=
Coq < | f_value : forall t,
Coq <     value t ->
Coq <     full_eval t t
Coq < | f_iftrue : forall t1 t2 t3 t,
Coq <     full_eval t1 tm_true ->
Coq <     full_eval t2 t ->
Coq <     full_eval (tm_if t1 t2 t3) t
Coq < | f_iffalse : forall t1 t2 t3 t,
Coq <     full_eval t1 tm_false ->
Coq <     full_eval t3 t ->
Coq <     full_eval (tm_if t1 t2 t3) t.
full_eval is defined
full_eval_ind is defined

Coq <
Coq < Require Import Bool.

Coq <
```

A procedure $\mathcal{P}$ (abstracting on states) is expressed in the following by `proc` as element of type `Prop`: it is defined by the characteristic function that takes a boolean term (i.e. a content of a state) and returns a procedure that construct that term. This way, a procedure is parametrized as a function from a term to a proposition, namely the proposition corresponding to the specification which is validated by the procedure. We also define extensionality properties on specifications in terms of their procedure: equality of specifications is given by identity of procedures; subset relation by procedural inclusion; an empty specification corresponds to no procedure available. Finally, given the implicit definition of a goal as the last state of a procedure (see Definition 4), we also define the term to which no proper procedure terminates as a non-goal.

```
Coq <
Coq < Set Implicit Arguments.

Coq <
Coq < Inductive proc : Type :=
Coq <     Charac : (A -> bool) -> proc.
proc is defined
proc_rect is defined
proc_ind is defined
proc_rec is defined

Coq <
Coq < Definition charac (s:proc) (a:A) : bool := let (f) := s in f a.
charac is defined

Coq <
Coq < Parameter In : proc -> tm -> Prop.
In is assumed

Coq <
Coq < Definition Equal A A' := forall t:proc, In t A <-> In t A'.
Equal is defined

Coq <
Coq < Definition Subset A A' := forall t:proc, In t A -> In t A'.
Subset is defined

Coq <
Coq < Definition Empty A := forall t:proc, ~ In t A.
Empty is defined

Coq <
Coq < Definition no_goal := exists t, forall t1, value t -> full_eval t1 tm_false.
no_goal is defined
```

```
Coq <
Coq <
```

We now proceed with giving inductive definitions for our error categories. The inductive definition of mistake includes three cases:

1. a *missing type*, defined as the function that, given a type, returns as value that no process exists for that type;
2. an *ill-defined type*, defined as the function that, given any valid process, returns as value a non-validly terminating term (i.e. no goal is in turn definable);
3. and a *retyping term function*, which given a process and a specification, expresses that the latter has a non-valid process description.

```
Coq <
Coq < Inductive mistake : Type -> Type :=
Coq < | missing_type : mistake (exists A, Empty A)
Coq < | type_illdefined : mistake (forall t:proc, no_goal)
Coq < | term_retype : mistake (exists A, exists t:proc,
Coq <                  In t A <-> ~ In t A).
mistake is defined
mistake_rect is defined
mistake_ind is defined
mistake_rec is defined

Coq <
```

The inductive definition of failure includes four cases:

1. *application of a wrong rule*, defined as the function that matching a rule to a type, invalidates it;
2. *bad application of a correct rule*, defined as the function that matching a rule in a valid context for a type, invalidates it;
3. *bad addressing*, defined as the matching of a type with another type, which returns an invalid value for the latter;
4. and *failing to reach appropriate resources*, defined as the matching of a type with a process, which returns an invalid value for the corresponding process term.

```
Coq <
Coq < Inductive failure : Type -> Type :=
Coq < | wrong_rule : failure (match A with match_rule => ~ A end)
Coq < | bad_rule : failure (match A with context_rule =>  ~ A end)
Coq < | bad_address : failure (match A with B => ~ B end)
Coq < | no_resources : failure (match A with t => ~ t end).
Warning: pattern B is understood as a pattern variable
failure is defined
failure_rect is defined
failure_ind is defined
failure_rec is defined

Coq <
```

The inductive definition of slip includes four cases:

1. the *exception rule*, which says that it is not the case that for any two process terms, given an evaluation of the first, it reduces safely to the second;
2. the case of *bad location*, which says that it is not the case for *multiple* distinct dependent process terms, that they each safely evaluate to another final term;
3. the case of *redundant processes*, which says that it is not the case that the same rule matches safely to any type;
4. and the case of *recurrent data*, which says that it is not the case that the same process term matches safely to any type.

```
Coq <
Coq < Inductive slip : Type -> Type :=
Coq < | exception_rule : slip (~forall t t', value t -> full_eval t t')
Coq < | bad_location : slip (~forall t1 t2 t3 t,
Coq <              full_eval t1 tm_true ->
Coq <      full_eval t2 t ->
Coq <      full_eval (tm_if t1 t2 t3) t)
Coq < | redundant_process : slip (forall A, match A with match_rule => ~A end)
Coq < | recurrent_data : slip (forall A, match A with t => ~t end).
slip is defined
slip_rect is defined
slip_ind is defined
slip_rec is defined

Coq <
```

We now define the algorithm `check`, which requires inductive recursions over types, terms and locations to address the right type of error. This checking will have cases corresponding to error definitions and will ideally proceed backwards from the simplest to the most complex error case.

For slips:

- if a case of exception rule occurs, then search for the right process term that is safely evaluated by an appropriate rule;

- if a case of bad location occurs over multiple dependently evaluated terms, check recursively for slips on the local evaluation on each of the dependent processes;
- if a case of redundant process occurs, check for the type that matches with the given rule;
- if a case of recurrent data occurs, check for the process term that matches with the given type.

```
Coq <
Coq < Inductive Check_slip: Type -> Type :=
Coq < | check_exception_rule : ~ full_eval t t' ->
Coq <                           Check_slip (full_eval t t'')
Coq < | check_bad_location : ~ full_eval (tm_if t1 t2 t3) t ->
Coq <                        Check_slip (bvalue t1) ->
Coq <                        Check_slip (bvalue t2) ->
Coq <                        Check_slip (bvalue t3)
Coq < | check_redundant_process : forall A, match A with match_rule => ~A end ->
Coq <                              Check_slip (match A1 with match_rule => A1 end)
Coq < | check_recurrent_data : forall A, match A with t => ~A end ->
Coq <                     Check_slip (match A with t1 => A end).
Warning: pattern t1 is understood as a pattern variable
Warning: pattern t is understood as a pattern variable
Check_slip is defined
Check_slip_rect is defined
Check_slip_ind is defined
Check_slip_rec is defined

Coq <
```

For failures:

- if a case of application of wrong rule occurs, check for a rule returning a valid value;
- if a bad application of rule occurs, check recursively for the valid context rule returning a valid value;
- if bad addressing occurs, check for the type value that returns an appropriate value for the defined type;
- if fail on resources occur, check for the process value that returns an appropriate value for the defined type.

```
Coq <
Coq < Inductive Check_failure: Type -> Type :=
Coq < | check_wrong_rule : match A with match_rule => ~ A end ->
Coq <                      Check_failure (match A with match_rule => A1 end)
Coq < | bad_rule : match A with context_rule =>  ~ A end ->
Coq <              Check_failure (match A with context_rule => A1 end)
Coq < | bad_address : match A with B => ~ B end ->
Coq <                 Check_failure (match A with B1 => B1 end)
Coq < | no_resources : match A with t => ~ t end ->
Coq <                  Check_failure (match A with t1 => t1 end).
Warning: pattern t1 is understood as a pattern variable
Warning: pattern t is understood as a pattern variable
Check_failure is defined
Check_failure_rect is defined
Check_failure_ind is defined
Check_failure_rec is defined

Coq <
```

For mistakes:

- if a missing type case occurs, check for a process value that returns a valid type;
- if ill-definition on type occurs, check for a process value that returns a new valid type;
- if term retyping is needed, check for a process value that be valid.

```
Coq <
Coq <
Coq < Inductive Check_mistake: Type -> Type :=
Coq < | check_missing_type : forall A, Empty A ->
Coq <                         Check_mistake (match A1 with t1 => ~ Empty A end)
Coq < | type_illdefined : forall t:proc, no_goal->
Coq <                     Check_mistake (match A with t => A1 end)
Coq < | term_retype : (exists A, exists t:proc, In t A <-> ~ In t A) ->
Coq <                  Check_mistake (match A with t1 => A end).
Warning: pattern t1 is understood as a pattern variable
Warning: pattern t is understood as a pattern variable
Warning: pattern t1 is understood as a pattern variable
Check_mistake is defined
Check_mistake_rect is defined
Check_mistake_ind is defined
Check_mistake_rec is defined

Coq <
```

We now define the algorithm `resolve`, that after an error recognition proceeds with the resolution of ill-typed expressions to well-typed ones. To resolve mistakes:

- on a case of missing type, produce the valid process on a new type;
- on a case of ill-defined type, produce two new process terms that are new states for the system (one of them possibly final);
- on a term retyping case, search for the new process that is not invalid in view of a given type.

```
Coq <
Coq <
Coq < Inductive resolve_mistake : Type -> Type :=
Coq < | resolve_missing_type : resolve_mistake (forall A, Empty A ->
Coq <                           exists t', exists A', In t' A')
Coq < | resolve_type_illdefined : resolve_mistake (forall t:proc, no_goal ->
Coq <                              exists t', exists t'', full_eval t' t'')
Coq < | resolve_term_retype : resolve_mistake (exists A, exists t:proc,
Coq <                         In t A <-> ~ In t A -> exists A', exists t':proc,
Coq <                         In t A <-> In t' A').
resolve_mistake is defined
resolve_mistake_rect is defined
resolve_mistake_ind is defined
resolve_mistake_rec is defined

Coq <
```

To resolve failures:

- on a case of reciprocally invalid types in view of the wrong rule application, match the first with a new type that invalidates the second;
- on a case of bad application of a rule, match the first type (on which the rule is applied) with a new type (as the result of the correct rule application) that invalidates the result of the faulty application;
- on a case of reciprocally invalid types in view of bad addressing, match the first with a new context rule that invalidates the second type;
- on a case of reciprocally invalid types in view of failing resources, match the first with a new process term that invalidates the second type.

```
Coq <
Coq < Inductive resolve_failure : (Prop*Prop) -> Type :=
Coq < | resolve_wrong_rule : resolve_failure (A, ~A ->
Coq <                         match A with A' => ~~A end)
Coq < | resolve_bad_rule : resolve_failure (A, ~B ->
Coq <                       match A with A' => ~~B end)
Coq < | resolve_bad_address : resolve_failure (A, ~A ->
Coq <                          match A with context_rule
Coq <                          => ~~A end)
Coq < | resolve_no_resources : resolve_failure (A, ~A ->
Coq <                           match A with t' => ~~A end).
resolve_failure is defined
resolve_failure_rect is defined
resolve_failure_ind is defined
resolve_failure_rec is defined

Coq <
```

To resolve slips:

- on a case of an application of rule on process terms that are exceptions, match the starting process with a new terminating process that is safely evaluated;
- on a case of bad locations, run the (re-evaluated) dependent process terms to the appropriate evaluation process;
- on a case of reciprocally invalid types by case of redundant process, match with the new rule that invalidates the previously faulty evaluated type;
- on a case of reciprocally invalid types by case of recurrent data, match with the new process that invalidates the previously faulty evaluated type.

```
Coq <
Coq < Inductive resolve_slip : Prop -> Type :=
Coq < | resolve_exception_rule : resolve_slip (~forall t t', value t ->
Coq <                             full_eval t t' -> match t with t'' =>
Coq <                             full_eval t'' t' end)
Coq < | resolve_bad_location : resolve_slip (~forall t1 t2 t3 t,
Coq <                           full_eval (tm_if t1 t2 t3) t ->
Coq <                           match t with t' =>
Coq <                           full_eval (tm_if t1 t2 t3) t' end)
Coq < | resolve_redundant_process : resolve_slip (forall A,
Coq <                                match A with match_rule => ~A end ->
Coq <                                match A with match_rule => ~~A end)
Coq < | resolve_recurrent_data : resolve_slip (forall A,
Coq <                             match A with t => ~A end ->
Coq <                             match A with t' => ~~A end).
resolve_slip is defined
resolve_slip_rect is defined
resolve_slip_ind is defined
resolve_slip_rec is defined

Coq <
Coq <
```

6 Conclusion

We have introduced an extended taxonomy of error states for computational systems based on an informational semantics. The latter allows for an understanding of processes that generalizes with respect to both procedural and denotational semantics. The next stage of this research is the formulation of a proper formal language for error-handling based on the coded definitions and algorithms introduced above. Long-term objectives are the design of resolution strategies for multi-agent information processing systems and of consensus reaching models.

7 Acknowledgements

Drafts of this paper were discussed at the *Fourth Workshop in the Philosophy of Information*, University of Hertfordshire and at the *Conference on Judgement and Justification*, University of Tampere. I wish to thank the participants for helpful discussions. Two anonymous referees have offered criticisms and remarks that have helped clarifying various passages of this work. My personal thanks to Patrick Allo for his comments and observations.

References

[1] B.B. Agarwal, M. Gupta, and S.P. Tayal. *Software engineering and testing: an introduction.* Jones & Bartlett Learning, 2009.

[2] D. Allchin. The epistemology of errors. In *Philosophy of Science Association*, 2000.

[3] D. Allchin. Error types. *Perspectives on Science*, 9:38–59, 2001.

[4] P. Allo and E. Mares. Informational semantics as a third alternative? *Erkenntnis*, 2011.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (eds.). *The Description Logic Handbook. Theory, Implementation and Applications.* Cambridge University Press, 2003.

[6] L.R. Baker. The metaphysics of malfunction. *Techne*, forthcoming.

[7] D. Beaver. *Presupposition and Assertion in Dynamic Semantics.* Stanford: CSLI Publications., 2001.

[8] D. Bonnay and P. Egre'. Knowing One's Limits - An analysis in Centered Dynamic Epistemic Logic. *Synthese, Springer.*, 2011.

[9] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968.

[10] H.B. Curry. Functionality in combinatory logic. In *Proceedings of the National Academy of Science USA*, volume 20, pages 584–590, 1934.

[11] L. Floridi. The logic of being informed. *Logique & Analyse*, 196:433–460, 2006.

[12] L. Floridi. Philosophical Conceptions of Information. In G. Sommaruga, editor, *Formal Theories of Information*, volume 5363 of *Lectures Notes in Computer Science*, pages 13–53. Springer Verlag, 2009.

[13] M. Franssen. Design, use, and the physical and intentional aspects of technical artifacts. In A. Light P. E. Vermaas, P. Kroes and S. A. Moore, editors, *Philosophy and Design: From Engineering to Architecture*, pages 21–35. Springer, 2008.

[14] W. Howard. The formulae-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[15] B. Jespersen. A new logic of technical malfunction. *Studia Logica*, forthcoming.

[16] B. Jespersen and M. Carrara. Two conceptions of technical malfunction. *Theoria*, 77:117–138, 2011.

[17] D.G. Mayo. *Error and the Growth of Experimental Knowledge.* Chicago University Press, 1996.

[18] D.G. Mayo. Learning from error, severe testing, and the growth of theoretical knowledge. In D. Mayo and Spanos, editors, *Error and Inference.* Cambridge University Press, 2010.

[19] G. Michaelson. *Functional programming through λ-calculus.* Dover, 1989.

[20] C.S. Peirce. Illustrations of the logic of science vi: Deduction, induction, and hypothesis. *Popular Science Monthly*, 13, 1878.

[21] K. R. Popper. *Conjectures and Refutations.* Routledge & Keagan, London, 1963.

[22] G. Primiero. An epistemic logic for becoming informed. *Synthese (KRA)*, 167(2):363–389, 2009.

[23] G. Primiero. Offline and online data: on upgrading functional information to knowledge. *Philosophical Studies*, 2012. DOI:10.1007/s11098-012-9860-4.

[24] J. Reason. *Human Error.* Cambridge University Press, 1990.

[25] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry–Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, 2006.

[26] B.G. Sundholm. Error. *Topoi*, 2012.

[27] R. Turner. Specification. *Minds & Machines*, 21(2).:135–152, 2011.

[28] T. Williamson. Inexact knowledge. *Mind*, 101(402).:217–241, 1992.

[29] T. Williamson. *Vagueness.* Routledge, 1994.

[30] T. Williamson. *Knowledge and its Limits.* Oxford University Press, 2002.

[31] H. Woods. *The Death of Argument: Fallacies in Agent-based Reasoning.* luwer Academic Publishers., 2004.