FI^{4}FA: a Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis

Barbara Gallina and Sasikumar Punnekkat
MRTC, School of Innovation, Design and Engineering,
Mälardalen University,
Västerås, Sweden
{barbara.gallina, sasikumar.punnekkat}@mdh.se

***Abstract*— To architect dependable distributed component-based, transactional systems, failures as well as their mitigation behaviors must be analyzed. Analysis helps in planning if, where and which mitigation means are needed to increase quality, by reducing the failures that threaten the system's dependability. Fault Propagation and Transformation Calculus (FPTC) is a technique for automatically calculating the failure behavior of the entire system from the failure behavior of its components [1]. FPTC, however, considers few failure types and offers no support to analyse the mitigation behaviour.**

To overcome these limitations and support the mitigation's planning, we introduce a new formalism, called FI^{4}FA. FI^{4}FA focuses on failures avoidable through transaction-based mitigations. FI^{4}FA extends FPTC by enabling the analysis of I^4 (incompletion, inconsistency, interference and impermanence) failures as well as the analysis of the mitigations, needed to guarantee completion, consistency, isolation and durability. We also illustrate the usage of FI^{4}FA on a set of examples.

Keywords-dependability, component-based systems, failures types, failure behaviour analysis techniques, (relaxed) ACID properties.

I. INTRODUCTION

Techniques to automatically calculate the failure behaviour of a system from the failure behaviour of its components are of utmost importance. In the framework of distributed component-based engineering, for instance, the importance of these techniques is even more evident since they offer a means to support a careful selection and composition of the components. Architects, human beings or automatic (re)configuration applications, have two options: 1) select those components that, if composed, do not manifest as a whole a failure behaviour that goes beyond the threshold of tolerance; 2) select accurate mitigation means to be introduced to avoid or at least reduce or control the failure behaviour.

In the framework of distributed real-time transactional systems, quality-requirements can be in conflict and have to be traded-off. Logical consistency may, for instance, be traded-off in favour of temporal consistency or availability [29]. Architects may accept wrong data values in favour of fresh data values.

The knowledge of the system's failure behaviour permits architects to evaluate the system's dependability and plan the mitigation means that eventually should be introduced to avoid/reduce failures [4]. More specifically, the more detailed the system's failure behaviour is, the more accurate the mitigation means can be planned. Knowing, for instance, that the value of the system's output is deviating from the expected set of values is useful but may not be enough to choose mitigation means meant to guarantee the weakest but acceptable level of dependability.

By knowing that the wrong value is the result of an incomplete computation, the architect has the means to choose strategically the mitigation (e.g., commit protocol) to be introduced to prevent/mitigate the failure's occurrence.

Besides the means to specify the failure behaviour, as discussed in [28], means to specify the mitigation (masking) behaviour of a component should also be included within the techniques to analyse the failure behaviour.

Fault Propagation and Transformation Calculus (FPTC) [1] is one of the techniques that allows the system's failure behaviour to be automatically calculated from the failure behaviour of its components. FPTC, however, presents two limitations: it only supports a coarse-grained analysis in terms of failure behaviour and it has no support to analyse the mitigation behaviour.

In this paper, we provide a new formalism, called FI^{4}FA, which extends FPTC. The advantage of FI^{4}FA with respect to FPTC is twofold: FI^{4}FA enriches the failure types and it introduces the capability to analyse mitigation behaviour, by offering mitigation types. More specifically, thanks to FI^{4}FA, the user can analyse incompletion, inconsistency, interference and impermanence failures, as well as the behaviour of the eventually introduced corresponding mitigation means. The additional failure types and the mitigation types are crucial to plan accurately focused mitigation means in terms of component-based transactional support. It is well known that transactions and their extensions offer means to address the I^4 failures by selecting and combining, respectively, the desired Atomicity, Consistency, Isolation and Durability (properties, which are widely known under the acronym ACID).

FI^{4}FA has been conceived within the framework of the CHESS project [27] and it is expected to be one of the analysis techniques available in the integrated development environment of the project (under development).

The rest of the paper is organized as follows. Section II presents essential background. Section III introduces the extension of FPTC, called FI^{4}FA. Section IV shows, on the basis of small examples, the potential effectiveness of FI^{4}FA in discriminating failures and in specifying mitigation behaviour. Section V discusses related work. Finally, Section VI presents conclusions and directions for future work.

II. BACKGROUND

In this section, we briefly present the essential background on which we base our work. In particular, we present FPTC in Section II.A and the ACID spectra in Section II.B.

A. FPTC

FPTC is a technique for automatically calculating the failure behaviour of an entire system from the failure behaviour of its components. A twofold motivation makes this technique relevant: it offers a qualitative means to evaluate the dependability of a system and it can be used to plan how and where to introduce failure avoidance means within the system.

Before explaining the steps that have to be followed to apply FPTC, to avoid any misconceptions, it is crucial to explain what is meant by 'failure' within this paper.

The term *failure* identifies an event that occurs when the delivered service (the behavior of the system as perceived by the user) deviates from correct service (the system specification) [5]. In the framework of component-based systems, a component can be considered as a system when studied in isolation and as a sub-system when the whole system is considered. Therefore we believe that while considering the single components' input-output behavior, the term 'failure' is correct as well as sufficient to denote system's deviations/violations and there is no need to introduce the other dependability threats (fault and error).

As depicted in Fig.1, when the system as a whole (C) is considered, a failure in a sub-system (A or B) represents a fault elsewhere. For instance, a failure of component A, recursively, represents a fault for the component B [5].

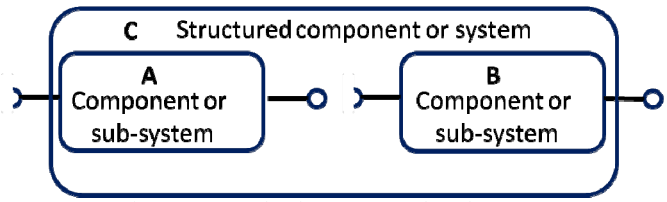

Figure 1. Example of a component-based system

According to the causality chain that inter-relates the dependability threats, a fault may become active and produce an error which if not handled may lead to the system's failure. So the failure generated by A, if not handled, propagates itself within B and beyond B's boundary until reaching and exiting the system's boundary.

Since FPTC only considers the input-output behavior, to avoid confusions in the terminology as well as to enable ease of syntax-level synthesis, we consistently use the term 'failure' only (we consider that the fault in input comes from a failure generated elsewhere).

To apply the FPTC technique, the following three steps have to be followed:

1) *Analysis of the failure behaviour of the components*. The failure responses of a component to its input is analysed in isolation from the rest of the system. From this analysis it is possible to establish if a component:
   a. propagates a failure received in input as it is (passing on a failure from input to output);
   b. transforms a failure (changing the nature of the failure from one type to another from input to output);
   c. behaves as a source of failure, by generating a failure despite the absence of failure in input;
   d. behaves as a sink, by avoiding the failure to be either propagated of transformed.
2) *Specification of the component's behaviour*. The component's behaviour analysed at step 1 must be specified as a collection of propagation or transformation expressions, using FPTC syntax.
3) *Calculus of the failure behaviour of the whole system*. The inter-dependent components are considered as a token (failure)-passing network and a fixed-point calculation is performed.
   To guarantee convergence, at each node of the network it must be ensured that exactly one expression matches the input failure behaviour (see [1] for further details).

The FPTC syntactical rules (in EBNF) to specify the failure behaviour of a component are presented in the text box below.

behaviour = expression+
expression = LHS '→' RHS
LHS = bL| '(' bL (',' bL)+ ')'
RHS = bR| '(' bR (',' bR)+ ')'
bL = '_' | bR
bR = no-failure |alphachar | failure | '{'failure (',' failure)+ '}'
failure = 'subtle' | 'coarse' | 'early' | 'late' | 'omission' | 'commission'
no-failure = '*'

These rules are adapted from [1, 2] and they take into consideration the additional constraints available in [3], regarding the right-hand-side.

The behaviour of a component is specified as a collection of expressions and each expression is composed of two parts: the left-hand-side part specifies the behaviour received in input and the right-hand side specifies the behaviour in output. The behaviour on each input port (bL) may be normal or failure behaviour. The wild card symbol '_' is used when the type of behaviour in input does not play a crucial role to determine the behaviour in output. This symbol cannot appear on the right-hand-side because otherwise it would contradict the assumption that wants the FPTC expressions at each node to be purely functional.

The rule *failure* defines the failure types (also known as failure modes [5]) that can be used. FPTC considers the majority of the failure types (except for arbitrary) presented in [24, 26] and briefly explained in Table 1. These failures, as extensively discussed in Section V, are the result of several extensions and revisions.

TABLE 1. Failure Types

Failure type	Description
Arbitrary	Any violation from the specified behavior
Early	The output is provided too early
Late	The output is provided too late
Coarse	The output deviates from the expected range of values in a detectable way (by the user)
Subtle	The output deviates from the expected range of values in an un detectable way (by the user)
Omission	no output is provided
Commission	an output is provided when not expected

As shown in Fig. 2, the failures types presented in Table 1 can be hierarchically ordered.

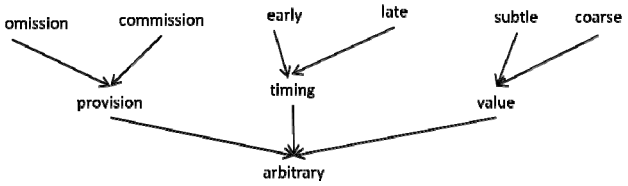

Figure 2. Failure types hierarchy

Early and *late* are sub-types of the more general *timing* failure type. *Coarse* and *subtle* are sub-types of the more general *value* failure type. *Omission* and *commission* are sub-types of the more general *provision* failure type (known also as sequence failure type). Finally, value, timing and provision are sub-types of the more general *arbitrary* failure type.

The following FPTC expressions illustrate the usage of FPTC syntax to specify the four possible behaviours:

early→early	(propagation)
omission→coarse	(transformation)
*→coarse	(source)
coarse→*	(sink)

As it can be noticed, the sink behaviour is simply specified by stating that no failure will occur on the output. No information concerning the mitigation types is available.

Fig. 3 shows how FPTC analysis works. The failure behaviour of the simple system, introduced in Fig. 1, is calculated on the basis of the failure behaviour of its two single-input single-output chained components (A and B). For space and simplicity reasons, the failure behaviour of each component consists of only one rule, which will be depicted inside the component. The reader, interested in deepening more complex examples, may refer to [3].

During the initialization phase, the system is fed with a normal behaviour and then during the propagation phase the transformation rules are considered. Component A behaves as a source and generates a coarse failure. Component B transforms the coarse failure into a late failure.

At the end of the analysis, we see that the system as a whole (which was denoted as C in Fig. 1) exhibits a late timing failure behaviour.

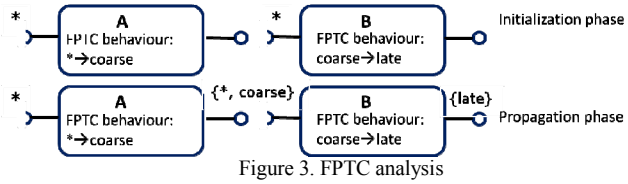

Figure 3. FPTC analysis

FPTC allows users to trace back the failure's source(s).

B. *The ACID spectra*

Atomicity, Consistency, Isolation and Durability, widely known under the acronym ACID [7], are four properties, which, if satisfied together, ensure high dependability (more specifically, high reliability [8]). These properties combine fault tolerance and concurrency control.

We use the term *work-unit* to identify the set of possible executions of a partially ordered set of logically related events (executions of a single operation on the state, which is a mapping from storage unit names to values storable in those units) [12]. A work-unit in the context of component-based systems may represent the set of possible executions of a component.

The definitions of ACID properties, adapted from [9] using the term work-unit, are presented below:

Atomicity: a work-unit's changes to the state are atomic: either all happen or none happen (all-or-nothing semantics, known as failure atomicity). Atomicity guarantees that in case of failure, intermediate/incomplete work is undone bringing the state back to its initial consistent value.

Consistency: a work-unit is a correct transformation of the state. All the a priori constraints on the input state must not be violated by the work-unit (intra-work-unit, local, consistency).

Isolation: a set of work-units either is executed sequentially (no interference) or is executed following a serializability-based criterion (controlled interference). This criterion ensures that conflicting work among work-units is executed in a controlled way to avoid undesired interference (known as anomalies or phenomena in the concurrency control literature and detectable as cycles [9]).

Durability: once a work-unit completes successfully, its changes to the state are permanent.

ACID properties preserve a consistent state (global data consistency), that is the state that satisfies all the predicates on objects (that is single pairs <name, value>).

In a component-based system development, ACID properties can represent 4 different functionalities. These functionalities, as depicted in Fig. 4, can be achieved through 4 different components, which cooperate to ensure global data consistency.

Figure 4. Component-based architecture

These separate but interacting functionalities were initially identified in [9] and had rather design-specific names. The lock manager, for instance, was in charge of ensuring isolation on the basis of locking strategies. Similarly, the resource manager was in charge of checking consistency on the basis of consistency tests and the log manager was in charge of ensuring durability on the basis of logging strategies. In [10, 11], authors propose to enclose some of these functionalities within components.

ACID properties, however, do not always represent the right solution to achieve reliability. In several application or technological domains, ACID properties are considered too strict and therefore relaxed versions are introduced, forming the ACID spectra. Distributed real-time applications, for instance, require relaxed ACID properties to meet distribution and timeliness requirements.

Relaxed versions of atomicity accept *incomplete* work. Relaxed versions of consistency accept integrity violation (*inconsistency*). Relaxed versions of isolation accept *interference*. Relaxed versions of durability accept *impermanence*.

By adopting a product line perspective [12], these ACID spectra may be considered as variants to be selected and composed to achieve the desired combination of fault tolerance and concurrency control that guarantees the desired reliability. A component can be represented by one of the variants for our analysis purpose.

The selection and composition of the desired components could be eased by having at disposal means to analyze the failure as well as the mitigation behavior with respect to incompletion, inconsistency, interference and impermanence.

III. FI^{4}FA

As seen in Section II, FPTC is a powerful technique to analyze the failure behavior of a system. FPTC, however, presents two limitations: it only supports a coarse-grained analysis in terms of failure behaviour and it has no support to analyse the mitigation behaviour. As discussed previously, these limitations need to be overcome to provide a better support to architect dependable distributed component-based, transactional systems. To address the first limitation, as also pointed out by Wallace [2], additional failure types may be considered. As discussed in [28], to address the second limitation, new specification capabilities need to be introduced. FPTC, therefore, should be extended.

This section presents FI^{4}FA, which is an extension of FPTC aimed at tailoring FPTC to enable the analysis of well-defined classes of failures as well as the analysis of the mitigation means.

FI^{4}FA focuses on the collectively called I^4 (incompletion, inconsistency, interference and impermanence) failures and their corresponding mitigation means. The I^4 failures are avoidable through (relaxed) ACID-based mitigation means.

Similar to the detectability point of view, represented by the *coarse* and *subtle* failure sub-types, I^4 introduces an additional point of view to further characterize failures as well as the corresponding mitigation means. As Fig. 5 depicts, each failure type considered within FPTC can be further analyzed according to the I^4 point of view.

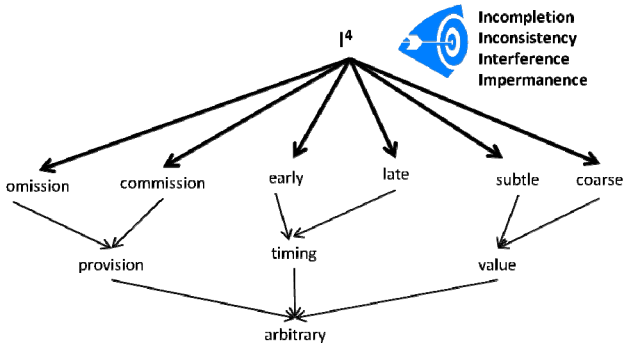

Figure 5. The I^4 perspective of failure types hierarchy

Before presenting the syntactical extension of FPTC and discussing the impact in terms of semantics (Subsection III.E), a conceptual introduction of the extension is given. In particular, the I^4 failures and their corresponding mitigation means are discussed in the following four sub-sections.

A. *Failure avoidable through atomicity*

Incompletion failures occur when a component does not execute to completion its expected work.

The identification of an "incompletion" failure is essential to decide the adequate mitigation in terms of atomicity that avoids it. According to the needs of the application domain, an incompletion failure can be avoided by introducing a component that satisfies either *all-or-nothing* semantics (by implementing for instance a 2-phase commit protocol) or *all-or-compensation* semantics, etc.. The interested reader may refer to [13] for further details. The choice between these two semantics highly depends on the characteristics of the components involved. Highly distributed, autonomous and heterogeneous components are not suitable to manage an incompletion failure through an *all-or-nothing* semantics since it would contradict their autonomy and would also block or delay their work, causing a timing failure.

On the basis of the characteristics of the components involved and on the basis of the identification of an incompletion failure, the accurate atomicity semantics can be selected. A specific atomicity semantics then can be designed in different ways. As discussed in [14], for instance, the all-or-nothing semantics can be achieved through different component-based commit protocols, which

have different costs in terms of the number of messages to be exchanged to achieve an agreement. This difference in terms of the number of messages leads to a difference in terms of the time to complete the agreement.

B. *Failure avoidable through consistency*

Inconsistency failures occur when a component executes some events that violate the data´s integrity constraints.
The identification of an "inconsistency" failure is essential to decide the adequate mitigation in terms of consistency that avoids it. According to the needs of the application domain, an inconsistency failure can be avoided by introducing a component that satisfies full consistency or ε-data consistency (range violation). The interested reader may refer to [16] for further details.

C. *Failure avoidable through isolation*

WW-cycle-based interference failures occur when a component executes an event of type *write* that is in conflict with other events of type write that are in execution and are invoked by concurrent components. The different conflicts form a cycle.
All-cycle-based interference failures occur when a component executes an event that is in conflict with other events that are in execution and are invoked by concurrent components. The different conflicts form a cycle.
The identification of an "interference" failure is essential to decide the adequate mitigation in terms of isolation that avoids it. An interference failure can be avoided by introducing a component that satisfies either *PortableLevel1* semantics or *serializable* semantics, etc.. The interested reader may refer to [17,18] for further details.

D. *Failure avoidable through durability*

Impermanence failures occur when a component does not save permanently the work executed successfully.
The identification of an "impermanence" failure is essential to decide the adequate mitigation in terms of durability that avoids it. An impermanence failure can be avoided by introducing a component that satisfies, for instance, full durability or either relaxed durability [19] or ephemeral durability [20].

E. *FI⁴FA syntax and semantics*

The syntax of FI⁴FA, shown in textbox below, is an extension/redefinition of the syntax of FPTC, presented in Section II. FI⁴FA, in particular, redefines two syntactical rules: *failure* and *no-failure*. As mentioned in the introduction, FI⁴FA allows users to analyze the failure behavior with finer granularity (I^4 failures) and it introduces the possibility to analyze the mitigation behaviour. Moreover, FI⁴FA allows users to consider combinations of failures and therefore it also implicitly considers the arbitrary failure.

failure = basic-standard | combined
basic-standard = timing | value | sequence
timing ='early' | 'late'
value ='coarse' | 'subtle'
sequence ='omission' | 'commission'
combined = basic-standard '.' basic-standard | basic-standard'.'A-avoidable'.'C-avoidable'.'I-avoidable'.'D-avoidable
A-avoidable = 'incompletion' | no-failure
C-avoidable = 'inconsistency' | no-failure
I-avoidable = 'ww-cycle-based-interference' | 'all-cycle-based-interference' | no-failure
D-avoidable = 'impermanence' | no-failure

no-failure = basic-star | detailed-star
basic-star = '*'
detailed-star = basic-star'.'A-mitigation'.'C-mitigation'.'I-mitigation'.'D-mitigation
A-mitigation = 'all-or-nothing' | 'all-or-compensation' | 'none'
C-mitigation = 'full-consistency' | 'range-violation' | 'none'
I-mitigation = 'PortableLevel1' | 'serializable' | 'none'
D-mitigation = 'no-loss' | 'partial-loss' | 'none'

The terminal *"none"* in the above rules represents absence of mitigation.
From a semantic point of view, intuitively, the new failures introduced within FI⁴FA as well as the mitigation types are mapped into additional tokens and the failure behavior of the system is still obtained as a fixed-point calculation.

IV. FI⁴FA USAGE

This section shows how FI⁴FA can be used to specify more granularly the failure propagation and transformation that may take place within the system and beyond the system boundary. FI⁴FA is used within a component diagram-like context as presented in [27]. Examples are adapted from [12]. The usefulness of this fine-grained specification is then motivated by discussing how counter-measures (failure sinks) can be accurately planned and introduced to avoid failures.

Examples:

Before presenting the examples, we explain the notation used throughout them:

- read1[x, v] refers to a read operation which belongs to a work-unit labeled with the number 1 and which reads an object x (where the read value is v)
- write1[x, v] refers to a write operation which writes an object x (where the written value is v)
- the symbol "<>" denotes inequality
- the symbol "x" denotes multiplication.

Two objects x and y of type integer are related by the constraint: y=2x

The initial state of the two objects is: x=1 and y=2.

Since 2 = 2 x 1, in the initial state, the constraint holds.

Incompletion failure

Work-unit 1 (structured component) is supposed to execute to completion the following set of operations: {write1[x, 10], write1[y, 20]}. An omission takes place during the execution of the operation write1[y, 20]. As a consequence, as depicted in Fig. 6, the whole system will witness a coarse failure due to incompletion, detected by the consistency checker.

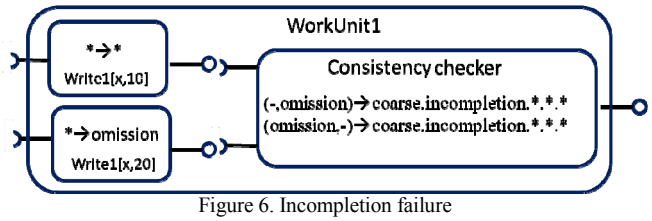

Figure 6. Incompletion failure

To mitigate an incompletion, as depicted in Fig. 7, an all-or-nothing semantic-based wrapper can be introduced.

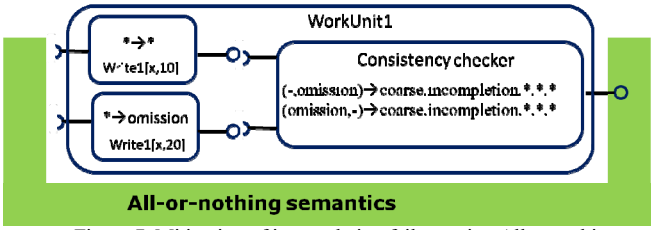

Figure 7. Mitigation of incompletion failure using All or nothing

The FPTC behavior of the wrapper would be:

coarse.incompletion..*.*→*.all-or-nothing.none.none.none*

As mentioned before, according to the needs, other wrappers, based on other customizable semantics [13], can be chosen.

Inconsistency failure

Work-unit 1 is supposed to execute consistently the following set of operations: {write1[x, 10], write1[y, 20]}. A coarse failure takes place during the execution of the operation write1[y, 20]. As a consequence, as depicted in Fig. 8, the whole system will witness a coarse failure due to inconsistency.

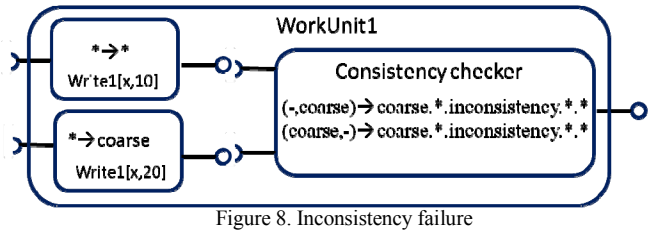

Figure 8. Inconsistency failure

To mitigate an inconsistency, a traditional consistency semantic-based wrapper can be introduced.

Interference failure

Work-unit 1 is supposed to execute the following set of operations {write1[x, 10], write1[y, 20]}. Work-unit 2 is supposed to execute the following set of operations {write2[x, 30], write2[y, 60]}.

The two work-units are supposed to execute concurrently without interfering. Their execution is, however, as follows: write1[x, 10] write2[x, 30] write2[y, 60] write1[y, 20] and as a consequence, as depicted in Fig. 9, the whole system will witness a ww-cycle-based-interference takes place.

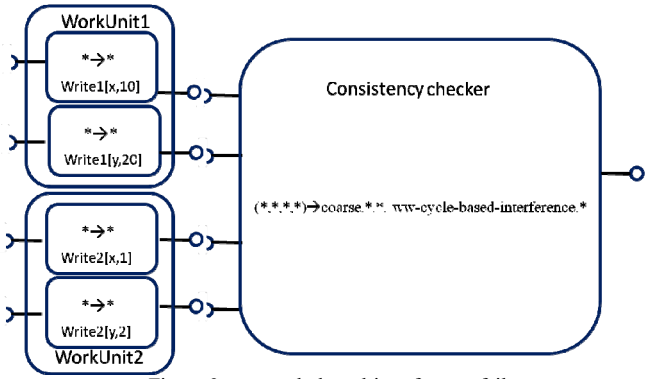

Figure 9. ww-cycle-based-interference failure

To mitigate an interference failure, a Portable Level 1 semantic-based wrapper could be conceived. This wrapper, for instance, could consist of a component that manages the inter-component dependencies (conflicts), as discussed in [15], and in particular it could monitor and prevent ww-cycle-based-interference.

Impermanence failure

Work-unit 1 is supposed to save permanently the following set of operations: {write1[x, 10], write1[y, 20]}. An omission takes place during the storing of the operation write1[y, 20]. As a consequence, as depicted in Fig. 10, the whole system will witness a coarse failure due to impermanence.

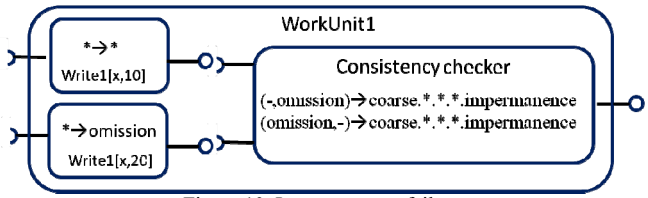

Figure 10. Impermanence failure

To avoid this failure a traditional durability-based wrapper can be introduced.

As discussed previously, incompletion, inconsistency, interference and impermanence are failures which all lead to coarse value failures. Having at disposal a means to discriminate them permit the value failures to be avoided in a specific way.

V. RELATED WORK

A fundamental step in engineering dependable systems consists of modelling the system's failure behaviour. The knowledge of the failure behaviour is essential to plan appropriate mitigation means. The assumed type or nature of failures dictates the type of mitigation means. In the literature, several proposals have been provided to categorize failures and their corresponding mitigation means.

Failures have been ordered by identifying structures, mainly lattices. In [21], a nested structure of potential failure classes is defined to enable a systematic comparison of the power of a family of fault-tolerant protocols based on atomic broadcast. More specifically, three nested types of failures are identified, respectively omission, timing and Byzantine failures. In [22], an extended classification is provided. In particular, value failures (called timely failures) are considered. This is further extended in [6, 23, 25], to discriminate additional subclasses of timing (i.e. early, late, etc.) as well as value (subtle, coarse, etc.) failures. Subsequently, in [24], a slightly different classification is given. This last classification, which was considered as a starting point in the previous sections of this paper, introduces a service perspective and classifies failures according to three main classes: timing (early, late), value (coarse, subtle) and provision (omission and commission).

The progressive extensions/reconsiderations of the failure types, which emerges from the literature as summarized above, reveals the necessity of having at disposal fine-grained classifications of failures.

In [26], an interesting remark is given. It is argued that a rich set of guidewords (failure types) might be attractive from an academic perspective but is proved difficult to interpret and manage.

Despite this remark, our work represents an additional ring of this chain of extensions and as its predecessors it enriches the types of failures. Its originality lies in its focus on failures that can be avoided through (relaxed) ACID-based mitigation means. Due to its sharp focus, FI^{4}FA should be considered a means specifically conceived for assessing the effectiveness of the selection and composition of transactional components with respect to the dependability requirements. Users are supposed to be knowledgeable enough regarding the fundamental concepts of different failure types to avoid misconceptions and are expected to manage better and take advantage from the analysis of the I^4 failures.

FI^{4}FA differs with respect to other failure behaviour analysis techniques also because it includes means to specify the mitigation behaviour. By including this specification capability, FI^{4}FA contributes in overcoming some limitations discussed in the comparative work presented in [28].

VI. CONCLUSION AND FUTURE WORK

To architect dependable component-based, transactional systems, the failure as well as the mitigation behaviour must be analyzed. The analysis is necessary to plan if, where and which mitigation means are needed to avoid or at least reduce the failure behaviour. Moreover, the analysis can be used by architects, either human beings or automatic (re)configuration applications, during the selection and composition of components.

As discussed in the background, FPTC is a relevant technique that allows a user to analyze the failure behaviour of an entire system on the basis of the failure behaviour of its components. FPTC, however, presents two limitations: it only supports a coarse-grained analysis in terms of failure behaviour and it has no support to analyse the mitigation behaviour.

To overcome these limitations and ease a better planning of the mitigation means, in this paper, we have introduced a new formalism, called FI^{4}FA. The focus of FI^{4}FA is on those failures that are avoidable through transaction-based mitigations. More specifically, FI^{4}FA extends FPTC by enabling the analysis of I^4 (incompletion, inconsistency, interference and impermanence) failures as well as the analysis of the mitigations, needed to guarantee completion, consistency, isolation and durability. We have also illustrated the usage of FI^{4}FA using a set of examples.

FI^{4}FA, similarly to other techniques for failure behaviour analysis, is part of a recent research direction aimed at providing the most appropriate architecture-based dependability analysis technique. Its practicability in industrial settings remains to be tested.

In the immediate future, we aim at providing tool-support for FI^{4}FA and evaluate its effectiveness and usability/practicability within industrial settings. FI^{4}FA is expected to be one of the analysis techniques available within the tool (under development) in the framework of the CHESS project [27]. An industrial case study will be considered and the full potential of FI^{4}FA will be evaluated and compared with the one of the standard FPTC.

In a mid-term future, additional refinements will be considered to further characterize the ACID-avoidable failures to achieve an even more granular analysis. An incompletion (impermanence) failure, for instance, can involve work that has to be executed (saved) mandatorily, optionally, etc.. Similarly, an inconsistency failure may involve the violation of the range of values that are allowed for a data item as well as the violation of a referential constraint. Since these additional refinements might make the formalism heavy, they will be integrated in a second iteration in case of positive feedback from the industrial partners.

Finally, we intend to integrate the I^4 point of view also within other techniques for the analysis of failure behaviour (e.g. FPTA [30] as well as HiP-HOPS, etc. reviewed in [24,28]) and make a comparative study. The integration of the I^4 point view within FPTA would allow users to obtain a quantitative evaluation of the I^4 failure behaviour in addition to the qualitative one presented in this paper.

ACKNOWLEDGMENT

This work has been partially supported by the European Project ARTEMIS-JU100022 CHESS [27].

The authors thank Thomas Leveque and Hüseyn Aysan for fruitful discussions on FPTC.

REFERENCES

[1] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. Electronic Notes in Theoretical Computer Science (ENTCS), volume 141 n.3, p.53-71, December, 2005.

[2] R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J. Brooke. FPTC: automated safety analysis for domain-specific languages. In Models in Software Engineering, M. R. Chaudron (Ed.). Lecture Notes In Computer Science, Vol. 5421. Springer-Verlag, Berlin, Heidelberg, pp. 229-242, 2009.

[3] FPTC implementation's documentation

http://sourceforge.net/apps/mediawiki/epsilonlabs/index.php?title=FPTC

[4] F. Ye and T. Kelly. Component failure mitigation according to failure type. Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC), pp.258-264 vol.1, 28-30 Sept. 2004.

[5] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing. In: IEEE Trans. Dependable Sec. Comput. 1(1): 11-33. 2004.

[6] A. Bondavalli and L. Simoncini. Failures classification with respect to detection. In 2nd. IEEE Workshop on Future Trends in Distributed Computing Systems, pages 47-53, IEEE Computer Society Cairo, Egypt, 1990.

[7] T. Härder, A. Reuter. Principles of transaction-oriented database recovery, ACM Computing Survey, pp. 287-315, 1983.

[8] V. Hadzilacos, A theory of reliability in database systems. Journal of the ACM (JACM) Volume 35, Issue 1, Pages: 121 – 145, January 1988.

[9] J. Gray, A. Reuter; Transactions processing: concepts and techniques. Morgan Kaufmann Publishers, 1993.

[10] A. Tesanovic, D. Nyström, J. Hansson and C. Norström; Embedded databases for embedded real-time systems: a component-based approach. Mälardalen University, ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Technical Report, January, 2002.

[11] A. Tesanovic, D. Nyström, J. Hansson, and C. Norström, Towards aspectual component-based development of real-time systems, in Proceeding of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), Springer-Verlag, Feb. 2003.

[12] B. Gallina, PRISMA: a software product line-oriented process for the requirements engineering of flexible transaction models, PhD thesis, Université du Luxembourg, 2010.

[13] W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflow. In Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'01), pp. 140-147. IEEE Computer Society, 2001.

[14] P. Serrano-Alvarado, R. Rouvoy, and P. Merle. Self-adaptive component-based transaction commit management. In Proceedings of the 4th workshop on Reflective and adaptive middleware systems (ARM '05). ACM, New York, NY, USA, 2005.

[15] F. Kon, R. H. Campbell. Dependence management in component-based distributed systems. Concurrency, IEEE, vol.8, no.1, pp.26-36, Jan-Mar 2000.

[16] B. Sadeg, S. Saad-Bouzefrane. Relaxing correctness criteria in real-time DBMSs. In Sung Y. Shin, editor, Proceedings of the ISCA 15th International Conference Computers and Their Applications, New Orleans, Louisiana, USA, pp. 64–67, ISCA, 2000.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A critic of ANSI SQL isolation levels, Proceedings of the ACM SIGMOD international conference on Management of Data, San Jose, California, USA, pp. 1-10, 1995.

[18] A. Adya and B. Liskov and P. O'Neil, Generalized isolation level definitions, In: IEEE Intnl. Conf. on Data Engineering, San Diego, CA, USA, pp. 67–78, 2000.

[19] IBM SolidDB 6. Relational Database http://soliddb.com/en/carrier-grade/index1.asp.

[20] J. R. Garbus, In-memory database option for Sybase Adaptive Server Enterprise. Database Journal, 2010.

[21] F. Cristian, H. Aghili, R. Strong, D. Volev. Atomic broadcast: from simple message diffusion to Byzantine agreement. 25th International Symposium on Fault-Tolerant Computing, Highlights from Twenty-Five Years, pp.431, 1985.

[22] P.D. Ezhilchelvan and S.K. Shrivastava, "A characterisation of faults in systems", Proceedings of Symposium on Reliability in Distributed Software and Database Systems, pp.215-222, 1986.

[23] D. Powell. Failure mode assumptions and assumption coverage. Twenty-Second International Symposium on Fault-Tolerant Computing. FTCS. Digest of Papers., pp.386-395, 8-10 Jul 1992.

[24] D. J. Pumfrey. The principled design of computer system safety analyses, PhD Thesis, University of York, 1999.

[25] H. Aysan, S. Punnekkat, R. Dobrin. Error modeling in dependable component-based systems. In Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC). IEEE Computer Society, Washington, DC, USA, pp.1309-1314, 2008.

[26] J. A. McDermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon. Experience with the application of HAZOP to computer-based systems. Proceedings of the 10 th Annual Conference on Computer Assurance, Gaithersburg, MD, pp. 37-48, IEEE, 1995.

[27] ARTEMIS-JU-100022 CHESS- Composition with guarantees for High-integrity Embedded Software components aSsembly. https//www.artemis-ju.eu/chess.

[28] L. Grunske, J. Han. A comparative study into architecture-based safety evaluation methodologies using AADL's Error Annex and failure propagation models. 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp.283-292, 2008.

[29] B. Gallina and N. Guelfi. Reusing transaction models for dependable cloud computing. In: Software reuse in the emerging cloud computing era. IGI Global, in press.

[30] X. Ge, R. F. Paige, and J. A. Mcdermid. Probabilistic failure propagation and transformation analysis. In Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Bettina Buth, Gerd Rabe, and Till Seyfarth (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 215-228, 2009.