A general trace-based causality analysis framework for component systems

Yoann Geoffroy

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE la Communauté UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Yoann GEOFFROY

Thèse dirigée par **Gregor Gössler**

préparée au sein **INRIA Grenoble Rhône-Alpes**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Un cadre général de causalité basé sur les traces pour des systèmes à composants

Thèse soutenue publiquement le **7 Décembre 2016**,
devant le jury composé de :

Dr. Éric FABRE
Directeur de recherche, IRISA, Rapporteur
Dr. Louise TRAVÉ-MASSUYÈS
Directeur de Recherche, LAAS-CNRS, Rapporteur
Dr. Oded MALER
Directeur de recherche, CNRS-VERIMAG, Président
Prof. Oleg SOKOLSKY
Research Professor, University of Pennsylvania, Examinateur
Dr. Gregor Gössler
Chargé de recherche, INRIA Grenoble Rhône-Alpes, Directeur de thèse

Acknowledgements

La thèse est un animal bizarre. Une aventure de trois ans dont on connaît le point de départ mais pas vraiment la destination. Ce voyage n'a pas toujours été aisé, mais cela ne l'a pas empêché d'être enrichissant.

Je pense que les premières personnes a remercier sont celles qui m'ont donné le goût de la recherche, cette curiosité d'essayer de comprendre les choses. Mes parents ont été, a travers l'éducation qu'ils m'ont prodiguée, les principaux acteurs de la construction de cette facette de ma personnalité. Si mon père n'a pas vu la fin de mes pérégrinations, il a été présent tout au long de ma thèse, ne serait-ce que par tout ce qu'il m'a enseigné et apporté. Mes différents professeurs, durant mes (nombreuses) années d'études y ont aussi participé.

Mais si ces derniers m'ont amené jusqu'à la thèse, les personnes que j'ai côtoyé, et qui ont rendu ce périple plus agréable au jour le jour sont les membres de l'équipe SPADES. J'ai pris beaucoup de plaisir à travailler dans cette équipe. Que ce soit les repas, animés de discussions diverses et souvent passionnées, les thés, les pauses et tous ces moments de vie du quotidien, qui rendent le travail plus agréable. Je suis extrêmement reconnaissant à Helen, qui a transformé la corvée des tâches administratives en de simples formalités.

Je tiens à remercier particulièrement les non-permanents, avec qui nous avons fait de nombreuses et parfois longues pauses, qui ont étés des distractions bienvenues aux divers problèmes du travail de thèses. En espérant n'oublier personnes, merci à Dima, Vagelis, Gideon, Quentin, Willy, Adnan, Cricri, Judas, Martin et Stephan.

Je remercie aussi Gregor Goësler, mon directeur de thèse, sans qui rien n'aurait été possible, ainsi que pour les nombreux échanges et débats scientifiques que nous avons eu.

Un grand merci à tous mes amis et ma famille. Ils m'ont soutenu, tout au long de ma thèse, même si Grenoble n'est pas la porte à côté pour nombre d'entre eux.

Enfin, un énorme merci à Marie-Cécilia. Si nos chemins ce sont maintenant séparés, elle a été mon équipière dans la vie pendant 7 ans, dont l'intégralité de la thèse. C'est elle qui a dû supporter au quotidien tous les tracas de la thèse et a été d'un immense réconfort lors de la mort de mon père. Sans elle ce voyage aurait été moins facile et moins heureux, et je lui suis reconnaissant d'avoir partagé ce dernier avec moi.

Contents

Section 1

Introduction

A widespread solution to be able to create complex systems is to resort to component-based architecture. The idea is to create subsystems, namely components, that will ensure certain functions of the system. This makes the design process easier, as some smaller part can be designed and then merged together. However, if components are a boon for designing systems, it is a bane to analyse them. Indeed, the operation of assembling the components, that will be referred as the composition, is generally complex and hard to reverse. Therefore, it is hard to associate system level behaviour with the components, as the components interact with one another, and the expression of a bad behaviour is often not its cause.

Would a bad behaviour (called failure) be observed, it is desirable to to be able to identify the components that are responsible for it. Indeed the failure may occur in a safety critical system, thus endangering human lives, or disrupt a service being provided, which is generally bad for the provider it, both economically and in term of image. Identifying the components responsible for a failure is the first step in being able to avoid the occurrence of the failure, and repair it.

This identification problem is mainly studied by diagnosis and fault localisation. Even though those techniques can identify from which components the failure stems from, they are not able to assess whether those components are responsible for the failure or not. The ability to ascribe responsibility is useful to chose the components that should be modified, or repaired, in order to fix the failure, as well as in a legal framework, to be able to held responsible the designer/builder of the incriminated components. One of the approaches that is able to attribute responsibility in a component-based setting is causality analysis. It uses a counter-factual approach ("what would have happened, would this component not be faulty?") to assess whether or not a set of components is responsible for a system failure.

This thesis expands this framework in two ways. The first one is to extend the causality analysis framework from a black-box component setting (components for which we know the expected behaviour, usually the ones diagnosis consider) to a mixed framework with both black-box and white-box components (components for which we know the actual behaviour, usually the ones considered in fault localisation). This extension is interesting because a lot of systems use off-the-shelf components (black-box) with custom ones (white-box), and this approach provides a tool to ascribe responsibility to components in mixed systems, which, to our knowledge is a novelty. Using a game, one is able to assess responsibility for systems composed of both black-box and white-box components. This approach is a generalisation of the causality analysis approach for the black-box components, and works similarly to fault localisation techniques for the white-box components. The approach relies on techniques that are close to the one used in controller synthesis, and thus can generate fixes for the bug observed.

The second main axis is to study the impact of the amount of information accessible on the construction of the counter-factuals. Having more information, like the memory state of the components or a fault model for the components, can be used to build more refined counter-factuals. Those counter-factuals will be closer to what would have happened if components had been fixed, thus yielding some more accurate responsibility assignment. The other end of the spectrum is what happens with less information (missing variables in the trace, or missing portions of the trace). Some general results are given, as well as a way of performing causality analysis on a partial trace that yield the same results as the one performed on the full trace, under certain assumptions.

This thesis will be divided as follow:

Section 2 will give an overview of the state of the art diagnosis, fault localisation and causality-based approaches.

Section 3 develops the base definitions used in the whole thesis, as well the causality framework from [Gössler and Métayer, 2015].

Section 4 presents the implementation of the causality framework introduced in the previous section.

Section 5 expands the causality analysis from a black-box approach to a mixed one (black and white-box). It builds from the causality framework from Section 3 and uses controller synthesis to assess causality.

Section 6 extends further the approach from the previous section, by considering a finer grain description of the system and using game to be able to assess responsibility in the mixed systems.

Section 7 presents the impact of having less, or more, information on the results of the causality analysis.

Section 8 concludes this thesis and gives some future perspectives and possible extensions of this work.

Section 2

State of the art

The approach developed in this thesis aims at blaming components of a system for a global system failure. Another way to look at it is that some effects are observed (namely the failure), that reflect some faults in the system, and the approach aims at determining which faulty components are the cause for the failure. In that respect, it is similar to the goal of both diagnosis and fault localisation.

According to [Reiter, 1987], given a system description and an observation which contradicts this description, the diagnosis problem is to determine the components of the system which, when assumed to be functioning abnormally, will explain the discrepancy between the observed and correct system behaviour. This problem is close to the Causality Analysis approach, and is then of interest in this state of the art.

The thesis approach is in a black-box component setting, i.e. we have access to a description of the correct behaviour of the components. Fault localisation focuses on a white-box setting, where the actual behaviour can be used, like the source code for instance. Fault localisation has a goal similar to Causality Analysis: identifying the parts of the system where the faults occurred. As defined in [Steinder and Sethi, 2004] Fault Localisation is a process of deducing the exact source of a failure from a set of observed failure indications, in a network context. In a program context, it can be defined, as in [Alipour, 2012], as a task in software debugging to identify the set of statements in a program that cause the program to fail.

The state of the art will focus on the fields of diagnosis, and fault and bug localisation.

The approach presented in this thesis uses causality analysis, that aims at linking causes to effect. There are different causal frameworks, but the one used here is a counter-factual reasoning, i.e. a "what if" reasoning. For instance, "what would have happened if this component had not behaved

abnormally?". Therefore, the state of the art will only develop causal frameworks that use counter-factuals.

This section will be divided in three subsections. The first one will focus on diagnosis. The second one will discuss of fault localisation. The last one will present approaches from both fields that use a notion of causality.

The split between approaches that rely, or not on a causality notion is motivated by the fact that causality approach are the closer to the one developed in this thesis, hence the fact of treating them in an independent subsection, but they should be compared to the other techniques (gathered the other subsections). What's more, certain specific concepts need to be explained, in order to fully understand the causality approaches, which is why the subsection will begin by a quick discussion on the notion of causality itself.

Vocabulary definitions Throughout this document, the notions of error, fault and failure will be used. Definitions will then be given for them in this paragraph, inspired by the ones of [Laprie et al., 1990].

We suppose we are given a system. This system can be a program, a network, a cyber-physical system,... This system is composed of sub-systems (statements for a program, nodes for a network,...) that will be called components. Both the system and the components have an expected behaviour.

We also suppose we are given an observation of the functioning of the system, or the components. E.g. an execution logs for a program, the outputs of the sensors over time for a cyber-physical system... This observation is called a trace.

An *error* is a defect in a hardware device, a program, a component,...

A *fault* is a component behaviour, in a component trace, that differs from the component expected behaviour. It is the manifestation of an error.

Given a component, a trace is said to be *faulty* if it contains, at least, one fault. By extension, the component is said to be faulty.

A *failure* is a system behaviour, in a system trace, that differs from the system expected behaviour.

A *failing* trace is a system trace that contains, at least, one failure.

Here is an example of instantiation of those definitions to software: an error is a mistake in the code, a fault is a bug, that is the result of the execution of errors, and a failure is a software crash.

The different approaches presented in this section have in common that given at least one failing trace, they aim at identifying the components, or

parts of the code, that are (likely) responsible for the failure, or the locations where the faults occured.

A component that is a cause for a failure is also said to be responsible for the failure.

2.1 Diagnosis

Diagnosis is a vast field, which has been widely studied. However, it is way too extensive to fully cover it in this state of the art. Therefore, this state of the art will focus on the precursor papers, and several others that are related to fault localisation in programs.

One of the founding paper of the field is [Reiter, 1987]. It gives both a definition of the diagnosis problem, and a general way of solving it, alongside an application example to the medical field. The inputs of the approach are a description of the system and a (failing) observation. A diagnosis is a minimal set of components such that, if they are considered as behaving abnormally, and all the other components behave normally, the observation is consistent with the system description. This work formalises the notion of diagnosis, in a way which is consistent with the non-formal definitions used at the time. One important feature of this approach is that it is model independent, as it externalises the queries that are specific to the model.

At a similar time as Reiter, [de Kleer and Williams, 1987] proposed a way of diagnosing multiple faults. The approach has the same inputs as the previous one, with the addition of a probability of failure for each component. It uses an inference procedure to give rise to symptoms (differences between the observation and the system description). From those symptoms, conflict sets (set of components that cannot be all behaving normally) and candidate sets (minimal sets of abnormally behaving components that explain the symptoms) are computed. The algorithm chooses the best measurement to be performed, in order to refine the diagnosis, using Shannon entropy to discriminate between the different candidates. It gives a sequential way to makes the diagnosis, in order to conclude. Note that it implies that you can have access to the system, to actually perform the measurements, that it will not change its behaviour over time, and that measuring does not affect the outcome either. Note that those hypotheses apply to all incremental diagnosis that necessitate measurements, regardless of the theory of diagnosis used.

The two previous approaches are said to be consistency-based diagnosis: they try to find diagnoses that are consistent with the observation.

[Poole et al., 1987] proposed an abduction-based diagnosis. All the fault models for each components are given, alongside an observation. This approach finds the component and the corresponding faults that explains the observation. However, in the system that Causality Analysis aims at studying, it is generally easier to have access to a system description, than all the failure modes. What's more, all the failure modes for each component generally make for a bigger state space than a system description. Since the abduction-based approaches differ greatly from the one developed in this thesis, they will not be further treated.

In [Console et al., 1993], the authors showed that those theory of diagnosis are applicable to programs. If you derive a specification from the source code, and possibly fault models, alongside a failing trace, you have the inputs for the diagnosis problem. Those approaches can therefore be used as fault localisation techniques in programs.

Diagnosis is often tightly connected to fixing faults in physical systems. This is the case of [Madre et al., 1989] that performs a diagnosis on the system, and proposes a fix for the faulty components, by replacing them with a Boolean equation. [Heh-Tyan et al., 1990] improved this technique by using the topology of the system and proposing a new algorithm for the fix.
In [Jobstmann et al., 2012] the idea is transposed to components that are finite-state machines, and properties that are either invariants or linear-time temporal logic (LTL) formulae. The authors model the diagnosis and fixing problem as an infinite game. The approach is complete for invariant, and "works well in practice" for LTL formulae.

[Pons et al., 2015] proposed a way of performing diagnosis on hybrid systems. The core of the paper lies in the way they transform the hybrid system into an automaton, and how they build a causal system description (that reflects influence of variables over variables, rather than causal relations). This causal model description is then used to perform an incremental diagnosis, using Reiter framework. One of the most interesting aspect of this paper is that using model transformation, they are able to apply Reiter's approach, which was mostly used on simple models, to hybrid systems, which are arguably a very complicated model.

2.2 Fault localisation

In fault localisation, the effect of the fault (the failure) is observed, and the goal is to pinpoint the fault(s) from which the failure stems from. There are

two main approaches to fault localisation: spectrum-based and model-based. The first one consist in analysing a big set of traces from the system, in order to pinpoint the possible localisation of the fault(s). The second one uses the model to infer the fault(s) localisation that induced the failure present in a given trace.

2.2.1 Spectrum-based fault localisation

In today's context, it is not unusual to have access to a large set of traces. In the design phase, systematic, or at least large, testing is widely used, creating big sets of traces for which the outcome, failing or non-failing (passing), is known. Another way to have access to a lot of traces are the reports from the users.

Those big sets of traces can be analysed using suitable statistical algorithm (machine learning, pattern recognition, data-mining,...), in order to extract information about the program, that can be used to track the bugs. Those approaches will be called statistical techniques, as suggested in [Pal and Mohiuddin, 2013]. All the techniques presented here are dynamical techniques, i.e. they analyse specific runs, static techniques will be treated in other subsections.

Those techniques do not compare directly to the approach developed in this thesis, as they need a set of traces, and not a single faulty trace. Nevertheless, it is interesting to make a quick presentation of such techniques, as, given those set of traces, they are very efficient at pinpointing the fault location. What's more, those techniques aim at reducing the amount of information the developer needs to look at, in order to locate the bugs in a program, e.g. it isolates a set of statements, of a path in the control-flow graph. It is close to the blaming objective.

It is worth mentioning that similar techniques that aim at generating invariants for programs exists (such as DySy [Csallner et al., 2008], DAIKON [Ernst et al., 2007] or PRECIS/PREAMBL [Sagdeo et al., 2011]). However, since their goal does not fit in the blaming framework, I will not detail them.

General principle The statistical techniques have common ground. The code is first instrumented to gather some information about the program execution. Then, a large number of traces is generated. Afterwards, this set of traces is analysed to extract likely invariants or bugs (via some form of predicators) of the programs. Even though the number of traces is large, it does not cover the whole state-space of the program, and not necessarily all the errors, as the error needs to be executed, and result in a fault to be detected. Consequently, it is possible to have generated invariants that are

only true for this set of traces, but not in the general case. Similarly, the bug predicators might be true on non-buggy traces that are not present in the considered set, giving rise to false-positives.

Several of those approaches use an iterative process, where the set of traces is analysed, and the code re-instrumented to get finer invariants/bug predicators.

One of the main advantages of those techniques is that they can tackle multiple bugs/infer multiple invariants. It stems from the fact that they have access to a large set of traces, where statistical tools can be used.

Where those techniques differ is in the information they use to both instrument the program and analyse the traces. In term of instrumentation, we can mention the data, the control-flows, the program paths, the memory state,...

In the next paragraphs, some of those techniques will be presented.

Testing phase approaches During the test phase of a program, most of the code is instrumented, and test case are run. The traces are labelled as passing (no failure) or failing (a failure occurred). Since this instrumentation is generally done during the testing phase, the overhead of fault localisation is just running the localisation algorithms, and it helps pinpointing the origins of the bugs quite accurately.

The approaches used in the testing phase have access to both the source code and a big set of labelled traces.

TARANTULA have been introduced in [Jones et al., 2002]. The underlying idea behind this approach is that statements that are more executed in faulty traces are more likely to be faulty. The statements are then sorted according to how often they are executed in faulty runs, helping the programmer on the selection of which statements to explore to fix the bugs.
This technique has been compared to four other similar techniques in [Jones and Harrold, 2005
namely nearest neighbour ([Renieris and Reiss, 2003]), set union ([Agrawal et al., 1995]), set inclusion ([Pan and Spafford, 1992]) and Cause Transitions ([Cleve and Zeller, 2005]).
Set union performs the set difference between the set of statements executed in a faulty trace, and the set of the statements executed in a passing traces. The resulting set is used as an initial set of suspicious statements.
Set intersection uses dynamic-slice-based heuristic that identify a set of "blamed" node, and then rank the other nodes according to how the "blamed" nodes are dependent on the node being ranked.
Nearest neighbour takes a faulty trace and a set of passing traces as input. It selects the passing trace the closest to the failing one, and perform a set

union on those traces
Cause transitions performs a binary search of the memory states of a program between a passing test case and a failing test case.
TARANTULA outperforms those four techniques, in term of effectiveness of the fault localisation, and has the same efficiency as the least computationally expansive one.

A crosstab-based statistical method is proposed in [Wong et al., 2008]. For each statement, a degree of association with the outcome (be it failing or passing), and a quantity reflecting if the statement is associated with the failure or the success of the run. The statements are then split in five classes, according to how much the statements are associated with the failure or the success. The statements highly associated with the failure are to be examined first.
This technique is more effective than TARANTULA (it pinpoints fault more accurately) and has a comparable efficiency (computational cost).

CBI and related techniques In [Liblit et al., 2005], the authors have presented an approach that was later implemented in CBI (Cooperative Bug Isolation), in 2007. This approach first instruments the code, then gathers a big set of traces, containing the information instrumented. Lastly, an algorithm ranks the different bug predicators, given how well they predict bugs. Then the best predicator of the most severe bugs are removed, and the next most severe bug is considered, and so on.
This approach is able to capture most of the bugs from a thousand traces, up to several dozen thousands, for the rarest bugs. It has been used in several widely used software.

[Ball and Larus, 1996], [Jiang and Su, 2007] and [Arumuga Nainar et al., 2007] advocate that control flow paths give more information to locate bug than predicates. Therefore, PATHGEN ([Jiang and Su, 2007]) extends CBI by associating the bugs with control flow paths, instead of bug predicators. In terms of performance, it is better than CBI, as paths are more informative than a collection of statements. However, the choice of which parts of the code to instrument might prove to be difficult.

In [Chilimbi et al., 2009], another extension of CBI is proposed. HOLMES associate bugs with control flow paths, like PATHGEN, but the instrumentation performed is different. HOLMES instruments control flow paths, instead of predicates, leading to a way smaller overhead. The performances are comparable to PATHGEN, though it is not proven to be better. However, the light instrumentation induce a smaller overhead on the execution of the pro-

gram.

The three previous approach only take into account the failing runs, while SOBER ([Liu et al., 2005, Liu et al., 2006]) also considers the passing runs. The authors associate each predicate with an evaluation bias (ratio of passing runs where the predicate is true). If the evaluation bias is very different from a normal distribution, it is a good bug predicator. Performance wise, this approach is better than the previous ones. However, it was compared on a "best case scenario" (namely the Siemens test suite), that provides a large number of traces, that are accurately labelled.

Non-parametric bug localisation techniques The approaches presented in the previous paragraph all rely on a distribution assumption of the occurrence of the bug predicates (namely a normal distribution). However, [Hu et al., 2008] and [Zhang et al., 2009] showed that this assumption is generally not true, and the distribution for the good bug predicator is very far from the normal distribution.

[Hu et al., 2008] uses an approach very similar from HOLMES, but with a Mann-Whitney test to rank the predicates, that does not rely on any normality hypothesis. This makes the bug location more accurate, while keeping a computational cost close to the one of HOLMES.

Bug localisation using evaluation sequence. In [Zhang et al., 2010], both CBI and SOBER are transformed by using evaluation sequence, instead of simple bug predicators. The resulting DES_CBI and DES_SOBER outperform the original ones, in term of accuracy of the bug localisation, while maintaining a similar computational cost.

We can conclude, from the papers presented here, that not making any assumption on the distribution of the bug predicators, and using execution paths, or evaluation sequences, further enhance already efficient techniques.

2.2.2 Model-based fault localisation

Contrary to the statistical approaches, the ones discussed here do not rely on a set of gathered traces, but on a single failing trace. Since they cannot perform a statistical analysis on a set of traces, some more information must be used, such as a specification, an implementation or the system itself. Those approaches are closer to the causality analysis framework, as it only has access to the same inputs.

The next paragraphs will present some of those techniques. They will not be fully developed, but their general principle will be given, and some recent influential papers.

Program slicing Program slicing was introduced by [Weiser, 1981, Weiser, 1982]. It consists in a backward search of the fault, starting from the program failure. Statements are pruned by using the control and data flow to keep the statements from which the failure has a dependency. This reduced number of statements is called a slice, and narrows the portion of code that must be examined for debugging.

Since slices tend to be quite big, [Lyle and Weiser, 1987] developed the notion of program dice. It uses a principle close to [Pan and Spafford, 1992], as it makes the set difference between static slices of correct and incorrect variables. However, some statements can only be removed by predicting run-time values. Following this observation, [Agrawal and Horgan, 1990] introduced the dynamic program slicing, that performs the slice using run-time information.

Dynamic program slicing has been extensively studied, some interesting papers in the field will be discussed in the remainder of the paragraph. Program slicing can capture the execution of bad statements, but cannot cope with the omission of a statement execution due to a fault.
[Zhang et al., 2005, Zhang et al., 2007] addressed this issue by introducing the concept of implicit dependencies and relevant slicing. Some missing dependencies are obtained using delta-debugging.
Another way to overcome this issue is proposed in [Agrawal et al., 1995, Wong and Qi, 2004]. The idea is to consider blocks instead of statements. A block depends on another block if a statement in this block depends on something contained in the block it depends on. The fact that blocks contain more than one statement means that the slice is less likely to miss relevant statements. On the other hand, it also means that the slices are bigger.

Though it is not exactly program slicing, [Wang et al., 2006] proposed a technique which relies on a similar principle. Given a failure and an execution trace, a weakest pre-condition is computed backwardly from the failure. A minimal proof of infeasibility is then computed. Since the trace is considered as a word of statements, the proof of infeasibility is a set of minimal words, thus a set of sequence of statements. Since this technique uses the code and some run-time information, it is close to dynamic program slicing. The authors claim that their technique is superior to dynamic program slicing. however, their claim is only based on two toy examples, and not formally proven.

Delta-debugging Given a set of possible inputs for a program, delta-debugging aims at reducing this set to the smallest subset of inputs that lead to the program failure. It was first introduced in [Zeller and Hildebrandt, 2002], and further expanded in [Zeller, 2002, Cleve and Zeller, 2005]. The underlying idea is that smaller inputs cover less code than bigger inputs. An important assumption is that inputs can be simplified, which is not always possible.
A shortcoming of this technique is that is treats all the possible inputs as a list, which might be big. Hierarchical delta-debugging ([Misherghi and Su, 2006]) addresses this problem by taking into account the structure of the input to reduce the set of inputs to explore. Using a coarser version of the algorithm at the beginning, it prunes a large portion of the inputs that is irrelevant. Besides the speedup, it also also give more precises results, as the output is a structured tree.

[Misherghi and Su, 2006] proposes a combination of both delta-debugging and slicing to narrow down the part of code to explore. The idea is to perform a delta-debugging, thus yielding a minimum set of inputs. A forward slice if performed on this minimum set, as well as a backward slice from the failure. The output of the approaches is the set of statements that are in both the backward and forward slice. This technique gives some more precise results than both delta-debugging and program slicing.

Counter-example simplification and explanation The approaches presented in this paragraph aim at making the faulty trace more understandable, rather than pinpointing the location of the faults. It is especially interesting if the system considered are very complex, or there are multiple faults, as then the automated fault localisation may fail. When it is the case, human must look into the complex traces, and simplifying, them is important to ease that examination.

In the context of concurrent program, one of the main difficulty with analysing the trace are all the context switches (change from the execution of one thread to the execution of another one). Following this observation, [Jalbert and Sen, 2010] proposed an algorithm that raises a trace equivalent to the faulty trace with a minimum number of context switches. The idea is to try to eliminate the context switches that do not contribute to the bug, beginning by the last one, and trying to add the statements from this context switch to the previous execution of the thread, or the next one. The problem being NP-hard, the authors proposed a heuristic that yields a simplified trace that is either easier to understand than the simplest one, or the simplest one itself.

In [Jin et al., 2004], the concept of fated and free segment in a trace is introduced. A free segment is a segment where the system could have made a "choice" that would have avoided the failure, a fated segment is one where there was no such possibility. The inputs of the system are split into two sets: those controlled by the environment and those controlled by the system. The authors proposed to build a game where the system tries to avoid the failure, and the environment tries to make the system fail. The states are partitioned in an "onion ring" fashion, characterised by the number of "mistakes" the system must make in order to enable the environment to have a winning strategy. The ring are the fated parts and the transition between the rings the free segments. This approach is especially interesting for very long traces, as the person analysing the trace can focus on the free segments. However, the partition between the system and the environment is not necessarily trivial to make, and the computational cost being too high, they must rely on a heuristic.

In the context of counter-examples produced by model-checking, [Ball et al., 2003] proposed to use additional information that is anyway computed by the model-checker to help understanding the traces. When a failing trace is found, it is compared with passing traces generated by the model-checker. "Halt" statements, that ensure that the same failing trace will not be yield again, are introduced. By adding those "halt" statements, this approach makes sure that every failing trace are generated. In a similar fashion to [Pan and Spafford, 1992], those failing traces are contrasted with the passing traces to highlight the problematic parts. This approach is very efficient, but since it uses model-checking does not scale very well.

[Groce and Visser, 2003] introduced an approach that resemble the previous one. Given a failing trace, it computes all the similar passing and failing traces using model-checking. It yields a big set of traces, like in Section 2.2.1. The authors proposed to perform an invariant, a transition and a minimal transformation analysis on the generated set of traces. The main contribution of this paper is the generation of the set of traces, that can have many application, such as making a query on a passing trace that the user suspects to be "close to" failing traces. Note that this approach could have been in Section 2.3, as the underlying idea can be seen as a causal reasoning. Whatsoever, it was presented in the current section, as it does not rely on an explicit causality definition.

Reduction of the fault localisation to Max-SAT [Jose and Majumdar, 2010] designed a reduction of the fault localisation problem to a Max-SAT problem. A Max-SAT-solver is a SAT-solver that supports "soft-constraints". A soft-

constraint, is a constraint that may be false. Soft-constraints are generally weighted, and the Max-SAT-solver returns a counter-example that maximise the cumulative weight of the satisfied soft-constraints. The input value and the program property are hard constraints, whereas the translation of the statements are soft constraints. The Max-SAT solver returns an Unsat core, i.e. a set of soft constraints that cannot be *true* while all other constraints are. Unsat core then reflect a set of statements that are problematic, with regards to the property, given the input of the program. Performance wise, the translation of the statements to a SAT problem can easily blowup computationally, so there are some scaling issues. However, the approach is general, and can be used at different granularity level. What's more, it is very efficient on small but complex problem, and since the solver part is externalised, this approach will benefit from the future progress of the Max-SAT Solvers.

2.3 Causality-based approaches

Causality have been extensively studied in philosophy (for instance, Aristotle treated this topic). However, one of the framework that is the more resilient to several problematic examples, and that suits well computing science is the counter-factual theory of causation ([Menzies, 2014]). The idea of counter-factual was first introduce in 1748, by Hume ([Hume, 2004]), however, the most influential theory is that of Lewis ([Lewis, 1973]), which was revised in [Lewis, 2000]. In [Lewis, 1973], Lewis exposed a definition of counter-factual which is: "Where c and e are two distinct actual events, e causally depends on c if and only if, if c were not to occur e would not occur". This is the basis of the counter-factual reasoning, the idea that if the cause is not present, then the effect is not either. Note that the previous proposition implies that the cause is always present if the effect is.
This notion of causation is pushed further, by the introduction of the notion of causal chain. "e causally depends on c if there exist a causal chain leading from c to e". This is basically a transitivity relation, i.e. if c is a cause of d, and d a cause of e, c is a cause of e. The definitions of Lewis are general, therefore, there might be some problem due to the fact that, in real life, you can always think of a crazy scenario. For instance, if the action c of a person is a cause for event e, one might argue that the fact that this person was not killed by a meteorite before doing her action is a cause for e. This problem was not entirely addressed in the 1973 definitions. In order to tackle this issue, Lewis defined, in [Lewis, 2000], the notion of influence: "Where c and e are distinct events, c influences e if and only if there is a substantial range of c_1, c_2, ... of different not-too-distant alterations of c (including

the actual occurrence of c) and there is a range of e_1, e_2, ... of alterations of e, at least some of which differ, such that if c_1 had occurred, e_1 would have occurred, and if c_2 had occurred, e_2 would have occurred, and so on.". This definition is able to deal with the "meteorite" example, because this scenario is very unlikely, and we can have a "substantial range" of similar causes without resorting to this crazy scenario. It formalises the idea of a set of world close enough to what actually happens. However, this model is still too general to be used in computing science. The Structural Equation Framework, developed in [Hitchcock, 2001, Woodward and Hitchcock, 2003] can more easily be applied to computing science, as it will be discussed in the next paragraph. The other paragraph will treat of counter-factual approaches that do rely on a model of the system as a normal behaviour description, rather than structural equations.

Structural Equation Framework One of the first, and most influential, paper on counter-factual reasoning in computer science is [Halpern and Pearl, 2001b, Halpern and Pearl, 2001a]. The authors formalised the structural equation framework, and proposed a definition of actual cause, and of explanation.

This framework makes the distinction between the exogenous variable U, the context of the system, the endogenous variable V, the variables belonging to the system. The system is associated with a causal model M, which links every variable x in V with a function F_x that deterministically returns the value it should take, given an assignation for every other variables.

Given a system, a context $\underline{u}$, a causal model M, a predicate φ, $\underline{X} = \underline{x}$ (the assignation $\underline{x}$ to the variable in $\underline{X}$) is an actual cause to φ if, AC1. the situation can actually happen, given M and $\underline{u}$, AC2. changing the values $\underline{X}$ to $\underline{x}'$ falsifies φ (removing the cause removes the effect) and modifying other variables in $V \setminus \underline{X}$ does not falsify φ, and lastly AC3. is minimal. Besides AC1., which is trivially necessary to make counter-factual analysis interesting, AC2. and AC3. define a rather strong definition for actual causes, as the actual cause must be minimal (AC3) and counter-factually responsible, and other variables must not be counter-factually responsible.

One could argue that this definition only yields causes in a given context, therefore, the authors introduced the concept of explanation. The inputs are the same as for the actual cause, besides the fact that $\underline{u}$ is replaced by a set of K of contexts. Roughly, the idea is that $\underline{X} = \underline{x}$ is an explanation for φ if, whenever $\underline{X} = \underline{x}$ is *true* in $\underline{u}$, $\underline{X} = \underline{x}$ is a sufficient cause for φ (it enforces AC1. and AC2., but not necessarily AC3.), and $\underline{X}$ must be the minimal set such that the previous holds.

Those two definitions accurately assign responsibility for most of the

causality examples that are generally hard to accurately treat. However, one of the main weaknesses of this approach is the causal model. Firstly, though it can model precedence between events, it cannot model timed systems. Secondly, the result of the analysis is highly dependant on the causal model, which is generally not available. Therefore, the person building the model could "orient" the outcome, by making certain modelling choices.

One of the interesting follow-up on Halpern and Pearl model is [Beer et al., 2012]. It extends the definitions to causal models described in Labelled Transition System (LTS) and formulae in Linear Temporal Logic (LTL). It fixes one of the main drawback of the previous approach, namely the incapacity to handle timed systems. The authors used those definitions to highlight events that are causes for the failure in a trace. The complexity of the approach being too high, they propose a heuristic that turns out to give more intuitive explanations than the full algorithm.

In later work, Chockler and Halpern ([Chockler and Halpern, 2004]) added a notion of degree of responsibility to the framework. Intuitively it quantifies the number of changes that need to be made to the context ($\underline{u}$), in order to make $\underline{X} = \underline{x}$ a cause of φ. For example, if an election where a given number of votes must be reached in order to win is won by one vote, the degree of blame is 1 for each voter (one change of vote changes the outcome of the election). If the election is won by 10 votes, each voter has a degree of blame of $\frac{1}{10}$ (10 persons need to change their vote to change the outcome).
The notion of degree of responsibility can be further extended with a probability associated with each context $\underline{u}$. By weighting the degree of blame, using this probabilities, a more accurate measure can be made. Indeed, the less likely scenarii will have a lesser weight in the degree of blame.
In [Chockler et al., 2008], the degree of responsibility is used in a Symbolic Trajectory Evaluation to determine which inputs to use to refine the model, in hardware model-checking. It enhanced the performance considerably, compared to other similar techniques.
The results that can be obtained using the degree are illustrated in [Chockler et al., 2015], where they apply the framework to a legal case, namely the death of Baby P. This paper shows that using a causality analysis approach, they get similar results to what the jury decided. However, it suffers the usual problems of the approaches with causal model, which is, "how to choose the causal model?", with an additional layer with the choice of the probabilities for the different possible contexts.
This notion of degree of responsibility has also been used in [Debbi and Bourahla, 2013]

to generate the "most indicative" counter-examples to a Probabilistic Computation Tree Logic (PCTL) formula applied to a Discrete-Time Markov Chain (DTMC). A probabilistic model-checker is used to generate counter-examples, and the notion of degree of blame is utilised to assess which counter-example is the "most indicative" (counter-example with the least number of paths, and higher probability). However, the authors fail to explain how to exploit this most indicative counter-example.
In [Kuntz et al., 2011], the framework is extended with a notion of order between events, thanks to priority AND operator ($P - AND$, which is *true* if its inputs become *true* in a given order). Using this priority notion and the actual cause from [Halpern and Pearl, 2001b], the authors are able to build fault trees from a probabilistic model-checker counter-example.

Model-based causality analysis Several causal models do not use the notion of structural equation, but a notion of "normality". That is a description of how the world usually is, or normally is. This causality notion is very close to the influential one, introduced by Lewis. [Kayser and Nouioua, 2005] advocate an actor based approach, whit the notions of *should*, *able* and *hold*. From those, the authors track back to a "primary anomaly", which is considered to be the cause. In [Dupin de Saint Cyr Bannay, 2008], a distance metric is used to compute the closest normal trajectory to a failing trajectory. Contrasting the two gives some information about the causes of the failure.
Those approaches have in common the fact that they contrast the failing trace with counter-factual traces that are normal, and close to the initial failing trace. They resemble diagnosis, with a causal relation added on top of the normality notion. What's more, the idea of contrasting failing and passing (or "normal") traces is present in numerous approaches presented in this state of the art.
Contrary to the Halpern and Pearl framework this framework has not received much attention.

In [Groce et al., 2006], the authors develop a technique close to delta-debugging, adding a notion of causality. Given a failing trace and a program specification, they build the closest passing trace, via a model-checker and a distance metric. The counter-example is found by altering the input values, until the closest passing trace is found. The counter-example and the passing trace are contrasted using delta-slicing, thus pinpointing the problematic parts of the code. This approach gives a narrower portion of code to explore (a couple of lines), compared to other techniques, in the examples treated by the authors. However, the user might need to give some assumptions for

the tool to conclude (since the specification is more abstract than the implementation). What's more, due to the way the algorithm is implemented, this approach cannot deal with more than one cause.

The approach used in this thesis was first introduced in [Gössler and Le Métayer, 2013]. It will be presented at great length in Section 3, but a broad idea will be developed here. Given a failing trace and a set of components, forming a system, this approach returns the sets of components that are causes of the system level violation. It proceeds as follows given a set of "suspected" components, the faults from those components are removed from the failing trace. From this "pruned" trace, all the possible traces are built, using the component specifications to prolong the traces. This gives rise to a set of counter-factuals on which we can apply causality definitions.
A similar approach is proposed in [Datta et al., 2015]. It considers program actions instead of components. Though I did not prove it formally, my intuition is that this approach, in its current state, yields coarser results than causality analysis. However, what is proposed in [Datta et al., 2015] seems to be in an early stage.

Conclusion

As shown in this state of the art, localising the cause(s) of a failure is a very complex problem that is studied by many communities. If you have access to big set of traces, obviously the approaches from Section 2.2.1 are the most efficient, since they can factor in statistical data about the passing and failing traces.

However, the approach developed in this thesis only has access to a faulty trace and a specification for the model. The model-based approaches are various but generally rely on principle close to causality, by contrasting failing and passing traces or failing traces with the specified behaviour.

Note that the frontier drawn between the causality-based approaches and some model-based is a bit arbitrary, since delta-debugging and the model-checking approaches use a notion of counter-factual, by contrasting the failing trace with passing traces close to it. This is not surprising, since counter-factual reasoning is quite intuitive, and very efficient at establishing causal relations.

However, Section 2.3 showed that generally, using an explicit causality notion, oftentimes Halpern and Pearl one, enhances the performance of existing approaches. This is due to the fact that an explicit causal relation usually is good way of determining responsibility or making choices in a fault localisation or diagnosis setup, since it describes well the links between the failure

and the faults.

This thesis will show further development of the approach introduced in [Gössler and Le Métayer, 2013]. This approach uses a causality notion different from Halpern and Pearl that does not relies on structural equation, which is the major drawback of their causality notion, while coping with problematic causality examples. What's more, Section 5 will introduce a way to mix white-box and black-box fault localisation, which, to my knowledge, has not been studied.

Section 3

Causality Analysis framework

In this section, the notions and notations that will be used in the remaining of the thesis will be introduced. The first subsection will focus on the formalisation of the system, and some general notations. The next one will contain definitions related to causality, as well as two ways of building the counter-factuals. The last one will illustrate the causality analysis with various example, showing some of the issues that can be tackled using causality analysis.

3.1 Notation and General definitions

Notation *Index:* The index are generally noted with i, j or k. The set of values indexes can take are generally noted I. We note $[n..m]$, with $n, m \in \mathbb{N}$ for $\{n, n+1, \ldots, m-1, m\}$. If $n > m$, $[n..m] = \emptyset$. The boundaries for integer value will generally be noted with n or m.

Vector: The vectors are are underlined, e.g. $\underline{v}$. Let $n = |\underline{v}|$ be the size of the vector. $\underline{v}$ can also be represented as a sequence of values indexed by $[0..n]$: $\underline{v} = (v_i)_{i \in [0..n]}$. We note $\underline{v}[i] = v_i$ the i^{th} value of $\underline{v}$. We note $\underline{v}[n..m] = (v_i)_{i \in [n..m]}$ the subvector of $\underline{v}$ from the n^{th} to the m^{th} value. For the sake of conciseness, we note $\underline{v}[n..]$ for $\underline{v}[n..|\underline{v}| - 1]$. We note $()$ for the empty vector.

Size and cardinal: The cardinal of a set I is noted $|I|$. Likewise, the size of a vector $\underline{v}$ is noted $|\underline{v}|$.

Sequence: The notations defined for vectors (e.g. $\underline{v}[i]$) will also be used for sequences. Let D be a set. We note D^* for the set of all possible finite sequences over D. We note D^ω the set of all infinite sequences over D. $D^\infty = D^* \cup D^\omega$.

General definitions This paragraph will present the formal definitions of a system, a trace,... that will be used throughout this document.

The base model chosen for this thesis is data flow synchronous language ([Halbwachs et al., 1991a]), like LUSTRE, LUCID, SIGNAL... Those language are used to program reactive system ([Berry, 1989, Halbwachs et al., 1989]). A reactive system reacts to the inputs of an environment as fast as the environment evolves.
The data flow part refers to a system architecture with components linked by data flows. Those data flows can be seen as pins, wires or, more generally, links that carry the data from one component to others . It means that the components take some input flows and transform them into output flows. This formalism is well suited to represent circuit, network,... The communication are handled by variable sharing.
The fact that the language is synchronous is an hypothesis that the computation in the system reacts instantaneously to the environment. It obviously cannot be true in practice, but if the computation is fast enough, then the system can react to an event from the environment before another one occurs.

This model was chosen for several reasons. Firstly, it is used in many safety critical domains, such as aircraft software, nuclear power plant commands or embedded systems in general. Those are domains that would greatly benefit from causality analysis, as it can help pinpointing the causes for a failure, using the logs. Another reason is that it is intuitive, and therefore easy to understand. This makes the definitions and examples simpler. Another advantage of this model is the synchronous communications (i.e. the communication are synchronised on a global clock for the whole systems). This removes several problems that come with asynchronous communication, such as interleaving. It means that we can build interesting causality definitions and framework that can be expanded afterwards to asynchronous communication models.

However, great care will be taken, in this manuscript, at making the definitions as independent as possible from the model. The attention of the reader will be drawn to the definitions that are directly dependent from the model. Nonetheless, the fact that the two causality frameworks developed in section 3.2 are adapted from other models ([Gössler and Le Métayer, 2013, Gössler and Métayer, 2015]) shows a good independence from the model.

Definition 1 (Flow f) *A flow f is a tuple $(name_f, D_f)$, with $name_f$ the name of the flow and D_f its domain.*

A flow can be seen as a typed variable, with $name_f$ its name and D_f its type. Note that a flow can be shared by different components.

Definition 2 (Possible behaviours $\mathcal{B}$) *Let $f = (name_f, D_f)$ be a flow, the possible behaviours on this flow are $\mathcal{B}_f = D_f^\infty$.*

By notation abuse, we note $\mathcal{B}_{\mathbb{F}} = \left(\bigtimes_{f \in \mathbb{F}} D_f\right)^\infty$, with $\mathbb{F}$ a set of flows.

A trace over a flow is a sequence of value from domain of the flow ($\underline{tr}_f \in \mathcal{B}_f$).

Given a set of flows $\mathbb{F}$, an element $\underline{tr}$ of $\mathcal{B}_{\mathbb{F}}$ is called a trace over $\mathbb{F}$. A trace over $\mathbb{F}$ is one of all the possible sequences over $\mathbb{F}$

Example 1 *For instance, here is a trace $\underline{tr}$ over a Boolean flow f:*

Time	*0*	*1*	*2*	*3*
Value	*true*	*true*	*false*	*true*

Which can also noted $\underline{tr} = (true, true, false, true)$

Definition 3 (Prefix $\sqsubseteq$) *Let $\underline{tr}, \underline{tr}' \in \mathcal{B}$ be two traces. $\underline{tr}$ prefixes $\underline{tr}'$, noted $\underline{tr} \sqsubseteq \underline{tr}'$, if $|\underline{tr}| \leqslant |\underline{tr}'| \wedge \underline{tr} = \underline{tr}'[0..|\underline{tr}| - 1]$.*

We note $\sqsubset$ for strict prefixes. $\underline{tr} \sqsubset \underline{tr}' = \underline{tr} \sqsubseteq \underline{tr}' \wedge \underline{tr} \neq \underline{tr}'$.

Let $\mathbb{F}$ be a set of flows and $\mathbb{TR} \subseteq \mathcal{B}_{\mathbb{F}}$ be a set of traces, $\mathbb{TR}$ is *prefix-closed* if $\forall \underline{tr} \in \mathbb{TR}, \{\underline{tr}' \in \mathcal{B}_{\mathbb{F}} \mid \underline{tr}' \sqsubseteq \underline{tr}\} \subseteq \mathbb{TR}$.

Let $\mathbb{F}$ be a set of flows and $\mathbb{TR} \subseteq \mathcal{B}_{\mathbb{F}}$ be a set of traces, its supremum is defined as follow: $sup(\mathbb{TR}) = \{\underline{tr} \in \mathbb{TR} \mid \forall \underline{tr}' \in \mathbb{TR}, \underline{tr} \not\sqsubset \underline{tr}'\}$. Similarly, its infimum is $min(\mathbb{TR}) = \{\underline{tr} \in \mathbb{TR} \mid \forall \underline{tr}' \in \mathbb{TR}, \underline{tr}' \not\sqsubset \underline{tr}\}$.

Definition 4 (Projection π) *Let $f \in \mathbb{F}$ be a flow and $\underline{tr} = (tr_g)_{g \in \mathbb{F}}$ be a trace such that $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$, the projection $\underline{tr}$ over f is $\pi_f(\underline{tr}) = tr_f$.*

We also define a projection on a set of flows as follow: let $\mathbb{E} \subseteq \mathbb{F}$ be a set of flows and $\underline{tr} = (tr_g)_{g \in \mathbb{F}}$ be a trace over $\mathbb{F}$. The projection of $\underline{tr}$ over $\mathbb{E}$ is $\pi_{\mathbb{E}}(\underline{tr}) = (tr_g)_{g \in \mathbb{E}}$.

Definition 5 (Upward projection $\pi^\uparrow$) *Let $\mathbb{E}$ and $\mathbb{F}$ be two sets of flows such that $\mathbb{E} \subseteq \mathbb{F}$, and $\underline{tr} \in \mathcal{B}_{\mathbb{E}}$ a trace over $\mathbb{E}$. The upward projection of $\underline{tr}$ over $\mathbb{F}$ is $\pi^\uparrow_{\mathbb{F}}(\underline{tr}) = \{\underline{tr}' \in \mathcal{B}_{\mathbb{F}} \mid \pi_{\mathbb{E}}(\underline{tr}') = \underline{tr}\}$*

The upward projection consist in projecting on a "bigger" set of flows $\mathbb{F}$ than the one of the trace ($\mathbb{E}$). It raises a set of traces that match the initial traces on $\mathbb{E}$.

By extension, let $\mathbb{TR}$ be a set of traces over $\mathbb{E}$, $\pi^\uparrow_{\mathbb{F}}(\mathbb{TR}) = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \exists \underline{tr}' \in \mathbb{TR}, \pi_{\mathbb{E}}(\underline{tr}) = \underline{tr}'\}$

Definition 6 (Receptive) *Let $\mathbb{E}$ and $\mathbb{F}$ be two set of flows, such that $\mathbb{E} \subseteq \mathbb{F}$ and $\mathcal{S} \subseteq \mathcal{B}_{\mathbb{F}}$. $\mathcal{S}$ is receptive, with respect to $\mathbb{E}$ if $\forall \underline{tr} \in \mathcal{B}_{\mathbb{E}}, \exists \underline{tr'} \in \mathcal{S}, \pi_{\mathbb{E}}(\underline{tr'}) = \underline{tr}$.*

Definition 7 (Component C) *A component C is a tuple $(I_C, O_C, \mathcal{S}_C)$ with:*

- *I_C the set of its input flows,*
- *O_C the set of its output flows,*
- *$\mathcal{S}_C$ its specification. $\mathcal{S}_C$ is such that $\mathcal{S}_C \subseteq \mathcal{B}_{I_c \cup O_c}$, $\mathcal{S}_C$ is prefix-closed and receptive with respect to I_C*

The specification must be prefix-closed because the system evolves step by step, and if it was not prefix-closed, there would be possible behaviours that would be allowed, but not reachable in a step-wise manner. E.g. if $(1,1) \in \mathcal{S}_C$, but $(1) \notin \mathcal{S}_C$, it is not possible to reach $(1,1)$, since the component cannot do (1) in the first time-step.
The specification must be receptive so that the component cannot be blocked by the reception of inputs. This hypothesis alongside the prefix-closed one implies that any trace generated by the component, that respects the specification, can be continued, whatever are the inputs.

Note that the receptive hypothesis does not imply that the component must be able to perform meaningful computation for any inputs. It can have a fail state if the inputs it is given do not respect certain criteria. For instance, a square root component could output *error* if it is given a negative input.

The component formalisation here is non-deterministic, which is more general than the data flow synchronous languages. It can be easily changed to be deterministic, by having the specification be a complete prefix-closed function from $\mathcal{B}_{I_c}$ to $\mathcal{B}_{O_c}$. However, it does not impact the approach, so for the sake of generality, non-deterministic specifications are accepted.

Let C be a component, by notation abuse, we write π_C for $\pi_{I_C \cup O_C}$ and similarly everywhere the subscript is supposed to be a set of flows.

Property 1 *Let C be a component. $\forall \underline{tr} \in \mathcal{B}_C, \underline{tr} \notin \mathcal{S}_C \implies \forall \underline{tr'} \in \mathcal{B}_C, \underline{tr} \sqsubseteq \underline{tr'} \implies \underline{tr'} \notin \mathcal{S}_C$*

This property is a direct consequence of the prefix-closed hypothesis on $\mathcal{S}_C$.

This property means that once a component is faulty, it remains faulty. It means that the specifications behave as a safety property.

Example 2 *An example of component would be a pump, that takes in input a command com_{pump}, in the form of a natural, and outputs a volume of liquid out_{pump}, which is the minimum between the command, and the maximum output* 10*. We then have:*

- $I_{pump} = (com_{pump}, \mathbb{N})$
- $O_{pump} = (out_{pump}, [0..10])$
- $\mathcal{S}_{pump} = \{\underline{tr} \in \mathcal{B}_{pump} \mid \forall i \in [0..|\underline{tr}|-1], \pi_{out_{pump}}(\underline{tr}[i]) = \min(\pi_{com_{pump}}(\underline{tr}[i]), 10)\}$

The specification is clearly both prefix-closed and receptive with respect to I_C.

Definition 8 (System S) *A system S is a tuple $(\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ with:*

- $\mathbb{C}$ *is the set of components of the system.*
- $\mathbb{F}$ *is the set of flows of the system.* $\mathbb{F} = \bigcup_{C \in \mathbb{C}} (I_C \cup O_C)$ *such that,* $\forall f \in \mathbb{F}, \exists C, C' \in \mathbb{C}, (f \in O_C \wedge f \in O'_C) \implies C = C'$ *and* $\forall f, f' \in setF, name_f = name_{f'} \implies f = f'$.
- $\mathcal{P}$ *is a safety property on the system. It is such that* $\mathcal{P} \subseteq \mathcal{B}_{\mathbb{F}}$.
- BM *is the behavioural model of the system.* $BM \subseteq \mathcal{B}_{\mathbb{F}}$ *prefix-closed.*

The hypothesis on the flows, in the second item, means that a flow is, at most, the output of one component. Each flow must have a unique name, hence ensuring that no different flows can be equal.
$\mathcal{P}$ is a safety property, i.e. if $\underline{tr} \notin \mathcal{P}$, then $\forall \underline{tr'} \in \mathcal{B}_{\mathbb{F}}, \underline{tr} \sqsubseteq \underline{tr'} \implies \underline{tr'} \notin \mathcal{P}$. It means that if the system failed, it will remain in this failing state as the system evolves.
The behavioural model represents all the possible behaviours that the system can achieve. It reflects the physical, hardware, software... limitations of the system. Note that contrary to the specification, there is no receptive hypothesis, as the some system inputs might not be possible to achieve.

Example 3 *Here is an example of a system.*

The components are the pump presented in example 2 and a tank.
The tank takes out_{pump} in input and outputs the volume it currently contains, which is the sum of the volume from the pump and the volume the tank contained on the previous instant. The tank is initially empty. We then have

- $I_{tank} = \{(out_{pump}, [0..10])\}$

- $O_{tank} = \{(vol_{tank}, \mathbb{N})\}$
- $\mathcal{S}_{tank} = \{\underline{tr} \in \mathcal{B}_{tank} \mid \underline{tr} \neq () \implies \pi_{vol_{tank}}[0] = \pi_{out_{pump}}[0]) \wedge |\underline{tr}| > 0 \implies (\forall i \in [1..|\underline{tr}| - 1], \pi_{vol_{tank}}[i] = \pi_{vol_{tank}}[i-1] + \pi_{out_{pump}}[i])\}$

$S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, *with:*

- $\mathbb{C} = \{pump, tank\}$
- $\mathbb{F} = \{(com_{pump}, \mathbb{N}), (out_{pump}, [0..10]), (vol_{tank}, \mathbb{N})\}$
- $\mathcal{P} = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}| - 1], \pi_{vol_{tank}}(\underline{tr}[i]) \leqslant 5\}$
- $BM = \mathcal{B}_{\mathbb{F}}$

The hypothesis on $\mathbb{F}$ is verified, since the out_{pump} is part of two components, but is only an output of pump.
$\mathcal{P}$ means that the tank should not overflow, if we suppose it has a capacity of 5. Here, there is no restriction on the behavioural model.

3.2 Causality Analysis definitions

As explained in the previous section, the approach developed in this thesis, uses a counter-factual approach. The approach takes a system, a specification for each component, a system property, a faulty system trace, and a set of suspected components as inputs. The faults from the suspected components are removed from the faulty system trace. The set of all possible traces prolonging the resulting trace is build, and called the counter-factuals. In those counter-factuals, the suspected components should behave according to their specification and no non-faulty component should become faulty. A causality definition is then checked on those counter-factuals.

Example 4
Let us illustrate a very naive way of applying this approach to an example. The system we consider is the one described in example 3, composed of a pump and a tank, with $\mathcal{P}$ being that the tank should not overflow, i.e. $vol_{tank} \leqslant 5$ at all time.

The correct scenario is the following, the tank is initially empty ($vol_{tank} = 0$), the command asks the pump to deliver 2 during 2 time-frame, and then to stop. In the end the tank should have $vol_{tank} = 4$, which does not violate the property.

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	*0*
vol_{tank}	*0*	*2*	*4*	*4*

The faulty scenario is the same, except the pump does not stop at the instant 3 *and outputs* 2*. The tank then overflows, as it volume become* 6*. The component faults are underlined, the system property violation are in red. This convention will be used throughout this manuscript.*

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	$\underline{2}$
vol_{tank}	*0*	*2*	*4*	*6*

The set of suspected components is $\{pump\}$*, since it is the only one that is faulty. We start by removing the fault from this component, which gives us the following trace:*

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	
vol_{tank}	*0*	*2*	*4*	*6*

This trace is problematic, as we cannot build a consistent trace where the pump behaves accordingly to its specification as out_{pump} *should be* 0 *at time* 3*, but it would make the tank faulty, as* vol_{tank} *should then be* 4*. Therefore, we need a way of assessing the impact of the faults from the suspected components on the rest of the system. What our intuition tells us here is that if we remove the faulty value for the pump, we must also remove the resulting* 6 *for* vol_{tank}*, since the value of* vol_{tank} *depends on* out_{pump}*, leading to the following trace:*

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	
vol_{tank}	*0*	*2*	*4*	

From this pruned trace, we can build the correct scenario. By fixing the behaviour of the pump, the failure does not happen in the counter-factuals, then $\{pump\}$ *is a necessary cause (it is necessary for pump to be faulty for the failure to happen). Other causality definition will be presented later in this section.*

This section will be divided in three subsections. The first one will present the general principle of causality analysis. The next ones will present two approaches that instantiate this general principle. The first one is adapted from [Gössler and Le Métayer, 2013]. It relies on the construction of a cone of influence that over-approximates the impact of the component faults on the rest of the system. This approach is both easy to understand and apply, but sometimes results in causality analysis that does not match the intuition. The second approach is adapted from [Gössler and Métayer, 2015], which enhances [Gössler and Le Métayer, 2013], by being less pessimistic on the impact of the faults on the rest of the system, resulting in more intuitive results for causality analysis.

3.2.1 General principle of Causality Analysis

As explained previously, the idea of the causality analysis approach is to build counter-factuals from a failing trace, using the specification of the system to complete the part of the trace that have been removed. We assume we are given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. The counter-factuals will be noted CF. CF takes a trace $\underline{tr}$ and a set of suspected components $\mathcal{I}$ as arguments. Here are some properties over the counter-factuals that hold for every instanciation of the causality analysis framework.

Requirement 1 (Behaviour enforcement) $CF \subseteq BM$.

Trivially, the counter-factuals should be possible behaviours of the system.

Requirement 2 (No fault introduction) $\forall C \in \mathbb{C}, \pi_C(\underline{tr}) \in \mathcal{S}_C \implies \pi_C(CF(\underline{tr}, \mathcal{I})) \in \mathcal{S}_C$.

The CF cannot "break" components which were behaving according to their specification. It means that the construction of the counter-factual does not introduce any fault. Note that the hypothesis that the specifications are receptive is important here, so that we can always prolong traces, without violating the specifications.

Requirement 3 (Suspect repair) $\forall C \in \mathbb{C}, C \in \mathcal{I} \implies \forall \underline{tr'} \in CF(\underline{tr}, \mathcal{I}), \pi_C(\underline{tr'}) \in \mathcal{S}_C$.

The counter-factuals should respect the specification of the components in $\mathcal{I}$. It means that the counter-factuals remove the faulty behaviours from the suspected components. Removing the faults of the suspected components is the core idea of the approach. This requirement enforces this idea.

Requirement 4 (No impact from non-faulty components) $\forall(\mathcal{I}, \mathcal{I}') \in \mathbb{C}, \mathcal{I} \subseteq \mathcal{I}' \wedge (\forall C \in \mathcal{I}' \setminus \mathcal{I}, \pi_C(\underline{tr}) \in \mathcal{S}_C) \implies CF(\underline{tr}, \mathcal{I}) = CF(\underline{tr}, \mathcal{I}')$.

This means that components that respect their specification do not impact the CF. This is important, as we only want to blame components that are faulty.

Requirement 5 (Conservation of the non-impacted trace parts)

$$\forall C \in (\mathbb{C} \setminus (\mathcal{I}), C \textit{ not "impacted" by the faults from } \mathcal{I} \implies \pi_C(CF(\underline{tr}, \mathcal{I})[0..|\underline{tr}| - 1]) = \pi_C(\underline{tr})$$

Note that this definition is not formal, since natural language is used in it. It is due to the fact that the way of assessing the impact of the faults on the rest of the system is the difference between the instanciations of causality analysis. This hypothesis is central in making this approach counter-factual, as it means that we want to stay as close as possible to the initial trace, by not modifying its "non-impacted" parts.

Once the construction of the counter-factual has been defined, we can build different definitions of causality. For all the remaining definitions in this subsection, we assume we are given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, a failing trace $\underline{tr}$ such that $\underline{tr} \in BM \setminus \mathcal{P}$, and a set of suspected components $\mathcal{I} \in \mathbb{C}$.

Definition 9 (Weak Necessary Causality (nec_{weak})) *$\mathcal{I}$ is a weak necessary cause for the failure in $\underline{tr}$, noted $\text{nec}_{weak}(\underline{tr}, \mathcal{I})$ if $CF(\underline{tr}, \mathcal{I}) \cap \mathcal{P} \neq \emptyset$.*

A set suspected component $\mathcal{I}$ is a weak necessary cause of the failure in $\underline{tr}$ if some traces in the counter-factuals, where the component in $\mathcal{I}$ have been fixed, respect the property. It means that removing the faults from the components in $\mathcal{I}$ sometimes makes it possible to avoid the failure.

Definition 10 (Necessary Causality (nec)) *$\mathcal{I}$ is a necessary cause for the failure in $\underline{tr}$, noted $\text{nec}(\underline{tr}, \mathcal{I})$ if $CF(\underline{tr}, \mathcal{I}) \subseteq \mathcal{P}$.*

A set of suspected components $\mathcal{I}$ is a necessary cause of the failure in $\underline{tr}$ if all the traces in the counter-factuals, where the component in $\mathcal{I}$ have been fixed, respect the property. It means that removing the faults from the components in $\mathcal{I}$ always avoids the failure. It is a necessary cause, as it is necessary that those components' faults are there, in order for the failure to happen.
This notion of causality is stronger that the weak one $(CF(\underline{tr}, \mathcal{I}) \subseteq \mathcal{P}) \implies CF(\underline{tr}, \mathcal{I}) \cap \mathcal{P} \neq \emptyset$, therefore, $\text{nec}(\underline{tr}, \mathcal{I}) \implies \text{nec}_{weak}(\underline{tr}, \mathcal{I})$.

Definition 11 (Sufficient Weak Cause (suff_{weak})) *$\mathcal{I}$ is a weak sufficient cause for the failure in $\underline{tr}$, noted* $\text{suff}(\underline{tr}, \mathcal{I})$*, if* $sup(CF(\underline{tr}, \mathbb{C} \setminus \mathcal{I})) \not\subseteq \mathcal{P}$

A set of suspected components $\mathcal{I}$ is weak sufficient cause of the failure in $\underline{tr}$ if some finished traces in the counter-factuals, where the components not in $\mathcal{I}$ have been fixed, do not respect the property. It means that repairing all the components, but those in $\mathcal{I}$, does not always fix the system. Note that the *sup* here is important, as the prefixes of the longest traces may be non-failing (for instance, the empty trace is non-failing). We want to check that their are some traces for which the system eventually fails.

Definition 12 (Sufficient Cause (suff)) *$\mathcal{I}$ is a sufficient cause for the failure in $\underline{tr}$, noted* $\text{suff}(\underline{tr}, \mathcal{I})$*, if* $sup(CF(\underline{tr}, \mathbb{C} \setminus \mathcal{I})) \cap \mathcal{P} = \emptyset$

A set of suspected components $\mathcal{I}$ is sufficient cause of the failure in $\underline{tr}$ if all the traces in the counter-factuals, where the components not in $\mathcal{I}$ have been fixed, do not respect the property. It means that repairing all the components, but those in $\mathcal{I}$, does not fix the system. We check that for all traces, their is eventually a system failure. It is a sufficient cause, since the faults of the components in $\mathcal{I}$ are enough to make the system fail.
This notion of causality is stronger that the weak one ($(CF(\underline{tr}, \mathbb{C} \setminus \mathcal{I}) \cap \mathcal{P} = \emptyset) \implies CF(\underline{tr}, \mathbb{C} \setminus \mathcal{I}) \not\subseteq \mathcal{P}$, therefore, $\text{suff}(\underline{tr}, \mathcal{I}) \implies \text{suff}_{weak}(\underline{tr}, \mathcal{I})$.

Example 5 *If the take the example 4, presented earlier in this section, with the pump and the tank. The failing trace is:*

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	*2*
vol_{tank}	*0*	*2*	*4*	*6*

As we saw previously, $\mathcal{I} = \{pump\}$ fixes the system, i.e. $CF(\underline{tr}, \{pump\}) \in \mathcal{P}$. Then, $\{pump\}$ is a necessary cause.

Given the "non-impact from non-faulty components" (requirement 4), $CF(\underline{tr}, \{pump, tank\}$ $CF(\underline{tr}, \{pump\})$, since $\pi_{tank}(\underline{tr}) \in \mathcal{S}_{tank}$. Then $\{pump, tank\}$ is also a necessary cause. Note that we used a requirement to derive that $\{pump, tank\}$ was a necessary cause, but in the general, a superset of a necessary causes is not always a necessary cause itself.

From requirement 4, it also follows that $CF(\underline{tr}, \{tank\}) = CF(\underline{tr}, \emptyset)$. Given "the conservation of the non-impacted trace parts" (requirement 5), $CF(\underline{tr}, \emptyset) = \{\underline{tr}\}$. Since no component has been repaired, the trace has not been impacted, and thus remains the same. We then have $\text{suff}(\underline{tr}, \{pump, tank\})$ *and* $\text{suff}(\underline{tr}, \{pump\})$*, since $CF(\underline{tr}, \mathbb{C} \setminus \{pump, tank\}) = CF(\underline{tr}, \emptyset) = \{\underline{tr}\}$, $CF(\underline{tr}, \mathbb{C} \setminus \{pump\}) = CF(\underline{tr}, \{tank\}) = \{\underline{tr}\}$ and $\underline{tr} \notin \mathcal{P}$.*

Property 2 *Given a system* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. *Let* $\underline{tr} \in BM \setminus \mathcal{P}$ *be a faulty trace, and* $\mathcal{I} \subseteq \mathbb{C}$. $\text{nec}(\underline{tr}, \mathcal{I}) \implies \neg\,\text{suff}(\underline{tr}, \mathbb{C} \setminus \mathcal{I})$ *and* $\text{suff}(\underline{tr}, \mathcal{I}) \implies \neg\,\text{nec}(\underline{tr}, \mathbb{C} \setminus \mathcal{I})$.

The proof is derived from the definitions of causality. $\text{nec}(\underline{tr}, \mathcal{I}) = CF(\underline{tr}, \mathcal{I}) \in \mathcal{P}$ and $\text{suff}(\underline{tr}, \mathbb{C} \setminus \mathcal{I}) = CF(\underline{tr}, \mathbb{C} \setminus (\mathbb{C} \setminus \mathcal{I})) \cap \mathcal{P} = \emptyset) = (CF(\underline{tr}, \mathcal{I}) \cap \mathcal{P} = \emptyset)$. Since $CF(\underline{tr}, \mathcal{I}) \in \mathcal{P} \implies \neg(CF(\underline{tr}, \mathcal{I}) \cap \mathcal{P} = \emptyset)$, and thus the first part of the property. The second one is proved similarly.

This property means that if $\mathcal{I}$ is a necessary (resp. sufficient) cause, the complementary of $\mathcal{I}$ is not a sufficient (resp. necessary) cause.

Note that all the definitions and properties from this subsection are independent from the model chosen. They only rely on the fact that the system is made of components ($\mathbb{C}$) with a specification ($\mathcal{S}_C$), that there is a system property ($\mathcal{P}$), that the system has a set of possible behaviours (BM), and the ability to project a system trace on a component (π_C). As long as $\mathcal{S}_C$, $\mathcal{P}$ and BM are sets of traces, the definitions hold. All those elements should be provided by the chosen model, which is usually true.

Definition 13 (Well-designed system) *Let* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ *be a system.* S *is well-designed if* $\bigcap_{C \in \mathbb{C}} \pi^{\uparrow}_{\mathbb{F}}(\mathcal{S}_C) \subseteq \mathcal{P}$.

A well-designed system is a system where the composition of the specification refines the system property. Here, the composition is simple, since the communications are done by variable sharing, but a constraint representing the communication might be necessary for other models. However, this definition can be generalised, by requesting a composition definition alongside the system model.

Theorem 1 (Soundness) *If the system is well-designed, each cause contains a component that is faulty.*

Proof 1 *Let* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ *be a system, and* $\underline{tr}$ *a failing trace over this system.*
Let $\mathcal{I}$ *be a necessary cause, such that* $\forall C \in \mathcal{I}, \pi_C(\underline{tr}) \in \mathcal{S}_C$. *By requirement 4 (no impact from non-faulty components),* $CF(\underline{tr}, \mathcal{I}) = CF(\underline{tr}, \emptyset)$. *By requirement 5 (conservation of the non-impacted trace parts),* $CF(\underline{tr}, \emptyset) = \{\underline{tr}\}$. *Thus,* $CF(\underline{tr}, \mathcal{I}) \not\subseteq \mathcal{P}$, *which is a contradiction, since* $\mathcal{I}$ *is a necessary cause.*
Let $\mathcal{I}$ *be a sufficient cause, such that* $\forall C \in \mathcal{I}, \pi_C(\underline{tr}) \in \mathcal{S}_C$. *It means that* $\mathcal{I}' = \mathbb{C} \setminus \mathcal{I}$ *contains all faulty components. Thus, by requirement 3 (suspect repair),* $\forall C \in \mathcal{I}', \forall \underline{tr}' \in CF(\underline{tr}, \mathcal{I}'), \pi_C(\underline{tr}') \in \mathcal{S}_C$. *By requirement 2 (no fault introduction),* $\forall C \in \mathcal{I}, \forall \underline{tr}' \in CF(\underline{tr}, \mathcal{I}'), \pi_C(\underline{tr}') \in \mathcal{S}_C$. *Then,*

$\forall C \in \mathbb{C}, \forall \underline{tr'} \in CF(\underline{tr}, \mathcal{I}'), \pi_C(\underline{tr'}) \in \mathcal{S}_C$. *Hence, since the system is well-designed,* $CF(\underline{tr}, \mathcal{I}') \subseteq \mathcal{P}$, *which is a contradiction. Since* $sup(CF(\underline{tr}, \mathcal{I}')) \subseteq CF(\underline{tr}, \mathcal{I}')$, $sup(CF(\underline{tr}, \mathcal{I}')) \in \mathcal{P}$, *which is a contradiction.*

Note that the proof of soundness for the necessary cause does not use the well-designed hypothesis, and is then a general result.

Theorem 2 (Completeness) *If the system is well designed, each violation of* $\mathcal{P}$ *has at least one necessary and at least one sufficient cause.*

Proof 2 *Let* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ *be a system, and* $\underline{tr}$ *a failing trace over this system.*
Let $\mathcal{I}$ *be* $\mathbb{C}$. *By requirement 5,* $CF(\underline{tr}, \mathbb{C} \setminus \mathbb{C}) = CF(\underline{tr}, \emptyset) = \{\underline{tr}\}$. $\{\underline{tr}\} \cap \mathcal{P} = \emptyset$, *hence* $\mathcal{I}$ *is a sufficient cause to the failure in* $\underline{tr}$.
Let $\mathcal{I}$ *be* $\mathbb{C}$. *With the same reasoning as for the sufficient cause soundness proof,* $\forall C \in \mathbb{C}, \forall \underline{tr'} \in CF(\underline{tr}, \mathbb{C}), \pi_C(\underline{tr'}) \in \mathcal{S}_C$. *Given the well-designed hypothesis,* $CF(\underline{tr}, \mathbb{C}) \subseteq \mathcal{P}$, *and* $\mathcal{I}$ *is a necessary cause.*

Symmetrically to the soundness proof, the sufficient cause completeness proof does not rely on the well-designed hypothesis, and is thus a general result.

3.2.2 Cone of influence approach

This subsection presents an instanciation of causality analysis adapted from [Gössler and Le Métayer, 2013]. The idea is to build a cone of influence spanning from the first fault of each suspected component, and adding to the cone each component after it received data from a component in the cone. The parts which are not in the cone are removed, and the counter-factuals are built from this pruned trace. The cone is a coarse over-approximation of the impact of the faults of the suspected components on the system, but is both intuitive and easy to build.

Definition 14 (Cone of influence cone**)** *Given a system* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. *Let* $\underline{tr} \in BM \setminus \mathcal{P}$ *be a faulty trace, and* $\mathcal{I} \subseteq \mathbb{C}$. $\text{cone}(\underline{tr}, \mathcal{P}) = (\text{cone}_C)_{C \in \mathbb{C}}$ *is the vector of maximal indexes such that,* $\forall C \in \mathbb{C}$*:*

$$C \in \mathcal{I} \implies \text{cone}_C \leqslant fv_C(\underline{tr}) \wedge \tag{3.1}$$

$$\forall f \in I_C, \forall C' \in \mathbb{C}, \text{cone}_{C'} \leqslant fv_C(\underline{tr}) \wedge f \in O_{C'} \implies \text{cone}_C \leqslant \text{cone}_{C'} \tag{3.2}$$

With $fv_C(\underline{tr}) = min(\{i \in [0..|\underline{tr}| - 1] | \pi_C(\underline{tr}[0..i]) \notin \mathcal{S}_C\} \cup \{|\underline{tr}|\}$

$fv_C(\underline{tr})$ returns the index of the first violation of component C specification in $\underline{tr}$, or $|\underline{tr}|$ if $\mathcal{S}_C$ is not violated in $\underline{tr}$.
The constraint 3.1 makes sure that the suspected components enter the cone at the latest when they violate their specification. It means that the cone necessarily contains every violation of $\mathcal{S}_C$, if C is a suspected component.
Constraint 3.2 means that a component that is not faulty at a given time-step, and receives data from a component inside the cone must be in the cone as well.
The maximality of the indexes ensures that we make the component enter the cone as late as possible.

This definition is not independent from the model chosen. However it can easily be generalised by replacing constraint 3.2 as follow:

$$\forall f \in I_C, \forall C' \in \mathbb{C},$$
$$comm(C', C, l_{c'}) \leqslant fv_C(\underline{tr}) \wedge f \in O_C \implies l_C \leqslant comm(C', C, l_{c'})$$

with $comm(C', C, l_{c'})$ the minimum index such that C receives data from C' sent at or after the instant $l_{C'}$, or $|\underline{tr}|$ if no communication occurs after $l_{C'}$. Since the communications are synchronous and done by sharing a flow, the first instant a component C receives data from a component C' at instant t or after is either t if one of the output flow of C' is an input flow of C, or $|\underline{tr}|$. This gives us a definition equivalent to constraint 3.2.

Definition 15 (Cone Counter-factuals CF_{cone}) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace, and $\mathcal{I} \subseteq \mathbb{C}$. Let $\text{cone}(\underline{tr}, \mathcal{P}) = (\text{cone}_C)_{C \in \mathbb{C}}$. The counter-factuals are defined as follow:*

$$CF_{\text{cone}}(\underline{tr}, \mathcal{I}) = \{\underline{tr}' \in BM \mid \forall C \in \mathbb{C},$$
$$\pi_C(\underline{tr}[0..\,\text{cone}_C - 1]) = \pi_C(\underline{tr}[0..\,\text{cone}_C - 1]) \wedge \tag{3.3}$$
$$\pi_C(\underline{tr}[0..\,\text{cone}_C - 1]) \in \mathcal{S}_C \implies \pi_C(\underline{tr}') \in \mathcal{S}_C\} \tag{3.4}$$

Constraint 3.3 makes sure that the projection of the counter-factual traces of each component begins with the prefix of the initial trace not in the cone, i.e. the non-impacted parts of the trace are kept as they were in the initial trace.
Constraint 3.4 ensures that a component that respects its specification in the pre-cone portion of the trace, does so on all the counter-factual traces.

Example 6 *We apply the counter-factuals construction using cone to example 4. We still have a system composed of a pump and a tank, with a faulty trace as follow:*

Time	*0*	*1*	*2*	*3*
com_{pump}	*0*	*2*	*2*	*0*
out_{pump}	*0*	*2*	*2*	$\underline{2}$
vol_{tank}	*0*	*2*	*4*	*6*

$\text{cone}(\underline{tr}, \{pump\}) = (3, 3)$. *At time-step* 3, *pump enters the cone, as it does not respect its specification. Since tank receives data from pump, it enters the cone as soon as pump does, that is at time-step* 3. *If we build the counter-factuals from this cone, we get the correct scenario, as expected.* $\{pump\}$ *then is a necessary cause.*

$\text{cone}(\underline{tr}, \{tank\}) = (4, 4)$. *tank never enters the cone. Since a non-suspected component can enter the cone only if it receives data from a component in the cone, pump never enters the cone.* $CF(\underline{tr}, \{tank\}) = \{\underline{tr}\}$ *and* $\{tank\}$ *is not a necessary cause, as expected.*

$\text{cone}(\underline{tr}, \{pump, tank\}) = \text{cone}(\underline{tr}, \{pump\}) = (3, 3)$. *Since pump never enters the cone by being faulty, it enters the cone when receiving potentially "corrupted" data from the faulty pump. This illustrates the "No impact from non-faulty components requirement" (4).*

This definition of counter-factuals does respect the requirements we defined from the counter-factuals:

Behaviour enforcement (requirement 1): Trivially, $CF \subseteq BM$.

No fault introduction (requirement 2): A component is either non-faulty in the prefix to the cone, or does not enter the cone (Constraints 3.1 and 3.2). Therefore, the counter-factuals only build trace where the component trace is left as it was, or a non-faulty part of the trace is prolonged with a non-faulty behaviour (constraint 3.4). Thus no fault is introduced.

Suspect repair (requirement 3): Constraint 3.1 ensures that the prefix of the initial trace for the suspected components trace is fault free. Constraint 3.4 ensures that the prolongation remains in the specification of the components. Accordingly, the suspected components are repaired in the counter-factuals.

No impact from non-faulty components (requirement 4): Constraint 3.1 only has an impact if the suspected component is faulty, as $fv_C(\underline{tr}) = |\underline{tr}|$ if the component is non-faulty. Therefore $\text{cone}(\underline{tr}, \mathcal{I}) = \text{cone}(\underline{tr}, \mathcal{I} \cup \{C\})$ if C is non-faulty. It follows that $CF\text{cone}(\underline{tr}, \mathcal{I}) = CF\text{cone}(\underline{tr}, \mathcal{I} \cup \{C\})$, which is the requirement.

Conservation of the non-impacted trace part (requirement 5): The cone computes the part of the trace that are considered impacted, and constraint 3.3 ensures that the non-impacted parts of the trace are kept as the initial trace, hence enforcing this requirement.

3.2.3 Unaffected prefix approach

This approach differs from the cone construction by the fact that, after removing the faults from the faulty components, it shortens the traces of each component if there is no consistent global trace for those components. It uses the behavioural model to give keep a longer part of the initial trace, thus being less pessimistic than the cone approach.

Definition 16 (Partial trace) *Let $\mathbb{C}$ be a set of components. $\underline{tr}$ is a partial trace over $\mathcal{B}_{\mathbb{C}}$ if $\exists \underline{tr'} \in \mathcal{B}_{\mathbb{C}}, \forall C \in \mathbb{C}, \pi_C(\underline{tr}) \sqsubseteq \pi_C(\underline{tr'})$.*

A partial trace is a trace that may not have the same length on each component.

We note $\mathcal{B}_{\mathbb{F}}^{partial}$ for the set of all partial traces over the set of flows $\mathbb{F}$

Definition 17 (Critical prefix *cp*) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace, and $\mathcal{I} \subseteq \mathbb{C}$. The critical prefix is a partial trace $cp(\underline{tr}, \mathcal{I}) \in \mathcal{B}_{\mathbb{F}}^{partial}$ such that:*

$$(\forall C \in \mathbb{C} \setminus \mathcal{I}, \pi_C(cp(\underline{tr}, \mathcal{I})) = \pi_C(\underline{tr})) \ \wedge$$

$$\forall C \in \mathcal{I}, (\pi_C(cp(\underline{tr}, \mathcal{I})) \sqsubseteq \pi_C(\underline{tr}) \wedge \pi_C(cp(\underline{tr}, \mathcal{I})) \in \mathcal{S}_C \wedge$$
$$\forall i \in \big[(|\pi_C(cp(\underline{tr}, \mathcal{I}))| + 1)..(|\underline{tr}| - 1)\big] \pi_C(\underline{tr}[0..i]) \notin \mathcal{S}_C$$

The critical prefix is the partial trace for which the trace of the component not suspected are left untouched, and the traces of the suspected component are the longest that respect their specification.

Definition 18 (Trace extension (*extend*)) *Let C be a component, and $\underline{tr}, \underline{tr}^0 \in \mathcal{B}_C$ be traces over that component.*

$$extend_C(\underline{tr}^0, \underline{tr}) \begin{cases} \{\underline{tr'} \in \mathcal{S}_C \mid \underline{tr} \sqsubseteq \underline{tr'}\} & \text{if } \underline{tr} \neq \underline{tr}^0 \wedge \underline{tr} \in \mathcal{S}_C \\ \{\underline{tr}\} & \text{otherwise} \end{cases}$$

The trace extension takes an initial trace $\underline{tr}^0$ and a trace $\underline{tr}$ to extend. If the $\underline{tr}$ is different from the initial trace, and respects the component specification, the trace extension is the set of all traces of the component specification that are prefixed by $\underline{tr}$. Otherwise, it returns $\underline{tr}$.

Definition 19 (Unaffected prefixes UP) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace and $\mathcal{I} \subseteq \mathbb{C}$. We define the unaffected prefixes as follow. $\forall C \in \mathbb{C}$,*

$$\underline{tr}_C^1 = \pi_C(cp(\underline{tr}, \mathcal{I}))$$

And $\forall C \in \mathbb{C}, \forall j > 1$:

$$\underline{tr}_C^{j+1} = \max\{\underline{tr}' \in \mathcal{B}_C \mid \underline{tr}' \sqsubseteq \underline{tr}_i^j \wedge (\exists \underline{tr}'' \in \mathcal{B}^j, \underline{tr}' \sqsubseteq \underline{tr}''\}$$

Where $\mathcal{B}^j = \sup\{\underline{tr}' \in BM \mid \forall C \in \mathbb{C}, \exists \underline{tr}_C \in extend_C(\pi_C(\underline{tr}), \underline{tr}_C^j), \pi_C(\underline{tr}') \sqsubseteq \underline{tr}_C\}$

Let $UP(\underline{tr}, \mathcal{I}) = \underline{tr}' \in \mathcal{B}_{\mathbb{F}}^{partial}$ such that $\forall C \in \mathbb{C}, \pi_C(\underline{tr}') = tr_C^*$, with $tr_C^* = \min_j(tr_C^j)$ be the partial trace that is unaffected by the failures of the components in $\mathcal{I}$.

Intuitively, the unaffected prefixes are built by removing the fault from the suspected components, and then shortening the traces of the different components. For each component, if the trace cannot be extended, the prefixes are shortened. Then the impact of those changes is checked, and the process is repeated if needed. $\mathcal{B}^j$ has a central role in checking whether a trace can be extended or not. It corresponds to all the traces such that, for each component, a trace from $\mathcal{B}^j$ prefixes a possible extension of the trace for this component.
The result is a partial trace that could have been observed (i.e. there are possible traces that are consistent with this partial traces) if the suspected components would not have been faulty. The "suffixes" of this partial trace have been impacted by the faults of the suspected components.

Definition 20 (Cone from unaffected prefixes (cone_{UP})) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace and $\mathcal{I} \subseteq \mathbb{C}$. Let $UP(\underline{tr}, \mathcal{I})$ be the unaffected prefixes. $\underline{\text{cone}}_{UP} = (\text{cone}_C)_{C \in \mathbb{C}}$ is a vector of indexes such that, $\forall C \in \mathbb{C}$,*

$$\text{cone}_C = |\pi_C(UP(\underline{tr}, \mathcal{I}))|$$

As shown in this definition, it is very easy to go from the unaffected prefixes to the cone, and vice-versa. Therefore, we can use the definition 15 of counter-factuals from the cone approach.

The detail of the reasons why this instantiation of the causality framework enforces the requirements will not be given, as the arguments are very similar

to the one given for the cone approach. Indeed, we have the same way of building the counter-factuals and the unaffected prefixes do not contain the faults from the suspected components (corresponding to constraint 3.1) for the cone, neither the parts of the trace that are impacted by those faults (corresponding to constraint 3.2 for the cone). Those are the only things we used to prove that the properties were enforced, and therefore the proofs are similar.

Similarly to the cone approach, the unaffected prefix one is independent from the model. It relies on the same parts of the system as the cone one. It is even "more" independent, as it has no need for a *comm* function, since the communication check is embedded in the use of global traces during the construction of the unaffected prefixes.

3.3 Examples

This subsection presents problematic examples, which will be treated with both the cone and the unaffected prefixes approaches.

Example 7 (Non-suspected faulty component) *This example illustrates the fact that the unaffected prefixes are a tighter over-approximation of the impact of the faults on the rest of the system than the cone one, and that it deals better with non-suspected faulty components*

This example has the same architecture as the running example 4, with the pump and the tank. Here, we restrict the behavioural model, by adding a constraint that reflects the fact that the tank cannot "create water", i.e. $\forall \underline{tr} \in BM, \forall i \in [0..|\underline{tr}| - 1], vol_{tank}[i] \leqslant vol_{tank}[i-1] + out_{pump}[i]$*, with* $vol_{tank}[-1] = 0$*, by convention. This constraint means that volume in the tank is, at most, the previous volume, plus the water pumped into it.*

The scenario is the following failing trace $\underline{tr}$*:*

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	$\underline{2}$	$\underline{2}$
vol_{tank}	*0*	*2*	$\underline{3}$	$\underline{5}$	$\underline{7}$

At the instants 3 *and* 4*, the pump keeps pumping. At instant* 2 *the tank leaks* 1*, postponing the system failure by one instant. Our intuition is that the tank is not a cause, as its fault actually delays the property violation (i.e. the tank overflowing). On the other hand, pump should be a cause, as it should have stopped, hence preventing the overflow. Out of the multiple faults, we want*

to know which ones or which combinations of faults are responsible for the systems failure.

Let us build the counter-factuals with the cone.
If only the pump is suspected, $\text{cone}(\underline{tr}, \{pump\}) = (3, 5)$. *The tank never enters the cone, as it is faulty upon receiving data from the faulty pump. Whatever we do, since the failure is not in the cone, the counter-factuals will necessarily be failing. Hence,* $CF(\underline{tr}, \{pump\}) \notin \mathcal{P}$ *and* $\{pump\}$ *is not a cause, which does not match our intuition.*

If only the tank is suspected, $\text{cone}(\underline{tr}, \{tank\}) = (5, 2)$. *The counter-factuals are then*

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	$\underline{2}$	$\underline{2}$
vol_{tank}	*0*	*2*	*4*	*6*	*8*

The failure actually happens earlier, and $\{tank\}$ *is not a cause, which matches our intuition.*

$\{pump, tank\}$ *is a necessary cause, as all the faults are removed (*$\text{cone}(\underline{tr}, \{pump, tank\}) = (3, 2)$*), we end up the following non-failing scenario:*

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	*0*	*0*
vol_{tank}	*0*	*2*	*4*	*4*	*4*

Now, let us try with the unaffected prefixes.
If only pump is suspected, we have the following unaffected prefixes:

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*		
vol_{tank}	*0*	*2*	$\underline{3}$		

The projection of the trace over vol_{tank} *is shorten, as the only possible value for* $out_{pump}[3]$ *is* 0*, and* $vol_{tank}[3] = 5$ *which is greater than* $vol_{tank}[2] + out_{pump}[3] = 3 + 0 = 3$*. Since this is not a possible trace in* BM*, we remove this part of the trace.*
Then $CF(\underline{tr}, \{pump\})$ *are the traces such that:*

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	*0*	*0*
vol_{tank}	*0*	*2*	$\underline{3}$	$\underline{[0..3]}$	$\underline{[0..vol_{tank}[3]]}$

Here, for concision's sake, some values are actually sets of possible values for the flow, at a given instant. We then represent a set of traces, instead of a unique one.
The values of vol_{tank} always remain lesser than 3, given the constraint in BM. All those traces are non-failing, thus $\{pump\}$ is a necessary cause, as expected.

In a similar fashion to the cone approach, $\{tank\}$ is not a necessary cause, and $\{pump, tank\}$ is.

The unaffected prefixes approach gives us a more intuitive result here because it uses the information given by BM to build the unaffected prefixes. The part of the trace for vol_{tank} at instant 3 and 4 is removed because there is no possible behaviour of the system, where the pump is not faulty that would be consistent with the values of vol_{tank} at instant 3 and 4 in the initial trace. What's more, we do not need an explicit way of knowing when components communicate, contrary to the cone approach, because it is embedded in BM.

This example also illustrates the fact that the causality analysis approach can tackle the problem of faults that do not cause the failure, but delays it. In this example, the result of the analysis matches the intuition, as tank is not a necessary cause, nor a sufficient cause (since $\{pump\} = \mathbb{C} \setminus \{tank\}$ is a necessary cause).

Example 8 (Causal over-determination) *We consider a system similar to the running example 4, where there are two pumps.*

The definition of the system is the following. The set of components is $\{pump_1, pump_2, tank\}$. The two pumps have the same definition as in 4. We just add $_1$ and $_2$ to differentiate them.
The new definition for tank is the following:

- $I_{tank} = \{(out_{pump_1}, [0..10]), (out_{pump_2}, [0..10])\}$
- $O_{tank} = \{(vol_{tank}, \mathbb{N})\}$
- *For simplicity sake, we define $vol_{tank}[-1] = 0$.* $\mathcal{S}_{tank} = \{\underline{tr} \in \mathcal{B}_{tank} \mid \forall i \in [0..|\underline{tr}| - 1], vol_{tank}[i] = vol_{tank}[i-1] + out_{pump_1}[i] + out_{pump_2}[i])\}$

The system definition is $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, with:

- $\mathbb{C} = \{pump_1, pump_2, tank\}$
- $\mathbb{F} = \{com_{pump_1}, out_{pump_1}, com_{pump_2}, out_{pump_2}, vol_{tank}\}$
- $\mathcal{P} = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}|], \pi_{vol_{tank}}(\underline{tr}[i]) \leqslant 5\}$

- $BM = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}| - 1], \pi_{vol_{tank}}(\underline{tr}[i]) \leqslant \pi_{vol_{tank}}(\underline{tr}[i-1]) + \pi_{out_{pump_1}}(\underline{tr}[i]), \pi_{out_{pump_2}}(\underline{tr}[i]))\}$

The specification is similar to the one pump case, the volume inside the tank is the sum of the previous volume, plus the volume pumped by the two pumps.

The scenario is the following failing trace:

Time	*0*	*1*	*2*
com_{pump_1}	*0*	*2*	*0*
out_{pump_1}	*0*	*2*	$\underline{2}$
com_{pump_2}	*0*	*2*	*0*
out_{pump_2}	*0*	*2*	$\underline{2}$
vol_{tank}	*0*	*4*	*8*

If the set of suspected components is $\{pump_1\}$, the counter-factuals are the following (with the cone and the unaffected prefixes):

Time	*0*	*1*	*2*
com_{pump_1}	*0*	*2*	*0*
out_{pump_1}	*0*	*2*	*0*
com_{pump_2}	*0*	*2*	*0*
out_{pump_2}	*0*	*2*	$\underline{2}$
vol_{tank}	*0*	*4*	*6*

The failure still occurs, therefore $\{pump_1\}$ is not a necessary cause, but $\{pump_2\}$ and $\{pump_2, tank\}$ are sufficient causes.

Symmetrically, we have $\{pump_2\}$ is not a necessary cause, but $\{pump_1\}$ and $\{pump_1, tank\}$ are sufficient causes.

If the set of suspected components is $\{pump_1, pump_2\}$, the counter-factuals are the following:

Time	*0*	*1*	*2*
com_{pump_1}	*0*	*2*	*0*
out_{pump_1}	*0*	*2*	*0*
com_{pump_2}	*0*	*2*	*0*
out_{pump_2}	*0*	*2*	*0*
vol_{tank}	*0*	*4*	*4*

Where the failure does not occur. Therefore, $\{pump_1, pump_2\}$ is a necessary cause, and $\{tank\}$ is not a sufficient cause (which is trivial, since tank is not faulty).

This example illustrates the concept of causal over-determination. It means that both $pump_1$ and $pump_2$ fault are enough to make the system fail,

which is reflected by the fact that they are sufficient causes (fixing every component but one of the pumps still leads to a failure). Therefore, in order to fix the system, you need to fix both $pump_1$ *and* $pump_2$*, which is reflected by the fact that* $\{pump_1, pump_2\}$ *is a necessary cause (fixing both pumps prevents the failure from happening).*

Example 9 (Joint causation) *We consider a similar system to the two pumps example 8, where the tank loses two units of liquid per unit of time (there is a fix output flow of liquid from the tank).*

The pumps have the same definition as in 8.
The new definition for tank is the following:

- $I_{tank} = \{(out_{pump_1}, [0..10]), (out_{pump_2}, [0..10])\}$
- $O_{tank} = \{(vol_{tank}, \mathbb{N})\}$
- *For simplicity sake, we define* $vol_{tank}[-1] = 0$. $\mathcal{S}_{tank} = \{\underline{tr} \in \mathcal{B}_{tank} \mid \forall i \in [0..|\underline{tr}|-1], vol_{tank}[i] = max(0, vol_{tank}[i-1] + out_{pump_1}[i] + out_{pump_2}[i]) - 2)\}$

The system definition is $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$*, with:*

- $\mathbb{C} = \{pump_1, pump_2, tank\}$
- $\mathbb{F} = \{com_{pump_1}, out_{pump_1}, com_{pump_2}, out_{pump_2}, vol_{tank}\}$
- $\mathcal{P} = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}|], \pi_{vol_{tank}}(\underline{tr}[i]) \leqslant 5\}$
- $BM = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}|-1], \pi_{vol_{tank}}(\underline{tr}[i]) \leqslant max(0, \pi_{vol_{tank}}(\underline{tr}[i-1]) + \pi_{out_{pump_1}}(\underline{tr}[i]) + \pi_{out_{pump_2}}(\underline{tr}[i])) - 2)\}$

The scenario is following failing trace:

Time	*0*	*1*	*2*	*3*
com_{pump_1}	*0*	*2*	*2*	*0*
out_{pump_1}	*0*	*2*	*2*	*2*
com_{pump_2}	*0*	*2*	*2*	*0*
out_{pump_2}	*0*	*2*	*2*	*2*
vol_{tank}	*0*	*2*	*4*	*6*

At instant 3*, both pumps do not stop, thus overflowing the tank.*

Let us build the counter-factuals (with the same result with both approaches) for $\mathcal{I} = \{pump_1\}$.

Time	*0*	*1*	*2*	*3*
com_{pump_1}	*0*	*2*	*2*	*0*
out_{pump_1}	*0*	*2*	*2*	*0*
com_{pump_2}	*0*	*2*	*2*	*0*
out_{pump_2}	*0*	*2*	*2*	$\underline{2}$
vol_{tank}	*0*	*2*	*4*	*4*

The failure does not occur anymore, so $\{pump_1\}$ *is a necessary cause. Symmetrically,* $\{pump_2\}$ *is a necessary cause.*

Similarly, $\{pump_1, pump_2\}$ *is a necessary cause. And since tank is not faulty,* $CF(\underline{tr}, \{tank\}) = \{\underline{tr}\}$*, and since* $\underline{tr}$ *is faulty,* $\mathbb{C}\backslash\{tank\} = \{pump_1, pump_2\}$ *is a sufficient cause. However,* $\{pump_1\}$ *and* $\{pump_2\}$ *are not sufficient causes, since* $\{pump_2\}$ *and* $\{pump_1\}$ *(and thus* $\{pump_i, tank\}$*) are necessary causes.*

This example illustrates the joint causality, i.e. a set of components that must be faulty together in order to make the system fail. It means that fixing one pump fixes the system, which is showed by the fact that $\{pump_1\}$ *and* $\{pump_2\}$ *are necessary causes. What's more, the combined fault of the two components is enough to make the system fail, which is reflected by the fact that* $\{pump_1, pump_2\}$ *is a sufficient cause. However, nor* $\{pump_1\}$ *or* $\{pump_2\}$ *are sufficient causes, since repairing one pump ensures that the system property is not violated anymore.*

Example 10 (Early preemption) *We go back to the over-determination example (8), but add another constraint to the behavioural model. It defines the behaviour of the faulty pumps as stuck, i.e. they keep outputting the same value once they are faulty. We add the following constraint to the BM:* $\forall \underline{tr} \in BM, \forall i \in [0..|\underline{tr}| - 1], \forall k \in \{1, 2\}, \pi_{out_{pump_k}}(\underline{tr})[O..i] \notin \mathcal{S}_{pump_k} \implies \forall l \in [i..|\underline{tr}| - 1], \pi_{out_{pump_k}}(\underline{tr})[l] = \pi_{out_{pump_k}}(\underline{tr})[i]$

The scenario is the following failing trace:

Time	*0*	*1*	*2*
com_{pump_1}	*0*	*0*	*0*
out_{pump_1}	*0*	*0*	$\underline{4}$
com_{pump_2}	*0*	*0*	*0*
out_{pump_2}	*0*	$\underline{2}$	$\underline{2}$
vol_{tank}	*0*	*2*	*8*

Let us build the counter-factuals for $\mathcal{I} = \{pump_1\}$*:*

$Time$	0	1	2	3
com_{pump_1}	0	0	0	0
out_{pump_1}	0	0	0	0
com_{pump_2}	0	0	0	0
out_{pump_2}	0	$\underline{2}$	$\underline{2}$	$\underline{2}$
vol_{tank}	0	2	4	6

We build counter-factual that are longer than the initial trace. We suppose that the input for each pump will remain 0 *for all instant after instant* 2*, in order to respect the property.*

Here there are two ways to consider the counter-factuals. We either consider infinite traces, or we stop it as soon as the initial trace ends. In the infinite trace, $pump_1$ *is not a cause, as the failure eventually happens. In the case of a length* 3 *trace,* $pump_1$ *is a cause, as the failure does not happen anymore.*

This notion is called early preemption. $pump_2$ *would have eventually caused the failure, but the failure of* $pump_1$ *was so severe that it ended up making the system fail before* $pump_2$ *would have. This notion of early preemption is important in a legal framework, as* $pump_1$ *would generally be considered as responsible, and not* $pump_2$*. However, in a design context, or in a post-mortem analysis, the fact the the failure would eventually have happened is generally more interesting. By choosing if we consider infinite or initial trace size counter-factuals, we can address both problems.*

Example 11 (Non-monotonicity of the causality analysis) *Let us consider a very simple example with three components* C_1*,* C_2*,* C_3*. Each component* C_i *has an output* out_i*, that should always be true. The property is* $\mathcal{P} = (out_1 \Leftrightarrow out_2) \wedge out_3$*. This example has no temporal aspect, so we will just consider one instant traces as vectors* (out_1, out_2, out_3)*.*

The failing scenario is that all components output false. Then $\mathcal{P} = false$*.*

Let us consider $\mathcal{I} = \{C_3\}$*, then the counter-factual is* $(false, false, true)$*,* $\mathcal{P} = (false \Leftrightarrow false) \wedge true = true$*. Therefore,* $\{C_3\}$ *is a necessary cause.*

Now, let us add C_1 *to the suspects. The counter-factual is* $(true, false, true)$*, and thus* $\mathcal{P} = false$*. Even-though* $\{C_3\}$ *is included in* $\{C_2, C_3\}$*,* $\{C_2, C_3\}$ *is not a necessary cause, whereas* $\{C_3\}$ *was. It shows that causality is not monotonic, with regard to the suspect set inclusion. The explanation here is simply that the error of* C_1 *is compensated by the one of* C_2*, as* $false \Leftrightarrow false = true$*, which explains the non-monotonicity.*

Note, that we can build a similar example for sufficient causality, as the non-monotonicity stems from the counter-factuals construction, which is used in both causality definitions.

Conclusion

This section presented the causality analysis principle, and two instantiations of it. As shown in the example subsection, the approach is able to give an intuitive result to all the problematic examples presented here, and inspired from the ones of [Halpern and Pearl, 2001b]. Though simpler and more intuitive, the cone approach makes a coarser approximation that the unaffected prefixes one, sometimes leading to counter-intuitive results.

Section 4

Implementation of the Causality Analysis Framework for a synchronous language

This section will present the Loca tool, developped by Gregor Gössler and Lacramioara Astefanoaei, that implements the unaffected prefixes approach, as well as an instantiation of the tool to the Lustre programming language.

This section will be divided in three subsection. The first one will present the restricted Lustre language instantiated in Loca. The next one will present the model transformation from Lustre to SMT. The last one will detail how the functions from Loca have been instantiated to Lustre.

The contributions from this section are the instantiation of a restricted version of Lustre into Loca, as well as a translation of restricted Lustre into SMT that is suitable for prefixing the specifications with traces.

4.1 The Lustre synchronous language

The Lustre language was introduced in [Halbwachs et al., 1991b]. It a synchronous data flow language, as presented in section 3.1. It has the particularity to be functional, which gives it the property to be deterministic both temporally and data-wise.

What was implemented is a subset of the Lustre language, that will be referred as "restricted Lustre". The main differences are that restricted Lustre does not support *current* and *when* operators, the use of library, external functions, arrays, custom types and declaring a constant without declaring its type (it would add to type-check steps, which would be costly for a not very important feature). Note that the library calls and array can

be emulated by, respectively, copying the functions/nodes called in the lustre file, and replacing the arrays by several variables (since LUSTRE only supports fix size arrays). *current* and *when* are , respectively, the up-sampling and down-sampling operators. The reason we do not support multiple clocks is not to have to perform clock calculus.

The idea of restricted LUSTRE is to keep the core of LUSTRE, while removing some features that would necessitate to compile the programs. Removing the up and down-sampling operators reduces the possible problems that can captured restricted LUSTRE. However, there are still a lot of interesting examples and causal problems that can be model by restricted LUSTRE.

The rest of this subsection will present LUSTRE features that are provided by restricted LUSTRE, as well as the syntax of restricted LUSTRE.

The two basic entities of and LUSTRE are the flows, presented earlier, and nodes. The nodes can be seen as components.

Example 12 (LUSTRE node) *Here is an example of the translation of the pump from example 4 into a* LUSTRE *node:*

```
node pump(com_pump: int) returns (out_pump: int);
let
  out_pump = min(10, com_pump);
  assert com_pump >= 0;
tel;
```

Node is a key-word that marks the beginning of the node. pump is the name of the node. $(com\_pump : int)$ *are the inputs of the node. After the colon, the type of the variable is given.* LUSTRE *supports the Boolean, integer and real types.* $(out\_pump : int)$ *after returns are the outputs of the node.*
The part between let and tel; is the body of the node, that contains equations that links the variables. Here, we have an equation that defines the value of out_pump *at each time-step as* $min(10, com\_pump)$*, as in the example.*
At this stage, we have defined com_pump *as an integer. However, in the example, it is a natural. To add a constraint to the input of the pump, we can use the assert construct. assert must be followed by a Boolean, that will always hold. Here, we assert that* $com\_pump >= 0$*, thus resulting in* com_pump *being a natural. We could add an assertion on the output flow, to ensure that it is in* $[0..10]$*, but it is verified by construction.*
This node is now equivalent to the definition given in Example 4, if we consider that the integers of LUSTRE *correspond to* $\mathbb{Z}$*.*

LUSTRE supports the call of nodes inside other nodes. In the previous example, *min* actually does not exists in the standard functions of LUSTRE. We can define a node *min* as follow:

Example 13
```
node min(a, b: int) returns (x: int);
let
  x = if (a < b) a else b;
tel;
```
Note that the if statement is in the right part of the equation defining x, contrary to the imperative languages. Here, at each instant, $x = a$ if $a < b$ and $x = b$ otherwise.

LUSTRE provides a function *pre* that returns the previous value of a flow. In the formalisation of the synchronous language, $pre(x)$ would translate into $\forall \underline{tr} \in \mathcal{B}_x, \forall i \in [0..|\underline{tr}| - 1], pre(x)[i+1] = x[i]$.
An initialising function $\rightarrow$ is also provided. $x = 0- > statement$ gives the constraint $\forall \underline{tr}, \pi_x(\underline{tr}[0]) = 0$.

Here is the translation of the tank into LUSTRE, that illustrates the use of *pre* and $\rightarrow$:

Example 14
```
node tank(out_pump: int) returns (vol_tank: int);
let
  vol_tank = out_pump -> out_pump + pre(vol_tank);
  assert 0 <= out_pump and out_pump <= 10;
tel;
```
This node has the expected behaviour to return $out\_pump[0]$ at instant 0 and $out\_pump[i] + vol\_tank[i-1]$ for the instant $i > 0$.

Constants and local variable can also be defined in LUSTRE.

LUSTRE needs the program to be "causal". The definition is different from the one used in this thesis. It means that there is no loop in the data dependencies that is not "broken" by a *pre*. For example, if a depends on b and b on a, the compilation will not take place. However, if a depends on b and b on $pre(a)$, there will be no problem. This constraint makes it easy to check whether a program is causal or not. However, it excludes some "causal" programs, for which their is a unique solution to the value of the variable from a dependency loop.

Restricted LUSTRE syntax The chosen syntax for a restricted LUSTRE program is the following, with $[x]*$ for any repetition of x, $[x]+$ any repetition of x at least one time, $[x]?$ for x at most once and 'x' for x written as is in the program:

```
program ::= [node]+

node ::= 'node' '(' params ')' 'returns' '(' params ')' node_body

params ::= param | param ';' params
param ::= idents ':' type
ident ::= ['A'-'Z''a'-'z']['A'-'Z''a'-'z''0'-'9''-''_']*
idents ::= ident [, ident]*
type ::= 'int' | 'real' | 'bool'

node_body ::= [local]* 'let' equations 'tel' [';']?
local ::= 'var' [param ';']+
  | 'const' [ident [':' type]? '=' expression ';']+

equations ::= [eq_or_as]+
eq_or_as ::= equation | 'assert' expression ';'
equation ::= left_part '=' expression ';'
left_part ::= '(' idents ')' | idents
expression ::= indent
  | value
  | unary expression
  | expression binary expression
  | 'if' expression 'then' expression 'else' expression
  | call
  | '(' expression ')'
call ::= ident '(' expressions ')'
expressions ::= expression [, expression]*
unary ::= 'pre' | '-' | 'not'
binary ::= '->' | '+' | '-' | '*' | '/' | 'div' | 'mod'
  | '<' | '>' | '<=' | '>=' | '<>' | '='
  | 'or' | 'and' | 'nor' | 'xor' | '=>'
value ::= 'true' | 'false' | ['0'-'9'][[.]['0'-'9']*]?
```

The most important definitions are the following. A program is a repetition of at least one node. A node is a declaration of inputs, one of outputs and a body. The body contains local variables and constants, followed by equations and assertions. An equation is a variable name on the left and an expression on the right.

This syntax is inspired from the LUSTRE reference manual ([Erwan Jahier, 2016]).

4.2 Translation from Lustre to SMTLib

As showed in [Hagen and Tinelli, 2008, Hagen, 2008], LUSTRE can be translated into first-order logic formulae. Therefore, we chose to use a SMT ([Biere et al., 2009]) translation for the implementation. This choice was also motivated by the fact that it means that several different "off the shelf" SMT-solvers could be used, and that it would prove useful if other models were to be instantiated in Loca using SMT, as some SMT function would have already been implemented.

The translation have been made into SMTLib ([Barrett et al., 2015]), which is a unified language for SMT-solvers that is supported by most of the existing SMT-solvers. Note that this thesis was not published when I designed the translation myself, but the results are similar, since I did it in early 2015.

The logic used by the SMT-solver must support uninterpreted functions, Booleans, integers and reals. The need for Booleans, integers and reals is straightforward, since they are the types supported by restricted LUSTRE.
Uninterpreted functions are used to be able to increasingly add some constraints to the functions, thus making it easier to prefix a specification, as well as being able to tackle the faulty components, for which we do not know the behaviour.
If properties on infinite traces are to be checked, it should also support quantifiers. When the properties are bounded, we can add manually enforce the constraints for each instant, up to the bound. Nevertheless, it is way more compact and convenient to have a quantification, if the bound is high, and is likely to be more efficient, since SMT solvers generally have some optimisations on quantifiers.
Note that this is an idealised LUSTRE that we describe, as the integer are unbounded, and the reals are not floating point numbers.

Here are the translations of LUSTRE into SMTLib inspired from [Barrett et al., 2015]:

Flow A flow is defined as an uninterpreted function, which take a natural as parameter:

```
(declare-fun flow_name (Nat) Type)
```

flow_name should be replaced by the name of the flow and *Type* by its type.
Since it is an uninterpreted function, we can add constraints on it, but do not need to fully define it.

Assertions The assertions are translated as follow:

```
(assert t(expression))
```

We note $t(x)$ for the translation of x into SMT. We simply assert the expression, requiring it to be *true*.

Equations For each equation we create a new uninterpreted function, that enforces the equality between the two members of the equation. Note that we use that in order to be able to choose when the equation constraint should be verified, as we need force the value sometimes (e.g. when prefixing a trace), due to causality analysis. If there is no need to remove the constraint, the equation can universally be quantified, using an *assert*, constraining the value of the uninterpreted function representing the flow.

```
(define-fun flow_name_eq (n Nat) Type
  (= (flow_name n) (t(expression) n)))
```

Expressions The expression are transformed from the LUSTRE operator notation, to the parenthesised notation used in SMTLib (i.e. *a and b* becomes (*and a b*). Most of the operators do not need any translations, as they are provided natively by SMTLib. The exceptions are the "if then else" statements, the temporal operator (pre and $->$) and the calls.

If then else To translate if c then t else e, we use the *ite* built-in as follow:

```
(ite t(c) t(t) t(e))
```

pre $pre(x)$ is translated as follow:

```
(t(x) n-1)
```

Supposing n is the index used.

$\rightarrow$ $a \rightarrow b$ is translated as follow:

```
(ite (= n 0) t(a) t(b))
```

Supposing n is the index used.
Note that $a = pre(x)$ will not be compiled by LUSTRE, since x does not have a value for $n = -1$. Therefore, there should always be an initialisation as follow: $a = b \rightarrow pre(x)$.

Call If a node call is made, like $node\_name(args)$, we replace the occurrence of the call by the output flow of the node that is called, and translate the node called. This may give rise to some problems which are solved by doing some pre-treatment, in order to list the instantiation needed by the node.

Nodes A node is translated by declaring all the variables of the node, translating the equations and the assertions, and if needed, translating the nodes that are called inside the initial node. Then, a function *node_step* is created, that supposes each equation, assertion and the called node step functions to be *true*. This is motivated by the same reason described for the equations.

For instance, let us suppose we want to translate a node *incr* that increment it input by one, calling an *add* node:

```
node incr(in: int) returns (out: int);
let
  out = add(in, 1);
tel;

node add(a, b: int) returns (x: int);
let
  x = a + b;
tel;
```

The translation would be:

```
(declare-fun in (Nat) Int)
(declare-fun out (Nat) Int)
(declare-fun add_x (Nat) Int)
(define-fun out_eq (n Nat) Int (= (out n) (add_x n)))
  (declare-fun add_a (Nat) Int)
  (declare-fun add_b (Nat) Int)
  (define-fun add_x_eq (n Nat) Int
   (= (add_x n) (+ (add_a n) (add_b n))))
  (define-fun add_step (n Nat) Bool (add_x_eq n))
(define-fun add_a_eq (n Nat) Bool (= (add_a n) (in n)))
(define-fun add_b_eq (n Nat) Bool (= (add_b n) 1))
(define-fun incr_step (n Nat) Bool
 (and (out_eq n) (add_step n) (add_a_eq n) (add_b_eq n))
```

The indented part corresponds to the translation of the called node *add*. In the node that calls, we add equations that constrain the values of the inputs of the called node, depending of the arguments passed in the call. Here, $add\_a = in$ and $add\_b = 1$, since the call is $add(in, 1)$. The last two lines are the *step* function. Note that it needs the equation functions, the arguments equations and the step function of the called node to be *true*.

Tuple Nodes might have more than on output. In that case, the left part of the equation is a tuple. For instance, would $node_name(args)$ have three outputs, an equation with this node would be:

```
(a, b, c) = node_name(args);
```

In order to be able to translate such an equation, a pre-treatment is performed, after the LUSTRE program have been fully parsed. The above equation into simple equations of the form $a = out1\_node\_called$, to instantiate the different calls (especially if a node is called several times).

4.3 Instantiation of LUSTRE in Loca

Loca is an implementation of the approaches from [Gössler and Le Métayer, 2013, Goessler and Astefanoaei, 2014] developed by Gregor Gössler and Lacramioara Astefanoaei. It uses a core algorithm that relies on functions that should be instantiated for the different models. The programming language used is SCALA ([Odersky, 2004]). It is a multi-paradigm language, that supports functional programming and is strongly and statically typed and compiles into java bytecode. The functional aspect and the strong type system makes it a good choice to implement formal methods algorithm.

As explained in the previous sections, a translation of LUSTRE into SMTLib was chosen. Since the SMT queries are externalised, the SMT-Solver z3 ([de Moura and Bjørner, 2008]) was selected. The reasons for this choice are manifold. Contrary to most of the solvers, that have a partial support of SMTLib, z3 provides a full support of SMTLib. It provides all the logics necessary to causality analysis, including the quantifiers. It is top tier, performance wise. Lastly, it supports MAX-SMT, that we consider using in the future, to implement the frameworks form Sections 5 and 6.

The argument of the program is a configuration file containing the path to the (compiling) LUSTRE file with all the node, the name of the main node, the name of the system property and a trace file. The specification is given as

a LUSTRE node, that should contain an assertion that is *true* if the property is satisfied, and *false* otherwise.

The rest of the section will present how the functions necessary to restricted LUSTRE have been instantiated.

initModel This function is the one that fetches the model and provides it to the rest of the program. The LUSTRE file is parsed into abstract types representing the different entities of LUSTRE (nodes, flows, equations, expressions,...).
Once the LUSTRE program have been parsed, a pre-treatment is performed, to split the equations of the form $(a, b, c) = node\_called(args)$. What's more, a list containing the instantiations of nodes that would be create by calling the program is created. This is the list of components that will be used throughout execution of Loca.
Every abstract types representing the LUSTRE entities have a **.toSMTLib** method. It is then easy to build a SMTLib version of any of these entities, by just calling this method.

sat The **sat** checks whether a trace satisfies a component specification. In order to verify it the node is translated into SMTLib, so is the trace. The node *step* function is supposed to be *true* for the length of the trace, and the solver is called on this model. If the solver returns "SAT", it means that the trace is consistent with the specification of the component. "Unsat" means that it is not the case.

Sat is used to check whether a trace is consistent with a component specification. $C.sat(\underline{tr})$ would translate in the formal causality framework as $\underline{tr} \in \mathcal{S}_C$.

semanticLength This function takes a trace as inputs and returns the length of the trace. Given that LUSTRE is synchronous, a trace is just a sequence of vectors of values. The length is trivially the length of the sequence. In other models, it may be more complicated.

getSemanticPrefix This function takes a trace and a length and returns the prefix of the trace of the given length. It is also simple here, we just return the prefix of the sequence of the given length.

prefixTr This function takes a component and a trace, and prefixes the trace to the component. The LUSTRE abstract node has a *trace* attribute. When it is present, the **toSMTLib** method prefixes the trace and does not suppose that the *step* function is *true*, for the length of the trace. We then just change the *trace* attribute to the trace given in argument

in a copy the component (not to change the initial object, and remain functional).

Example 15 *Let us suppose we have the following node:*

```
node add(a, b: int) returns (x: int);
let
  x = a + b;
tel;
```

and we want to prefix $((1,2,3),(3,2,5))$ *to it (with* (v,y,z) *corresponding to* $a = v$*,* $b = y$ *and* $x = z$*, at a given instant). We simply translate the node add into SMT, and add constraint for the first two ticks as follow:*

```
(declare-fun add_x (Nat) Int)
(declare-fun add_a (Nat) Int)
(declare-fun add_b (Nat) Int)
(define-fun add_x_eq (n Nat) Int
  (= (add_x n) (+ (add_a n) (add_b n))))
(define-fun add_step (n Nat) Bool (add_x_eq n))
(assert (and (= (a 0) 1) (= (b 0) 2) (= (x 0) 3)))
(assert (and (= (a 1) 3) (= (b 1) 2) (= (x 1) 5)))
```

The first six lines correspond to the translation of add into SMT. The last two ones correspond to constraining the first and second instant values to be as in the prefix.

Let us suppose that we want to prefix $\underline{tr}$ to the component C, it would translate in the formal causality framework as:

$$C.prefixTr(\underline{tr}) = \{\underline{tr}' \in \mathcal{B}_C \mid \underline{tr}' \in \mathcal{S}_C \wedge \pi_C(\underline{tr}'[0..|\underline{tr}| - 1) = \underline{tr}\}$$

Which is used in the building of the counter-factuals, as in Definition 15.

isReachableComplete This component takes a system trace, the list of all components, a component index and a complete predicate (a predicate provides a way of knowing that the end of the trace have been reached, for a given component), and returns *true* if the end of trace can be

reached for the considered component (designated by the index). We simply translate the system into SMTLib using the **compose** function, prefixing the trace, and adding a an assertion that the system step function should be *true* while the predicate holds (since a predicate is a length in the LUSTRE instantiation, it means from 0 to the predicate). "Sat" means that the end of the trace can be reached, and *true* is returned.

The intuition behind this function is that given a system trace prefix, component C is able to reach instant i of the initial trace, if all the components are prefixed with the system trace. This is used to build the unaffected prefixes, by checking whether C can reach instant i, or if the prefix must be shortened.

compose It is a method of the nodes that takes a list of nodes in argument, and returns the composition of the component with the components of the list. To do so, we call the **toSMTLib** function of the component, with the list of components in argument. When a node is called, if it is in the list of components, the one from the list is used. Otherwise, the specification is used. Given the pre-treatment, the main system node is the first one of the list of all nodes (since a call tree is build during the pre-treatment). Therefore, when we need to build the whole system, we can just call compose from the first component of the list of components.

Intuitively, given a set of all the system components (which might have been prefixed) this function returns the composition of all those components. It can be seen as the set of all the possible traces, rather than a composition. Note that to emulate the faulty components behaviour, we simply enforce the possible prefix, but do not constrain the rest of the behaviour at all (beside component input and output types).

refine This is a method of the nodes. It takes a specification and a set of predicates in arguments, and returns if the node refines the specification, up to the predicate. In order to verify it, we transform the node into SMTLib. The negation of the property is added to the SMTLib file, and the *step* function is supposed to be *true*, up to the predicates. If the solver call returns "Unsat", the node refines the property, as there is no behaviour that can falsify the property. The function then returns *true*.

Intuitively, since this method is called from a system composition (which can be seen as a set of traces $\mathbb{TR}$, if we suppose that the specification to

verify $\mathcal{P}$, $\mathbb{TR}.refine(\mathcal{P})$ can be seen in the formal causality framework as $\mathbb{TR} \in \mathcal{P}$. It is notably used to check necessary causality, by checking whether $CF(\underline{tr}, \mathcal{I}) \subseteq \mathcal{P}$).

inconsistentWith This function was not described in the presentation of Loca, but it is used when the sufficient causality is to be checked. It has the same arguments as the previous function, but verifies that the node has no behaviour that satisfies the property. Similarly to the **refine**, we translate the node and the property (not its negation) into SMTLib and make a solver call. If the result is "Unsat", it means that the node has no behaviour that satisfies the property, and **inconsistentWith** returns *true*.

Intuitively, if refine checks $\mathbb{TR} \in \mathcal{P}$, inconsistentWith checks $sup(\mathbb{TR}) \cap \mathcal{P}) = \emptyset$. It is notably used to check sufficient causality.

Conclusion

The instantiation was tested on several examples. It gave the expected causality analysis result for all of them. For a system with 8 nodes (4 of them faulty) and a trace of length 15, the result is given in a couple of seconds. However, the approach should not scale very well in number of faulty components, since the causality check is made on each subset of the set of faulty component, which result on a complexity of 2 to the power of faulty components ($O(2^{n_f})$, with n_f the number of faulty components).
Concerning the length of the trace, it depends on how non-deterministic the system is. LUSTRE systems generally are deterministic, however, non-determinism can be introduced by adding node inputs that are not "wired". If you want to add a non-deterministic variable to a node, you simply add an input to this node that is unconstrained. You can also consider the values of this input as being given by an oracle. For a deterministic system, the worst case complexity grow logarithmically to the length of the trace, since a dichotomous search is performed along the trace ($O(ln(l))$). This dichotomous search is performed for each component ($O(n_C)$, with n_C the number of component). For the number of components, the same determinism argument arise. For a deterministic system, the complexity of one search grows linearly to the number of equations and assertion in the call tree ($O(n_e)$). On deterministic systems, the SMT-Solver answers very fast, as long as there is enough memory. We end up with a complexity $2^{n_f} \times O(ln(l)) \times O(n_C) \times n_e \times c_{SMT})$, with c_{SMT} the cost of a SMT call.

As shown in [Hagen and Tinelli, 2008, Hagen, 2008], it is possible to translate the multi-clock model in SMT, except for the *when* operator (down-

sampling), that can only be translated in certain cases. It would be interesting to extend the current implementation, to take into account the clock calculus, so the failure induced by the clocks can be tackled.

Section 5

Combining white-box and black-box components in Causality Analysis

As shown in Section 2, their are numerous approaches that have goals similar to blaming for either the black-box components (e.g. components for which we know the specification, but not necessarily the model/implementation) or the white-box components (e.g. components for which we know the model/implementation, but not necessarily a specification). However, to my knowledge, their is no approach that treats both at the same time. Therefore, this section proposes a way of combining the causality analysis with white-box and black-box components.

This section presents such an approach, namely the mixed approach. It is close to the vanilla approach (the causality analysis approach introduced in Section 3), but adds a layer of controller synthesis to check whether or not the white-box components had a way of avoiding the failure. The idea is to use the counter-factuals to generate all the possible traces close to an initial failing trace ($\underline{tr}$), such that some black-box components (in $\mathcal{I}$) are fixed, and some white-box components (in $\mathcal{R}$) can act "as they want", regardless of what they did in the initial trace. It means that all the possible behaviours for the components in $\mathcal{R}$ are generated. From those counter-factuals, we try to make a controller synthesis, to check if the components in $\mathcal{R}$ had a way to prevent the failure.

This approach is rich enough to be used in a debugging or designing phase. It acts as a combination of diagnosis (which is usually used on black-box components) and fault localisation (usually used on white-box components). However, it lacks some tools to be usable as a responsibility assignment tool, as it will be discussed in Section 6.

This section is divided in three subsections. The first one presents the general definitions necessary to be able to perform a causality analysis in the mixed framework. The second one gives some causality definitions with this new framework. The third one shows how to perform the synthesis.

All the concepts and results that are developed here are contributions from this thesis.

5.1 Mixed framework definitions

This section will introduce some definitions and tools needed to perform the causality analysis in the mixed framework.

Definition 21 (Mixed System S) *A mixed system is a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ such that $\mathbb{C} = \mathbb{C}_W \uplus \mathbb{C}_B$, with C_W the set of white-box components and C_B the set of black-box component.*

A mixed system is a system for which the set of component is split between a set of black-box components and a set of white-box components. The black-box components are the ones we considered since the beginning of the manuscript. The white-box components are components for which we have access to a model, an implementation, or the actual system, instead of a more abstract specification. Formally, a white-box component is defined like a black-box component, though the $\mathcal{S}$ is referred as the model, instead of specification (of the black-box components).
We suppose that the white-box components always respect their model, contrarily to the black-box ones that can violate their specification.
It is important to understand that contrary to the black-box components, the white-box components do not have a notion of normality (see Section 2.3 for a discussion on notion of normality in causality analysis), since they do not have a specification. The model describes all the possible behaviours, but does gives us any information on the ones that are normal or expected, hence the hypothesis that the model is always respected.

In this section, we suppose we are a mixed system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$.

Definition 22 (Mixed framework counter-factuals (CF_{mixed})) *Let CF be a counter-factual definition, that takes a trace and a set of suspected components in arguments. Let $\underline{tr}$ be a system trace, $\mathcal{I} \subseteq \mathbb{C}_B$ a set of suspected components and $\mathcal{R} \subseteq \mathbb{C}_W$ a set of protagonists. $CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$ is the set of traces $CF(\underline{tr}, \mathcal{I} \cup \mathcal{R})$ where we consider that $\forall C \in \mathcal{R}, \pi_C(\underline{tr}) \notin \mathcal{S}_C$.*

In the mixed counter-factuals, we add the $\mathcal{R}$ arguments, that correspond to a set of white-box components. The idea is to build all the possible traces close to $\underline{tr}$, such that the faults from $\mathcal{I}$ are removed, and the protagonist can chose any possible behaviour.
By definition, the white-box components cannot be faulty. That is the reason we add the fact that they are considered as faulty for the whole trace, in the "vanilla" counter-factual definition, thus generating all the possible behaviours consistent with $\underline{tr}$ for the said protagonists. For instance, in the cone approach, if C is a protagonist, then $cone_C = 0$, since it is always considered as faulty. Therefore, all the possible behaviours will be built for C, since the counter-factuals will be generated from the very first instant for C.

Let us illustrate the Mixed framework counter-factuals with an example.

Example 16 *In order to ease the use of this example for the next definitions, we will leave the realm of pumps and tanks for a simpler system.*

The system is composed of two black-box components, C_1, that has no input and always output true, and C_3, a two input OR gate, alongside a white-box component C_2 that has no input, and a random Boolean output. The system property is that C_3 output should always be true.

Here are the formal definitions of the components:
C_1:

- $I_{C_1} = \emptyset$
- $O_{C_1} = \{(Out_1, \mathbb{B})\}$
- $\mathcal{S}_{C_1} = \{true\}^\omega$

C_2:

- $I_{C_2} = \emptyset$
- $O_{C_2} = \{(Out_2, \mathbb{B})\}$
- $\mathcal{S}_{C_2} = \mathbb{B}^\omega$

C_3:

- $I_{C_3} = \{(Out_1, \mathbb{B}), (Out_2, \mathbb{B})\}$
- $O_{C_3} = \{(Out_3, \mathbb{B})\}$
- $\mathcal{S}_{C_3} = \{\underline{tr} \in \mathcal{B}_{C_3} \mid \forall i \in [0..|\underline{tr}| - 1], \pi_{Out_3}(\underline{tr}[i]) = \pi_{Out_1}(\underline{tr}[i]) \vee \pi_{Out_2}(\underline{tr}[i])\}$

$\mathcal{P} = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \pi_{Out_3}(\underline{tr}) = (true)_{i \in [0..|\underline{tr}|-1]}\}$

Here is a failing trace $\underline{tr}$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	*false*	*true*	*true*	*false*
Out_3	*true*	*true*	*true*	*false*

In this trace, at instant 2, C_1 becomes faulty, and starts outputting false. This does not immediately create a failure, since C_2 outputs true. Then at instant 3, C_2 outputs false and the failure occurs.

Here is $CF_{mixed}(\underline{tr}, \{C_1\}, \emptyset)$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*true*	*true*
Out_2	*false*	*true*	*true*	*false*
Out_3	*true*	*true*	*true*	*true*

Since the protagonist argument is the emptyset, the behaviour is the same as the vanilla counter-factuals, i.e. $CF(\underline{tr}, \{C_1\})$, and it fixes the system.

Here is $CF_{mixed}(\underline{tr}, \emptyset, \{C_2\})$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	*true*	*true*	$Out_2[2]$	$Out_2[3]$

We note a set instead of a value, if a flow can take multiple values, and $f[i]$ for $\pi_f(\underline{tr}[i])$.
Here, one can see how the behaviour differs of $CF_{mixed}(\underline{tr}, \emptyset, \{C_2\})$ from the one of $CF(\underline{tr}, \emptyset)$ (i.e. the counter-factuals with no protagonist), as Out_2 can be true or false, regardless of the value in the initial trace.
Note that the model is a simple coin toss, however, it could be more complex, like never the same output three times in a row, or the possible output values could depend on inputs, and the mixed framework counter-factuals should respect this model.
Our intuition tells us that by always outputting true, C_2 can enforce the specification, however, this will be generated by the synthesis, that will be discussed later in this section.

Here is $CF_{mixed}(\underline{tr}, \{C_1\}, \{C_2\})$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*true*	*true*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	*true*	*true*	*true*	*true*

Here the fact that we fix the faulty C_1 ensures that the property is verified, whatever C_2 does. All the possible behaviours for C_2 have been generated, since $\mathcal{R}$ is equal to $\{C_2\}$. We can see the impact of $\mathcal{R} = \{C_2\}$, compared to $CF_{mixed}(\underline{tr}, \{C_1\}, \emptyset)$, in the fact that $Out_2 = \mathbb{B}$, instead of its initial value in $\underline{tr}$, for each instant.

Given that the formal definition of the synthesis necessitates several new concepts rather lengthy to introduce, the synthesis step will be abstracted by a function, in order to be able to introduce the causality definitions faster, thus giving a better understanding of the specificity and capabilities of this new framework, and the possible application to debugging/design, without flooding the reader with too many definitions. The method to actually perform the synthesis will be presented in Section 5.3.

Definition 23 (Winning strategy (WIN_{mixed})) *Let CF be a counter-factuals definition, that takes a trace and a set of suspected components in arguments. Let $\underline{tr}$ be a system trace, $\mathcal{I} \subseteq \mathbb{C}_B$ a set of suspected components and $\mathcal{R} \subseteq \mathbb{C}_W$ a set of protagonist.*
There exist a winning strategy, noted $WIN_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$, if there exist a way of controlling the choices of the components in $\mathcal{R}$ such that the set of controlled traces $controlled(\underline{tr}, \mathcal{I}, \mathcal{R}) \subseteq CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$ is prefix-closed and such that $\forall \underline{tr}' \in controlled(\underline{tr}, \mathcal{I}, \mathcal{R})$,:

$$\underline{tr}' \in \mathcal{P} \wedge \exists \underline{tr}'' \in (sup(CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})) \cap controlled(\underline{tr}, \mathcal{I}, \mathcal{R})), \underline{tr}' \sqsubseteq \underline{tr}''$$

There is a winning strategy, if the "choices" of the components in $\mathcal{R}$ can be made such that all the controlled traces are non-failing, and they all are prefixes of a "final" trace. The prefix-closure and the fact that all the traces in *controlled* prefix a "final" non-failing trace ensures that the controller is non-blocking, i.e. it does not enforce the property by preventing the system from evolving. A winning strategy is then a non-blocking way of constraining the system never to fail. What's more, in the current formalisation of the system, it is not possible to model a blocking behaviour.

This definition will be further formalised later on in Section 5.3, so let us illustrate it with an example, to give a good idea of the way it is supposed to work.

Example 17 *The system is the same as the one in Example 16.*
Given the following mixed counter-factuals for $\mathcal{R} = \{C_2\}$*:*

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	*true*	*true*	$Out_2[2]$	$Out_2[3]$

It is obvious that by always choosing true, C_2 *ensures that the system will never fail. This can be translated by a controller that forces the output of* C_2 *to true, at every instant.*

Let us now consider a slightly different system where C_3 *is an AND gate, instead of an OR gate.*
The mixed counter-factuals for this new system, if we consider the same initial trace as the previous ones and $\mathcal{R} = \{C_2\}$ *are:*

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	$Out_2[0]$	$Out_2[1]$	*false*	*false*

Here a way for C_2 *to ensure that no failure occurs is to output true if* Out_1 *is true, and "refuse" to output anything otherwise, thus blocking the system. This way,* Out_3 *would never be false. However, this is not a valid strategy, as the controller should not be blocking.*

Definition 24 (Spoiling behaviour (*$SPOIL_{mixed}$*)) *Let* CF *be a counter-factuals definition, that takes a trace and a set of suspected components in arguments. Let* $\underline{tr}$ *be a system trace,* $\mathcal{I} \subseteq \mathbb{C}_B$ *a set of suspected components and* $\mathcal{R} \subseteq \mathbb{C}_W$ *a set of protagonist.* $CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$ *is a spoiling behaviour, noted* $SPOIL_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$*, if* $\sup(CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})) \cap \mathcal{P} = \emptyset$*.*

The idea is that even with changing the non-deterministic values of the component in $\mathcal{R}$ and fixing the one in $\mathcal{I}$, the system always eventually fails. Note that we cannot get the result from this definition using *WIN* while trying to enforce the complementary of $\mathcal{P}$, since *SPOIL* introduces the notion of "eventually" violating $\mathcal{P}$.

The second counter-factuals of Example 17 (where C_3 is a *and* gate) is a spoiling behaviour, as the system will eventually fail, whatever is the behaviour of C_2.

5.2 Causality definitions for the mixed framework

This section will show how to adapt the previous causality definitions, and what the result of the causality analysis means, from a designer perspective, as well as some new definitions.

The design situation considered here is that there is one failing trace, like a bug report, or a result of a scenario, and the mixed framework causality is used to help the designer in choosing the components to fix/control, in order to avoid the failure.

In the whole section, we suppose we are given a mixed system $S = (\mathbb{F}, \mathbb{C}, \mathcal{P}, BM)$ with $\mathbb{C} = \mathbb{C}_W \uplus \mathbb{C}_B$, a faulty trace $\underline{tr} \in BM \setminus \mathcal{P}$ and a counterfactuals definition CF.

Definition 25 (Necessary cause (nec_{mixed})) *Let $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected components, $\mathcal{I}$ is a necessary cause, noted $\mathrm{nec}_{mixed}(\underline{tr}, \mathcal{I})$, if $WIN_{mixed}(\underline{tr}, \mathcal{I} \cap \mathbb{C}_B, \mathcal{I} \cap \mathbb{C}_W)$.*

Intuitively, it means that if we fix the black-box components in $\mathcal{I}$, it is possible to control the white-box components such that the failure from $\underline{tr}$ never happens.

From a designer perspective, would the black-box components be non-faulty, there is a way of controlling the white-box ones such that the failure is always avoided. Making sure that the black-box components are non-faulty can be achieved by some dependability means, like redundancy, choosing a more resilient off-the-shelf component that respect the initial specification. Controlling the white-box components can either be done locally, in certain cases, or by adding a global controller.
The fact that $\mathcal{I}$ is not a cause is actually interesting as well, since it gives the designer the information that more black-box components should be fixed, or more white-box components controlled.

Note that by construction, if $\mathcal{I} \subseteq \mathbb{C}_B$, $nec_{mixed}(\underline{tr}, \mathcal{I}) = \mathrm{nec}(\underline{tr}, \mathcal{I})$, where nec uses CF, since $CF(\underline{tr}, \mathcal{I}) = CF_{mixed}(\underline{tr}, \mathcal{I}, \emptyset)$. It means that this definition is a generalisation of the previous one (Definition 10).

Example 18 *Let us consider the first part of Example 17, where C_3 is an OR gate.*

Given the following failing trace:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	*true*	*true*	*true*	*false*
Out_3	*true*	*true*	*true*	*false*

We can build the mixed counter-factuals for $\mathcal{R} = \{C_2\}$*:*

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	*true*	*true*	$Out_2[2]$	$Out_2[3]$

As long as C_2 *outputs true when* Out_1 *is false, the system will not fail. This is controllable, since* $Out_2 = \neg Out_1$ *is a controller. Therefore,* $\mathcal{I} = \{C_2\}$ *is a mixed necessary cause.*

If we build the mixed counter-factuals for $\mathcal{I} = \{C_1\}$*, we get:*

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*true*	*true*
Out_2	*true*	*true*	*true*	*false*
Out_3	*true*	*true*	*true*	*true*

Those counter-factuals are the same as $CF(\underline{tr}, \mathcal{I})$*. Since they respect the system property,* $\mathcal{I} = \{C_1\}$ *is a mixed necessary cause, as it should be, since* $\{C_1\}$ *is a necessary cause in the vanilla framework.*

Definition 26 (Sufficient cause (suff_{mixed})**)** *Let* $\mathcal{I} \subseteq \mathbb{C}$ *be a set of suspected components,* $\mathcal{I}$ *is a sufficient cause, noted* $\text{suff}_{mixed}(\underline{tr}, \mathcal{I})$*, if* $SPOIL_{mixed}(\underline{tr}, (\mathbb{C} \setminus \mathcal{I}) \cap \mathbb{C}_B, (\mathbb{C} \setminus \mathcal{I}) \cap \mathbb{C}_W)$*.*

Intuitively, even by fixing the black-box components from the complimentary of $\mathcal{I}$, and considering all the possible behaviours of the white-box ones from $\mathcal{I}$ the failure is deemed to happen.

From a designer perspective the fact that suspect is a sufficient cause means that more components must be removed from $\mathcal{I}$, in order to fix the failure. The fact that suspect is not a sufficient cause means that subsets of $\mathbb{C} \setminus \mathcal{I}$ are possible candidates to fix the system. Since computing the counter-factuals for $\mathcal{I}$ is necessary to perform the synthesis, it is interesting to perform the *SPOIL* check (i.e. verifying if $\text{suff}_{mixed}(\underline{tr}, \mathbb{C} \setminus I)$) to make sure it is of use to actually perform the synthesis.

Note that this definition also is a generalisation of the vanilla causality framework sufficient causality one (Definition 12), for the same reasons as the necessary causality ones ($\text{suff}_{mixed}(\underline{tr}, \mathcal{I} \cup \mathbb{C}_W) = \text{suff}(\underline{tr}, \mathcal{I})$ if $\mathcal{I} \subseteq \mathbb{C}_B$).

Example 19 *Let us consider the second part of Example 17, where C_3 is an AND gate.*

Given the following failing trace:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	*true*	*true*	*true*	*false*
Out_3	*true*	*true*	*false*	*false*

Let us build the mixed counter-factuals for $\mathcal{I} = \mathbb{C} \setminus \{C_1\} = \{C_2, C_3\}$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	$Out_2[0]$	$Out_2[1]$	*false*	*false*

We can see that Out_3 is bound to be false at instant 2 and 3, whatever value C_2 chooses. Therefore, $\{C_1\}$ is a sufficient cause.

Let us build the mixed counter-factuals for $\mathcal{I} = \mathbb{C} \setminus \{C_2\} = \{C_1, C_3\}$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*true*	*true*
Out_2	*true*	*true*	*true*	*false*
Out_3	*true*	*true*	*true*	*false*

The counter-factuals eventually become faulty. Hence, $\{C_2\}$ is a mixed sufficient cause, as well as a sufficient cause in the vanilla framework.

Definition 27 (System fix fix_{mixed}) *$\mathcal{I}$ is a system fix, noted $fix_{mixed}(\underline{tr}, \mathcal{I})$, if $\forall \mathcal{I}' \subseteq \mathbb{C}, \mathcal{I} \subseteq \mathcal{I}' \implies \text{nec}_{mixed}(\underline{tr}, \mathcal{I}')$.*

It means that if we fix at least $\mathcal{I} \cap \mathbb{C}_B$, and at least the components in $\mathcal{I} \cap \mathbb{C}_W$ are controlled, the system cannot fail.
Note that the quantification could only be on $\mathcal{I} \cap \mathbb{C}_B$, because if $\mathcal{R}$ has a winning strategy, any superset $\mathcal{R}'$ of $\mathcal{R}$ trivially has one, by applying the same strategy as $\mathcal{R}$, for the component in $\mathcal{I}$, and keeping the behaviour as in the traces for the components in $\mathcal{I}' \setminus \mathcal{I}$. However, the black-box causality framework has shown that black-box cause are not monotonous (e.g. if two faults compensate one another, fixing one might make the system fail), hence the need for a quantification over the black-box components.

From a designer perspective, it is interesting because it means that fixing $\mathcal{I} \cap \mathbb{C}_B$ and controlling $\mathcal{I} \cap \mathbb{C}_W$ ensures that the system will not fail as in $\underline{tr}$.

We can define a notion of minimal fix. $\mathcal{I}$ is a minimal fix, if $\mathcal{I}$ is a fix and $\forall \mathcal{I}' \subset \mathcal{I}, \neg fix(\underline{tr}, \mathcal{I}')$. This is important for the designer as it gives him the minimum sets of components it has to control/fix, in order to avoid the failure.

If we consider Example 19 both $\{C_1\}$ and $\{C_2\}$ are system fixes, since any superset of them are mixed necessary causes.

Definition 28 (Unavoidable cause *una***)** *$\mathcal{I}$ is an unavoidable cause, noted $una(\underline{tr}, \mathcal{I})$, if $\forall \mathcal{I}' \subseteq \mathbb{C}, \mathcal{I} \subseteq \mathcal{I}' \implies \text{suff}(\underline{tr}, \mathcal{I}')$.*

It means that if you do not fix, any component from $\mathcal{I} \cap \mathbb{C}_B$ and do not control any component from $\mathcal{I} \cap \mathbb{C}_W$, the system necessarily fail.
As for the previous example, only the quantification on the black-box is necessary, because if the behaviour is spoiling while being able to control $\mathbb{C}_W \setminus \mathcal{I}$, it is also when controlling less components ($\mathbb{C}_W \setminus \mathcal{I}'$), since controlling more components means that more possible behaviours are generating, hence giving more possibilities for the behaviour not to be spoiling.

From a designer perspective, it means that there is no point in trying to prevent the failure if none of the components in $\mathcal{I}$ is fixed/controlled.

Similarly to the system fix, we can define a notion of minimal unavoidable cause. This gives sets of components from which at least one must be fixed/controlled, in order to have a shot at preventing the failure.

5.3 Strategy synthesis for the mixed framework

The causality definitions presented in Section 5.2 rely on the computation of a winning strategy. In that sense, they can be seen as two players games, with the protagonist against the rest of the system. The problem of finding a strategy can therefore be reduced to a controller synthesis problem, where the controllable moves are made by the player (namely the protagonist), and the uncontrollable moves are made by the rest of the system.

This section is divided into three subsections. The first one presents the principle of controller synthesis, as well as a short bibliography on the domain. The second one gives a method to translate traces into labelled transition systems (LTS). The last one introduces the synthesis on LTS, as well as its application to our framework

5.3.1 Controller synthesis

The community of Discrete-Event System (DES) has studied at great length the problem of controller synthesis. Ramadge and Wonham proposed an algorithm to compute controller for DES in [Ramadge and Wonham, 1987] that they enhanced in [Ramadge and Wonham, 1989]. [Cassandras and Lafortune, 1999] proposes an overview of the DES domain, with a chapter dedicated to controller synthesis.

The principle of controller synthesis is to split the events in two sets. The first one is the set of controllable events and the other one is the set of uncontrollable ones. The controller can choose the controllable events to generate (e.g. which value to output, in the mixed framework). A system property (oftentimes a safety one) is to be enforced by the controller. Controller synthesis consists in building a controller that ensures that the property is always verified, whatever the environment does. For safety property, a controller that does not permit any event might enforce the property. Thus, the controller synthesis problem is to find the most permissive controller. The algorithm proposed by [Ramadge and Wonham, 1987] works by first generating the state-space, and then removing recursively the controllable transitions that lead to the violation of the property.

[Asarin et al., 1995] proposed to use controller synthesis to perform program synthesis, using some symbolic tools from verification, as well as an extension from discrete time to continuous time. [Ehlers et al., 2016] is a comprehensive survey of the current state of supervisory control and reactive synthesis.

5.3.2 Translating traces into LTS

A formalism to describe how the system evolves during a step will be introduced, as well as a way of transforming a set of traces into a labelled transition system (LTS).

During the compilation of a Lustre program, the system is flattened (i.e. the node structured is translated in one big node). Then, a topological sort of the flows is performed, that reflects the dependencies over the flows ([Halbwachs et al., 1991b]). This flow order gives a partial order that reflects the dependencies between the flows. The following definition gives a way of representing the evolution of the system during a step.

Definition 29 (Partial step s) *Let $\mathbb{F}$ be a set of flows and $\prec$ be a partial*

order over $\mathbb{F}$. *A partial step* $s = (s_f)_{f\in\mathbb{F}}$ *is an element of* $\bigtimes_{f\in\mathbb{F}}(D_f \cup \{undef\})$ *such that:*

$$\forall f \in \mathbb{F}, s_f \neq undef \implies (\forall f' \in \mathbb{F}, f' \prec f \implies s_{f'} \neq undef)$$

$undef$ means that the flow has no value yet. $\prec$ corresponds to the partial order yield by the topological sort. If a flow f' is "greater" than another one f, it means that the evaluation of its value depends, possibly indirectly, on the value of f. A partial step captures what happens during the evaluation of a global step in a system, ensuring that if a flow is evaluated (i.e. different from $undef$), all its dependencies have already been evaluated too.

Let $v = (v_f)_{f\in\mathbb{F}} \in \bigcup_{f\in\mathbb{F}}(D_f \cup \{undef\})$ be a valuation for the flows in $\mathbb{F}$, we note $PS(v)$ the set of all possible partial steps over $\mathbb{F}$ that are consistent with $\prec$ such that $\forall s \in PS(v), \forall f \in \mathbb{F}, s_f = v_f \vee s_f = undef$.

Example 20 *If we consider the system from Example 16,* $Out_1 \prec Out_3$ *and* $Out_2 \prec Out_3$, *since* Out_3 *depends on* Out_1 *and* Out_2.

This is not the case in this system, but if there was another flow f *such that* $f \prec Out_1$, *then, we would have* $f \prec Out_3$, *since the partial order is transitive.*

Let $(true, false, true)$ *be a valuation for* $(Out_1, Out_2, Out_3$. *Then* $PS((true, false, true)) =$

$$\{(undef, undef, undef), (true, undef, undef), (undef, false, undef), (true, false, undef), (true, false, true)\}$$

Note that $(true, undef, true)$ *is not in* $PS((true, false, true))$, *because* Out_3 *would be valuated before* Out_2, *which is not possible, since* $Out_2 \prec Out_3$.

Definition 30 (Labelled Transition System (LTS)) *A LTS is a tuple* $L = (Q, q^0, \Sigma, \rightarrow)$ *where:*

- Q *is a set of states*
- q^0 *is an initial state*
- Σ *is a set of symbols*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *a transition function.*

As mentioned earlier, we need to describe the evolution of the system at a level within a step. This is called a microstep semantic. It was introduced because it ensures the closure of synchronous systems under concurrent composition ([Benveniste et al., 2003]). Though this is not the only solution to ensure the closure, this is the one we consider here, as it is simple, and is the one used in LUSTRE. The idea is to reduce a step to a sequence of elementary microsteps, here the valuation of one flow. The flow can be evaluated if its dependencies have been valuated, as defined in the partial step. A convenient and understandable way of representing the evolution of the system is to use a LTS where the states are labelled by a prefix (the already evaluated steps), alongside a partial step (that is being valued), and the transitions are labelled by the valuation of the flows (i.e. the name of the flow and the chosen value). It is easy to represent the possible interleaving of the valuation during a step with such a representation.

Definition 31 (System evolution LTS (LTS_{sys})) *We suppose we are given the partial order $\prec$ for the system S. Let $\mathbb{TR} \subseteq \mathcal{B}_{\mathbb{F}}$ be a set of system traces. The corresponding system evolution LTS, noted $LTS_{sys}(\mathbb{TR})$ is the tuple $l_{CF} = (Q, q^0, \Sigma, \rightarrow)$ such that:*

- *Q is the greatest set such that $Q \subseteq \left(BM \times \bigtimes_{f\in\mathbb{F}}(D_f \cup \{undef\})\right)$ and*

$$\forall \underline{tr}' \in BM, \forall \underline{tr}'' \in \mathbb{TR}, \Big(\underline{tr}' \sqsubseteq \underline{tr}'' \implies \big((\underline{tr}', (undef)_{f\in\mathbb{F}}) \in Q \wedge (\{tr'[0..|\underline{tr}'|-2]\} \times PS(\underline{tr}'[\underline{tr}'-1])) \subseteq Q\big)\Big)$$

- $q^0 = ((), (undef)_{f\in\mathbb{F}})$.
- $\Sigma = \{\varepsilon\} \cup \bigcup_{F\in\mathbb{F}}(\{name_f\} \times D_f)$
- *$\rightarrow$ is such that $\forall \underline{tr}' \in \mathcal{B}_{\mathbb{F}}$,*

$$\big((\underline{tr}', (undef)_{f\in\mathbb{F}}) \in Q \wedge (tr'[0..|\underline{tr}'|-2], \underline{tr}'[\underline{tr}'-1]) \in Q\big) \implies \big((tr'[0..|\underline{tr}'|-2], \underline{tr}'[\underline{tr}'-1]), \epsilon, (\underline{tr}', (undef)_{f\in\mathbb{F}})\big) \in \rightarrow \ \wedge \quad (5.1)$$

$$\forall s, s' \in \bigtimes_{f\in\mathbb{F}}(D_f \cup \{undef\}), \big((\underline{tr}', s) \in Q \wedge (\underline{tr}', s') \in Q \ \wedge enables(s, s) \neq \bot\big) \implies ((\underline{tr}', s), enables(s, s'), (\underline{tr}', s')) \in \rightarrow \quad (5.2)$$

With $enables(s, s') = (f, s'_f)$ *if* $\exists! f \in \mathbb{F}, \big((s_f = undef \wedge s'_f \neq undef) \wedge (\forall f' \in (\mathbb{F} \setminus \{f\}), s_{f'} = s'_{f'}) \wedge (\forall f' \in \mathbb{F}, f' \prec f \implies s_f \neq undef)\big)$ *and* $\perp$ *otherwise.*

Given a set of traces $\mathbb{TR}$, we build a LTS corresponding to all the possible evolutions of the system that lead to a trace in $\mathbb{TR}$. It means that it does not only represents the traces in $\mathbb{TR}$, but also the prefixes of those traces, and all the possible transitional evolution of a step from a trace to that trace extended by one step.
A state is a tuple containing a trace and a partial step. The state space corresponds to all the prefixes of the traces in $\mathbb{TR}$, alongside the transitional partial steps between those prefixes.
The initial state is the tuple with the empty trace and the totally undefined partial step.
The transitions are labelled with the name of the flow that becomes valued, alongside the chosen value.
We allow a transition between a fully valued partial step to the trace extended with this fully valued partial step, with an undefined partial step as the second part of the state (Constraint 5.1). It corresponds to finishing a global instant, and starting the new one. We also allow all the transitions between partial steps such that the differences between them correspond to the valuation of exactly one flow f, with the constraint that all the flows f depends on (i.e. the flows $f' \in \mathbb{F}$ such that $f' \prec f$) have been valued (Constraint 5.2).

Example 21 *Let us translate a trace of the system from Example 16 into a LTS.*

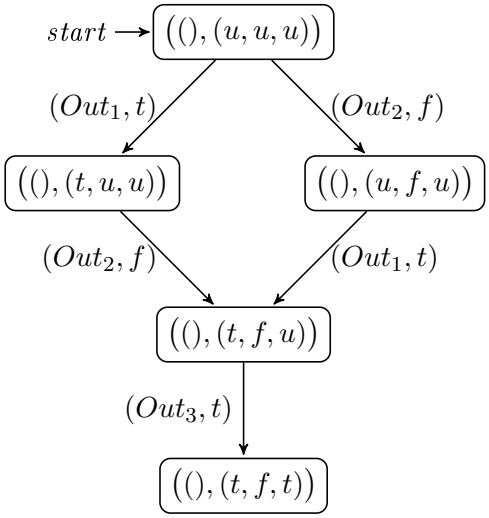
The trace $\underline{tr}$ *considered is the following:*

Time	*0*	*1*
Out_1	*true*	*false*
Out_2	*false*	$\mathbb{B}$
Out_3	*true*	$Out_2[1]$

A trace instant will be represented as (Out_1, Out_2, Out_3)*, with the* Out_i *replace by a Boolean value. For instance, instant* 0 *is* $(true, false, true)$*. A trace is represented as a sequence of instants. For instance, if* Out_2 *is false at instant* 1*, the* $\underline{tr}[0..1] = ((true, false, true), (true, false, true))$*.*

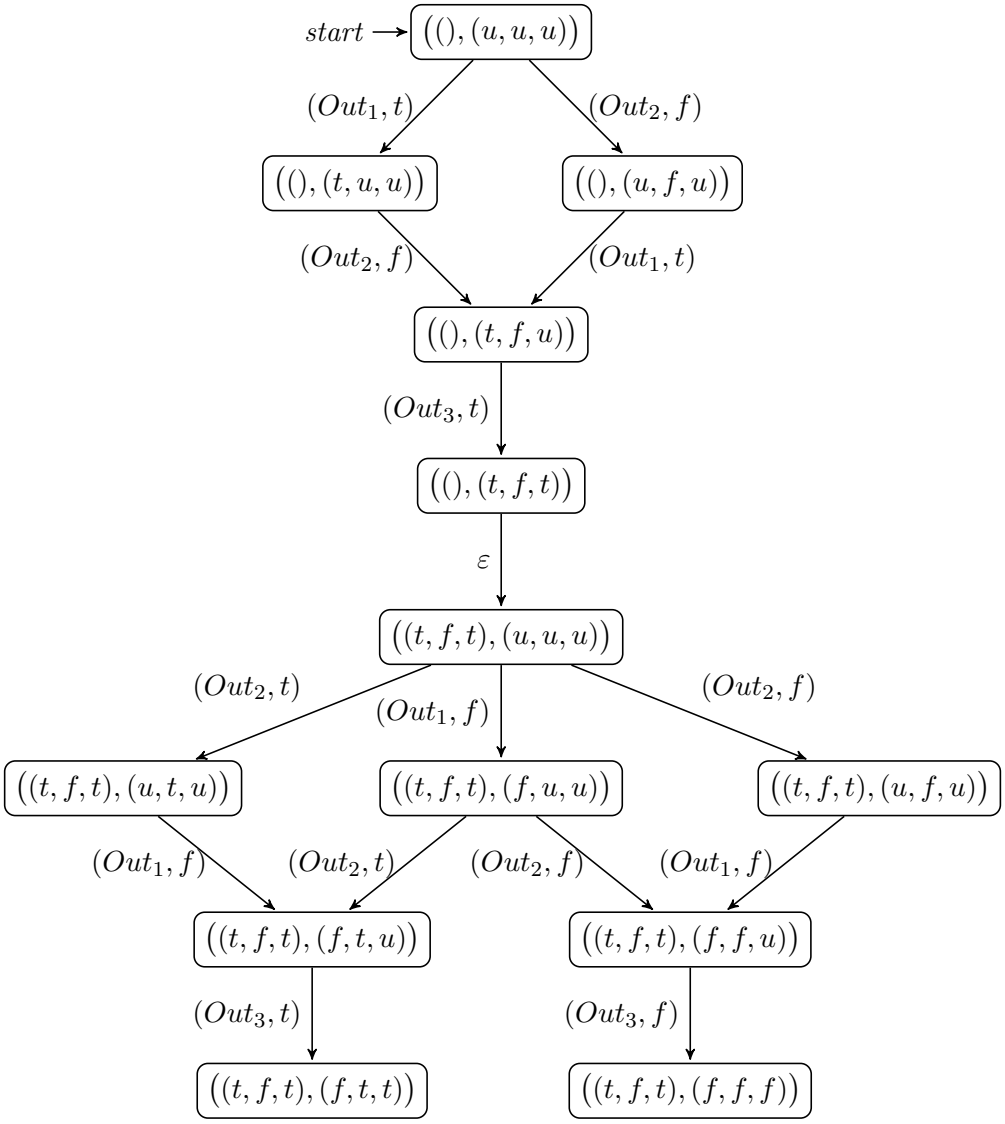
The partial order for the flows is $Out_1 \prec Out_3$ *and* $Out_2 \prec Out_3$*. The possible partial step for instant* 1 *are then* $PS((true, false, true)) = \{(undef, undef, undef), (tr$ $false, undef), (true, false, true)\}$*.*

For readability sake, we note t *for true,* f *for false and* u *for* $undef$ *in the LTS. Here is the LTS corresponding to the first step.*



The states correspond to $PS((true, false, true))$ *and the transitions are labelled by the name flow that becomes valued between the states and its value. Though it would be possible to value* Out_1 *and* Out_2 *at the same time (since they both have no dependency), there is no transition from* $((), (u, u, u))$ *to* $((), (t, f, u))$*, since it corresponds to the valuation of two flows at the same time. Nonetheless, it would be possible to optimise the graph by removing the "diamond" with a transition corresponding to two valuation, since the order is of no importance here (whatever is the ordering, the LTS ends up in state* $((), (t, f, u))$*). Another solution would be to choose one ordering that respects* $\prec$*, and use it throughout the graph. It makes the graph more compact, but it means that there are less possible ways of controlling the system.*

Let us prolong this LTS with the second step:



To go from instant 0 *to instant* 1*, an* ε*-transition finishes the previous step and starts a new one, with a fresh partial state.*

As we can see in the example, the system evolution LTS corresponding to finite traces are directed acyclic graphs (DAG). This is due to the fact that both the traces and the partial step can only "grow", i.e. the trace cannot become shorter and the partial step cannot have less valued flows than before.

We could introduce a more complex class of LTS to be able to represent a model as a cyclic LTS. They could be appended to DAG, since DAG are well suited to represent prefixes of traces. However, those cyclic LTS can ultimately be unfolded into (possibly infinite) DAG.

One could argue that we could drop the partial steps, and focus on the trace. However, it is not possible with a trace granularity to clearly define the possible "plays", as the values of some component flows depend on the values of other ones. Therefore, we need a finer granularity, namely the partial step.

5.3.3 Strategy synthesis

In this section we will present a formalism to compute the strategies suited for our problem that is close to the one proposed in [Asarin et al., 1995].

Definition 32 (Safe function ($safe$)) *Let $L = (Q, q^0, \Sigma, \rightarrow)$ be a LTS, $\Sigma_u \subseteq \Sigma$ a set of labels and $P \subseteq Q$ be a set of states. Let $Q_f = \{q \in P \mid \forall(\sigma, q') \in (\Sigma \times Q), (q, \sigma, q') \notin \rightarrow\}$ the set of final states of L.*
$safe(P) = \bigcap_{i=0}^{\infty} OK_i(P)$, *with* $OK_0(P) = P \cap Q$, $OK_{k+1}(P) = OK(OK_k(P))$
and $OK(X) = \{q \in X \mid$:

$$(q = q^0 \vee \exists q' \in X, \exists \sigma \in \Sigma, (q', \sigma, q) \in \rightarrow) \ \wedge \tag{5.3}$$

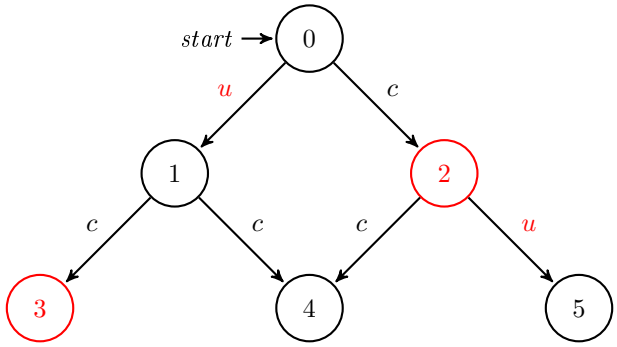
$$(q \notin Q_f) \implies \Big(\big(\forall \sigma \in \Sigma_u, \forall q' \in Q, (q, \sigma, q') \in \rightarrow \implies q' \in X\big) \wedge \big(\exists(\sigma', q'') \in (\Sigma, X), (q, \sigma', q'') \in \rightarrow\big)\Big)\} \tag{5.4}$$

Σ_u is the set of uncontrollable events.
Intuitively, given a set of traces, $safe$ returns a set of states included in P (i.e. states that verify a property) that cannot be exited using an uncontrollable transition, while being lively.
OK^{+1} enforces two properties. The first one corresponds to Constraint 5.3. It means that each state is either the initial state, or is the successor of a state in X. Constraint 5.4 means that either q is a final state ($q \in Q_f$), or it has a successor in X and every transition from q labelled by a symbol in Σ_u leads to a state in X.
If the considered LTS are DAG, the fact that each state has at least one predecessor and one successor means that $safe(P)$ returns a subset of states from P such that every state in $safe(P)$ is reachable from the initial state and some final state is reachable without the transition labelled by Σ_u leading outside of $safe(P)$. If the LTS are not DAG, if q^0 is in $safe(P)$, then some infinite cycle (the specification/composition) or final state is in $safe(P)$, as by definition, $safe(P)$ does not contain blocking state, nor state that have

no predecessor. Note that if the graph is not a DAG, there might be cycles that are not reachable from q^0, however, at least one cycle of final state is.
If the traces are infinite, there is no notion of final state, however, every state of an infinite trace always has a successor, therefore, $safe$ also works on infinite traces.
If the initial state is not in $OK_i(P)$, it does not make sense to pursue the search for the fix-point, since no "final" trace is reachable from the empty trace, so no system run is possible, and $safe(P)$ will converge to the empty set, or cycles not reachable from q^0 if the graph is not a DAG. Similarly, if every "final" states (or cycles for a non-DAG) are removed from $OK_i(P)$, the synthesis can stop, as it will converge to the empty set (for DAG) or non-reachable cycle (for non-DAG).

Example 22 *To illustrate the safe function, we will use arbitrary LTS that illustrates the functioning of the function, because the ones created from counter-factuals rapidly become big and hard to read.*

Let us consider the following LTS:

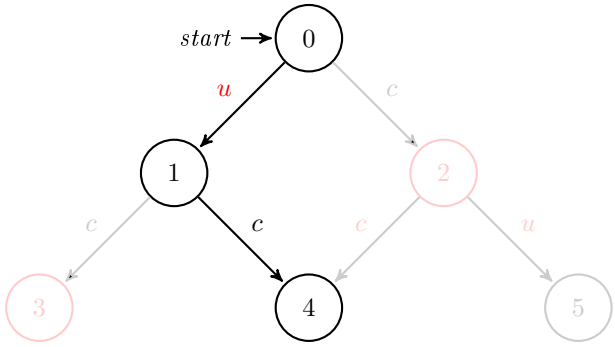


The controllable transitions are labelled by c, the uncontrollable are labelled by a red u. The "bad" states are red.

Here, $P = \{0, 1, 4, 5\}$. *Therefore,* $OK_0(P) = \{0, 1, 4, 5\}$.

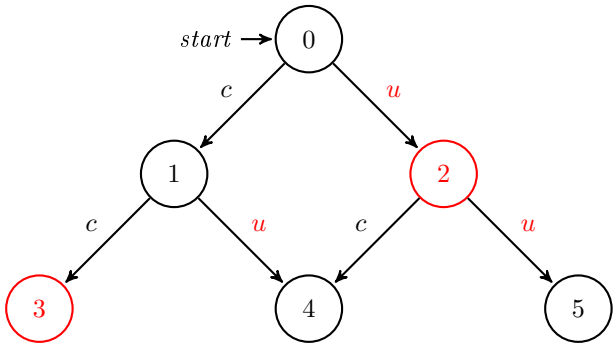
We then build $OK_1(P)$. 0 *remains in* $OK_1(P)$, *since its only uncontrollable successor,* 1 *is in* $OK_1(P)$. *The same goes for* 1, *since* 1 *is reachable from* 0, *its controllable outgoing transition leads to* 4, *that is in* $OK_1(P)$, *and the outgoing transition to* 3 *(not in* P*) is controllable (and thus can be forbidden).* 4 *is a final state and has* 1 *as predecessor, so* 4 *remains in* $OK_1(P)$. *Lastly,* 5 *is not reachable anymore, so it is removed from* $OK_1(P)$.

$OK_1(P) = \{0, 1, 4\}$, *which is the fix-point for this LTS. We get the following LTS, with the removed parts faded:*



We get $safe(P) = \{0, 1, 4\}$*, which contains at least one path from the initial state* 0*, to a final state, here,* 4*. Though* $safe(P)$ *is supposed to be a state set, it is more readable as a LTS, therefore the presented graph is the one where only the transition from* $safe(P)$ *to* $safe(P)$ *are kept.*

Let us consider a second LTS:



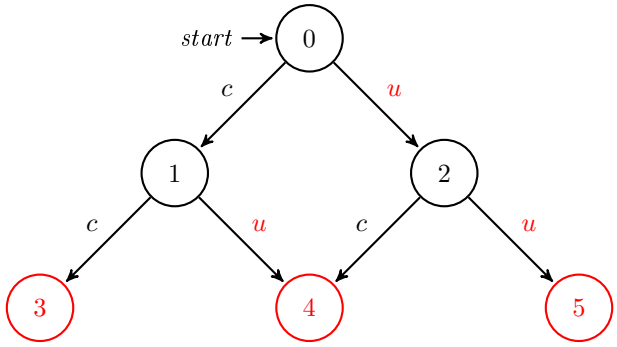
The difference with the first one is simply that the labelling of the outgoing transition from 0 *have been swapped.*

- $OK_0(P) = \{0, 1, 4, 5\}$.
- $OK_1(P) = \{1, 4\}$*, since* 0 *has an uncontrollable transition to* 2*, which is not in* $OK_1(P)$.
- $OK_2(P) = \{4\}$*, since* 1 *does not have a predecessor anymore.*
- $OK_3(P) = \emptyset$*, since* 4 *does not have a predecessor anymore.*

Then, $safe(P) = \emptyset$*. It shows that if the initial state is removed from* $OK_i(P)$*,* $safe$ *converges to the emptyset, since every state will eventually not have a predecessor for the DAG. We can then stop the computation of* $OK_i(P)$*, if no initial state remains in* $OK_i(P)$*. For graphs that are not DAG, it does*

not make sense to finish the computation if 0 *is not in* $OK_i(P)$*, since in our framework it would mean that this trace cannot be safely reached, since the LTS represents the traces growing in length.*

Lastly, let us consider another LTS:



Here all the final states are bad.

- $OK_0(P) = \{0, 1, 2\}$.
- $OK_1(P) = \{0\}$*, since* 1 *and* 2 *do not have any successor anymore.*
- $OK_2(P) = \emptyset$*, since* 0 *do not have any successor anymore.*

Then $safe(P) = \emptyset$*. It shows that the computation of* $OK_i(P)$ *can stop as soon as there is no final state or infinite path in* $OK_i(P)$ *anymore, since it will converge to the emptyset.*

Definition 33 (Strategy synthesis for LTS) *Let* $(Q, q^0, \Sigma, \rightarrow)$ *be a LTS,* $P \subseteq Q$ *be a safety property and* $\Sigma^c \uplus \Sigma^u = \Sigma$ *be a partition between controllable and uncontrollable labels. Let* Q_f *be the set of final states of the LTS. A strategy* $strat \subseteq (Q \times \Sigma^c)$ *to enforce* P *is synthesised as follow:* $\forall q \in safe(P)$,

$$strat(q) = \left\{\sigma \in \Sigma^c \mid \forall q' \in Q, \left((q, \sigma, q') \in \rightarrow\right) \implies q' \in safe(P)\right\}$$

Such that $\forall q \in safe(P), q \in Q_f \vee strat(q) \neq \emptyset$

A strategy ensures that for all state in $OK(P)$, all the controllable transitions that are enabled, and all the uncontrollable transitions, lead to a state in $OK(P)$. Whats more, it ensures that the controller is lively by ensuring that $strat(q)$ is not empty if q is not a final state.

Definition 34 (Mixed framework property ($\mathcal{P}_{mixed}$)) *Let $S = (\mathbb{F}, \mathbb{C}, \mathcal{P}, BM)$ be a mixed framework system. Let $LTS_{sys}(\mathcal{P}) = (Q, q^0, \Sigma, \rightarrow)$ be the LTS build from $\mathcal{P}$. $\mathcal{P}_{mixed} = Q$ is the set of states corresponding to the mixed framework property.*

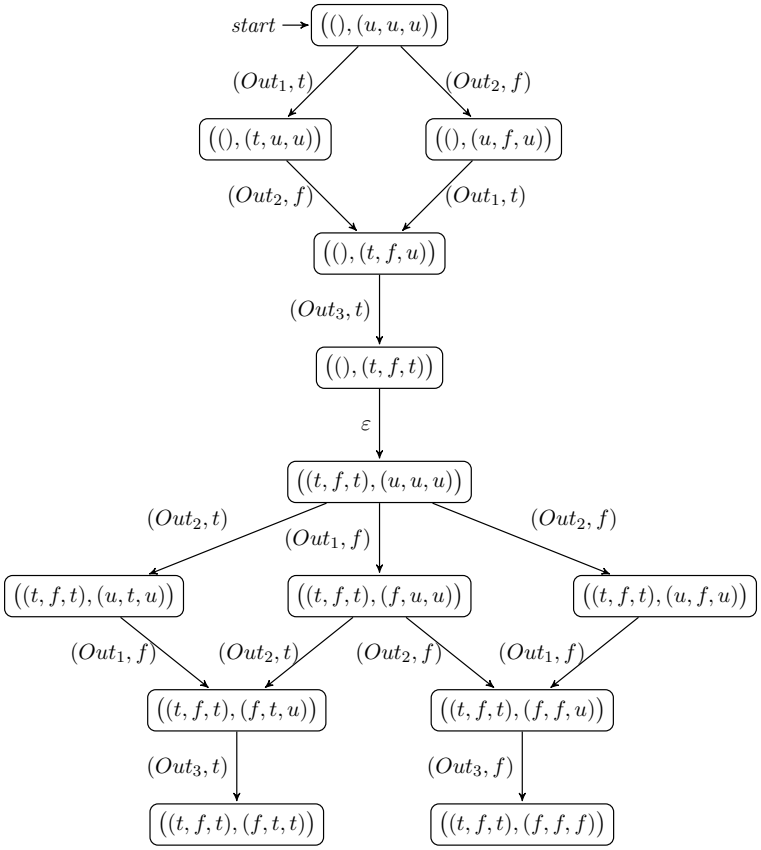
This is the translation of the property in the state space of the LTS. It represents all the states that respect the system property.
Note that since $\mathcal{P}$ is a safety property, it is prefix-closed, therefore $LTS_{sys}(\mathcal{P})$ only adds the partial step states to the property.

Definition 35 (Winning strategy ($\boldsymbol{WIN}_{mixed}$)) *Let CF be a counter-factual definition, that takes a trace and a set of suspected components in arguments. Let $\underline{tr}$ be a system trace, $\mathcal{I} \subseteq \mathbb{C}_B$ a set of suspected components and $\mathcal{R} \subseteq \mathbb{C}_W$ a set of protagonist. Let $L = LTS_{sys}(CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R}))$ be the LTS build from $CF_{mixed}(\underline{tr}, \mathcal{I}, \mathcal{R})$ and $\mathcal{P}_{mixed}$ is the LTS property build form $\mathcal{P}$. There is a winning strategy if $((), (undef)_{f \in \mathbb{F}}) \in safe(\mathcal{P}_{LTS})$, with $\Sigma_u = \bigtimes_{f \in \mathbb{E}} (D_f \cup \{undef\})\big)$, with $\mathbb{E} = \mathbb{F} \setminus \bigcup_{C \in \mathcal{R}} O_c$.*

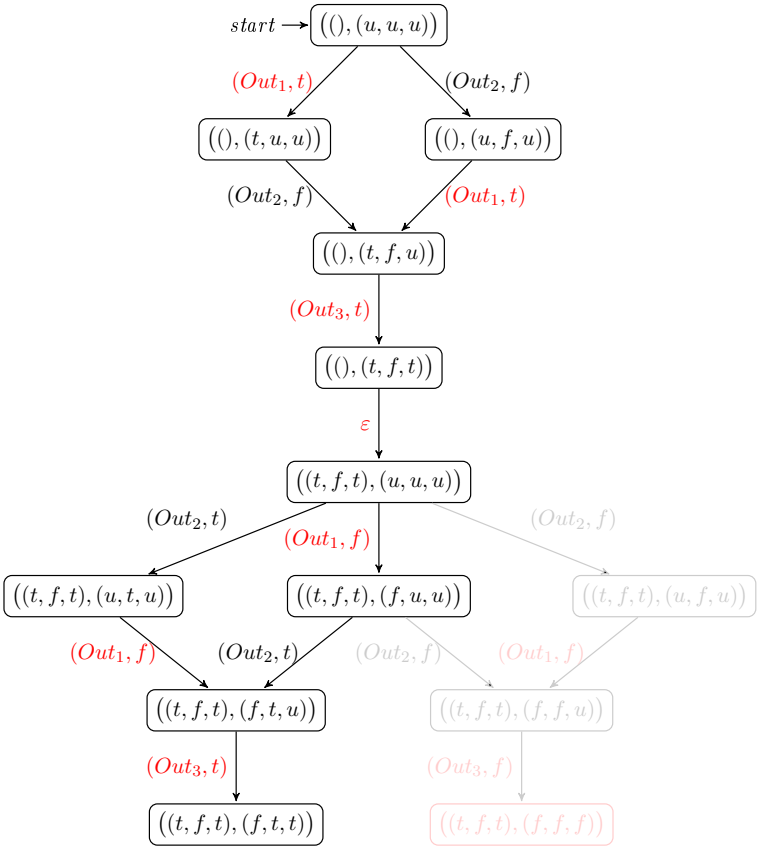
This formal definition is consistent with the one given earlier. The idea is that if $safe(\mathcal{P}_{LTS})$ contains the empty trace, there is a possible controller that ensures the system to remain in a subset of $\mathcal{P}_{LTS}$. It means that the controlled system is always non-failing, and the controller is non-blocking. The uncontrollable transitions are all the ones that do not correspond to an output from a component in $\mathcal{R}$. If the empty trace is not $safe(\mathcal{P}_{LTS})$, then the system cannot be controlled in a non-blocking fashion to ensure the system to be non-failing.

Let us illustrate the strategy synthesis with an example.

Example 23 *We consider the LTS produced in Example 21, on the or gate system:*



If we run the safe function on this LTS, with $\mathcal{R} = \{C_2\}$, we get:



As expected, this is controllable, if C_2 does not output false during the second instant. There is then a winning strategy.

Since we compute $safe$ to check if there is a winning strategy, we can actually build the winning strategy from $safe$, as shown in Definition 33.

Conclusion

This section developed a technique to mix white and black-box components in the causality approach. This approach uses the counter-factuals from the vanilla approach and builds on it. The result of the causality analysis gives some interesting information in a design/debugging framework.

A way of synthesising the strategy is also given.

As mentioned earlier, this framework cannot cope with responsibility assignment. The next Section will show how to deal with this issue.

Section 6

Game Framework for causality analysis

Section 5 introduced a framework that gives insight in a design perspective. However, the strategy synthesis supposes that a perfect knowledge of the system state is accessible to all the components. In a blaming framework, it is a problem, as it will be illustrated in the following example.

Example 24 *The system we consider is the simple three Boolean components setting from Example 16, where C_3 reflects that the outputs of C_1 and C_2 are equal.*

The failing trace is the following:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	*true*	*true*	*false*	*true*
Out_3	*true*	*true*	*true*	*false*

Let us build the counter-factuals for $\mathcal{I} = \emptyset$ and $\mathcal{R} = \{C_2\}$:

Time	*0*	*1*	*2*	*3*
Out_1	*true*	*true*	*false*	*false*
Out_2	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
Out_3	$Out_2[0]$	$Out_2[1]$	$\neg Out_2[2]$	$\neg Out_2[3]$

There is an obvious strategy, which is to always have $Out_2 = Out_1$.

The example showed that there is a possible strategy, which means that it is possible to add a controller, or modify the system such that the failure is avoided. However, C_2 is not supposed to have access to the value of Out_1

(since Out_1 is not an input of C_2), neither is it expected to be evaluated after Out_1 is valued. Even though C_2 is a necessary cause in the mixed framework, it does not make sense to hold C_2 responsible for the failure, as it does not have a local strategy that avoids the failure, with the information it has access to. What's more, holding it responsible assumes that Out_1 should be valued before Out_2, which is not necessary for the system to function as specified. It shows that we need a richer framework to be able to correctly assign responsibility, by both taking into account the information components have access to, and the actual constraints there are on the system.

This section will present such a framework. It will be divided in four subsections, in a similar fashion to Section 5. The first one will present the definitions we need to perform causality analysis in the game framework. The second one will adapt the causality definitions from Section 5.2, and will develop some new ones that are suited to a responsibility viewpoint. The third one will give a method to synthesise the strategies at a component level. The last one will present two ways of using this framework to find fixes.

6.1 Game Framework

As Example 24 showed, one of the issue of responsibility assignment lies in the fact that the model needs to be able to correctly assess the information each component has access to, in order to check whether the component had a way of avoiding the failure, given the information it had.

With this constraint in mind, we need to move to a framework that represents better how the system is actually evolving. Firstly, we move from the specification to "step function". There is extensive work in the literature about compiling synchronous dataflow system descriptions into actual code. A solution is to have a single-loop code that infinitely repeats itself, as discussed in [Halbwachs et al., 1991b]. This is the approach we chose here, as it is widely used, and specifically is the one used in Lustre.
The component specifications/models we now consider are abstractions of this single-loop model. We call those "single-loop" specifications/models step specifications/models.

The motivation to move to a "step function" framework is to better assess the information the component has access to when it computes its output. Indeed, in a responsibility mindset, it does not make sense to held a component responsible for a failure that could have been avoided would the component

have access to more information. The directing idea in this section is to be as cautious as possible when assigning responsibility, in particular, by having a finer grain model for the components.

Definition 36 (Step specification/model ($\hat{\mathcal{S}}$)) *Let I, O and M be distinct sets of flows. A Step specification/model is a function that takes an input from $D_I \times D_M$ and outputs a set of elements from $D_O \times 2^{D_M}$, with $D_{\mathbb{F}} = \bigtimes_{f \in \mathbb{F}} D_f$, such that:*

$$\forall (i, mem) \in (D_I \times MEM), \hat{\mathcal{S}}(i, mem) \neq \emptyset$$

With MEM the greater set such that $\forall mem \in MEM, \exists (i, mem') \in (D_I \times MEM), \exists o \in D_O, (o, mem) \in \hat{\mathcal{S}}(i, mem')$

Given a current memory state and inputs, the step specification returns all the possible outputs, alongside all the possible next memory states, for each possible output. It describes more realistically what the component has access to, while computing its output, than $\mathcal{S}$. The constraint is here to ensure that the step specification is $D_I \times MEM$ accepting, i.e. it has at least one possible output for each input and each possible memory states that $\hat{\mathcal{S}}$ can produce.
MEM is introduced because there is no need to be accepting for each possible memory state. Indeed, if a component is an integer memory flow that ends up taking its value in [0..3], it does not make sense to constraint the step specification to be accepting for each possible integer.

Given a component specification/model $\mathcal{S}$, we suppose we are able to infer the corresponding step specification/model $\hat{\mathcal{S}}$ such that recursively executing $\hat{\mathcal{S}}$ builds exactly $\mathcal{S}$. It is important to be able to go from the specifications to the step specifications, in order to compare the vanilla and the mixed framework to this one. Generally speaking, the system is not specified using a set of traces, and are also generally compiled into a single-loop code. Therefore, it is very likely to actually have access to a one step specification.
In the case of a step specification (i.e. for black-box components), we also suppose that M is minimal. This minimality criterion is here to ensure that $\hat{\mathcal{S}}$ uses the minimum memory possible to be able to compute the next step. It means that the implementation of the specification/actual system has, at least, as much information as $\hat{\mathcal{S}}$. This is important, because, as we do not know how the specification has been implemented in the actual system, we must use the "worst case scenario", in term of how much information the component has. Note that the minimality criterion definition here is a bit

fuzzy, given the generality of the framework. It could be the number of memory flows. However, some memory flows might represent complicated data structures. It could be the cardinal of D_M. However, if one chooses integer for a memory flow that is only three valued, any arbitrary number of three valued flows will make for a smaller domain, though retaining way more information. Anyhow, the idea is that the component has access to the minimal information necessary to ensure it is able to enforce its specification. E.g. a component that returns the previous value of its input will only have one memory flow, of the same type as the input, not a sequence of all the previous values. The sequence of all previous input values ensures to be able to enforce the specification, but only keeping the previous one is the minimal amount of data needed to ensure the specification, and thus the one we assume the component has access to.
For the models, we have access to the actual behaviour, and therefore the internal memory used in the system. The memory state is the one used in the system, and not necessarily the minimal one.

In practice, the specifications/models are not designed as sets of traces, but as set of constraints, programs,... A specification in LUSTRE is already a one step specification. Therefore, it is easy to have access to the "one step" specification.

Example 25 *This example will show a step specification derived from a specification, and a step model, as well as a step model.*

Let us consider the usual pump. The definition is the following:

- $I_{pump} = \{(com_{pump}, \mathbb{N})\}$
- $O_{pump} = \{(out_{pump}, [0..2])\}$
- $\mathcal{S}_{pump} = \{\underline{tr} \in \mathcal{B}_{pump} \mid \forall i \in [0..|\underline{tr}|], \pi_{out_{pump}}(\underline{tr}[i]) = \pi_{com_{pump}}(\underline{tr}[i])\}$

The corresponding step specification is:

- $I_{pump} = \{(com_{pump}, \mathbb{N})\}$
- $O_{pump} = \{(out_{pump}, [0..2])\}$
- $mem_{pump} = \emptyset$
- $\hat{\mathcal{S}}_{pump} = (out_{pump} = com_{pump})$

Here, the translation from the specification to the step specification is simple, since the pump is memoryless. It just consists in removing the universal quantifiers on the specification.

Let us now consider a more complex component, namely a white-box tank, which outputs some liquids, corresponding to a command out_{tank} and that can dump one unit of liquid, at each time-step.

- $I_{tank} = \{(out_{pump}, [0..2]), (out_{tank}, [0..2])\}$
- $O_{tank} = \{(vol_{tank}, \mathbb{N})\}$
- $mem_{tank} = \{(pre\_vol_{tank}, \mathbb{N}), (dump, [0..1])\}$, *with* pre_vol_{tank} *the previous value of* vol_{tank}. *We note* a' *for the previous value of* a.
- $\hat{\mathcal{S}}_{tank}$ *is such that:*
  - $dump = random([0..1])$, *with random that randomly returns a value from a set.*
  - $vol_{tank} = pre\_vol'_{tank} + out_{pump} - dump - out_{tank}$
  - $pre\_vol_{tank} = vol_{tank}$

We note f' for the previous value of f. Here, the dump variable represents the non-deterministic part of this white-box component, namely how much liquid the tank dumps at a given time-step. It is not necessary to create a dump variable, we can just replace dump by $random([0..1])$ in the definition of vol_{tank}. However, in order to ease the comprehension of the examples, the "non-determinism" will be made explicit, by using variables that reflects it, such as dump in this component.

Note that the order in which the statements in the specification are evaluated is important, as dump is used in vol_{tank}, and vol_{tank} in $pre\_vol_{tank} = vol_{tank}$.

Definition 37 (Step component (C)) *A step component is a tuple $C = (I, O, M, \hat{\mathcal{S}})$, with I the set of input flows, O the set of output flows, M the set of internal memory flows, and $\hat{\mathcal{S}}$ a one step specification/model.*

We simply add a set of memory flows to the tuple, to reflect the fact that they need to be added, to go from the specification/model framework to the step one.

Definition 38 (Game system (S)) *A game system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ is a mixed system (i.e. $\mathbb{C} = \mathbb{C}_B \uplus \mathbb{C}_W$) such that the components are step components, $\mathbb{F} = \bigcup_{C \in \mathbb{C}} (I_C \cup O_C \cup M_C)$, $\mathcal{P} \subseteq \mathcal{B}_{\mathbb{F}'}$ and $BM \subseteq \mathcal{B}_{\mathbb{F}'}$, with $\mathbb{F}' = \bigcup_{C \in \mathbb{C}} (I_C \cup O_C)$*

A game system is basically a mixed system for which the components have a step specification/model instead of a specification. This change is passed on to $\mathbb{F}$, since the internal memory is added to the set of flows of the system. However, the specification and behavioural models remain on the "initial" set of flows (namely $\mathbb{F}'$), since they reflect a global view-point, and do not have access to the internal state of the components.

Definition 39 (Causality game $\mathcal{G}$) *Given a game system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, a causality game $\mathcal{G}$ is a tuple $(\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P}')$, where*

- *$\underline{tr} \in (\mathcal{B}_{\mathbb{F}'} \setminus \mathcal{P})$ is the faulty trace (with $\mathbb{F}'$ defined as in Definition 38).*
- *$\mathcal{M} \subseteq \mathbb{C}_B$ is a set of black-box components.*
- *$\mathcal{R} \subseteq \mathbb{C}_W$ is a set of white-box components which are the protagonists.*
- *$\mathcal{A} \subseteq \mathbb{C}_W$ is a set of white-box components which are the antagonists.*
- *$\mathcal{P}' \subseteq \mathcal{B}_{\mathbb{F}'}$ is a safety property over the possible traces of the system.*

Such that $\mathcal{R} \cap \mathcal{A} = \emptyset$.

A causality game is a two players game, where the protagonist controls all the components in $\mathcal{R}$, and similarly, the antagonist controls all the components in $\mathcal{A}$. The goal of $\mathcal{R}$ is to ensure that $\mathcal{P}'$ remains true on all the traces built, given $\underline{tr}$ and $\mathcal{M}$. The goal of $\mathcal{A}$ is to ensure that $\mathcal{P}'$ will eventually be violated.
$\underline{tr}$ and $\mathcal{M}$ are the usual arguments we use to build the counter-factuals ($\mathcal{M}$ corresponds to $\mathcal{I}$, it is renamed to "modified", as it does not necessarily represent the suspects). $\underline{tr}$ is the initial faulty trace. $\mathcal{M}$ is a set of black-box components for which we will repair the faulty behaviour in $\underline{tr}$. The result of the counter-factuals reconstruction is a set of traces "consistent" with $\underline{tr}$, where the behaviour of the components in $\mathcal{M}$ is non-faulty.
As in the mixed-framework, $\mathcal{R}$ is a set of white-box components for which we authorise more behaviours than the one in $\underline{tr}$, i.e. behaviours that are consistent with the reconstructed traces from $\underline{tr}$, but not necessarily the one

chosen in $\underline{tr}$. $\mathcal{A}$ works in a similar fashion.
The shift to a game framework is motivated by the ability to have a set of components, $\mathcal{A}$, that are actively trying to make the system fail.
Note that this game definition does not explain how the game is played, but rather the setup of the game. We still need a one more definition before being able to actually explain how to "play" the game.

Definition 40 (Strategy *strat***)** *Let $C = (I, O, M, \hat{\mathcal{S}})$ be a step component, a strategy strat for this component is a step specification such that $strat \subseteq \hat{\mathcal{S}}$.*

A strategy is a restriction of the step specification that remains a step specification. It means that the "amount" of non-determinism is reduced, but a strategy remains I accepting.

In the white-box *tank* from Example 25, a strategy is more restrictive way of choosing *dump* than a total random. For instance, *dump* could the parity of out_{pump}.

Note that this definition of strategy explains why $\mathcal{A}$ and $\mathcal{R}$ do not form a partition of the white-box components because the behaviour of a "player" component can be different from the one in the initial trace $\underline{tr}$.

Even though the component strategy only relies on the information the component has access to, the choice of the strategies is coordinated. E.g. the strategies for all the components in $\mathcal{R}$ (similarly $\mathcal{A}$) are chosen as a whole, even if each strategy is local to the component. It is motivated by the fact that we want to check the responsibility of sets of components, that can choose strategies as a set, but perform the strategy locally. It is in the same line of thought as the remarks raised by Example 24, we want to blame a set of components on the information it had during the system run, and not on information it does not have access to.

Definition 41 (Outcomes *outcome***)** *Given a game system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ and a game $\mathcal{G} = (\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P})$ and $strat = \{strat_C\}_{C \in \mathcal{R} \cup \mathcal{A}}$ be a set of strategies for each protagonist and antagonist component. Let $MEM = \bigtimes_{f \in \mathbb{E}}(D_f)$ with $\mathbb{E} = \bigcup_{C \in \mathbb{C}}(mem_C)$. We suppose we are given the initial memory state $mem^0 \in MEM$. We also suppose we are given a similar$(\underline{tr}, \mathbb{TR})$ function from $(BM \times 2^{BM}) \times 2^{BM}$ that given a trace $\underline{tr}$ and a set of traces returns the set of traces that are the most similar to $\underline{tr}$.*
outcome$(\mathcal{G}, strat)$ builds all the possible traces, as follow:
$\mathbb{TR}^0 = \{((), \{mem^0\})\}$, *with $()$ the empty trace.*

The traces are then extended as follow:

- *We suppose* $\mathbb{TR}^i \in 2^E$ *with* $E = BM \times 2^{MEM}$ *has already been computed.*

- *Let* $POSSIBLE(\mathbb{TR}^i) = \bigcup_{(\underline{tr}', MEM_{\underline{tr}'}) \in \mathbb{TR}^i} (possible((\underline{tr}', MEM_{\underline{tr}'})))$ *With possible such that:*
  $\forall (\underline{tr}', MEM_{\underline{tr}'}) \in E, possible((\underline{tr}', MEM_{\underline{tr}'})) =$

$$\{(\underline{tr}'', MEM_{\underline{tr}''}) \in E \mid$$

$$|\underline{tr}''| = i + 1 \wedge \underline{tr}''[0..i-1] = \underline{tr}'[0..i-1] \wedge \tag{6.1}$$

$$\forall f \in \mathbb{F}, (\forall C \in \mathbb{C}, f \notin \mathbb{O}_C \wedge i < |\pi_f(\underline{tr})|) \implies (\pi_f(tr''[i]) = \pi_f(tr[i])) \wedge \tag{6.2}$$

$$\forall C \in (\mathcal{R} \cup \mathcal{A}),$$
$$\forall mem \in \pi_C(MEM_{\underline{tr}''}), \exists mem' \in \pi_C(MEM_{\underline{tr}'}) \ s.t.$$
$$(\pi_{O_C}(tr''[i]), mem) \in strat_C(\pi_{I_C}(tr''[i]), mem') \wedge \tag{6.3}$$

$$\forall C \in (\mathbb{C}_{nf}),$$
$$\forall mem \in \pi_C(MEM_{\underline{tr}''}), \exists mem' \in \pi_C(MEM_{\underline{tr}'}) \ s.t.$$
$$(\pi_{O_C}(tr''[i]), mem) \in \hat{\mathcal{S}}_C(\pi_{I_C}(tr''[i]), mem')\} \tag{6.4}$$

  with $\mathbb{C}_{nf} = \big(\mathbb{C}_W \setminus (\mathcal{R} \cup \mathcal{A})\big) \cup \mathcal{M} \cup \{C \in (C_B \setminus \mathcal{M}) \mid \pi(tr[0..k+1] \in \mathcal{S}_C\}$,
  with $k = max(\{l \in [0..i-1] \mid \pi_C(\underline{tr}[0..l] = \pi_C(\underline{tr}'[0..l]\})$

- $\mathbb{TR}^{i+1} = \{\underline{tr}', MEM_{\underline{tr}'}) \in POSSIBLE(\mathbb{TR}^i) \mid$

$$\underline{tr}' \in similar(\underline{tr}, TRACES^{i+1})\}$$

  With $TRACES^{i+1} = \bigcup_{(\underline{tr}', MEM_{\underline{tr}'}) \in POSSIBLE(\mathbb{TR})} (\{\underline{tr}'\})$.

outcomes$(\mathcal{G}, strat_{\mathcal{R}}, strat_{\mathcal{A}}) = \mathbb{TR}^k$ *such that* $\mathbb{TR}^k$ *cannot be extended anymore.*

The *outcome* function computes all the traces, given a game, an initial trace and a set of strategies.
The idea is to start from the empty trace, alongside the initial memory state (mem^0), which correspond to the memory state of the different components when they are initialised. The trace is then extended by one step, using

$POSSIBLE$. The output of $POSSIBLE$ is the set of all possible extensions of the input set of traces by one step. This set of possible traces is then filtered using *similar*, to keep only the traces that are the most consistent with the initial failing traces. The process of extension is then repeated. This extension assumes that the traces can be extended indefinitely, or that a maximum length has been chosen. It is not always possible to extend a trace, but taking it into account in this definition would further complicate this already involved definition. It would be easy to add another filtering that allow to keep traces that cannot be extended, to tackle this issue.
possible is built to respect certain rules. The constraint 6.1 is here to ensure that we are just extending the traces from $\mathbb{TR}^i$ by one time-step. The constraint 6.2 ensures that the input of the systems (flows that are not "plugged" to the output of any component) have the same values as in the initial trace $\underline{tr}$ (if there is one). Constraint 6.3 ensures that the "players", i.e. the components that are protagonist, or antagonist, respect the strategy they have chosen. Constraint 6.4 ensures that the white-box components which are not "players", the components that should be repaired (i.e. in $\mathcal{M}$) and the black-box components that respect their specification before their behaviour being modified, respect their specification. This is what C_{nf} (components non-faulty) represents: the non-player white-box ($\big(\mathbb{C}_W \setminus (\mathcal{R} \cup \mathcal{A})\big)$), the "modified" black-box components ($\mathcal{M}$) and the non-faulty black-box ones. The non-faulty black-box components are the one such that they were non-faulty the first time their behaviour was modified. k represents the last instant $\underline{tr}$ and $\underline{tr}'$ coincide on C, and at $k+1$, C was not faulty in $\underline{tr}$. This construction of C_{nf} ensures the no fault introduction requirement (2). Note that this check is made for every $\underline{tr}'$, it means that C could be non-faulty in a given trace that is being prolonged, but not in every prolonged ones. This is the main reason this framework builds some more accurate counter-factuals than the ones that use a notion of cone/unaffected prefixes. This approach is closer to the framework introduced in [Gössler and Stefani, 2016].
This operation gives us a set of traces, $POSSIBLE(\mathbb{TR}^i)$, that are all the possible extensions of the traces in $\mathbb{TR}^i$, and that respect the constraints above, alongside the possible memory states for those extensions. We now select only the traces that are the most similar to the initial trace. This similarity can be that the maximum number of flow values are similar to the initial trace, or that the maximum number of component behaviours are the same as in $\underline{tr}$. This similarity function correspond to 5 (Conservation of the non-impacted trace parts).
The state of the internal memories of the components are constrained by 6.3 and 6.4. For each possible extension $\underline{tr}''$ of a trace $\underline{tr}'$, one element of the projection of $MEM_{\underline{tr}''}$ over a component C must be consistent with one of

the previous possible memory states from $MEM_{\underline{tr}'}$. This ensures that all the possible memory states are generated, thus building as many traces as possible, while keeping a consistency with what information the components have. Note that the *similar* function does not take the memory state into account, as it only uses the traces in $TRACES(\mathbb{TR}^i)$, i.e. the trace extended, without the possible memory states. It is motivated by the fact that we do not have access to the internal state of the system from the initial trace $\underline{tr}$, and therefore, it cannot be used to assess the similarity to this initial trace. In this definition, *similar* is applied at each step. It means that less traces will be generated at each step, but that the similarity is "local", with respect to the time-step. It is also possible not to apply *similar* at each step, but only to the final set of traces. Overall, it will yield outcomes that are closer to the initial trace, as traces that were not the closest ones at a given instant might end up being the closest ones when fully extended. However, applying similar at the end means way more computation cost and memory usage.
The result of *outcome* corresponds to the output of CF_{mixed}, from the mixed framework, where the behaviour of the components in $\mathcal{R}$ and $\mathcal{A}$ is constraint by a strategy. Note that if we chose $\hat{\mathcal{S}}_C$ for those components, the results are the same as CF_{mixed} (all the possible behaviours of $\mathcal{R}$ and $\mathcal{A}$ that are consistent with $\underline{tr}$) but with a counter-factual definition that corresponds to *outcome*.

We note Π_C the set of all possible strategies for the component C, and $\Pi_{\mathbb{C}} = \bigtimes_{C \in \mathbb{C}} (\Pi_C)$ for the Cartesian product of all the sets of strategies of the components in $\mathbb{C}$.

Definition 42 (Winning strategy) *Given a causality game $\mathcal{G} = (\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P})$. $\mathcal{R}$ has a winning strategy $strat_{\mathcal{R}} \in \Pi_{\mathcal{R}}$ if $\forall strat_{\mathcal{A}} \in \Pi_{\mathcal{A}}, outcome(\mathcal{G}, strat_{\mathcal{A}} \cup strat_{\mathcal{R}}) \subseteq \mathcal{P}$*

Given a causality game $\mathcal{G}$, we define *WIN*($\mathcal{G}$) a Boolean function which returns *true* if their is a winning strategy for the protagonist in the causality game. It means that *WIN*($\mathcal{G}$) is *true* if $\mathcal{R}$ can always prevent the system from going in a state that violates $\mathcal{P}$. This definition is the extension of Definition 23 to the game framework.

Definition 43 (Spoiling strategy) *Given a causality game $\mathcal{G} = (\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P})$. $\mathcal{A}$ has a spoiling strategy $strat_{\mathcal{A}} \in \Pi_{\mathcal{A}}$ if $\forall strat_{\mathcal{R}} \in \Pi_{\mathcal{R}}, sup(outcome(\mathcal{G}, strat_{\mathcal{A}} \cup strat_{\mathcal{R}})) \cap \mathcal{P} = \emptyset$, with $sup(\mathbb{TR}) = \{\underline{tr}' \in \mathbb{TR} \mid \forall \underline{tr}'' \in TR, \underline{tr}' = \underline{tr}'' \vee \neg(\underline{tr}'' \sqsubseteq \underline{tr}')\}$*

Given a causality game $\mathcal{G}$, we define $SPOIL(\mathcal{G})$ a Boolean function which returns *true* if there is a spoiling strategy for the antagonist in the causality game. It means that $\mathcal{A}$ can always eventually force the system in a failing state where $\mathcal{P}$ is no longer verified. This definition is the extension of Definitions 24 to the game framework. It is necessary to define $SPOIL$, as it cannot be build from WIN and tweaking the game, because of the eventuality notion that is introduced with the *sup*. Contrary to Definition 24, where there was no real notion of strategy (thus the name spoiling behaviour), as no set of components was actively trying to make the failure happen, here $\mathcal{A}$ tries to create a system failure.

Example 26 *Let us consider a system composed of a controller, two pumps and a tank.*

The pumps are the usual ones ($pump_1$ and $pump_2$). When they are faulty, their output value never changes (which is reflected in BM).

The tank is the one described in Example 25, with two pumps instead of one:

- $I_{tank} = \{(out_{pump_1}, [0..2]), (out_{pump_2}, [0..2]), (out_{tank}, [0..4])\}$
- $O_{tank} = \{(vol_{tank}, \mathbb{N})\}$
- $mem_{tank} = \{(pre\_vol_{tank}, \mathbb{N}), (dump, [0..1])\}$
- $\hat{\mathcal{S}}_{tank}$ *is such that:*
  - $dump = random([0..1])$, *with random that choose randomly a value in a set.*
  - $vol_{tank} = pre\_vol'_{tank} + out_{pump_1} + out_{pump_2} - dump - out_{tank}$
  - $pre\_vol_{tank} = vol_{tank}$

The property $\mathcal{P}$ is that out_{tank} should be between 0 and 5.
The controller is as follow:

- $I_{control} = \{(vol_{tank}, \mathbb{N}), (out_{pump_1}, [0..2]), (out_{pump_2}, [0..2]), (out_{tank}, [0..4])\}$
- $O_{control} = \{(com_{pump_1}, [0..2]), (com_{pump_2}, [0..2])\}$
- $mem_{control} = \{(pump_1\_fault, \mathbb{B}), (pre\_pump_1\_fault, \mathbb{B}), (pump_2\_fault, \mathbb{B}), (pre\_pump_2\_fault, \mathbb{B}), (pre\_vol_{tank}, \mathbb{N}), (pre\_out_{pump_1}, [0..2]), (pre\_out_{pump_2}, [0..2])\}$, *with pre_name the precedent value of name and $pump_i\_fault$ a variable chosen by the controller to reflect the fact that it believes $pump_i$ to be faulty.*

- *The specification is complicated. However, the idea is to ensure that* vol_{tank} *should be as close as* 3 *as possible at the end of the instant.* $pump_1\_fault = pump_2\_fault = false$*, the controller will command to both pump to output a flow. If one of the two is true, the controller will take into account that it cannot change the value of the broken pump. If both are true, the controller does not have any impact on the system anyways.*

Note that we consider that $pump_1\_fault$ *and* $pump_2\_fault$ *are non-deterministic. However, the specification is deterministic, while the inputs values and the two internal values have been chosen.*

Let us consider the following trace:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*2*	*2*	*2*
$pump_1\_fault$	*false*	*false*	*false*	*false*	*false*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*1*	*1*	*1*
out_{pump_1}	*2*	*2*	*2*	*2*	*2*
com_{pump_2}	*2*	*2*	*1*	*1*	*1*
out_{pump_2}	*2*	*2*	*1*	*1*	*1*
dump	*false*	*false*	*false*	*false*	*false*
vol_{tank}	*0*	*3*	*4*	*5*	*6*

At instant 2*, the first pump becomes faulty, and stuck outputting* 2*. This ends up causing a system failure after* 3 *instants. However, would the* $pump_1\_fault$ *have actually reflected the fact that* $pump_1$ *was faulty, the command would have avoided the failure by taking the stuck value into account. Similarly, the tank could have dumped some liquid to prevent the overflow.*

Since we make the non-determinism of the white-box components explicit, we can use them in the similar function of the play-through. The similar function returns the traces of possible where the choices remains as close as possible and the black-box output values are kept as is. I.e. $similar(possible, \underline{tr}) = \{\underline{tr}' \in possible \mid \forall \underline{tr}'' \in possible, maintained(\underline{tr}', \underline{tr}) \geqslant maintained(\underline{tr}'', \underline{tr})\}$*, with* $maintained(\underline{tr}, \underline{tr}') = |\{f \in \mathbb{F} \cup ND \mid \pi_f(\underline{tr}[|\underline{tr}| - 1]) = \pi_f(\underline{tr}'[|\underline{tr}| - 1])\}|$*, with* ND *the set of flows representing the non-determinism of the non-playing white-box components (in* $\mathbb{C}_W \setminus (\mathcal{R} \cup \mathcal{A})$*). For instance,* $ND_{control} = \{(pump_1\_fault, \mathbb{B}), (pump_2\_fault, \mathbb{B})\}$*. maintained basically counts the number of flows for which that values are the same between the two traces.*

Let us consider the outcome if the strategy strat for control is to chose the correct value for $pump_1\_fault$, $outcome((\underline{tr}, \emptyset, \{control\}, \emptyset, \mathcal{P}), \{strat\}, \emptyset)$:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*2*	*2*	*2*
$pump_1\_fault$	*false*	*false*	*true*	*true*	*true*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*2*	*2*	*2*
out_{pump_1}	*2*	*2*	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	*2*	*2*	*0*	*0*	*0*
out_{pump_2}	*2*	*2*	*0*	*0*	*0*
dump	*false*	*false*	*false*	*false*	*false*
vol_{tank}	*0*	*3*	*3*	*3*	*3*

Since the $pump_1\_fault$ *is the right one, control adapts its commands accordingly and the volume in the tank, that remains at 3 after instant* 1*, as is aimed at. Note that tank does not changes its strategy, as reducing its output by one would not make it closer to the initial trace. Therefore, there exist a winning strategy, so* $WIN((\underline{tr}, \emptyset, \{control\}, \emptyset, \mathcal{P}))$.

Now, let us consider the outcome if we fix $pump_1$*, and that the strategy* $strat_{tank}$ *for the tank is to systematically dump,* $outcome((\underline{tr}, \{pump_1\}, \{control\}, \emptyset, \mathcal{P}), \{strat_{tank}\}, \emptyset)$

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*2*	*2*	*2*
$pump_1\_fault$	*false*	*false*	*false*	*false*	*false*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*2*	*2*	*2*
out_{pump_1}	*2*	*2*	*2*	*2*	*2*
com_{pump_2}	*2*	*2*	*1*	*1*	*1*
out_{pump_2}	*2*	*2*	*1*	*1*	*1*
dump	*true*	*true*	*true*	*true*	*true*
vol_{tank}	*-1*	*2*	*2*	*2*	*2*

At the first instant, vol_{tank} *has the value* -1*, thus violating the specification. Afterwards,* vol_{tank} *has the value* 2*, as the controller does not know that one unit of liquid will be dumped, and aims at* 3*. This trace is the closest to the initial one, since the choices are the same for control, and the outputs of the pumps are the same as in the initial trace. Therefore, this is a spoiling strategy, and* $SPOIL((\underline{tr}, \emptyset, \{control\}, \emptyset, \mathcal{P}))$.

Considering only the black-box aspect, this approach is similar to the one in [Gössler and Stefani, 2016], where the approach starts by removing

the faults (and not the suffixes) and then grafting the removed parts, and the one impacted by the changes, with correct behaviours. Here, we do that in one pass, through Constraint 6.4 (the one on black-box and non playing white-box components) and the *similar* function. However, the approach in [Gössler and Stefani, 2016] is able to keep bigger parts of the trace, by changing the trace in an early instant, if it means a reconstructed trace closer to the initial one. Since we build the trace incrementally here, we do not have this way of taking into account the "future" of the trace. However, when applying the *similar* function at the end of the trace extension, the result should be analogous.

6.2 Causality definitions

This paragraph will present causality definitions adapted from the mixed framework, and some new ones that the game framework enables. As for Section 5.1, the result of causality analysis will be discussed from a design perspective. In addition, they will be analysed from a responsibility point of view as well. As explained earlier, this game framework is largely motivated by the fact that the mixed framework does not allow us to assess the responsibility of the sets of components, since the synthesis is global. This is the usual responsibility assignment, with both black-box and white-box components, instead of only back-box ones.

The design setting we consider is the same as in Section 5.1. Concerning the responsibility assessment setting, we suppose that a failing trace is given, as well as black and white-box component specifications/models and a system property, and that we try to assess the responsibility of sets of components in the failure.

In those definitions, we suppose we are given a game system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$, and a faulty trace $\underline{tr} \in BM \setminus \mathcal{P}$.

Definition 44 (Necessary cause (nec)) *Let $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected components, $\mathcal{I}$ is a necessary cause, noted* $\mathrm{nec}(\underline{tr}, \mathcal{I})$*, if WIN($\mathcal{G}$), with* $\mathcal{G} = (\underline{tr}, \mathcal{I} \cap \mathbb{C}_B, \mathcal{I} \cap \mathbb{C}_W, \emptyset, \mathcal{P})$.

Intuitively, the set suspects are a fix for the system. For the black-box components, the idea is the same as the usual approach, repairing them fixes the system. On the white-box component side, it means that they had a set of local strategies that could have avoided the failure. Combined, it means that, would the black-box suspects have not been faulty, the white-box suspects had a way of avoiding the failure. $\mathcal{I} \cap \mathbb{C}_W$ can force the system to behave according to $\mathcal{P}$, if $\mathcal{I} \cap \mathbb{C}_B$ is fixed.

If $\mathcal{I} \subseteq \mathbb{C}_B$, and $CF_{black-box}(\underline{tr}, \mathcal{I}) = outcome((\underline{tr}, \mathcal{I}, \emptyset, \emptyset, \mathcal{P}), \emptyset, \emptyset)$, this definition gives the same result as the black-box one.
If $\mathcal{I} \cap \mathbb{C}_B = \emptyset$, it means that the white-box components in $\mathcal{I}$ could have chosen local strategies to ensure that the system would not fail, even with all the faulty component left as is.

From a designer viewpoint, it means that the models can be modified locally to be able to avoid the failure, as long as the black-box components in $\mathcal{I}$ are fixed.
From a responsibility assignment view-point, it means that there was a way of avoiding the failure, by doing forbidding some non-deterministic options, i.e. a winning local strategy was available, for the white-box components from $\mathcal{I}$ and repairing the black-box from $\mathcal{I}$. It means that the failure could have been avoided. However, $\underline{tr}$ might be a crazy scenario, which was not really foreseeable during the design phase. If not, it shows a defect from the designer, and assigning responsibility makes sense.

Let us illustrate this definition with an example.

Example 27 *The system considered and the trace is the one of Example 26.*

As show in Example 26, the controller had a winning strategy, therefore, $\text{nec}(\underline{tr}, \{control\})$.

If we consider $\mathcal{I} = \{pump_1\}$*, we get the following counter-factuals:*

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*2*	*2*	*2*
$pump_1\_fault$	*false*	*false*	*false*	*false*	*false*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*1*	*1*	*1*
out_{pump_1}	*2*	*2*	*1*	*1*	*1*
com_{pump_2}	*2*	*2*	*1*	*1*	*1*
out_{pump_2}	*2*	*2*	*1*	*1*	*1*
dump	*false*	*false*	*false*	*false*	*false*
vol_{tank}	*0*	*3*	*3*	*3*	*3*

The white-box components keep their choices as they were, and only the output of $pump_1$ *is modified, thus giving us the closest trace to the initial one.* $WIN((\underline{tr}, \emptyset, \{pump_1\}, \emptyset, \mathcal{P}))$*,* $\{pump_1\}$ *is a necessary cause.*

If $\mathcal{I} = \{tank\}$*, if* $strat_{tank}$ *is to dump if its volume would become strictly greater that* 3 *without dumping, we get* $outcome((\underline{tr}, \emptyset, \{tank\}, \emptyset, \mathcal{P}), strat_{tank}, \emptyset)$*:*

$Time$	0	1	2	3	4
out_{tank}	4	1	2	2	2
$pump_1\_fault$	$false$	$false$	$false$	$false$	$false$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	1	1	1
out_{pump_1}	2	2	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	2	2	1	1	1
out_{pump_2}	2	2	1	1	1
$dump$	$false$	$true$	$true$	$true$	$true$
vol_{tank}	0	3	3	3	3

The system property is not violated. The similar function prevents the controller from switching strategy, since it would mean that the trace is less similar to the initial one, thus producing the only possible outcome. Therefore, $\text{nec}(\underline{tr}, \{tank\})$.

Both white-box components had a possible strategy that lead to avoid the failure. Then, they both are necessary causes, so is $pump_1$*, the only faulty black-box component.*

Definition 45 (Sufficient cause suff**)** *Let* $\mathcal{I} \subseteq \mathbb{C}$ *be a set of suspected components,* $\mathcal{I}$ *is a sufficient cause, noted* $\text{suff}(\underline{tr}, \mathcal{I})$ *if* $SPOIL(\mathcal{G})$*, with* $\mathcal{G} = (\underline{tr}, \mathbb{C}_B \setminus \mathcal{I}, \mathbb{C}_W \setminus \mathcal{I}, \emptyset, \mathcal{P})$.

It means that even if we fix the components in $\mathbb{C}_B \setminus \mathcal{I}$, the system will eventually fail, whatever the components in $\mathbb{C}_W \setminus \mathcal{I}$ do.
If $\mathcal{I} \cap \mathbb{C}_W = \mathbb{C}_W$, and $CF_{black-box}(\underline{tr}, \mathbb{C}_B \setminus \mathcal{I}) = outcome((\underline{tr}, \mathbb{C}_B \setminus \mathcal{I}, \emptyset, \emptyset, \mathcal{P}), \emptyset, \emptyset)$, this definition has a similar behaviour as $\text{suff}_{black-box}(\underline{tr}, \mathcal{I})$ in the black-box only approach.

From a designer point of view, it means that in order to have local fixes, some more white-box components must be modified and more black-box components must be fixed.
From a responsibility assessment viewpoint, it means that the components not in $\mathcal{I}$ are not responsible, as they did not have any way of avoiding the failure, at a local level. On the other hand, the strategy/faults from the suspect are sufficient for the failure to happen. In order to assess the responsibility, it is interesting to contrast results with and without a component. For instance, if we have $\text{suff}(\underline{tr}, \mathcal{I} \cup \{C\})$ and $\neg\, \text{suff}(\underline{tr}, \mathcal{I})$, C can be considered as responsible, as its inclusion in the suspects or not changes the outcome of suff. We can define a notion of minimal sufficient cause (as usual, by the minimality of $\mathcal{I}$), thus giving us set of components for which the strategies/faults had a critical impact on the outcome. However, it can be a bit too fast to

arise responsibility for the white-box components from that, as the scenario might have not been foreseeable, and thus not taking it into account was not realistic.

Example 28 *The system is the same as in Example 26, with the following trace:*

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*4*	*3*	*3*
$pump_1\_fault$	*false*	*false*	*true*	*true*	*true*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*2*	*2*	*2*
out_{pump_1}	*2*	*2*	$\underline{0}$	$\underline{0}$	$\underline{0}$
com_{pump_2}	*2*	*2*	*2*	*2*	*2*
out_{pump_2}	*2*	*2*	*2*	*2*	*2*
dump	*false*	*false*	*false*	*false*	*false*
vol_{tank}	*0*	*3*	*1*	*0*	*-1*

Here, if we do not fix $pump_1$*, nothing can be done to avoid the failure, as* $pump_2$ *is already at its maximum output. Therefore,* $\text{suff}(\underline{tr}, \{pump_1\})$

Note that if the tank chooses to dump at instant 0*, the property is violated at the first instant. However, this trace is less similar to the initial one than the one where not dump occurs. Therefore,* $\{tank\}$ *is not a sufficient cause, as if we fix* $pump_1$*, the failure does not occur.*

Definition 46 (System fix fix**)** $\mathcal{I}$ *is a system fix, noted* $fix(\underline{tr}, \mathcal{I})$*, if* $\forall \mathcal{I}' \subseteq \mathbb{C}, \mathcal{I} \subseteq \mathcal{I}' \implies \text{nec}(\underline{tr}, \mathcal{I}')$.

It means that if we fix at least $\mathcal{I} \cap \mathbb{C}_B$, and at least the components in $\mathcal{I} \cap \mathbb{C}_W$ try to enforce $\mathcal{P}$, the system cannot fail. Note that The quantification could only be on $\mathcal{I} \cap \mathbb{C}_B$, because if $\mathcal{R}$ has a winning strategy, $\mathcal{R}' \supseteq \mathcal{R}$ also has one. However, the black-box causality framework has shown that black-box cause are not monotonous (e.g. if two faults compensate one another, fixing one might make the system fail).

We can define a notion of minimal fix. $\mathcal{I}$ is a minimal fix, if $\mathcal{I}$ is a fix and $\forall \mathcal{I}' \subset \mathcal{I}, \neg fix(\underline{tr}, \mathcal{I}')$.

From a designer point of view, a minimal fix is a set of components such that if the black-box ones are fixed, there is a way of constraining locally the model the white-box ones such that the failure is always avoided.
From a responsibility assessment point of view, a minimal system fix is a

set of components that can be considered as responsible, if the scenario is foreseeable, as their was a way of systematically preventing the failure.

If we consider the results from Example 27, $\{tank\}$, $\{control\}$ and $\{pump_1\}$ are system fix, as all there supersets are necessary causes. It means that fixing any of them is enough to fix the system.

Definition 47 (Unavoidable cause *una***)** *$\mathcal{I}$ is an unavoidable cause, noted $una(\underline{tr}, \mathcal{I})$, if $\forall \mathcal{I}' \subseteq \mathbb{C}, \mathcal{I} \subseteq \mathcal{I}' \implies \text{suff}(\underline{tr}, \mathcal{I}')$.*

It means that if you do not fix, at least, $\mathcal{I} \cap \mathbb{C}_B$ and at least $\mathcal{I} \cap \mathbb{C}_W$ are not protagonists, the system necessarily fail.
As for the previous example, only the quantification on the black-box is necessary, because if $\emptyset$ has a spoiling strategy (which can be considered as being a spoiling behaviour) against $\mathcal{R}$, it also does against $\mathcal{R}' \subseteq \mathcal{R}$.

Similarly to the system fix, we can define a notion of minimal unavoidable cause.

From a designer point of view, an unavoidable cause raises a set of components for which at least one must be fixed/modified, in order to be able to have a shot at avoiding the failure. A minimal unavoidable cause is more interesting, as removing one component form $\mathcal{I}$ ensures to be able to fix the system.
From a responsibility assessment point of view, $\mathbb{C} \setminus \mathcal{I}$ is a set of components that cannot be held responsible as they never had a way of avoiding the failure. On the other hand, if $\mathcal{I}$ is minimal, the strategies/faults from the components in $\mathcal{I}$ are critical to the failure. Therefore, the black-box components can be held responsible, and the white-box ones as well, if the scenario was foreseeable.

Example 29 *The system is the same as in Example 26, with the following trace:*

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*1*	*2*	*2*
$pump_1\_fault$	*false*	*false*	*false*	*false*	*false*
$pump_2\_fault$	*false*	*false*	*false*	*false*	*false*
com_{pump_1}	*2*	*2*	*1*	*1*	*1*
out_{pump_1}	*2*	*2*	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	*2*	*2*	*1*	*1*	*1*
out_{pump_2}	*2*	*2*	*1*	*1*	*1*
dump	*true*	*true*	*true*	*true*	*true*
vol_{tank}	*-1*	*2*	*3*	*3*	*3*

Here, whatever the over components do, they cannot prevent the tank from making the system fail.

An interesting aspect of this example is also that $pump_1$ is not a cause. It shows that with this approach, we can have a failure than are not caused by black-box component faults. What's more, the fault occurs after the failure of the system, showing that a failure can occur without any fault.

Definition 48 (Weak malicious cause mal_{weak}) *Let $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected component. Let $\mathcal{A} = \mathcal{I} \cap \mathbb{C}_W$. $\mathcal{I}$ is a weak malicious cause, noted $mal_{weak}(\underline{tr}, \mathcal{I})$, if:*

$$\forall strat_{\mathcal{A}} \in \Pi_{\mathcal{A}}, \exists \underline{tr'} \in outcome((\underline{tr}, \emptyset, \emptyset, \mathcal{A}), \emptyset, strat_{\mathcal{A}}), \underline{tr} \sqsubseteq \underline{tr'} \tag{6.5}$$

such that,

$$\forall \mathcal{R} \subseteq \mathbb{C}_W \setminus \mathcal{A}, \forall \mathcal{M} \subseteq \mathbb{C}_B \setminus \mathcal{I}, \forall strat_{\mathcal{R}} \in \Pi_{\mathcal{R}}, \\ outcome((\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}), strat_{\mathcal{R}}, strat_{\mathcal{A}}) \cap \mathcal{P} = \emptyset \tag{6.6}$$

Constraint 6.5 reflect the fact that $\underline{tr}$ is a trace which is consistent with $strat_{\mathcal{A}}$, as there is a trace $\underline{tr'}$ in $outcome((\underline{tr}, \emptyset, \emptyset, \mathcal{A}), \emptyset, strat_{\mathcal{A}})$ that is prefixed by $\underline{tr}$. Constraint 6.6 means that whatever set of white-box components not in $\mathcal{I}$ you oppose to $\mathcal{A}$, as long as the black-box components in $\mathcal{I}$ are not fixed, $strat_{\mathcal{A}}$ is a spoiling strategy.
This definition is close to the unavoidable cause, with an important difference, which is that the suspects strategy made the system fail.
This definition shows that $\mathcal{A}$ is used to add another level of granularity to the approach by being able to cope with sets of component that try to make the system fail.
Note that $una(\underline{tr}, \mathcal{I}) \not\Rightarrow weak\_mal(\underline{tr}, \mathcal{I})$, it might not be possible to build a local strategy that ensures the failure, even-though a global one was available (since the run resulted in a failure).

This definition does not really gives much information from a designer point of view. From a responsibility assessment point of view, $\mathcal{I}$ can be malicious, as one strategy consistent with the initial trace is always spoiling. If $\neg una(\underline{tr}, \mathcal{I}) \wedge weak\_mal(\underline{tr}, \mathcal{I})$ $\mathcal{I}$ is more likely to be malicious, as some strategies that are consistent with what happened in $\underline{tr}$ are spoiling ones, but not all the possible behaviours lead to a failure (since $\neg una(\underline{tr}, \mathcal{I})$).

The strategy used in Example 29 is a spoiling strategy that coincide with the initial trace, thus $\{tank\}$ is both an unavoidable cause and weak a malicious cause.

Definition 49 (Malicious cause *mal***)** *Let $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected component. Let $\mathcal{A} = \mathcal{I} \cap \mathbb{C}_W$. $\mathcal{I}$ is a malicious cause, noted $mal(\underline{tr}, \mathcal{I})$, if:*

$$\exists strat_{\mathcal{A}} \in \Pi_{\mathcal{A}}, \exists \underline{tr}' \in outcome((\underline{tr}, \emptyset, \emptyset, \mathcal{A}), \emptyset, strat_{\mathcal{A}}), \underline{tr} \sqsubseteq \underline{tr}' \tag{6.7}$$

such that,

$$\forall \mathcal{R} \subseteq \mathbb{C}_W \setminus \mathcal{A}, \forall \mathcal{M} \subseteq \mathbb{C}_B \setminus \mathcal{I}, \forall strat_{\mathcal{R}} \in \Pi_{\mathcal{R}}, \\ outcome((\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}), strat_{\mathcal{R}}, strat_{\mathcal{A}}) \cap \mathcal{P} = \emptyset \tag{6.8}$$

The difference with the weak malicious is the fact that the quantifier on the spoiling strategy is universal. It means that every possible strategies that are consistent with the initial trace lead to a failure, as long as the black-box components in $\mathcal{I} \cap \mathbb{C}_B$ are not repaired. It means that the suspected white-box components only had spoiling strategies. Note that the universal quantifier is here to ease the understanding, as using $\hat{\mathcal{S}}_C$ for each component in $\mathcal{A}$ ensures that all possible strategies are covered, as a strategy is a restriction of the specification, thus $\hat{\mathcal{S}}_C$ being a spoiling strategy implies that every strategy is.

From a responsibility assessment point of view, $\mathcal{I}$ is most likely malicious, or the design choices made cannot cope with the scenario represented by $\underline{tr}$, and the system must be modified.

Note that $mal(\underline{tr}, \mathcal{I}) \implies una(\underline{tr}, \mathcal{I})$, since the behaviours from $una(\underline{tr}, \mathcal{I})$ are necessarily produced if $\mathcal{A}$ has access to all the possible strategies.

$\{tank\}$ is a malicious cause as well, as any strategy that starts with a dump leads to a failure at instant 1.

Definition 50 (Resilient *resil***)** *Let $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected component. Let $\mathcal{A} = \mathcal{I} \cap \mathbb{C}_W$. The system is $\mathcal{I}$-resilient, noted $resil(\underline{tr}, \mathcal{I})$, if exists $\mathbb{C}' \subseteq (\mathbb{C} \setminus \mathcal{I})$, such that $WIN((\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{I} \cap \mathbb{C}_W))$, with $\mathcal{R} = \mathbb{C}' \cap \mathbb{C}_W$ and $\mathcal{M} = \mathbb{C}' \cap \mathbb{C}_B$.*

It means that there exists a set of protagonists $\mathcal{R}$ and black-box components $\mathcal{M}$, such that if the components in $\mathcal{M}$ are repaired, $\mathcal{R}$ always has a winning strategy against $\mathcal{I} \cap \mathbb{C}_W$ even though the components in $\mathcal{I} \cap \mathbb{C}_B$ remain faulty. Intuitively, the system is resilient to the fault and the malicious behaviours of the suspected components.

This definitions shows that the framework can capture even complicated definitions. From a designer point of view, it gives a tool to ensure that the system cannot fail because of a given set of components, for a given scenario. An example is that a system pacemaker-heart should be {pacemaker}-resilient, i.e. the pacemaker cannot make the heart fail (which would have some dire

consequences). Another example is that it ensures that some critical data cannot be accessed by a set of components trying to retrieve the said data.

Example 30 *If we consider Example 28 again, which relies on the following scenario:*

$Time$	0	1	2	3	4
out_{tank}	4	1	4	3	3
$pump_1\_fault$	$false$	$false$	$true$	$true$	$true$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	2	2	2
out_{pump_1}	2	2	$\underline{0}$	$\underline{0}$	$\underline{0}$
com_{pump_2}	2	2	2	2	2
out_{pump_2}	2	2	2	2	2
$dump$	$false$	$false$	$false$	$false$	$false$
vol_{tank}	0	3	1	0	-1

If we consider $\mathcal{I} = \{pump_2, tank\}$, we can build the following play-through, with $\mathcal{M} = \{pump_1\}$ and $\mathcal{R} = \{control\}$:

$Time$	0	1	2	3	4
out_{tank}	2	2	3	3	3
$pump_1\_fault$	$false$	$false$	$false$	$false$	$false$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	2	2	2
out_{pump_1}	2	2	2	2	2
com_{pump_2}	2	2	2	2	2
out_{pump_2}	2	2	2	2	2
$dump$	$true$	$true$	$true$	$true$	$true$
vol_{tank}	1	2	2	2	2

By repairing $pump_1$ and keeping the $pump_i\_faults$ at false, the tank cannot make the system fail. If it does not dump, the volume of the tank will be 3 if it does, it will be 2. Every other play-through with those players will always lead to a win. Therefore, $resil(\underline{tr}, \{pump_2, tank\})$.

6.3 Strategy synthesis for the game framework

As for the mixed approach, the game approach relies on the computation of strategies. Though the framework can be seen as a three players game (the protagonist, the antagonist and the environment), in all the definitions that

exist in the mixed framework only have a protagonist. Therefore, we can use a two players synthesis similar to the one used in Section 5.3. In the case of the malicious definitions, since the quantification is universal, by considering the antagonist as the only player and add the protagonist components to the environment, we actually get the expected result. Indeed, having a strategy against the most general strategy accessible by the protagonists (i.e. their model) encompasses any possible strategy.
We can also build a synthesis for the spoiling strategies. The idea is to eliminate recursively the states that lead to a state where the property cannot be violated.

This section will present a refinement to the synthesis proposed in Section 5.3, to fit the partial information constraint, as well as some illustrative examples. The section will be divided in two subsections. The first one will present how to synthesise the winning strategies, while the second will show how to synthesise the spoiling ones.

The LTS formalism to represent the counter-factual, as well as the definition of a strategy synthesis (Definition 33) will be reused as they were defined in Section 5.3.

6.3.1 Winning strategy synthesis

In order to compute the strategies, we use a $safe$ function (like Definition 32 of Section 5.3) that take into account the information accessible to the component.

Definition 51 (Safe function for the game framework ($safe_{game}$)) *Let $L = (Q, q^0, \Sigma, \rightarrow)$ be a LTS, $\Sigma_C \uplus \Sigma_u \subseteq \Sigma$ a partition of the labels and P be a set of states. Let $L' = (Q, q^0, \Sigma, \rightarrow)$.*
$safe_{game}(P, L', \Sigma_u) = \bigcap_{i=0}^{\infty} game\_OK_i(L')$ *with* $(Q, q^0, \Sigma, \rightarrow) \cap (Q', q^0, \Sigma', \rightarrow'$ $) = (Q \cap Q', q^0, \Sigma \cap \Sigma', \rightarrow \cap \rightarrow')$ *and:*

$$game\_OK_0(L') = (Q \cap P, q^0, \Sigma, \rightarrow),$$

$$game\_OK_{k+1}(L') = game\_OK^{+1}(game\_OK_k(L')) \text{ and}$$

$$game\_OK^{+1}((Q_x, q_x^0, \Sigma_x, \rightarrow_x)) = (OK^{+1}(Q_x), q_x^0, \Sigma_x, \rightarrow^{+1})$$

With $\rightarrow^{+1} = \rightarrow_x \setminus \{(q, \sigma, q') \in \rightarrow_x |$

$$q \notin OK^{+1}(Q_x) \vee q' \notin OK^{+1}(Q_x) \ \vee \tag{6.9}$$

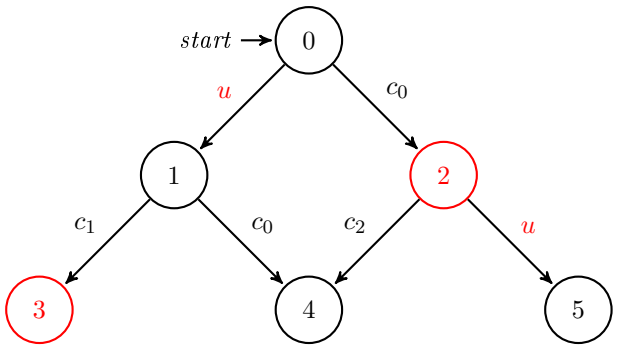
$$\exists (k, k') \in (OK^{+1}(Q_x) \times (Q \setminus OK^{+1}(Q_x))), \big((k \equiv_\sigma q) \wedge (k, sigma, k') \in \rightarrow_x \big) \tag{6.10}$$

With OK^{+1} as in Definition 32, with $Q_f = \{q \in (P \cap Q) \mid \forall (\sigma, q') \in (\Sigma \times Q), (q, \sigma, q') \notin \rightarrow\}$, and $q \equiv_\sigma q'$ being true meaning that q and q' are equivalent for the component that fires σ.

This definition shifts from considering a set of states to a full LTL. The idea is to restrict the state space (with OK^+1 from Definition 32) and then trim the transitions that have either the source or the destination that is no longer in the state space (Constraint 6.9).
It also removes the transition fired from q if a transition labelled similarly is fired from a state in $OK^{+1}(Q_x)$ k to a state outside of $OK^{+1}(Q_x)$ k. q is equivalent to k for σ (i.e. $(q \equiv_\sigma q') = true$) means that the state is the same from the perspective of the component that corresponds to σ. This is the idea of the $\equiv$ operator, but it will be further explained in the next definition.

Example 31 *To illustrate the $safe_{game}$ function, we adapt LTS from Example 22.*

Let us consider the following LTS:



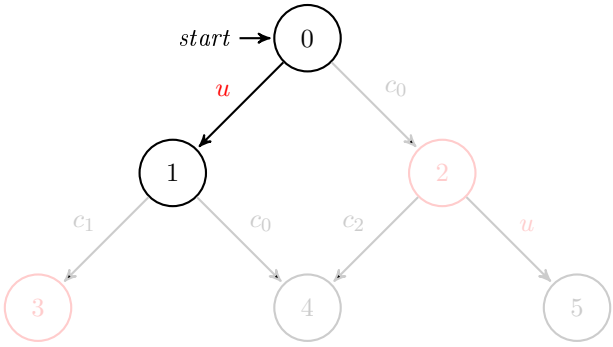
The red transitions are uncontrollable, and the black ones are.

We consider that all the states are equivalent, whatever is the label.

Here, $Q_0 = Q \cap P = \{0, 1, 4, 5\}$. $Q_1 = OK^{+1}(\{0, 1, 4\})$, since is not reachable from a state in Q_0

$\rightarrow_0 = \{(0, u, 1), (0, c_0, 2), (1, c_1, 3), (1, c_0, 4), (2, c_2, 4), (2, u, 5)\}$. We remove $(0, c_0, 2)$, $(1, c_1, 3)$, $(2, c_2, 4)$ and $(2, u, 5)$, as they originate, or lead to a state that is not in $OK^{+1}(P)$. We aslo remove $(1, c_0, 4)$, as it is labelled with c_0, and the transition $(0, c_0, 2)$ goes from 0 (in $OK^{+1}(P)$), which is c_0-equivalent to 4, to 2 (not in $OK^{+1}(P)$). We get $\rightarrow_1 = \{(0, u, 1)\}$.

We get the following LTS for game_$OK_1(P)$:



The result will then converge to the empty LTS.

The difference with Example 22 is that transition $(1, c_0, 4)$ *has been removed, making* 4 *unreachable. Since there is no reachable final state in* $game\_OK_1(P)$*, then* $safe_{game}(P)$ *will be the empty LTS.*

Definition 52 (σ-equivalent ($\equiv_\sigma$)) *Let* $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ *be a game system,* $\prec$ *be a partial order for the system and* $\mathbb{TR} \subseteq \mathcal{B}_\mathbb{F}$ *be a set of traces. Let* $L = (Q, q^0, \Sigma, \rightarrow)$ *be the LTS build from* $\mathbb{TR}$*.*

$$\forall q, q' \in Q, (q \equiv_\varepsilon q') = false \tag{6.11}$$

$$\forall q, q' \in Q, \forall (name, v) \in \sigma, q \equiv_{(name,v)} q' = \Big(\tag{6.12}$$

$$\exists C \in \mathbb{C}, \exists f = (name_f, D_f) \in O_C, (name = name_f) \wedge \tag{6.13}$$

$$\big(\forall e \in (I_C \cup O_C \cup M_C), e \prec f \implies (\pi_e(q) = \pi_e(q') \wedge \pi_e(q) \neq undef)\big)\Big) \tag{6.14}$$

The intuition for this definition is that $q \equiv_\omega q'$ is *true* if σ corresponds to the valuation of a component (C) output f and the values of the flows from C from which f depends on are the same in q and q'. I.e., q and q' are the same from C perspective on those flows.
Constraint 6.11 makes sure that the equivalence is false, if the label is ε. By definition, an epsilon transition is not fired by a component, and thus there is no component to compare q and q'.
Implicitly, we suppose that the label is not the epsilon one in Constraint 6.12, since $(name, v)$ cannot be the epsilon label.
Constraint 6.13 ensures that the label corresponds to a valuation by a component. Indeed, if not, it means that it is the valuation of a system input, which does not originate from a component and, as for Constraint 6.11, there is no component to compare q and q'.
Lastly, Constraint 6.14 ensures that the state q and q' are equivalent from the component prospective. They are if the values of the flows from which

f depends on are the same. We only take into account the flows from which f depends on, because they are the only ones relevant to the choice of the valuation of f.
Note that if q and k are σ-equivalent, their successors also are, since the difference between a state and its successor is exactly the valuation corresponding to the label.

Definition 53 (LTS property ($\mathcal{P}_{LTS}$)) *Let $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$ be a game system and $\prec$ be a partial order for the system. Let $\mathcal{P}' = \{\underline{tr} \in \mathbb{F} \mid \pi_{\mathbb{F}'}(\underline{tr}) \in \mathcal{P}\}$, with $\mathbb{F}' = \bigcup_{C \in \mathbb{C}} (I_C \cup O_C)$. $(Q, q^0, \sigma, \rightarrow) = LTS_{sys}(\mathcal{P}')$ is the mixed LTS build the set of traces $\mathcal{P}'$. $\mathcal{P}_{LTS} = Q$ is the set of states of the build LTS.*

First, we build $\mathcal{P}'$ by, essentially, an upward project $\mathcal{P}$ to $\mathbb{F}$. Note that in a game system, $\mathbb{F}$ includes the memory flows of the components, while $\mathcal{P}$ is defined on $\mathbb{F}$' (all the input and output flows of the components). Then, a LTS is build from $\mathcal{P}'$, using the LTS_{sys}. We then take the set of states from this LTS, which gives us the set of non-failing states upward projected on $\mathbb{F}$.

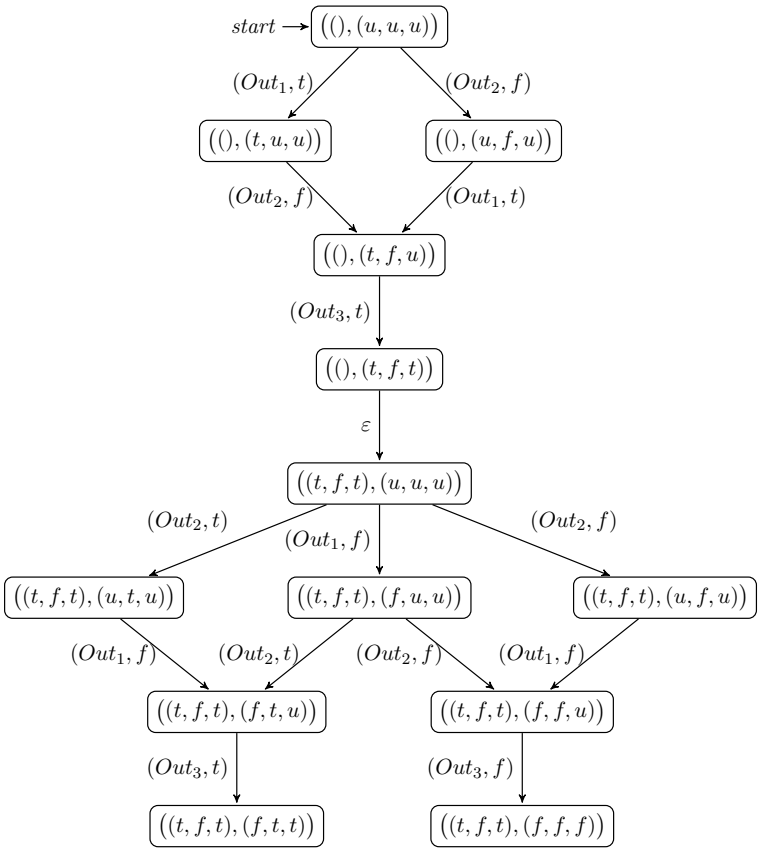
Definition 54 (Winning strategy ($\boldsymbol{WIN}_{game}$)) *Let $\underline{tr}$ be a system trace, $\mathcal{M} \subseteq \mathbb{C}_B$ a set of suspected components, $\mathcal{R} \subseteq \mathbb{C}_W$ a set of protagonist and $\mathcal{A} \subseteq (\mathbb{C}_W \setminus \mathcal{R})$.*
Let LTS_{game} be the LTS build from $outcome((\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P}), \{\hat{\mathcal{S}}_C\}_{C \in (\mathcal{R} \cup \mathcal{A})})$ and $\mathcal{P}_{LTS}$ is the LTS property build form $\mathcal{P}$.
There is a winning strategy if $((), (undef)_{f \in \mathbb{F}})$ is in the set of states of $safe_{game}(\mathcal{P}_{LTL}, LTS_{game}, \Sigma_u)$, with $\Sigma_u = \bigtimes_{f \in \mathbb{E}} (D_f \cup \{undef\})\big)$, with $\mathbb{E} = \mathbb{F} \setminus \bigcup_{C \in \mathcal{R}} (I_c \cup O_c \cup M_C)$.

This definition is very similar to the winning strategy definition from the mixed framework (33) . The differences are the initial LTS, here it is build with *outcome*, and the fact that the property is "upward projected" over $\mathbb{F}$ (which also contains the flows that correspond to the components memory). To generate the counter-factuals LTS, we use *outcome*, with their step specifications, $\hat{\mathcal{S}}_C$, as strategy for each protagonists and antagonists. It acts similarly to CF_{mixed}, i.e. it only constraints the protagonists and antagonists with their possible behaviours, regardless what they output in the initial trace. Since, by definition, the strategies are restrictions of the step specifications, all the possible strategies behaviours are generated. Since $safe_{game}$ generates a LTS for which each state has a successor, or is final, we indeed generate strategies, i.e. restrictions of the step specification that is lively (at

least the one "close" to the initial trace $\underline{tr}$). We just need to extend this "restricted" strategy to the other possible states, to get a full fledged accepting strategy.

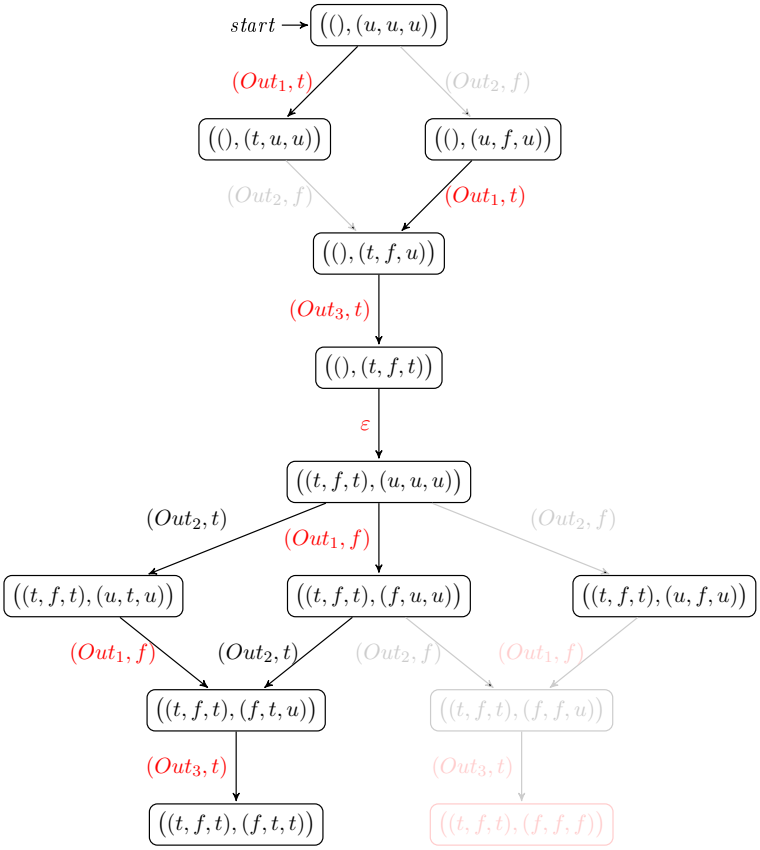
Let us illustrate the strategy synthesis with an example.

Example 32 *We consider the LTS used in Example 23, i.e. the one that illustrated the strategy synthesis for the mixed framework:*



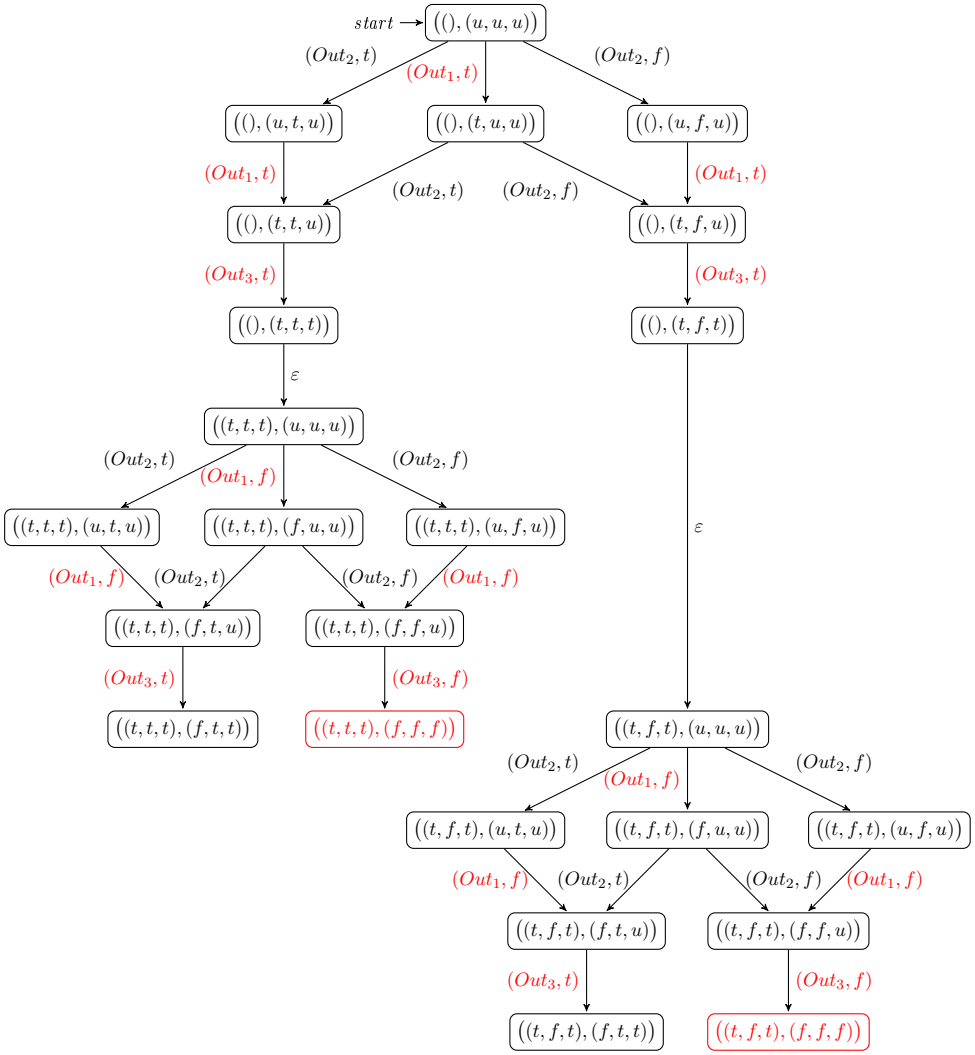
The system property is that Out_3 should always be true (i.e. the current valuation can be $(\_, \_, u)$ or $(\_, \_, t)$). The result for the mixed framework strategy synthesis was basically to forbid the (Out_2, f) transition in the lower part of the LTS (after the epsilon transition). Let us see how this example is treated by the game approach synthesis.

Here is the interesting step in the synthesis, it corresponds to game_OK_2 for the considered LTS:



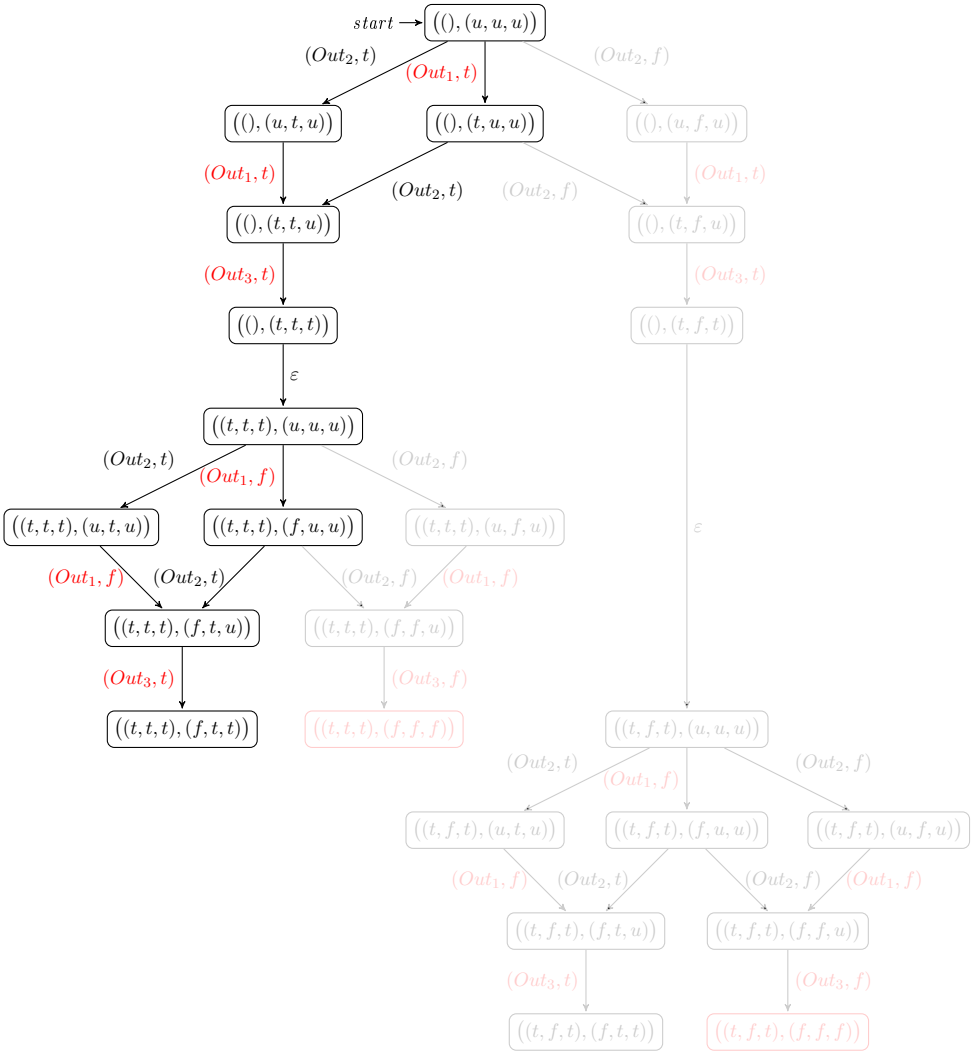
What happened at this step is that the state $\big((t, f, t), (f, f, u)\big)$ *got removed, as it had no successor. Therefore, the transition* (Out_2, f) *was removed. Since* C_2 *has no input, all the transitions labelled by* (Out_2, f) *are also removed (since all the states of* C_2 *where* out_2 *is not valued are equivalent), thus disconnecting the initial state from the rest of the graph. Therefore,* $safe_{game}$ *converges to the empty LTS.*
This was actually to be expected, as from C_2 *perspective, it is not possible to differentiate the transitions labelled* (Out_2, f)*, that are in the upper part of the graph (and are safe), from the one in the lower part of the graph (that are not safe). Therefore, there is no local strategy, whereas, there was a global one. Thus* C_2 *cannot be considered responsible for the failure, since it could not avoid it with the information it had.*

Let us now consider the following example, where C_2 *can output either true or false in the upper part of the graph:*



The uncontrollable transitions are labelled in red. The bad states (i.e. not respecting $\mathcal{P}$) are red.

Here is $safe_{game}(L, \mathcal{P}_{LTS}, \Sigma_u)$ for C_2 being the only controllable component:



The right part of the graph is totally removed, as the transitions labelled by (Out_2, f) *are forbidden. However, their is a possible strategy, that translates locally, for* C_2*, to always output true.*
This result is different from the one we would have using the mixed framework synthesis, since only the problematic transitions labelled (Out_2, f)*, in the lower right part of the graph would have been removed, thus allowing the transition labelled* (Out_2, f) *in the upper part of the graph. However, this controller cannot be translated to a local strategy for* C_2*, since* C_2 *does not have the information to decide whether it should, or not, perform a* (Out_2, f) *transition.*

With the Definition 54, we are able to synthesise a global strategy that is consistent with what information the components have access to. We can

then define a projection function, to give rise to local strategies for each component that is controllable.

By construction, $safe_{game}$ builds a LTS that is a restriction of the counterfactuals. Therefore, the "strategy" that is synthesised is not, per se, a strategy, as it is not enabling for all the possible inputs. However, by construction (since the LTS represents the execution of traces, and the empty trace is in $safe_{game}$) the generated strategy is accepting over the states from $safe_{game}$ (since a non-final state with no outgoing transition is removed). Therefore, to get a full fledged strategy, one just need to take a strategy that coincide with the one synthesised on the states that are the same as the one appearing in the synthesis, and $\hat{\mathcal{S}}$ otherwise.

6.3.2 Spoiling strategy synthesis

The problem of synthesis is very similar for the spoiling and the winning strategies. The idea is to reach a set of states in the two cases. For the winning strategies, the goal is to reach states that corresponds to a finished non-faulty trace. For the spoiling strategies, the goal is to reach states for which the property is not verified. Therefore, it is possible to compute the spoiling strategies by slightly modifying the functions used in the computation of the winning strategies.

Definition 55 (Unsafe function $unsafe$**)** *Let $L = (Q, q^0, \Sigma, \rightarrow)$ be a LTS, $\Sigma_u \subseteq \Sigma$ a set of labels and P be a set of states such that P represents a safety property. Let $Q_f = Q \setminus P$ the set of final states of L.*

$unsafe(P) = \bigcap_{i=0}^{\infty} KO_i(P)$ *is the fix-point of $KO_i(P)$, with $KO_0(P) = Q$, $KO_{k+1}(P) = KO^{+1}(KO_k(P))$ and $KO^{+1}(X) = \{q \in x \mid :$*

$$(q = q^0 \vee \exists q' \in X, \exists \sigma \in \Sigma, (q', \sigma, q) \in \rightarrow) \ \wedge \tag{6.15}$$

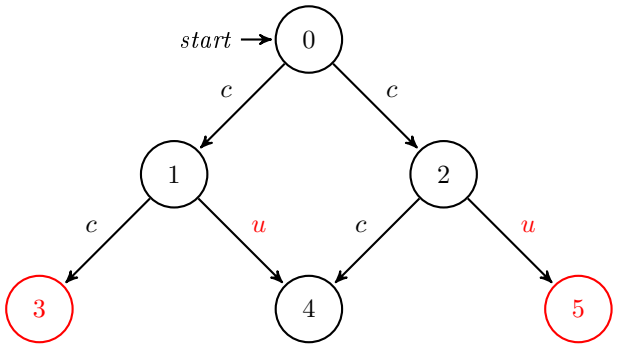
$$(q \notin Q_f) \implies \Big(\big(\forall \sigma \in \Sigma_u, \forall q' \in Q, (q, \sigma, q') \in \rightarrow \implies q' \in X\big) \wedge \big(\exists (\sigma', q'') \in (\Sigma, X), (q, \sigma', q'') \in \rightarrow\big)\Big)\} \tag{6.16}$$

From the set of all states in the LTS, $unsafe$ builds the set of states that eventually lead to a final state, whatever the uncontrollable transition are fired. Since the set of final states are the failing states, this function builds the set of states that eventually lead to a failure.

The differences with the safe function only lies in the initial set of states (Q instead of $P \cap Q$) and the final states (the failing states instead of the final non-failing traces). The difference in the initial set of states is just an "optimisation", as the properties we consider are safety ones, it is not possible to leave Q and then reenter it, therefore, $safe$ would converge to the same result, when starting from Q instead of $P \cap Q$ (up to parts of the graph unreachable from the initial state). However, we did not make any assumption over P in the safe definition, whereas we suppose it represents a safety property here.
The KO function is exactly the same as OK: it filters out non-final states with no successor, and the non-initial states with no predecessor. The reason it removes the states corresponding to non-failing finished trace, is because such a state is non-final and has no successor, and is therefore removed, like OK would do with the failing ones, would $safe$ start from Q. The predecessors of those states are themselves removed if they have no successor, for the same reason, and so on.
Note that we can stop when a failing state is reached because P represents a safety property (that is if a successor of a state in P leaves P, its successors never reenter P). If not, we would have needed the final states to correspond to finished failing traces.

Let us illustrate the $unsafe$ function with an example.

Example 33 *We consider a LTS that is similar to the ones in Example 22:*



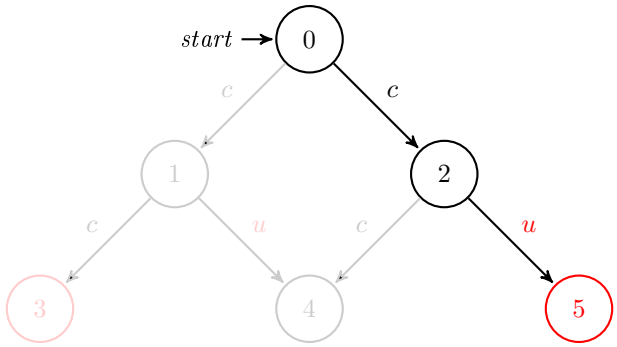
The controllable transitions are labelled by c, the uncontrollable are labelled by a red u. The "bad" states are red, i.e. the states not in P.
$Q_f = Q \setminus P = \{3, 5\}$, *and* $KO_0(P) = Q = \{0, 1, 2, 3, 4, 5\}$.
Let us now build the different $KO_i(P)$.

- $KO_1(P) = \{0, 1, 2, 3, 5\}$. *4 is removed, because it does not have a successor in* $KO_0(P)$ *and is not final.*

- $KO_2(P) = \{0, 2, 3, 5\}$. 1 *is removed, because it has an uncontrollable transition leaving* $KO_1(P)$.

- $KO_3(P) = \{0, 2, 5\}$. 3 *is removed, because it has no predecessor in* $KO_2(P)$.

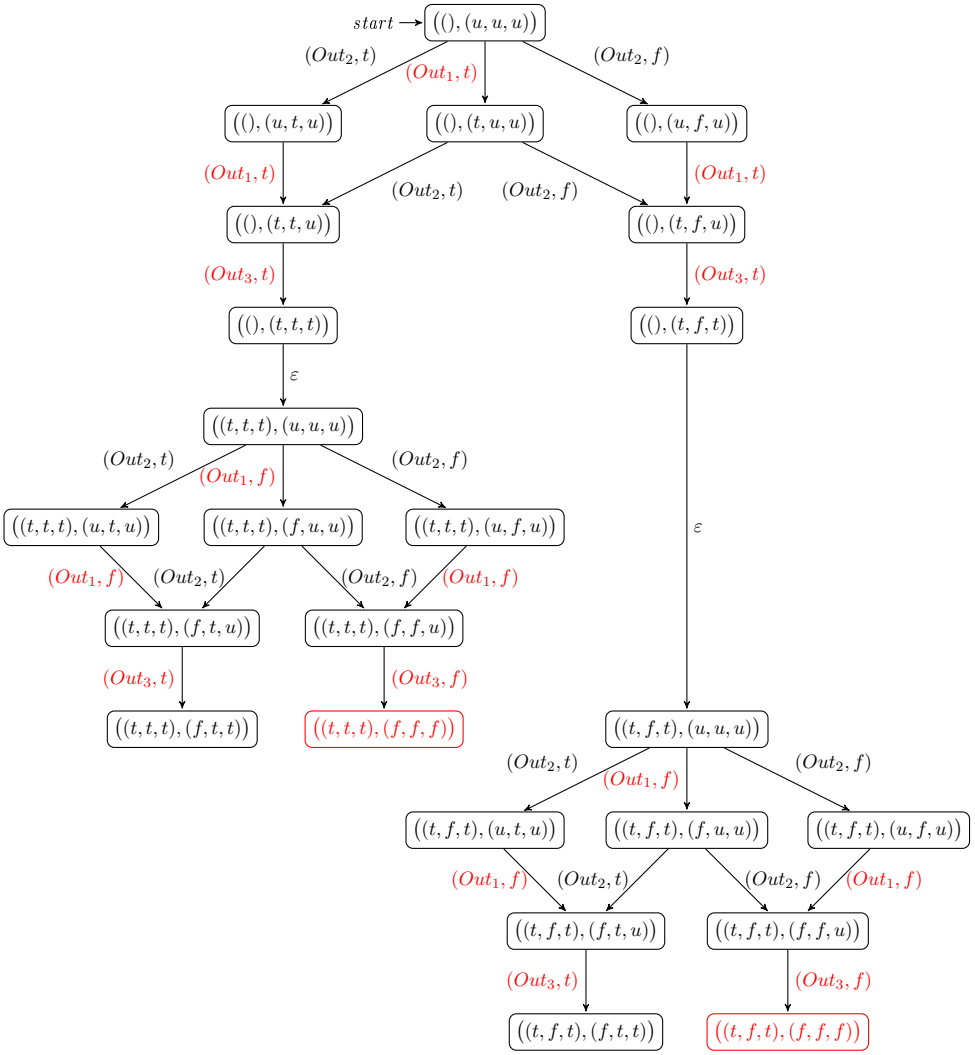
We have reached the fix-point here, and $unsafe(P) = \{0, 2, 5\}$, *which gives us the following LTS:*



Definition 56 (Spoiling strategy $SPOIL_{game}$) *Let* $\underline{tr}$ *be a system trace,* $\mathcal{M} \subseteq \mathbb{C}_B$ *a set of suspected components,* $\mathcal{R} \subseteq \mathbb{C}_W$ *a set of protagonist and* $\mathcal{A} \subseteq (\mathbb{C}_W \setminus \mathcal{R})$.
Let LTS_{game} *be the LTS build from outcome*$((\underline{tr}, \mathcal{M}, \mathcal{R}, \mathcal{A}, \mathcal{P}), \{\hat{\mathcal{S}}_C\}_{C \in (\mathcal{R} \cup \mathcal{A})})$ *and* $\mathcal{P}_{LTS}$ *is the LTS property build form* $\mathcal{P}$.
There is a winning strategy if $((), (undef)_{f \in \mathbb{F}}) \in unsafe_{game}(LTS_{game}, \mathcal{P}_{LTL}, \Sigma_u)$, *with* $\Sigma_u = \bigtimes_{f \in \mathbb{E}} (D_f \cup \{undef\})$, *with* $\mathbb{E} = \mathbb{F} \setminus \bigcup_{C \in \mathcal{A}} (I_c \cup O_c \cup M_C)$ *and* $unsafe_{game}$ *defined as* $safe_{game}$ *(Definition 51), with the* $safe$ *function replaced by the* $unsafe$ *one.*

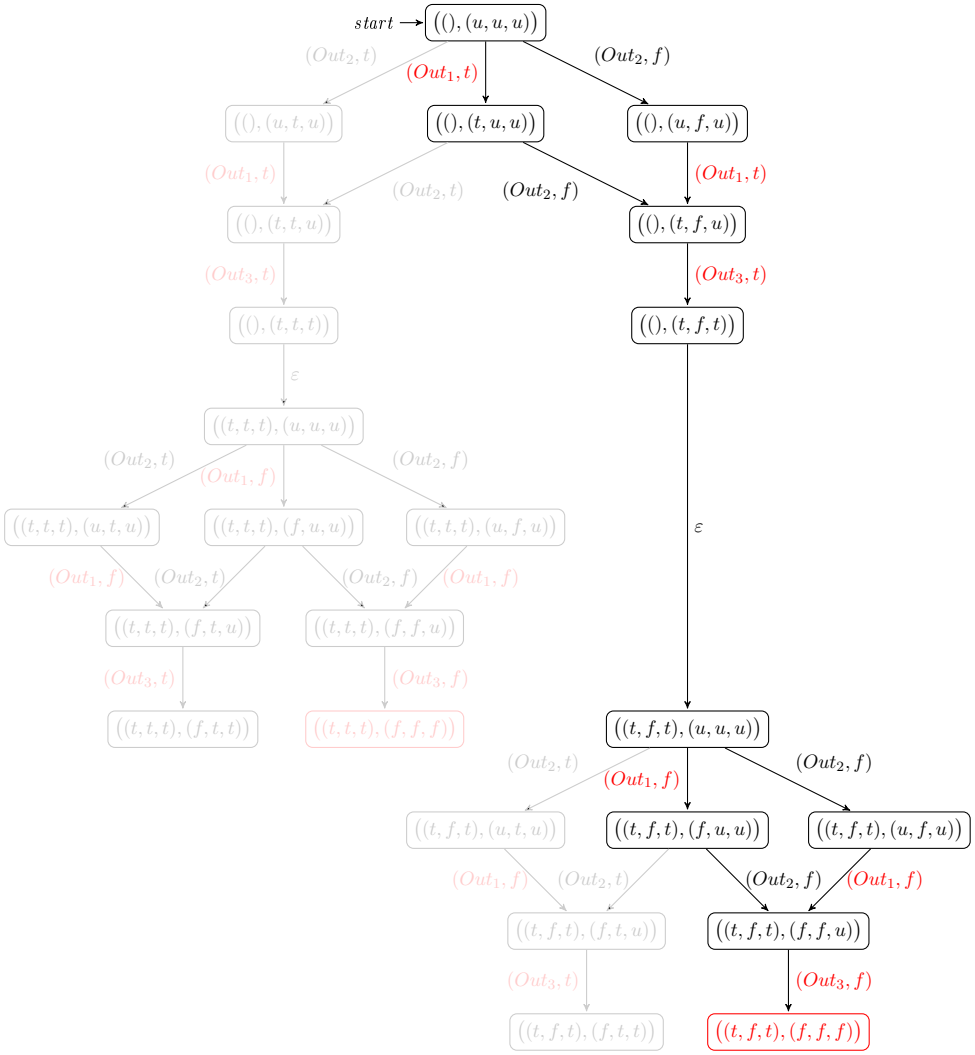
Since $unsafe$ builds a set a states that must not be left, similarly $safe$, we can build $unsafe_{game}$ just by replacing the use of $safe$ by the use of $unsafe$. The way of forbidding transition is the very same. This gives us a way of synthesising spoiling strategies.

Example 34 *Let us illustrate the functioning of* $SPOIL_{game}$ *on the second LTS of Example 32:*



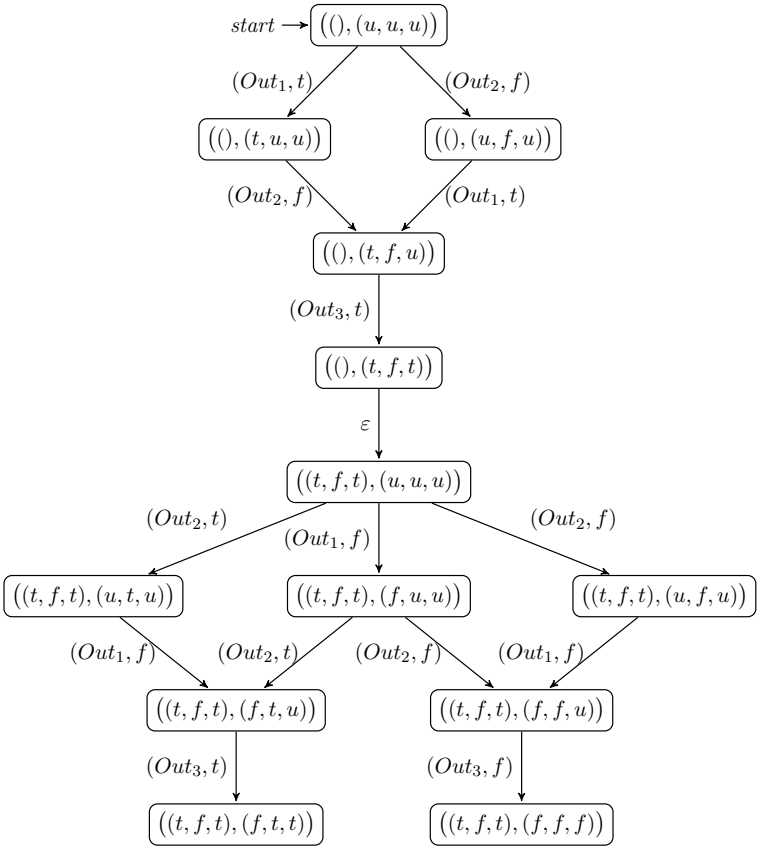
The uncontrollable transitions are labelled in red. The bad states (i.e. not respecting $\mathcal{P}$) are red.

Here is $unsafe_{game}$, for this LTS and C_2 being the controllable component.

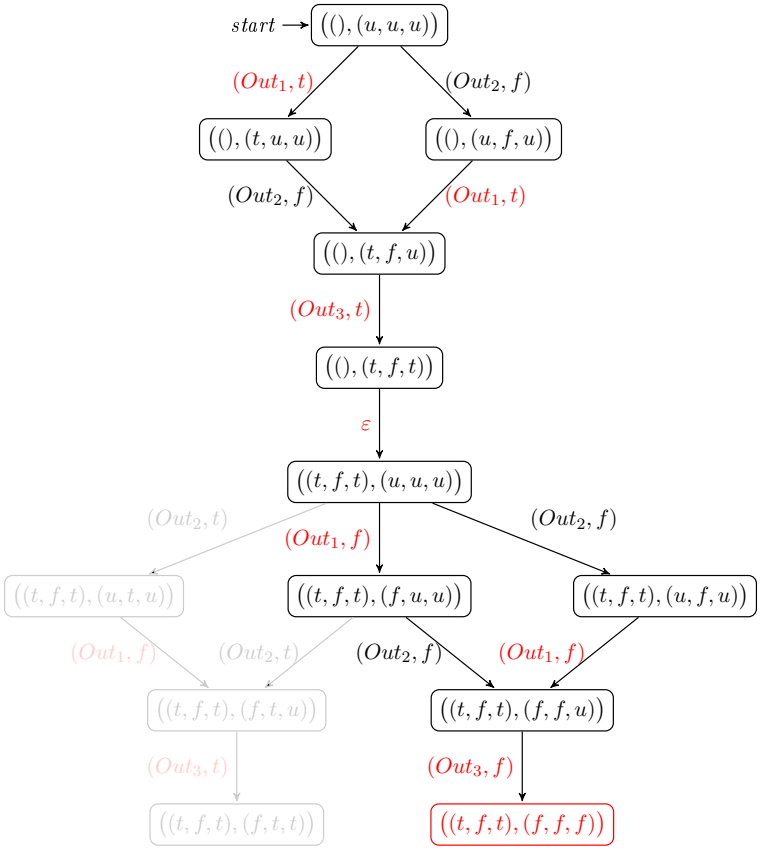


As expected, by always outputting false, C_2 can ensure that the system eventually fail. Note that this is the complimentary of the winning strategy. However, this is not a general result, and is due to the fact that this example is extremely simple.

For instance, if we consider the first LTS of Example 22, there is a spoiling strategy, whereas there was no winning one. The LTS is the following:



Here is $unsafe_{game}$, for this LTS and C_2 being the controllable component:



The reason there is a spoiling strategy and not a winning one, is because (Out_2, f) *is removed in the winning strategy synthesis, thus disconnecting the initial state from the rest of the graph. In the case of the spoiling strategy,* (Out_2, t) *is forbidden, which does not impact the upper part of the graph.*

6.4 Finding fixes with the game framework

This section will present two ways to find fixes with the game approach. The first one is to use the output of the game approach as the input of techniques that contrast passing and failing traces. The second one is to slightly modify the framework, to be able to find fixes "natively".

6.4.1 Using the game framework as input to approaches to find fixes.

In [Xuan et al., 2016], the authors propose a method to find and fix bugs in conditional statements (i.e. if-then-else statements) for Java programs.

The reason they focus on conditional statements stems from the observation that a large portion of the bugs (in their data-set, $12,5\%$ of the one change commits are on conditional statements). The approach needs a test suite, which contains at least one faulty trace that captures the bug to fix. The first step is to use angelic values (i.e. replacing values at runtime by arbitrary ones, namely the value of the conditions here) to transform a failing trace into a passing one. Once such a transformation is found, they use a SMT-based technique to synthesise, if possible, the new conditions.

The first step of the approach can be performed with the mixed approach or the game approach. Indeed, by changing the conditions in the white-box components to choices, we can get, if there is one, a strategy that fixes the bug. The second step of the [Xuan et al., 2016] approach can then be used to find a possible fix for the conditional statements.

Let us illustrate how it would function with an example.

Example 35 *The system is the same as in Example 26, $pump_1\_fault$ would represent a condition that recognise that $pump_1$ is faulty and $pump_2\_fault$ a condition that recognise that $pump_2$ is faulty.*
Let us consider the following trace.

$Time$	0	1	2	3	4
out_{tank}	4	1	2	2	2
$pump_1\_fault$	$false$	$false$	$false$	$false$	$false$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	1	1	1
out_{pump_1}	2	2	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	2	2	1	1	1
out_{pump_2}	2	2	1	1	1
$dump$	$false$	$false$	$false$	$false$	$false$
vol_{tank}	0	3	4	5	6

As seen in the example, if we consider $\mathcal{I} = \{control\}$, here is a possible passing trace:

$Time$	0	1	2	3	4
out_{tank}	4	1	2	2	2
$pump_1\_fault$	$false$	$false$	$false$	$true$	$true$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	1	2	2
out_{pump_1}	2	2	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	2	2	1	0	0
out_{pump_2}	2	2	1	0	0
$dump$	$false$	$false$	$false$	$false$	$false$
vol_{tank}	0	3	4	4	4

The angelic values for $pump_1\_fault$ *are* $(false, false, false, true, true)$.

From there, a fix could be computed for $pump_1\_fault$ *which would be* $pump_1\_fault = false \rightarrow \big(pre(pump_1\_fault) \vee (pre(com_{pump_1}) = pre(out_{pump_1}))\big)$. *The reason we use the pre in the synthesis is because we cannot add variables that have a dependency chain on themselves. Therefore, the controller cannot detect the fault when it occurs, but the next instant at the earliest.*

By using the mixed or the game approach for the first step of [Xuan et al., 2016], we replace the test suite by a failing trace and a model of the system.

Note that this is also possible to take into account several failing traces using the mixed or the game approach for the first step.

Example 36 *Let us consider the same system as in Example 35, with the fix taken into account, but with the addition of an output for the pumps,* KO_i, *with* i *the number of the pump, that reflects an auto-diagnosis of the pumps. These flows would be inputs of control.*

$Time$	0	1	2	3	4
out_{tank}	4	1	2	0	2
$pump_1\_fault$	$false$	$false$	$false$	$false$	$false$
$pump_2\_fault$	$false$	$false$	$false$	$false$	$false$
com_{pump_1}	2	2	1	1	1
KO_1	$false$	$true$	$true$	$true$	$true$
out_{pump_1}	2	2	$\underline{2}$	$\underline{2}$	$\underline{2}$
com_{pump_2}	2	2	1	1	1
KO_2	$false$	$false$	$false$	$false$	$false$
out_{pump_2}	2	2	1	1	1
$dump$	$false$	$false$	$false$	$false$	$false$
vol_{tank}	0	3	4	6	6

Here, $pump_1$ detects its fault at instant 1, and since this information is not taken into account by control, the tank overflows at instant 3

Let us build a play-through where $\mathcal{I} = \{Control\}$:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*4*	*1*	*2*	*0*	*2*
$pump_1\_fault$	*false*	*false*	*false*	*false*	*false*
$pump_2\_fault$	*false*	*false*	*true*	*true*	*true*
com_{pump_1}	*2*	*2*	*2*	*2*	*2*
KO_1	*false*	*true*	*true*	*true*	*true*
out_{pump_1}	*2*	*2*	*2*	*2*	*2*
com_{pump_2}	*2*	*2*	*0*	*0*	*0*
KO_2	*false*	*false*	*false*	*false*	*false*
out_{pump_2}	*2*	*2*	*0*	*0*	*0*
dump	*false*	*false*	*false*	*false*	*false*
vol_{tank}	*0*	*3*	*3*	*5*	*5*

The angelic values for $pump_1\_fault$ are $(false, false, true, true, true)$.

From there, a fix could be computed for $pump_1\_fault$ which would be $pump_1\_fault = false \rightarrow (pre(pump_1\_fault) \vee (pre(KO_1))$.

Note that this fix is compatible with the previous one, and we can combine them into:

$$pump_1\_fault = false \rightarrow \big(pre(pump_1\_fault) \vee (pre(KO_1) \wedge (pre(com_{pump_1}) = pre(out_{pump_1})\big)$$

This Subsection showed that the counter-factuals build with the mixed or the game approach can also be used as an input to approaches that rely on a passing trace close from a failing one. Using the game approach is necessary when you want to ensure that the conditions added only rely on information from the component. If you can modify the amount of information the component has access to, the mixed approach is enough.

The argument to use the mixed/game approach is the following: since the synthesis is necessary to most of the causality definition, its result can then be used as a set of passing traces close to the initial one. The "closeness" notion relies on the one used by the way of building the counter-factuals. Those passing traces can then be used as the input of the methods that contrast passing with failing traces. The reason those approaches can be a good way of generating the traces is because they use an explicit causality definition to build the said traces, and as Section 2.3 has shown, results are generally better when using an explicit causality definition. This claim, however is not based on testing or any proof, but on a general observation.

6.4.2 Extending the game framework to find fixes

In Section 6, we defined the strategies as respecting the specification. It means that the strategy only controls the non-deterministic choices of the component. It could be interesting the have richer strategies in order to be able to propose fixes to a buggy white-box component.

Definition 57 (expanded strategy $\widetilde{strat}$) *Let $C = (I_C, O_C, M_C, \hat{\mathcal{S}})$ be a component, let $\widetilde{\mathcal{S}}$ be a step mode over (I_C, O_C, M_C) such that $\hat{\mathcal{S}} \subseteq \widetilde{\mathcal{S}}$. A expanded strategy is a a step model such that $\widetilde{strat} \subseteq \widetilde{\mathcal{S}}$*

Here, the idea is to expand the possible behaviours for the component C from $\hat{\mathcal{S}}$ to $\widetilde{\mathcal{S}}$, and then taking the strategies in this new bigger set of behaviours. $\widetilde{\mathcal{S}} \backslash \hat{\mathcal{S}}$ is basically some slack that is authorised on the model of the component.

Let us consider an example.

Example 37 *The system we study has the same architecture as Example 26, i.e. a controller, two pumps and a tank. For simplicity sake, we remove the non-deterministic choice from the controller.*

However, we consider $\widetilde{\mathcal{S}}_{tank}$ which is the same as $\hat{\mathcal{S}}_{tank}$ with the exception that can dump 0, 1 or 2 unit of liquid (instead of 0 or 1).

The considered trace is the following:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*0*	*4*	*2*	*2*	*2*
com_{pump_1}	*2*	*2*	*1*	*1*	*1*
out_{pump_1}	*2*	*2*	*2*	*2*	*2*
com_{pump_2}	*1*	*2*	*1*	*1*	*1*
out_{pump_2}	*1*	*2*	*2*	*2*	*2*
dump	*0*	*0*	*-1*	*-1*	*-1*
vol_{tank}	*3*	*3*	*4*	*5*	*6*

Here, since the two pumps become faulty, the tank cannot prevent the overflow, even when dumping at each timestep. Therefore, there is no winning strategy with $\hat{\mathcal{S}}_{tank}$.

If we consider the generalised strategies for $\mathcal{R} = \{tank\}$, there are several winning strategies, like:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*0*	*4*	*2*	*2*	*2*
com_{pump_1}	*2*	*2*	*1*	*1*	*1*
out_{pump_1}	*2*	*2*	*2*	*2*	*2*
com_{pump_2}	*1*	*2*	*1*	*1*	*1*
out_{pump_2}	*1*	*2*	*2*	*2*	*2*
dump	*0*	*0*	*-1*	*-1*	*-2*
vol_{tank}	*3*	*3*	*4*	*5*	*5*

Here, dump takes the value -2 *at the last timestep. It solves the problem, but is not really satisfactory, in term of a general fix.*

A strategy that better matches the intuition is:

Time	*0*	*1*	*2*	*3*	*4*
out_{tank}	*0*	*1*	*4*	*4*	*4*
com_{pump_1}	*2*	*1*	*2*	*2*	*2*
out_{pump_1}	*2*	*1*	*2*	*2*	*2*
com_{pump_2}	*1*	*0*	*2*	*2*	*2*
out_{pump_2}	*1*	*0*	*2*	*2*	*2*
vol_{tank}	*3*	*3*	*3*	*3*	*3*

Here, the tank dumps 2 *whenever not doing so would result in its volume becoming greater than 3.*

Obviously, the second version is better, however the first one does solve the problem.

This example showed that there generally are many possible solutions for the fix. This problem could be tackled by having multiple failing traces, which may refine the fix. Another way would be to propose several strategies to the user, and let him choose the best one.

Though the framework can find fixes in theory, in practice, it may be hard to actually synthesise the strategies, with the slack introduced increasing the size of the state-space. However, with very specific slack (like here, adding one more possible value to *dump*), it may be possible to synthesise the fixes on small systems.

Conclusion

This section presented an extension to the mixed-framework that is twofold. The first one is to move from the global ticks perspective to a finer grain description of the system that takes into account the evolution of the system during a step. This required to change the specifications/models to step

specifications/models, that better represent the functioning of the components. It enabled us to be able to assess the responsibility of components or set of components on a failure. The way we treat the sets of components is to suppose that they can coordinate on a global level the choice of their strategies, however, the strategies are locals, meaning that the components only have access to the information they are supposed to have access to at runtime.

The second extension is to move to a game framework, that enables us to add a second player. We can then emulate behaviours where a set of components chose their strategies to try to make the system fail, while the protagonists try to prevent the failure. It opens up a lot of possibilities, in terms of new definitions, as was shown in Section 6.2. Those definitions are interesting both in a design mindset (can we prevent this set of components from making the system fail) or in a responsibility one (are the strategies that may correspond to what happened in the initial trace always spoiling ones).

This section also describes a way of computing the winning and spoiling strategies in practice. Nevertheless, the offset, in term of computation cost, of eliminating the transitions in the LTS has not been assessed, and we cannot use the classic results from the controller synthesis.

Lastly, extensions to automated fix generator are sketched in Section 6.4.

Section 7

Impact of information on the Causality Analysis framework

This section presents some studies on the impact of the amount of information we have access to on the accuracy of causality analysis (being able to accurately assess responsibility), and some extensions that can be derived from this impact. This section will be divided in two subsections.
The first one will present some results on causality analysis performed on reduced logs. For systems that have a lot of components and/or that run for an extensive period of time, the traces can become big. Being able to perform causality on reduced logs is then important, to reduce the amount of data necessary, and possibly reducing the cost of the analysis, by having less data to process .
The second subsection is a generalisation of the work in [Wang et al., 2015]. Though the faulty behaviour can be embedded in the system behavioural model, it is not convenient to use in practice, since a new behavioural model must be computed if a fault model changes, and local fault models are easier to manipulate than a global behavioural model. This subsection gives a general framework to use fault models in causality analysis, as well as a method to refine the causes and a way to be able to reduce the number of analyses performed, if components have multiple fault modes.

All the results presented in this section are contributions from this thesis.

7.1 Causality Analysis on reduced logs

The idea is to perform causality analysis on partial logs, without losing too much in accuracy. Here the notion of partial log is either traces that do not

cover the whole execution (e.g. only the last 10 ticks, or every other tick), that are called partial in time, or traces that do not contain every variable (e.g. only the inputs of the components), those are called partial in space. There is very little to change to the current framework to allow the use partial logs, since it reasons on global traces from the behavioural model. The traces in the counter-factuals must match the unaffected-prefixes, and even-though the unaffected prefixes are partial, the counter-factuals can still be built. Nonetheless, the set of counter-factuals will be bigger if the logs are partial, since the possible behaviours will be less constrained.

The first subsection will present general properties on partial log. The second one will feature some ideas and thoughts about partial logs in space. The third one will introduce a method to keep the same precision with partial logs in time, under certain assumptions. The last one will present some thoughts on how adding some information to the logs may help reducing them overall.

7.1.1 General results

This section will formalise causality analysis on reduced logs and give some results about necessary and sufficient causality on those logs.

Definition 58 (Log) *Let $\mathbb{F}$ be a set of flows. A log is a element of $LOG_{\mathbb{F}} = \left(\bigtimes_{f\in\mathbb{F}} D_f \cup \{\bot\}\right)^{\infty}$, with D_f the domain of a flow, and $\bot$ denoting an absence of logged information.*

A log is a trace which can lack some information. For instance, if f is a Boolean flow, $(true, \bot, false)$ is a log over f. The $\bot$ value could be any possible value from the domain of f, i.e. *true* of *false*, here.

Definition 59 (Counter-factuals on log (CF_{log})) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{log} \in LOG_{\mathbb{F}}$ be a log over $\mathbb{F}$, and $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected components. The counter-factuals on $\underline{log}$ is defined a follow:*

$$CF_{LOG}(\underline{log}, \mathcal{I}) = \bigcup_{\underline{tr}\in\mathbb{TR}(\underline{log})} (CF(\underline{tr}, \mathcal{I}))$$

With $\mathbb{TR}(\underline{log}) = \{\underline{tr} \in BM \mid |\underline{tr}| = |\underline{log}| \wedge \forall f \in \mathbb{F}, \forall i \in [0..|\underline{tr}|-1], \pi_f(\underline{log}[i]) \neq \bot \implies \pi_f(\underline{log}[i]) = \pi_f(\underline{tr}[i])\}$ the set of all possible traces corresponding to the log.

The counter-factuals on a log is the union of all counter-factuals of the traces which can corresponds to the log. What $\mathbb{TR}(\underline{log})$ is to build the set of possible system traces that may corresponds to $\underline{log}$ and have the same length.

The definition of necessary and sufficient causality are transposed to the logs by replacing $CF(\underline{tr}, \mathcal{I})$ by $CF(\log, \mathcal{I})$ in the definitions. We then get nec_{LOG} and suff_{LOG}.

Theorem 3 *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{log} \in LOG_{\mathbb{F}}$ be a log and $\underline{tr} \in \mathbb{TR}$ be a corresponding trace (with $\mathbb{TR}$ as in Definition 59).*

$$\text{nec}_{LOG}(\log, \mathcal{I}) \implies \text{nec}(\underline{tr}, \mathcal{I})$$

By definition, $CF(\underline{tr}, \mathcal{I}) \subseteq CF_{LOG}(\log, \mathcal{I})$, therefore, $(CF_{LOG}(\log, \mathcal{I}) \subseteq \mathcal{P}) \implies (CF(\underline{tr}, \mathcal{I}) \subseteq \mathcal{P})$, which given the definitions of necessary causality gives us the theorem.

This theorem means that if $\mathcal{I}$ is a necessary cause on the log, it is as well on the corresponding trace. It means that the necessary cause is conservative over the logs.

Theorem 4 *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{log} = LOG_{\mathbb{F}}$ be a log and $\underline{tr} \in \mathbb{TR}$ be a corresponding trace (with $\mathbb{TR}$ as in Definition 59).*

$$\neg\, \text{nec}_{LOG}(\underline{log}, \mathcal{I}) \not\Longrightarrow \neg\, \text{nec}(\underline{tr}, \mathcal{I})$$

$$\text{suff}_{LOG}(\underline{log}, \mathcal{I}) \not\Longrightarrow \text{suff}(\underline{tr}, \mathcal{I})$$

$$\neg\, \text{suff}_{LOG}(\underline{log}, \mathcal{I}) \not\Longrightarrow \neg\, \text{suff}(\underline{tr}, \mathcal{I})$$

$\neg\, \text{nec}_{LOG}(\underline{log}, \mathcal{I})$ means that $CF_{LOG}(\underline{log}, \mathcal{I}) \not\subseteq \mathcal{P}$. However, basic set theory tells us that $(A \subseteq B \wedge B \not\subseteq \mathcal{P}) \not\Longrightarrow A \not\subseteq \mathcal{P}$, hence we have do not have any information on $CF(\underline{tr}, \mathcal{I}) \not\subseteq \mathcal{P}$, thus the result from the theorem.
It is easy to build a counter example to $\text{suff}_{LOG}(\underline{log}, \mathcal{I}) \implies \text{suff}(\underline{tr}, \mathcal{I})$, as it will be shown in Example 38.
$\neg\, \text{suff}_{LOG}(\underline{log}, \mathcal{I})$ means that $sup(CF_{LOG}(\underline{log}, \mathcal{I})) \cap \mathcal{P} \neq \emptyset$. However, $(A \subseteq B \wedge (B \cap \mathcal{P} \neq \emptyset) \not\Longrightarrow (A \cap \mathcal{P} \neq \emptyset)$, hence we have do not have any information on $CF(\underline{tr}, \mathcal{I}) \cap \mathcal{P} = \emptyset$, thus the result from the theorem.
Those two theorems shows that only the fact that $\mathcal{I}$ is a necessary cause on the log can be transposed to the actual trace. Thus, we need some other hypotheses to be able to transpose the causality analysis results from the log the trace actual trace that is represented by the log.

Example 38 *Let us consider a system with two components, C_1 and C_2. C_1 has only one Boolean output, Out_1, that should always be true. C_2 is the identity and has Out_1 as its Boolean input, and Out_2 as its Boolean output. The system property is that Out_2 should always be true. The behavioural model is that C_2 cannot output true more that two times in a row, Let us suppose we have the following trace $\underline{tr}$:*

Time	*0*	*1*	
Out_1	*true*	*true*	*false*
Out_2	*true*	*true*	*false*

The partial log $\underline{log}$ is the following:

Time	*0*	*1*	*2*
Out_1	*true*	*true*	$\perp$
Out_2	*true*	*true*	*false*

Let us suppose that $\mathcal{I} = \{C_1\}$. The counter-factual are the following for $\underline{tr}$:

Time	*0*	*1*
Out_1	*true*	*true*
Out_2	*true*	*true*

The counter-factuals must stop at the second tick, because of the BM. If the trace is extended, C_2 must become faulty, which is not authorised in the cone, nor unaffected-prefixes approach.

Now, let us build the counter-factuals if we suppose that $\perp$ is replaced by false at the second tick:

Time	*0*	*1*	*2*	...
Out_1	*true*	*true*	*true*	...
Out_2	*true*	*true*	*false*	...

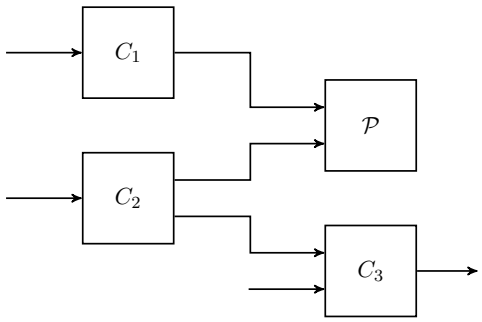
This is the prefix to any counter-factual produced with the trace where $\perp$ correspond to true. They are always failing, and prefixed by the counter-factuals from the trace.

We have $sup(CF_{LOG}(\underline{log}, \mathcal{I})) \cap \mathcal{P} = \emptyset)$ and $\neg sup(CF(\underline{tr}, \mathcal{I})) \cap \mathcal{P} \neq \emptyset)$, i.e. $\text{suff}_{LOG}(\underline{log}, \mathcal{I}) \wedge \neg\, \text{suff}(\underline{tr}, \mathcal{I})$*, which is a counter-example to* $\text{suff}_{LOG}(\underline{log}, \mathcal{I}) \implies \text{suff}(\underline{tr}, \mathcal{I})$.

7.1.2 Reduced logs in space

As explained previously, the idea is not to log certain variables while keeping an accurate causality analysis.

Example 39 *Let us consider the following architecture:*



The nodes are components and the directed arcs are flows. If an arc is directed from a component, the corresponding flow is an output flow of the component. If an arc is directed at a component, the corresponding flow is an input of the component.

The node $\mathcal{P}$ corresponds to the system property. It only depends on the outputs from the components C_1 and C_2. Since C_1 and C_2 do not receive data from C_3, not logging C_3 will not change the outcome of the causality analysis.

One might be tempted not to log the output from C_2 to C_3, but it might be important to verify whether C_2 satisfies its specification or not.

The previous example shows how we can perform causality analysis on partial logs in space, without losing any accuracy.

Here is an algorithm that could be applied to compute which flows should be logged, supposing we are given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$.

1. Recursively compute the set of flows on which $\mathcal{P}$ depends. I.e. compute the set flows, named $\mathbb{E}$, $\mathcal{P}$ depends on, then the set of flows $\mathbb{E}$ depends on, and so forth. The result is stored $\mathbb{E}_{\mathcal{P}}$.

2. $\mathbb{C}' = \{C \in \mathbb{C} \mid \exists f \in \mathbb{E}_{\mathcal{P}}, f \in I_C \cup O_C\}$, for all component C in $\mathbb{C}'$ compute $\mathbb{E}_C$, in a similar fashion to $\mathbb{E}_{\mathcal{P}}$, adding to $\mathbb{C}'$ each component that has a flow in a $\mathbb{E}_C$.

3. $\mathbb{E}_{log} = \mathbb{E}_{\mathcal{P}} \cup \bigcup_{C \in \mathbb{C}'} \mathbb{E}_C$ is the set of flows that should be logged.

This is naive algorithm to compute the dependencies. However there might more efficient (eliminating more flows) ways to do so, such as slicing for programs. Nonetheless, it shows an example of an offline analysis that might reduce the number of flows that needs to be logged. If applied to example 39, it would remove the flow from C_3, and the one going from C_2 to C_3, depending on whether or not the respect of the specification of C_2 depends on it.

A more refined algorithm could use further information, for instance taking into account the behavioural model, to perform a finer analysis of the dependencies. It is also worth considering online tools, like monitoring ([Liao and Cohen, 1992]), in order to decide whether a flow should be logged or not. This is a similar idea to the one of dynamic slicing here, the monitoring is used to assess whether or not a flow needs to be logged, based on runtime information.

7.1.3 Reduced logs in time

Contrary to the logs reduced in space, every variable are logged, however the logs do not contain the whole trace for each of them, but a sub-trace. Some definitions will first be introduced, and an approach that enable causality analysis, with no loss of accuracy, on partial logs in time will be presented.

There will be a lot of new definitions introduced in this section, so here is the underlying idea, so that the reader can know where all this is going. The way we ensure that we can perform the same causality analysis with partial logs and a full trace, is to make sure that we build the same counter-factuals. To do so, the grey-components are introduced, that have some information on how long a trace is needed to compute their outputs (namely the critical length). Using the critical length, it is possible to build logs called "critical logs" that will enable the construction of counter-factuals that are the same as the ones build from the full trace.

For the remainder of the subsubsection, we suppose we are given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$

Definition 60 (Grey-box component) *A grey-box component is a component $C = (I_C, O_C, \mathcal{S}_C, \underline{L}_C)$, with a critical length vector $\underline{L}_C = (l_f^C)_{f \in I_C \cup O_C}$ such that $\forall \underline{tr}, \underline{tr}' \in \mathcal{S}_C, \forall i \in [0..(min(|\underline{tr}|, |\underline{tr}'|) - 2)]$,,*

$$\big(\forall f \in I_C, (\pi_f(\underline{tr}[max(0, i - l_f^C + 1)..i]) = \pi_f(\underline{tr}'[max(0, i - l_f^C + 1)..i]) \wedge$$

$$\forall f \in O_C, \pi_f(\underline{tr}[max(0, i - l_f^C + 1)..i - 1]) = \pi_f(\underline{tr}'[max(0, i - l_f^C + 1)..i - 1]))\big)$$
$$\implies \underline{tr}[i] = \underline{tr}'[i]$$

Intuitively, a grey-box component is a component for which the output at an instant t only depends on the inputs f from $t - l_f^C + 1$ to t and the outputs f' from $t - l_{f'}^C + 1$ to $t - 1$.
Note that if the premise is *true*, $\underline{tr}[i] = \underline{tr'}[i]$ only corresponds to the constraint $\forall f \in O_C, \pi_f(\underline{tr}[i]) = \pi_f(\underline{tr'}[i])$, since the equality is supposed in the premise for the input flows.
This definition implicitly supposes that the component only has one initial state. It would be possible to take into account different initial states, by adding them as a prerequisite (like $initial\_state_{\underline{tr}} = initial\_state_{\underline{tr'}}$). This only impacts the trace from 0 to $l_f^C - 1$. However, for simplicity sake, we only consider components with one initial state (like in LUSTRE).
The property only needs to hold on the non-faulty behaviours.
A flow for which all the values must be kept will have a critical length $l_f^C = \infty$.

Example 40 *Here are examples of components, with their respective critical length.*

*Let add*0 *be a component with one input a and one output x. This component returns to x the value of a plus one.*
*An example of trace on add*0 *is* $Tr = ((1,2),(4,5),(2,3),(12,13))$.
The output at a given instant only depends on the current value of a. The critical lengths are thus $l_a^{add0} = 1$ *and* $l_x^{add0} = 0$

*Let add*1 *be such that x is the sum of a, and the previous value of a (i.e.* $x[t] = a[t] + a[t-1]$*). x*[0] *is initialised at a*[0].
*An example of trace on add*1 *is* $Tr = ((1,1),(4,5),(2,6),(12,14))$*. The output at a given instant only depends on the current value of a, and the previous one. Therefore,* $l_a^{add1} = 2$ *and* $l_x^{add1} = 0$.

*Let us now consider add*2 *which waits two values of a to output the sum on x (i.e. if* $x[t-1] = \perp$, $x[t] = a[t] + a[t-1]$, *else* $x[t] = \perp$*). We note* $\perp$ *when x does not output a value.*
*An example of trace on add*2 *is* $Tr = ((1,\perp),(4,5),(2,\perp),(12,14))$.
The output at a given instant only depends on the current and previous value of a, and the previous value of x (actually the value of the output is not important, only the presence or absence of a value is). Therefore, $l_a^{add2} = 2$ *and* $l_x^{add2} = 2$.

*Let us consider a last example add*3*. Every two values of a, x is updated to be sum of a and its previous value (i.e. if* $(t \mod 2) = 1$, $x[t] = a[t] + a[t-1]$, *else,* $x[t] = x[t-1]$*).*
*An example of trace on add*3 *is* $Tr = ((1,0),(4,5),(2,5),(12,14))$.
If we consider a sub-trace $((0,0),(0,0),(0,0))$*, we cannot determine if the next x should be the current a, or* 0*. We need to know the parity of the number of a received. Therefore,* $l_a^{add3} = 2$ *and* $l_x^{add3} = \infty$.

This example illustrates that components with close specifications can have different critical lengths, depending on the information needed to compute the outputs of the component, and what we can infer on the internal state of the component from its inputs and outputs. In those examples, l_a corresponds to the data needed to compute the current value of x, and l_x corresponds to the data we need to keep (alongside the last l_a values of a), in order to have an idea of the current internal state of the component.

In order to be conservative, the critical length is the worst case value. If we consider *add*3, most of the traces actually have a critical length of 2 for x. However, the only sub-trace for which l_x^{add3} need to be ∞ instead of 2 is $((0,0),(0,0),(0,0))$.

Definition 61 (Property critical length $\underline{L_{\mathcal{P}}}$) *Let $\mathbb{F}$ be a set of flow, and $\mathcal{P}$ a system property. Let $\mathbb{F}_{\mathcal{P}} = \{f \in \mathbb{F} \mid \exists(\underline{tr}, \underline{tr'}) \in (\mathcal{B}_{\mathbb{F}}, \mathcal{B}_{\mathbb{F}\setminus\{f\}}), \pi_{\mathbb{F}\setminus\{f\}}(\underline{tr}) = \underline{tr'} \wedge \underline{tr'} \in \pi_{\mathbb{F}\setminus\{f\}}(\mathcal{P}) \wedge \underline{tr} \notin \mathcal{P}\}$. $\underline{L_{\mathcal{P}}} = (l_f^{\mathcal{P}})_{f\in\mathbb{F}_{\mathcal{P}}}$ is the critical length of the component $C_{\mathcal{P}} = (\mathbb{F}_{\mathcal{P}}, \{(OK_{\mathcal{P}}, \mathbb{B})\}, \mathcal{S}_{\mathcal{P}}, \underline{L_{\mathcal{P}}})$ such that*

$$\mathcal{S}_{\mathcal{P}} = \{\underline{tr} \in \mathcal{B}_{C_{\mathcal{P}}} \mid \forall i \in [0..|\underline{tr}|-1], OK_{\mathcal{P}}[i] = \pi_{\mathbb{F}_{\mathcal{P}}}(\underline{tr}[0..i]) \in \pi_{\mathbb{F}_{\mathcal{P}}}(\mathcal{P})\}$$

Intuitively, $\mathbb{F}_{\mathcal{P}}$ is the set of flows the property depends on. We build a component that has $\mathbb{F}_{\mathcal{P}}$ as input, and a Boolean flow $OK_{\mathcal{P}}$. $OK_{\mathcal{P}}$ is *true* when the trace is in $\mathcal{P}$, and false otherwise. Since $\mathcal{P}$ only depends on $\mathbb{F}_{\mathcal{P}}$, restricting the property to this set gives the same result as if we evaluate $\mathcal{P}$ on all the flows. The critical length of $\mathcal{P}$ is the critical length of the component $C_{\mathcal{P}}$. By construction, $\mathcal{S}_{\mathcal{P}}$ is prefix-closed and $\mathbb{F}_{\mathcal{P}}$ receptive.

We also suppose that for each flow f, we have a critical length l_f^{BM}, computed from the constraints in BM, using the construction of a similar component $C_{BM} = (F_{BM}, OK_{BM}, \mathcal{S}_{BM})$ for the behavioural model. For instance, in Example 10 (early preemption), the constraint that states that a faulty *pump* keeps the same output when faulty means that we need the previous value of the *pump* output to be able to compute the next (faulty) one. Then $l_{out_{pump}}^{BM} = 2$.
Note that if $BM = \mathcal{B}_{\mathbb{F}}$, then $\mathbb{F}_{BM} = \emptyset$, and thus, the behavioural model has no impact on the critical lengths.

We build the vector of maximal critical length as follow: $\underline{L} = (l_f)_{f\in\mathbb{F}}$ with $\forall f \in \mathbb{F}, l_f = max(\{l_f^C \mid C \in \mathbb{C} \wedge f \in I_C \cup O_C\} \cup \{l_f^{\mathcal{P}} \mid f \in \mathbb{F}_{\mathcal{P}}\} \cup \{l_f^{BM} \mid f \in \mathbb{F}_{BM}\})$. This is the set containing the maximum critical length for every flow. The max function is applied to the set of all critical lengths that are related to the flow.

Let us illustrate the notion of critical length with an example.

Example 41 *Let us consider the system that is similar to the one from Example 26, but with only one pump (i.e. pump, a controller and a tank). However, here, the specification will be regular ones, and not step ones.*

pump *The pump has the usual input, the usual output, but its domain is* $[0..2]$ *and the output can only be altered by* 1 *at each tick. The specification is the following:*

- $I_{pump} = \{(com_{pump}, [0..2])\}$
- $O_{pump} = \{(out_{pump}, [0..2])\}$
- $\mathcal{S}_{pump} = \{\underline{tr} \in \mathcal{B}_{pump} \mid \forall i \in [O..|\underline{tr}|-1], \pi_{out_{pump}}(\underline{tr}[i]) = \pi_{out_{pump}}(\underline{tr}[i-1]) + sgn(\pi_{com_{pump}}(\underline{tr}[i]), \pi_{out_{pump}}(\underline{tr}([i-1]))\}$, *with* $\pi_{out_{pump}}(\underline{tr}[-1]) = 0$ *and* $sgn(a, b) = -1$ *if* $a - b < 0$, 1 *if* $a - b > 0$ *and* 0 *otherwise.*

tank *The tank now has a second input corresponding to the output of liquid taken from the tank.*

- $I_{tank} = \{(out_{pump}, [0..2]), (out_{tank}, [0..2])\}$
- $O_{tank} = \{(vol_{tank}, \mathbb{Z})\}$
- $\mathcal{S}_{tank} = \{tr \in \mathcal{B}_{pump} \mid \forall i[0..|\underline{tr}| - 1], \pi_{vol_{tank}}(\underline{tr}[i]) = \pi_{vol_{tank}}(\underline{tr}[i - 1]) + \pi_{out_{pump}}(\underline{tr}[i]) - \pi_{out_{tank}}(\underline{tr}[i])\}$, *with* $\pi_{vol_{tank}}(\underline{tr}[-1]) = 0$

control *The controller takes* out_{pump}, vol_{tank} *and* out_{tank} *as inputs and outputs* com_{pump}.

- $I_{control} = \{(out_{pump}, [0..2]), (vol_{tank}, \mathbb{Z}), (out_{tank}, [0..2])\}$
- $O_{control} = \{(com_{pump}, [0..2])\}$
- *The specification is a bit complicated. The idea that sums it up is that it tries to get the future value* vol_{tank} *as close as possible from* 3, *supposing all components are non-faulty.*

System property *The system property is that the volume in the tank should always be between* 0 *and* 5. $\mathcal{P} = \{\underline{tr} \in \mathcal{B}_{\mathbb{F}} \mid \forall i \in [0..|\underline{tr}|-1]\pi_{vol_{tank}}(\underline{tr}[i] \in [0..5]\}$.

Behavioural model *BM is such that the tank can "leak", at maximum,* 1 *unit of liquid over a period of two time-step.*

Here are the different critical lengths:

pump $l^{pump}_{com_{pump}} = 1$ *and* $l^{pump}_{out_{pump}} = 2$, *since* out_{pump} *depends on it previous value and the current value of* com_{pump}.

tank $l^{tank}_{out_{pump}} = 1$, $l^{tank}_{out_{tank}} = 1$ *and* $l^{tank}_{vol_{tank}} = 2$. vol_{tank} *depends on its previous value, the current one of* out_{pump} *and* out_{tank}.

control $l^{control}_{out_{pump}} = 2$, $l^{control}_{out_{tank}} = 1$, $l^{control}_{vol_{tank}} = 2$ *and* $l^{control}_{com_{pump}} = 1$. *Control tries to keep the future value of* vol_{tank} *as close as possible from* 3. *To do so, it predicts the values that should be output by the different components. Therefore, the critical length are the maximum of those of the other components.*

System property $\mathcal{P}$ *only depends on the current value of* vol_{tank}. *Hence,* $l^{\mathcal{P}}_{vol_{tank}} = 1$.

Behavioural model BM *depends on the previous and current value of* vol_{tank}, *and its previous and current expected value. Then, the critical lengths are the same as the tank one, incremented by one (to get the previous expected value).* $l^{B}_{out_{pump}}M = 2$, $l^{B}_{out_{tank}}M = 2$ *and* $l^{B}_{vol_{tank}}M = 3$.

By taking the maximum of the different critical lengths for each flow, we get the following vector of maximum critical length: $\underline{L} = (l_{com_{pump}}, l_{out_{pump}}, l_{out_{tank}}, l_{vol_{tank}})$ *with* $l_{com_{pump}} = 1$, $l_{out_{pump}} = 2$, $l_{out_{tank}} = 2$ *and* $l_{vol_{tank}} = 3$.

Definition 62 (Partial log) *Let* $\mathbb{F}$ *be a set of flow* $\underline{tr}$ *is a partial log over* $\mathcal{B}_{\mathbb{F}}$ *if* $\exists \underline{tr}' \in \mathcal{B}_{\mathbb{F}}, \forall f \in \mathbb{F}, \pi_f(\underline{tr}) = \pi_f(\underline{tr}'[(|\underline{tr}'| - |\pi_f(\underline{tr})|)..])$

A partial log is a trace that contains suffixes of a complete trace for each flow, such that all the suffixes finishes at the same instant. We do not need to resort to an absence value $\perp$ as in Definition 58 (log), since all the unknown value are in prefixes of the log.

We note $\mathcal{B}^{part}_{\mathbb{F}}$ all the possible partial logs over $\mathbb{F}$.

Definition 63 (Critical log $\underline{cl}$) *Let* $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ *be a trace and* $\underline{L} = (l_f)_{f\in\mathbb{F}}$ *the vector of maximal critical length for each flow. Let* $\mathbb{F}_{\mathcal{P}}$ *and* $\mathbb{F}_{BM}$ *as in Definition 61. Let* $(cone_C)_{C\in\mathbb{C}} = \text{cone}(\underline{tr}, \mathbb{C})$ *(with* cone *the function that builds the cone for the counter-factuals),* $cone_{\mathcal{P}} = min(\{i \in [0..|\underline{tr}| - 1] \mid \underline{tr}[0..i] \notin \mathcal{P}\} \cup \{|\underline{tr}|\})$ *and* $cone_{BM} = min(\{cone_C \mid C \in \mathbb{C} \wedge (\exists f \in \mathbb{F}_{BM}, f \in I_C \cup O_C)\} \cup \{|\underline{tr}|\})$. *The critical log, noted* $\underline{cl}(\underline{tr}, \underline{L}) \in \mathcal{B}^{part}_{\mathbb{F}}$ *is the partial log such that,* $\forall f \in \mathbb{F}$,

$$(fe(f, \underline{tr}) \neq |\underline{tr}| \implies \pi_f(\underline{cl}(\underline{tr}, \underline{L})) = \pi_f(\underline{tr}[max(fe(f, \underline{tr}) - l_f], 0)..])) \wedge \tag{7.1}$$

$$(fe(f, \underline{tr}) = |\underline{tr}| \implies \pi_f(\underline{cl}(\underline{tr}, \underline{L})) = \varepsilon)) \tag{7.2}$$

With $fe(f, \underline{tr}) = min(\{cone_C \mid C \in \mathbb{C} \wedge f \in I_C \cup O_C\} \cup \{cone_{\mathcal{P}} \mid f \in \mathbb{F}_{\mathcal{P}}\} \cup \{cone_{BM} \mid f \in \mathbb{F}_{BM}\})$

fe corresponds to the first entry of the flow into the cone.
The critical log is a partial log that contains, for each flow, the suffix of the initial trace containing the first entry of the flow in the cone, and the previous data corresponding to the critical length for that flow. Intuitively, this is the minimal data we need to build the counter-factuals, if we know the first fault for each components.
Constraint 7.1 is here to ensure that if a flow enters the cone, it is logged, as well as the previous data corresponding to the critical length. Constraint 7.2 treats the case of the flows that never enters the cone, and are thus not logged. If we treated them as in Constraint 7.1, their critical length would be logged, which is not necessary.
Note that for the grey-box approach, contrary to the regular of causality, the granularity shifted from the components to the flows. This is due to the fact that a flow might belong to several components and/or $\mathcal{P}$ and/or BM. Therefore, it is important to partially log for each flow, as reasoning with the components might result in logging too much. Indeed, any information we need is in the critical log, since all the part that are in the cone are, and all the data needed to prolong the traces is also, thanks to the critical length.

Example 42 *This example shows the critical logs for a trace on the system presented in Example 41.*

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*2*	*1*	*1*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*5*	*5*	*6*

At instant 7*, the pump does not reduce its output and at instant* 8 *the tank leaks one unit of fluid. For* $\text{cone}(\underline{tr}, \mathbb{C})$*,* $cone_{pump} = cone_{tank} = cone_{control} =$ 7*, since the pump communicates with the two other components.* $cone_{\mathcal{P}} = 9$*, since the property is violated at instant* 9*. Lastly,* $cone_{BM} = 7$*, since the minimum value of a component cone that contains a flow that impact the* BM *is* 7*. If we take into account the critical length, we get* $f_e(com_{pump}, \underline{tr}) = 7 - 1 + 1 = 7$*,* $f_e(out_{pump,\underline{tr})} = 7 - 2 + 1 = 6$*,* $f_e(out_{tank}, \underline{tr}) = 7 - 2 + 1 = 6$ *and* $f_e(vol_{tank}, \underline{tr}) = 7 - 3 + 1 = 5$*.*

Here is the corresponding critical log, with the removed part of the trace greyed:

$Time$	0	1	2	3	4	5	6	7	8	9
com_{pump}	2	2	2	2	1	1	2	0	0	0
out_{pump}	1	2	2	2	1	1	2	$\underline{2}$	$\underline{1}$	$\underline{1}$
out_{tank}	0	1	2	1	1	1	2	0	0	0
vol_{tank}	1	2	2	3	3	3	3	5	$\underline{5}$	$\underline{6}$

We can see in this example that the size of the critical logs is about one third of the initial trace. This stems from the fact that a big part of the trace, where the components behave accordingly to their specification, is removed. Would the non-faulty part be longer, the reduction would be even more important. Because this is a toy example with only three components, all the components are logged as soon as one component is faulty. However, on bigger systems, with more complex topology, not all the components are impacted when one component is faulty.

Property 3 *Let $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$, $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected components and $\underline{fv} = (fv_C)_{C\in\mathbb{C}}$ with $fv_C = min(i \in [0..|\underline{tr}| - 1] \mid \pi_C(\underline{tr}[0..i]) \notin \mathcal{S}_C \cup \{|\underline{tr}|\})$. We can compute* $\text{cone}(\underline{tr}, \mathcal{I})$ *using $\underline{fv}$ and the information of the system.*

From definition 14 (Cone of influence), it comes that $\text{cone}(\underline{tr}, \mathcal{I})$ only depends on the first instant the components in $\mathcal{I}$ become faulty (Constraint 3.1), the topology of the system and the fact that non-suspected component that should enter the cone are faulty or not (Constraint 3.2). Therefore, we can infer $\text{cone}(\underline{tr}, \mathcal{I})$ from $\underline{fv}$.

This result only applies to the synchronous framework, since we can derive the fact that components communicate using the topology. For this property to hold in the general case, we would have to introduce another critical length, corresponding to the communication model, and do a more thorough analysis of the critical logs, to determine if components communicate with one another.

Definition 64 (Counter-factuals with partial logs (CF_{pl})) *Let $\underline{tr} \in \mathcal{B}^{part}_{\mathbb{F}}$ be a partial log, n be the number of instants since the beginning of the execution, when $\underline{tr}$ ends, $\mathcal{I} \subseteq \mathbb{C}$ be a set of suspected components, $\underline{fv} = (fv_C)_{C\in\mathbb{C}}$ the first violations as defined in property 3 and $\underline{L} = (l_f)_{f\in\mathbb{F}}$ the vector of maximal critical length for each flow. The counter-factuals using partials logs are defined as follow: $CF_{pl}(\underline{tr}, \mathcal{I}, \underline{fv}, n) = \{\underline{tr'} \in BM, \forall C \in \mathbb{C},$*

$$\forall f \in I_C \cup O_C, \pi_f(\underline{tr'}[n_f .. \text{cone}_C - 1]) = \pi_C(\underline{tr}[0.. \text{cone}_C - 1 - n_f]) \wedge \quad (7.3)$$

$$fv_C \geqslant \text{cone}_C \implies \pi_C(\underline{tr'}) \in \mathcal{S}_C\} \quad (7.4)$$

With $\forall f \in \mathbb{F}, n_f = n - |\pi_f(\underline{tr})|$ and $(cone_C)_{C\in\mathbb{C}}$ computed from the first violations and the system topology.

The definition is the usual one, besides the fact that Constraint 7.3 is only on the parts that are logged. Constraint 7.4 is equivalent to the constraint usually used, since $fv_C \geqslant \text{cone}_C$ means that the component is not faulty upon entering the cone.

Contrary to the counter-factuals on log definition (CF_{LOG}, Def. 58) from Section 7.1.1, we do need to resort to the union of the counter-factuals for all the possible corresponding traces because we have access to the exact cone, via the first violations. This means that all the possible corresponding trace uses the same cone, which is the real one.

Lemma 1 *Let $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ be a trace, $\underline{L} = (l'_f)_{f\in\mathbb{F}}$ be the set of maximal critical length and $\underline{fv} = (fv_C)_{C\in\mathbb{C}}$ the first violations as defined in property 3. Let $(cone_C)_{C\in\mathbb{C}}$ be the cone for $\mathcal{I}$, build using $\underline{fv}$.*
Let $\underline{TR} = \{\underline{tr'} \in \mathcal{B}^{part}_{\mathbb{F}} \mid \exists \underline{tr''} \in CF(\underline{tr}, \mathcal{I}), \forall f \in \mathbb{F}, \pi_f(\underline{tr'}) = \pi_f(\underline{tr''}[max(fe(f, \underline{tr}) - l'_f], 0)..])\}$ and:
$\underline{TR'} = \{\underline{tr'} \in \mathcal{B}^{part}_{\mathbb{F}} \mid \exists \underline{tr''} \in CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1), \forall f \in \mathbb{F}, \pi_f(\underline{tr'}) = \pi_f(\underline{tr''}[max(fe(f, \underline{tr}) - l'_f], 0)..])\}$. *$\underline{TR} = \underline{TR'}$, with fe as in 63, using* $\text{cone}(\underline{tr}, \mathcal{I})$.

The result of this lemma is that the counter-factuals coincide after the entry of the flows in the critical logs, be they build with the critical logs, or the full trace.
$\mathbb{TR}$ (respectively $\mathbb{TR}'$) is the set of partial traces that start for each flow the first time it becomes logged, and coincide with a full trace from CF ($\underline{tr}$, $\mathcal{I}$) (resp. $CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1)$).
Since the demonstration is quite complicated, an example will be introduced first to give the intuition behind the lemma first.

Example 43 *This example uses the critical log presented in Example 42:*

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*2*	*1*	*1*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*5*	*5*	*6*

We are also given the first violation for each component, i.e. $fv_{pump} = 7$, $fv_{tank} = 8$ and $fv_{control} = 10$. We are also given $|tr| = 10$.

Let us build the counter-factuals for $\mathcal{I} = \{pump\}$.

Using $\underline{fv}$, we compute $\text{cone}(\underline{tr}, \mathcal{I}) = (7, 7, 7)$. *It give us the following unaffected prefixes:*

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*			
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*			
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*			

out_{tank} being only a component input, it is not shorten. For instant 7*, the value of out_{tank} being* 0*, and the previous volume of the tank being* 3*, the control component will output* 0*. Since the pump respects its specification, it will reduce its output by one, resulting in* 1 *for out_{pump}. Similarly, the tank respects its specification, and will increase its volume by* 1*, i.e. the value output by the pump, and so on. We get the following counter-factual:*

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*1*	*0*	*0*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*4*	*4*	*4*

Would the grey part have been used to build this counter-factual, the reconstructed part would be the same, since the value computed only depends on the data that have been critically logged.

To show how the BM is handled, let us consider the counter-factuals for $\mathcal{I} = \{tank\}$*.* $cone_{tank} = cone_{control} = 8$ *and* $cone_{pump} = 10$*, which give us the following unaffected prefixes::*

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*		
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*2*	*1*	*1*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*5*		

Note that out_{pump} is not cut down, since it is an output from the pump, which is already faulty when it should enter the cone. However, since the command from the control will remain the same as in the initial trace, it is sensible to consider that the pump would not have changed its behaviour.

The counter-factuals are the following:

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*2*	*1*	*1*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*5*	[5..6]	*6 if* $vol_{tank}[8] = 5$*, else* [6..7]

Since $vol_{tank}[7] = 5$ *is the expected value (computed using* $vol_{tank}[6]$, $out_{pump}[7]$ *and* $out_{tank}[7]$*), it is possible that the tank leaks at instant* 8*, hence the* [5..6]*. If* $vol_{tank}[8] = 5$*, the tank cannot leak at instant* 9*, since it can only leak once over two ticks, therefore the* $vol_{tank}[9] = 5$*. If* $vol_{tank}[8] = 6$*, we are in the same situation as in the previous instant, where the tank can leak, thus the* [6..7]*.*
Similarly to the counter-factuals for $\mathcal{I} = \{pump\}$*, the data used has been logged. Therefore the counter-factuals on the critical logs and on the full trace coincide after the flows are logged, and the counter-factuals would have been the same if the grey parts had been used.*

Proof 3 *Given Property 3, the cone in* CF_{cone} *and* CF_{pl} *are the same. It is noted* $\text{cone} = (cone_C)_{C\in\mathbb{C}}$*.*
$\forall f \in \mathbb{F}, i_f = fe(f, \underline{tr}) - l_f$*, the first instant* f *is logged in the critical prefix. To make the proof more readable, we note* $\underline{cl}$ *for* $\underline{cl}(\underline{tr}, \underline{L})$*,* CF_{cone} *for* $CF_{cone}(\underline{tr}, \mathcal{I})$ *and* CF_{pl} *for* $CF_{pl}((\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1))$*.*

Given Constraints 3.3 and 7.3 (conservation of the non-impacted parts), $\forall \underline{tr}' \in CF_{cone}, \forall f \in \mathbb{F}, \pi_f(\underline{tr}'[i_f..fe(f, \underline{tr}) - 1]) = \pi_f(\underline{tr}[i_f..fe(f, \underline{tr}) - 1])$ *and* $\forall \underline{tr}' \in CF_{pl}, \forall f \in \mathbb{F}, \pi_f(\underline{tr}'[i_f..fe(f, \underline{tr}) - 1]) = \pi_f(\underline{cl}[0..l_f - 1])$*. Since, by definition of* $\underline{cl}$*,* $\forall f \in \mathbb{F}, \pi_f(\underline{cl}[0..l_f - 1]) = \pi_f(\underline{tr}[i_f..fe(f, \underline{tr}) - 1])$*, then,* $\forall (\underline{tr}', \underline{tr}'') \in (CF_{cone} \times CF_{pl}), \forall f \in \mathbb{F}, \pi_f(\underline{tr}'[i_f..fe(f, \underline{tr})] - 1) = \pi_f(\underline{tr}[i_f..fe(f, \underline{tr})] - 1)$*. For each flow, from the first instant it is logged to the first instant it enters the cone, all the traces in both counter-factuals coincide.*

Given Definitions 63 (critical logs) and 61 (property critical length, used as it is for BM*),* $\forall \underline{tr}', \underline{tr}'' \in BM, \forall i \in [i_f..|\underline{tr}| - 1], (\forall f \in \mathbb{F}_{BM}, \pi_f(\underline{tr}'[i_f - l_f..i]) = \pi_f(\underline{tr}''[i_f - l_f..i])) \implies (\forall f \in \mathbb{F}, \pi_{OK_{BM}}(\underline{tr}'[i]) = \pi_{OK_{BM}}(\underline{tr}'[i]))$*, i.e.* $(\exists \hat{\underline{tr}} \in BM, \underline{tr}'[i_f..] = \hat{\underline{tr}}[i_f..]) = (\exists \hat{\underline{tr}} \in BM, \underline{tr}''[i_f..] = \hat{\underline{tr}}[i_f..])$*. Since* $\underline{tr}$ *is an actual trace and* BM *is prefix closed,* $\big((\forall i \in [i_f..|\underline{tr}| - 1], \pi_{OK_BM}(\underline{tr}[i]) = true) \wedge (\forall f \in \mathbb{F}, \pi_f(\underline{tr}[0..i_f - 1]) = \pi_f(\underline{tr}[0..i_f - 1])) \implies \underline{tr}' \in\big)$*, and a trace prefixed by* $\underline{tr}$ *before the* i_f *and for which* OK_{BM} *is always true after the* i_f *is a valid* $\hat{\underline{tr}}$*. Hence, given what was shown in the previous paragraph,* $\{\underline{tr}' \in \mathcal{B}^{part}_{\mathbb{F}} \mid \exists \underline{tr}'' \in BM, \forall f \in \mathbb{F}, \pi_f(\underline{tr}''[i_f..fe_{cone}(\underline{tr}, f) - 1]) = \pi_C(\underline{cl}[0..fe_{cone}(\underline{tr}, f) - i_f - 1]) \wedge \pi_C(\underline{tr}''[n_C..]) = \pi_C(\underline{tr}'))\} = \{\underline{tr}' \in \mathcal{B}^{part}_{\mathbb{F}} \mid \exists \underline{tr}'' \in BM, \forall f \in \mathbb{F}, \pi_f(\underline{tr}''[i_f..fe_{cone}(\underline{tr}, f) - 1]) = \pi_C($ $tr[i_f..fe_{cone}(\underline{tr}, f) - 1]) \wedge \pi_f(\underline{tr}''[i_f..]) = \pi_f(\underline{tr}'))\}$*. It means that the set of partial logs that respects Constraint 3.3 (the first constraint for the cone counter-factuals) and the one that respect Constraint7.3 (the first constraint for the counter-factuals with partial logs) coincide on the parts that are after the entry in the critical logs.*

Similarly, we prove the same property the second constraint of the counter-factuals (constraints 3.4 and 7.4). Since $\{\underline{tr}' \in BM \mid constraint_1 \wedge constraint_2\} = \{\underline{tr}' constraint_1\} \cap \{\underline{tr}' constraint_2\}$*, with* $constraint_i$ *a Boolean over* BM*, we have the result of the lemma.*

Lemma 2 *Let* $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ *be a trace,* $\underline{L} = (l'_f)_{f \in \mathbb{F}}$ *be the set of maximal critical length and* $\underline{fv} = (fv_C)_{C \in \mathbb{C}}$ *the first violations as defined in property 3. Let* $\text{cone}(\underline{tr}, \mathcal{I}) = (cone_C)_{C \in \mathbb{C}}$ *be the cone for* $\mathcal{I}$*, build using* $\underline{fv}$*. Let* $\mathcal{I} \subseteq \mathbb{C}$ *be a set of suspected component. If the system is well designed (the composition of the specification refines the property), then*

$$(CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1) \subseteq \mathcal{P}) = (CF_{\text{cone}}(\underline{tr}, \mathcal{I}) \subseteq \mathcal{P})$$

$$(sup(CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1)) \cap \mathcal{P} = \emptyset) = (sup(CF_{\text{cone}}(\underline{tr}, \mathcal{I})) \cap \mathcal{P} = \emptyset)$$

Intuitively, if the system is well-designed, the inclusion of counter-factuals on the critical logs in the property is equal to the inclusion of counter-factuals on the initial trace in the property. The same is true for the disjunction with the property.

Proof 4 *We keep the same notation as in the previous proof.*
Using the well-designed hypothesis and the definition of the cone, the unaffected prefixes UP *corresponding to* $\text{cone}(\underline{tr}, \mathbb{C})$ *respect the specification of all the components. Therefore,* $UP \in \mathcal{P}$*. In a similar fashion to the part on the previous proof dealing with the* BM*, we can prove the that* $\forall \underline{tr}', \underline{tr}'' \in BM, \forall i \in [i_f..|\underline{tr}| - 1], (\forall f \in \mathbb{F}_{\mathcal{P}}, \pi_f(\underline{tr}'[i_f - l_f..i]) = \pi_f(\underline{tr}''[i_f - l_f..i])) \implies (\forall f \in \mathbb{F}, \pi_{OK_{\mathcal{P}}}(\underline{tr}'[i]) = \pi_{OK_{\mathcal{P}}}(\underline{tr}'[i]))$*. It means that the respect of the property coincide on the logged part for the counter-factuals with the critical log and the trace. Hence, since* $\mathcal{P}$ *is a safety property, and we know that the trace respect the property before entering the log, we get the result of the lemma.*

Theorem 5 *Let* $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ *be a trace,* $\underline{L} = (l'_f)_{f \in \mathbb{F}}$ *be the set of maximal critical length and* $\underline{fv} = (fv_C)_{C \in \mathbb{C}}$ *the first violations as defined in property 3. Let* $(cone_C)_{C \in \mathbb{C}}$ *be the cone for* $\mathcal{I}$*, build using* $\underline{fv}$*. Let* $\mathcal{I} \subseteq \mathbb{C}$ *be a set of suspected component. If the system is well designed (the composition of the specification refines the property), then*

$$\text{nec}(\underline{tr}, \mathcal{I}) = \text{nec}_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1))$$

$$\text{suff}(\underline{tr}, \mathcal{I}) = \text{suff}_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1))$$

This theorem means that the causality analysis on the critical logs and the initial trace gives the same result.
This theorem is directly deduced from the two previous lemmas.

Let us illustrate this result with an example.

Example 44 *We consider the followup of example 42, and the counter-factuals we build.*

For $\mathcal{I} = \{pump\}$, the counter-factual is:

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	*1*	*0*	*0*
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*4*	*4*	*4*

The property being that the volume of the tank must be between 0 and 5, this trace respect the system property. We have all the data we need to compute that the trace respects the property, and since it was necessarily true before entering the log, we know that $CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1) \in \mathcal{P}$, thus $CF_{\text{cone}}(\underline{tr}, \mathcal{I}) \in \mathcal{P}$ and $\{pump\}$ is a necessary cause.

For $\mathcal{I} = \{tank\}$, the counter-factuals are:

Time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
com_{pump}	*2*	*2*	*2*	*2*	*1*	*1*	*2*	*0*	*0*	*0*
out_{pump}	*1*	*2*	*2*	*2*	*1*	*1*	*2*	$\underline{2}$	$\underline{1}$	$\underline{1}$
out_{tank}	*0*	*1*	*2*	*1*	*1*	*1*	*2*	*0*	*0*	*0*
vol_{tank}	*1*	*2*	*2*	*3*	*3*	*3*	*3*	*5*	[5..6]	*6 if $vol_{tank}[8] = 5$, else* [6..7]

As the red parts shows, $\sup(CF_{cl}(\underline{cl}(\underline{tr}, \underline{L}), \mathcal{I}, \underline{fv}, |\underline{tr}| - 1)) \cap \mathcal{P} = \emptyset$. Thus $sup(CF_{\text{cone}}(\underline{tr}, \mathcal{I})) \cap \mathcal{P} = \emptyset$ and $\{pump\}$ is a sufficient cause.

This result is strong, since under the assumption that we have the critical log, we get the exact result of the causality analysis on the full trace. The result goes further than that, since the counter-factuals are the same. Therefore, even with other causality definitions, it should be possible to keep this equivalence between the critical logs and the full trace, as long as the data to evaluate all the elements of the causality are logged.

The critical length is information that can be computed offline, and might be available from the design phase.

One of the main drawbacks of this approach is that we need the first violations ($\underline{fv}$). However, there are classes of system, where it is available, e.g. monitored systems. It might be hard to have a monitored system that

can detect the faults right away. Nevertheless, as long as the fault detection is just delayed by a bounded amount of time, we can add that delay to the critical length.

The assumption of the well designed system can be removed if we can have access the fact that the system property is violated, in a similar fashion to the way we have access to the fact that the components are faulty. For instance, if the system property is that the program should not crash, this is easy to check, without the need of the trace.

Besides reducing the size of the logs, these results also opens the door to online causality analysis. If it is possible to have access to enough computing power, and that the system is deterministic, it is possible to start the causality analysis as soon as a component becomes faulty. When components become faulty, the causality analysis can be performed on the critical logs, and if the system is deterministic, it means that the counter-factuals is a unique trace, that can be prolonged every time new data is acquired.

Lastly, even though the grey-box approach here have been presented and proved for the cone, it is also true for the unaffected prefixes approach, since the unaffected prefixes are "longer" than the one computed with the cone, thus the data needed for them is available, since the data is available for the cone.

7.1.4 Reducing the logging by using extra information

In this section two ways of reducing the logs have been proposed: in space and in time. Extra information can help for both approaches.

For reduced log in space, one of the problems is that some information is lost because some flows are not logged. If a component has different modes of functioning, and some of the modes do not rely on some of the component flows, logging a flow corresponding to the mode and the relevant component flows is enough, thus reducing the logs in space. This is typically where the monitoring could be used, to check, online, if the flows should be logged.

In a similar fashion to the grey-box components, we could use the information that the component is behaving according to its specification, not to log certain flows. For instance, if we can infer the outputs from the inputs, it is not necessary to log the outputs, as long as the component is not faulty. This might be used to enhance the benefit of the grey-box component even further.

Concerning the grey-box component, extra information may help reduce the critical length. Indeed, if we go back to Example 40, specifically to $add3$. This component outputs the sum of the current value and previous input

value every two tick and maintain the output the other ticks. The critical length was $+\infty$, because for a sequence of inputs that are 0, the output will remain 0, and we cannot infer if the output is maintained or computed, at a given instant. If this information is output, via a Boolean flow (called *parity*), for instance, the critical length becomes 2. Whats more, we do not need to log the parity at every tick, but just the one when the component become logged, since we can infer the parity just from one correct value of it. Another solution would be to use monitoring to log the necessary data. As long as there is no sequence of the form $((0,0),(0,0),(0,0))$, only the last two ticks are enough. If such a sequence arises, the values just before the sequence must be kept, as well as the length of the sequence.

7.2 Causality Analysis using fault models

One of the problems we face while building the counter-factuals is that we do not know the faulty behaviour of the components. BM might constraint the possible faulty behaviours, but it is supposed to reflect system level restriction of the behaviour, e.g. if the maximum power accessible by the system is a given value, then the sum of the power consumed must be lesser than that value. It would be interesting to be able to build more accurate counter-factuals by using explicit fault models to prolong the traces. It would be the faulty version of using the specification.

Another interesting addition is that it would enable to perform horizontal causality analysis. It could be phrased as: "does the faults from those components caused the fault from another one". It shifts the focus from the system level property (vertical causality) to the component specifications. We need to be able to use a fault model for the components to be able to modify the behaviour of faulty components more accurately, with the fault models, which yields more interesting results for horizontal causality.

This section will present a generalisation of the work from [Wang et al., 2015]. It will be divided in five subsections. The first one will introduce the notion of fault model. The next one will present how to adapt the definitions for the cone and unaffected prefixes to take into account the fault models. Horizontal causality will be discussed in the third one. The fourth one will show how to tackle the fact that components might have multiple fault modes, that could be used in the causality analysis. The last one will present some idea on how to enhance further the accuracy, by adding extra information.

7.2.1 Fault models

A fault model describes the behaviour of a component when it is faulty. This paper ([Avizienis et al., 2004]) gives an overview of the possible failure modes. Failure can be seen as system level fault models. However they can be transposed at a component level, since those definitions relies on specifications and services.
[Powell, 1992] discusses the coverage of failure modes. The notion of partial order between the failure modes, reflecting the relative severity of the failure modes, developed in [Powell, 1992], will be transposed to the fault modes from this section.

Definition 65 (Fault model) *Let $C = (I_C, O_C, \mathcal{S}_C)$. A fault model for this component is a partially ordered set of behaviours, called fault modes, included in $\mathcal{B}_C$, noted $\mathbb{FM}_C = (\mathbb{M}, \prec)$ such that $\mathbb{M} = \{\mathbb{M}^0, \ldots, \mathbb{M}^n\}$ and $\mathbb{M}^i \prec \mathbb{M}^k \implies \mathbb{M}^i \subseteq \mathbb{M}^k$.*

Let us illustrate the fault models with an example:

Example 45 *Let us suppose we have a component C, that is the Boolean identity, with one input and one output. We will consider five possible fault modes:*

Stuck at *true* *The component always outputs true.*

Stuck at *false* *The component always outputs false.*

Stuck a *true* when switched *The component becomes stuck at true if it starts outputting true.*

Stuck a *false* when switched *The component becomes stuck at false if it starts outputting false.*

Random output *The component randomly outputs true or false.*

The less severe modes are the struck at true, or false, as they are easier to spot. The moment the component became faulty is any instant during the first sequence of true (false) where the component can be diagnosed as faulty.

Time	*0*	*1*	*2*	*3*	*4*
Input	*false*	*true*	*false*	*true*	*false*
Expected output	*false*	*true*	*false*	*true*	*false*
Actual output	*false*	*true*	*false*	*true*	*true*

We suppose the fault mode here is stuck at true. The underlined values are the possible instants where the component can be faulty. The red values are the one where the component is recognised as faulty. Here the component became faulty either at instant 3 *or* 4*, but is faulty for sure at instant* 4*, as its behaviour derives from the expected one.*

The stuck when "switched" modes are more severe than the "stuck at" ones, as the component can be faulty for longer, without being detected. Let us consider the same trace as previously for stuck at true when the mode is stuck at true when switched.

Time	*0*	*1*	*2*	*3*	*4*
Input	*false*	*true*	*false*	*true*	*false*
Expected output	*false*	*true*	*false*	*true*	*false*
Actual output	*false*	*true*	*false*	*true*	*true*

We know for sure that the component was not faulty at instant 1*, because if it were faulty, it should have been stuck, and would have output true at instant* 2*. However, it could be faulty at instant* 2*, as it outputs false, which is one tick earlier than stuck at true. Therefore, this mode is mode severe than the stuck at one.*

Lastly, let us consider the random output fault mode on the trace:

Time	*0*	*1*	*2*	*3*	*4*
Input	*false*	*true*	*false*	*true*	*false*
Expected output	*false*	*true*	*false*	*true*	*false*
Actual output	*false*	*true*	*false*	*true*	*true*

Here, the component could have been faulty since the very beginning of the trace, since (*false*, *true*, *false*, *true*) *is a possible faulty behaviour for this mode. It is the hardest mode to detect, and thus the most severe.*

The partial order is:

- *Stuck at true* $\prec$ *Stuck at true when switched* $\prec$ *Random output and*
- *Stuck at false* $\prec$ *Stuck at false when switched* $\prec$ *Random output*

This example illustrates how fault models work. Given an observation, some modes might be ruled out, but more than one can apply.

In the next two subsections, we suppose that the mode can be identified, as well as when the component becomes faulty. Ways to deal with the multiple possible fault modes will be discussed in Subsection 7.2.4.

7.2.2 Including fault models in causality analysis

In this section, since we suppose the fault mode is identified, we consider that for a component C, the fault model is a single behaviour $\mathbb{FM} \subseteq \mathcal{B}_C$. We suppose that components come with a fault model, and are of the following form $C = (I_C, O_C, \mathcal{S}_C, \mathbb{FM}_C)$.

Definition 66 (Cone with fault model ($\text{cone}_{\mathbb{FM}}$)) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace, and $\mathcal{I} \subseteq \mathbb{C}$. $\text{cone}(\underline{tr}, \mathcal{P}) = (\text{cone}_C)_{C \in \mathbb{C}}$ is the vector of maximal indexes such that, $\forall C \in \mathbb{C}$:*

$$C \in \mathcal{I} \implies cone_C \leqslant fv_C(\underline{tr}) \wedge \tag{7.5}$$

$$\forall f \in I_C, \forall C' \in \mathbb{C}, f \in O_{C'} \implies cone_C \leqslant cone_{C'} \tag{7.6}$$

The only difference with the initial definition 14 is that constraint 7.6 has been relaxed, and any component is added to the cone, even when it is faulty. It means that the suspect components enter the cone when they become faulty, and any component that receive data from a component inside the cone is added to the cone.

To have a similar behaviour with the unaffected prefixes, we only need to modify the extend function as follow:

Definition 67 (Trace extension with fault model ($extend_{\mathbb{FM}}$)) *Let C be a component, and $\underline{tr}, \underline{tr}^0 \in \mathcal{B}_C$ be traces over that component.*

$$extend_C(\underline{tr}^0, \underline{tr}) \begin{cases} \{\underline{tr}' \in \mathcal{S}_C \mid \underline{tr} \sqsubseteq \underline{tr}'\} & \text{if } \underline{tr} \neq \underline{tr}^0 \wedge \underline{tr} \in \mathcal{S}_C \\ \{\underline{tr}' \in \mathbb{FM}_C \mid \underline{tr} \sqsubseteq \underline{tr}'\} & \text{if } \underline{tr} \neq \underline{tr}^0 \wedge \underline{tr} \notin \mathcal{S}_C \\ \{\underline{tr}\} & \text{otherwise} \end{cases}$$

We add to the extend function a way to extend the trace for the component that are faulty that uses the fault model. This change is enough, since *extend* is used in all the subsequent definition, to check whether the trace can be extended or not.

As explained in section 3.2 the definition of the counter-factuals is the same for both approached, since the cone can easily be transformed in unaffected prefixes, and vice-versa. Therefore, only a definition for the cone will be given.

Definition 68 *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in BM \setminus \mathcal{P}$ be a faulty trace, and $\mathcal{I} \subseteq \mathbb{C}$. Let $(\text{cone}_C)_{C \in \mathbb{C}} = \text{cone}(\underline{tr}, \mathcal{P})$. The counter-factuals are defined as follow:*

$$CF_{\mathbb{FM}}(\underline{tr}, \mathcal{I}) = \{\underline{tr}' \in BM \mid \forall C \in \mathbb{C},$$

$$\pi_C(\underline{tr}[0..\text{cone}_C - 1]) = \pi_C(\underline{tr}[0..\text{cone}_C - 1]) \wedge \tag{7.7}$$

$$\pi_C(\underline{tr}[0..\text{cone}_C - 1]) \in \mathcal{S}_C \implies \pi_C(\underline{tr}') \in \mathcal{S}_C\} \wedge \tag{7.8}$$

$$\pi_C(\underline{tr}[0..\text{cone}_C - 1]) \notin \mathcal{S}_C \implies \pi_C(\underline{tr}') \in \mathbb{FM}_C\} \tag{7.9}$$

Constraint 7.9 is added, to prolong the traces of the faulty components. This will yield tighter counter-factuals, since the faulty traces are constrained by both BM and $\mathbb{FM}_C$.

Example 46 (Non-suspected faulty component, with fault model) *This example is the same as the example 7, i.e. the running example with one pump and a tank, and the faulty behaviour of the tank embedded in BM. Here we transfer this faulty behaviour from BM to $\mathbb{FM}_{tank}$ (it can leak, anytime, any quantity). BM is all the possible behaviours of the system once again, and the following fault models is added to the tank:*

$$\mathbb{FM}_{tank} = \{\underline{tr} \in \mathcal{B}_C \mid \forall i \in [0..|\underline{tr}| - 1], vol_{tank}[i] \leqslant vol_{tank}[i-1] + out_{pump}[i]\}$$

With $vol_{tank}[-1] = 0$, by convention.

The cone approach failed to give rise to the intuitive causes. Let us apply the approach taking the fault model into account, for the cone approach.

The failing scenario is the following:

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	$\underline{2}$	$\underline{2}$
vol_{tank}	*0*	*2*	$\underline{3}$	$\underline{5}$	$\underline{7}$

The problematic causality analysis happened when the suspected component were $\mathcal{I} = \{pump\}$. Here the cone is $(3, 3)$. pump enters the cone at instant 3 because it is suspected and faulty. Since tank receives data from a component in the cone, it enters it, in the fault model approach, without regards to the fact that it is faulty or not.

Thus the counter-factuals are the following:

Time	*0*	*1*	*2*	*3*	*4*
com_{pump}	*0*	*2*	*2*	*0*	*0*
out_{pump}	*0*	*2*	*2*	*0*	*0*
vol_{tank}	*0*	*2*	$\underline{3}$	$\underline{[0..3]}$	$\underline{[0..vol_{tank}[3]]}$

The counter-factuals respect the system property, and $\{pump\}$ is a necessary cause, as expected.

With the fault model approach, an example that was not treated properly by the cone approach now is. It also shows that the reason the unaffected prefixes could tackle this example and not the cone approach is because the unaffected prefixes take into account the behavioural model in the construction of the "cone".

Note the the unaffected prefixes is still superior to the cone approach, even taking into account the fault model, because it gives rise to a cone that is less pessimistic. In the worst case the unaffected prefixes give the same cone as the cone approach, but generally, the cone built by the unaffected prefixes is included in the cone one.

7.2.3 Horizontal causality, and cause minimisation

As presented previously, the horizontal causality aims at showing the causal relations between the faults of different components. Those relations, as well as causality definition properties, can be used to minimise the causes, in order to return minimal sets of components for causes.

Definition 69 (Horizontal counter-factuals (CF_h)) *Let $\underline{tr}$ be a system trace, $\mathcal{I}$ a set of suspected components, and $C \in \{C' \in \mathbb{C} \setminus \mathcal{I} \mid \pi_C(\underline{tr}) \notin \mathcal{S}_C\}$ a non-suspected faulty component. The horizontal counter-factuals, noted $CF_h(\underline{tr}, \mathcal{I}, C)$, are the same as the one of definition 68 (counter-factuals with fault models), except for C, for which the Constraints 7.8 and are 7.9 are replaced by $\forall \underline{tr}' \in CF_h(\underline{tr}, \mathcal{I}, C)$:*

$$ff(C, \underline{tr}) \leqslant \text{cone}_C \implies \pi_C(\underline{tr}') \in \mathbb{FM}_C \wedge$$

$$ff(C, \underline{tr}) > \text{cone}_C \implies \pi_C(\underline{tr}') \in \mathcal{S}_C$$

With $ff(C, \underline{tr})$ the first instant where the component can be faulty, according to $\mathbb{FM}_C$.

Intuitively, if the component can have been faulty before entering the cone, or when it entered the cone, we prolong its behaviour using the fault model. If not we use the specification. This choice is driven by the fact that if the component already have received corrupted data before becoming faulty, we cannot know whether it became faulty because of it, or was bound to become faulty.

Note that the fact that we use $\mathbb{FM}_C$ does not necessarily implies that the counter-factuals do not respect the specification of the component. If we consider the stuck at *true* when switched, as long as no *true* have to be output, followed, at some point, by a *false*, the faulty and the correct behaviours coincide.

Definition 70 (Horizontal necessary causality (nec_h)) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ a trace and C such that $\pi_C(\underline{tr}) \notin \mathcal{S}_C$ a faulty component and $\mathcal{I} \subseteq \mathbb{C}$ a set of suspected components such that $C \notin \mathcal{I}$. $\mathcal{I}$ is a horizontal cause for the faults of C, noted $\text{nec}_h(\underline{tr}, \mathcal{I}, C)$, if $\pi_C(CF_h(\underline{tr}, \mathcal{I})) \subseteq \mathcal{S}_C$.*

$\mathcal{I}$ is an horizontal necessary cause for the faults of C, if repairing the components in $\mathcal{I}$ in the counter-factuals fixes the component. If the component could not be faulty (according to $\mathbb{FM}_C$) upon entering the cone, $\mathcal{I}$ will always be a necessary horizontal cause. If not, the faulty behaviour of the component might coincide with the correct one, $\mathcal{I}$ thus being a necessary horizontal cause for the fault of C.

Definition 71 (Horizontal sufficient cause suff_h) *Given a system $S = (\mathbb{C}, \mathbb{F}, \mathcal{P}, BM)$. Let $\underline{tr} \in \mathcal{B}_{\mathbb{F}}$ a trace and C such that $\pi_C(\underline{tr}) \notin \mathcal{S}_C$ a faulty component and $\mathcal{I} \subseteq \mathbb{C}$ a set of suspected components such that $C \notin \mathcal{I}$. $\mathcal{I}$ is a horizontal sufficient cause for the faults of C, noted $\text{suff}_h(\underline{tr}, \mathcal{I}, C)$, if $\pi_C(sup(CF_h(\underline{tr}, (\mathbb{C} \setminus \mathcal{I}) \setminus \{C\}))) \cap \mathcal{S}_C = \emptyset$.*

$\mathcal{I}$ is a horizontal sufficient cause for the faults of C if fixing every components, but the one in $\mathcal{I}$ and C does not fix C. As usual with sufficient causality, we consider only the "finished" traces, as we want to check that the component specification is eventually violated.

When analysing the causality on a system, we do if for each possible subset of the faulty components ($2^{\mathbb{C}'}$, with $\mathbb{C}'$). It may give rise to a lot of possible causes, especially a the faults of some components is a cause of the faults of other ones. It can be hard to analyse a very large set of causes, and some may not be relevant. Thus, it would be interesting to be a able to minimise the sets of causes.

Let $\underline{tr}$ be a system trace, and $\mathbb{S}$ be the set of sets of components such that $\forall \mathcal{I} \in \mathbb{S}, cause(\underline{tr}, \mathcal{I})$, with *cause* a causality definition (e.g. nec). Here is an algorithm to minimise the sets of causes:

1. **Elimination of the non-faulty components** We build $\mathbb{S}' = \left\{\mathcal{I} \subseteq \mathbb{C} \mid \exists \mathcal{I}' \in \mathbb{S}, \mathcal{I} = \mathcal{I}' \cap \{C \in \mathbb{C} \mid \pi_C(\underline{tr}) \notin \mathcal{S}_C\}\right\}$. Here, the non-faulty components are removed, since they are not relevant to causality analysis.

2. **Minimisation using horizontal cause** We build $\mathbb{S}'' = \bigcup_{\mathcal{I} \in \mathbb{S}'} (reduce_h(\mathcal{I}))$, with $reduce_h(\mathcal{I}) = \{\mathcal{I}' \subseteq \mathcal{I} \mid \forall C \in (\mathcal{I} \setminus \mathcal{I}'), \exists \mathcal{I}'' \subseteq \mathcal{I}', \text{nec}_h(\underline{tr}, \mathcal{I}'', C) \vee (\text{suff}_h(\underline{tr}, \mathcal{I}'', C) \wedge \neg\, \text{suff}_h(\underline{tr}, \emptyset, C))\}$.

$reduce_h$ generates all the susbsets of $\mathcal{I}$ where all the components for which there is a horizontal cause in the subset are removed from $\mathcal{I}$.
If $\mathcal{I}'$ is in $reduce_h(\mathcal{I})$, then for all components C in $\mathcal{I}'\backslash\mathcal{I}$, the fault of C is caused by a subset of $\mathcal{I}'$. To assess if the faults of C is caused by $\mathcal{I}''$, we check $\text{nec}_h(\underline{tr}, \mathcal{I}'', C) \vee (\text{suff}_h(\underline{tr}, \mathcal{I}'', C) \wedge \neg\, \text{suff}_h(\underline{tr}, \emptyset, C))$, i.e. *suspect*$''$ is a necessary cause for the faults of C, or *suspect*$''$ is a sufficient cause for the faults of C and C is not faulty by itself ($\neg\, \text{suff}_h(\underline{tr}, \emptyset, C)$)).
The idea behind this reduction is that if a component fault is caused by another set of components, then it should not be a cause, and should then be removed from the suspect set. The fact that the horizontal causes are not empty ensures that $\emptyset$ is not in $\mathbb{S}''$. However it means that we do not necessarily have $\text{nec}(\underline{tr}, \mathcal{I})$ for all set in $\mathbb{S}''$ anymore.

3. **Set minimisation** $\mathbb{S}_{min} = inf(\mathbb{S}'')$. We remove all the set for which there is a subset that also is a cause. Indeed, $\mathbb{C}$ is always a necessary cause. However, it is not an interesting cause. We rather have minimal causes, which the infimum provides.

4. **Second horizontal minimisation**

$$\mathbb{S}'_{min} = \{\mathcal{I} \in \mathbb{S}_{min} \mid \forall \mathcal{I}' \in \mathbb{S}_{min},$$

$$(\forall C \in \mathcal{I}, cause_h(\underline{tr}, \mathcal{I}', C)) \implies (\forall C' \in \mathcal{I}', cause_h(\underline{tr}, \mathcal{I}, C'))\}$$

With $cause_h(\underline{tr}, \mathcal{I}, C) = \exists \mathcal{I}' \subseteq \mathcal{I}, \text{nec}_h(\underline{tr}, \mathcal{I}', C) \vee (\text{suff}_h(\underline{tr}, \mathcal{I}', C) \wedge \neg\, \text{suff}_h(\underline{tr}, \emptyset, C))$.
This new minimisation removes the cause $\mathcal{I}$ for which their exists another cause $\mathcal{I}'$ that provides an horizontal cause for each components of $\mathcal{I}$, unless $\mathcal{I}$ provides an horizontal cause for each component of $\mathcal{I}'$. However, if $\mathcal{I}$ is a horizontal cause for each components $\mathcal{I}'$, and reciprocallly , it would not make sense to remove any of them (their is no reason to prioritise one over the other), or both (two minimal causes would be removed).

Let us illustrate the cause minimisation with an example.

Example 47 *Let us consider a system formed of four components a, b, c and d. Let $\underline{tr}$ be a failing trace over the system. d is the only non-faulty components in $\underline{tr}$. The necessary causes are $\mathbb{S} = \big\{\{a\}, \{a, d\}, \{a, b\}, \{a, b, d\}, \{b, c\}, \{b, c, d\}, \{a, b, c\},$*
Let us minimise this set of causes.

1. *We remove the non-faulty components, i.e. d. We get $\mathbb{S}' = \big\{\{a\}, \{a, b\}, \{b, c\}, \{a, b, c\}\big\}$.*

2. *The horizontal cause relation are that b is an horizontal cause for a. It has an impact on $\{a,b\}$ and $\{a,b,c\}$. $reduce_h(\{a,b\}) = \big\{\{a,b\},\{b\}\big\}$ and $reduce_h(\{a,b,c\}) = \big\{\{b,c\},\{a,b,c\}\big\}$. We get $\mathbb{S}'' = \big\{\{a\},\{b\},\{a,b\},\{b,c\},\{a,b,c\}\big\}$*

3. *We pursue by minimising this set $\mathbb{S}_{min} = \big\{\{a\},\{b\}\big\}$*

4. *Lastly, we perform the second horizontal minimisation $\{\{b\}\}$, since $\{b\}$ is an horizontal cause to the faults of a.*

$\{a,b\}$ was a cause, but there is a reduced cause $\{b\}$, since b is a cause of the fault in a. $\{b,c\}$ is removed, because $\{b\}$ is a cause. Here, we see the importance of the order, since $\{a,b\}$ would have been removed, since it is a superset of $\{a\}$, if the step 3. was performed before the 2.
The intuition to reduce $\{a,b\}$ to $\{b\}$ is that the faults of b cause the faults in a, thus only b should be held responsible. Since $\{b\}$ is a cause, $\{b,c\}$ should be eliminated, as it is bigger than $\{b\}$.
The reason we remove $\{a\}$ is because $\{b\}$ is a horizontal cause of the faults of a.
Note that $\{b\}$ is the minimal cause, but note a fix. indeed, step 2. removes some components from the causes, thus maybe changing the cause to a new set which is not a fix.

This example shows that it is possible to reduce significantly the set of causes. However, a lot of information is lost. It would be interesting, when presenting this minimised set of cause, to give access to the reason some causes were removed.

7.2.4 Dealing with multiple fault models

As explained earlier, being able to isolate one fault mode for a given trace is very unlikely, and generally, several fault modes are possible. In order to tackle this issue, three methods will be proposed.

A general note is that only the fault modes of the non-suspected faulty (before the cone) components are relevant, as they are the only one for which a fault mode is used.

Conservative method In this method, we only consider the most severe fault mode, when doing the analysis. This is conservative, since it assumes the worst. However, it does not necessarily yield better results than the normal approach, since generally the worst fault mode is a random one (up to some component constraints), which end up being $\mathcal{B}_C$.
It gives the same result as constraining BM with all the different most severe fault modes.

Exhaustive approach Here, we simply test every possible combination of fault modes for every non-suspected faulty component. The obvious problem is the combinatorial explosion, since another exponential blowup is added to the fact that we have to do the analysis for every subset of the faulty component. Here, in addition, we have to test every possible combination of fault modes for each possible non-suspected faulty (before the cone) components.

However, it gives more accurate results, since some suspect might be cause only for certain fault modes, but not for other.

Mode coverage approach Oftentimes, the fault modes are given with a probability of occurrence. For instance, the FMECA standard ([US Department of Defense, 19 is widely used in safety critical sector, such as aerospace, the military sector or the medical field. Another widely used technique to assess possible failures during the design phase is the fault tree analysis, which comes with probabilities of failure.

Usually, the most severe modes have probabilities that are significantly lesser than the one of the less severe ones. This can be used to assess which mode should be used during the causality analysis. Using the probabilities and some properties of causality analysis, a coverage of the modes, given the chosen mode, can be computed, thus guiding the fault modes to chose, or giving a confidence measure in the result of the analysis. The notion of mode coverage will be introduced to show how this idea can be applied.

In this section, the cone approach will be treated, as it only impacts the counter-factuals construction. However, those results hold for the unaffected prefixes approach (where the "cone" construction is impacted by the fault mode).

Property 4 (Necessary causality inclusion) *Let $\underline{tr}$ be a system trace, $\mathcal{I} \subseteq \mathbb{C}$, $C \in \mathbb{C} \setminus \mathcal{I}$ such that $\pi_C(\underline{tr}) \notin \mathcal{S}_C$ be a non suspect faulty component and $\mathbb{M}_C^0, \mathbb{M}_C^1$ be two fault modes for C, such that $\mathbb{M}_C^0 \prec \mathbb{M}_C^1$. We note $\mathcal{I}^i$ when $\mathbb{M}_C^i$ is used in the counter-factuals.*

- $\text{nec}(\underline{tr}, \mathcal{I}^1) \implies \text{nec}(\underline{tr}, \mathcal{I}^0)$

The only difference between $CF(\underline{tr}, \mathcal{I}^0)$ and $CF(\underline{tr}, \mathcal{I}^1)$ comes from constraint 7.9 (prolongation for faulty components) from definition 68 (counter-factuals with fault model). Therefore, $\forall \underline{tr}' \in CF(\underline{tr}, suspect^i), \pi_C(\underline{tr}') \in \mathbb{M}^i$. Since $\mathbb{M}^0 \prec \mathbb{M}^1$, $\mathbb{M}^0 \subseteq \mathbb{M}^1$. Hence, $CF(\underline{tr}, \mathcal{I}^0) \subseteq CF(\underline{tr}, \mathcal{I}^1)$, from which the properties follow.

This property means that if for a given mode, the suspect is a necessary cause, it is also the case for the less severe modes.
Note that it follows from the property that $\neg \operatorname{nec}(\underline{tr}, \mathcal{I}^0) \implies \neg \operatorname{nec}(\underline{tr}, \mathcal{I}^1)$.

Definition 72 (Necessary cause fault mode coverage *cover***)** *Let $\underline{tr}$ be a system trace, $\mathcal{I}$ the set of suspected component. Let $\mathbb{C}' = \{C \in \mathbb{C} \setminus \mathcal{I} \mid \pi_C(\underline{tr}) \notin \mathcal{S}_C\}$ the set of faulty non-suspected components. Let $\underline{mode} = (\mathbb{M}_C)_{C \in \mathbb{C}'}$ be the chosen mode for the counter-factuals. For each component, we suppose we are given a probability of occurrence function p for all the fault modes, such that $\forall C \in \mathbb{C}, \big(\sum_{\mathbb{M} \in \mathbb{FM}_C} p(\mathbb{M})\big) = 1$. cover is defined as follow:*

$$cover = \begin{cases} \prod_{C \in \mathbb{C}'} set\_p(\{\mathbb{M} \in \mathbb{FM}_C \mid \mathbb{M} \preceq \mathbb{M}_C\}) & \text{if } nec(\underline{tr}, \mathcal{I}) \\ \prod_{C \in \mathbb{C}'} (1 - set\_p(\mathbb{FM}_C \setminus \{\mathbb{M} \in \mathbb{FM}_C \mid \mathbb{M}_C \preceq \mathbb{M}\})) & \text{if } \neg \operatorname{nec}(\underline{tr}, \mathcal{I}) \end{cases}$$

$$\text{With } set\_p(\mathbb{FM}') = \sum_{\mathbb{M} \in (\mathbb{FM}' \cup \{\mathbb{M}_C\})} p(\mathbb{M}).$$

Using the previous property, we can compute the coverage of the chosen fault modes. Let us look at the case where the suspect is a necessary cause. The coverage is the product of the probability of all the modes for which we know the result of the causality analysis for sure. Given a fault mode, for each mode that are less severe than it, the suspect are still a necessary cause. The total probability is set_p, when $nec(\underline{tr}, \mathcal{I})$, since we consider all the modes that are supersets of $\mathbb{M}_C$. When $\neg \operatorname{nec}(\underline{tr}, \mathcal{I})$, it gets all little trickier, since we want to exclude all the supersets of $\mathbb{M}_C$, hence the use 1 minus the probability of being a subset of $\mathbb{M}_C$. The product of the coverage of each components gives us the total coverage.

Let us illustrate the notion of coverage with an example.

Example 48 *The system we consider has component with three fault modes such that $\mathbb{M}^0 \prec \mathbb{M}^1 \prec \mathbb{M}^2$, and with an order of magnitude between the different mode probabilities, i.e. $p(\mathbb{M}^0) = 10p(\mathbb{M}^1)$ and $p(\mathbb{M}^1) = 10p(\mathbb{M}^2)$. This is not unusual to have such difference between the modes, since the more severe modes generally means that the component have been more severely damaged or impacted, which is less likely.*

Let $\underline{tr}$ be a system trace and $\mathcal{I}$ be a set of suspected component. Here is a table giving the coverage, in function of the result of the causality analysis, the number of faulty non-suspected components, and the mode chosen.

Result of the analysis	$\mathrm{nec}(\underline{tr}, \mathcal{I})$			$\neg\,\mathrm{nec}(\underline{tr}, \mathcal{I})$		
Mode chosen	$\mathbb{M}^0$	$\mathbb{M}^1$	$\mathbb{M}^2$	$\mathbb{M}^0$	$\mathbb{M}^1$	$\mathbb{M}^2$
0	*1*	*1*	*1*	*1*	*1*	*1*
1	*0.9*	*0.99*	*1*	*1*	*0.09*	*0.009*
2	*0.81*	*0.98*	*1*	*1*	10^{-3}	8×10^{-5}
5	*0.59*	*0.96*	*1*	*1*	10^{-6}	6×10^{-11}
10	*0.35*	*0.91*	*1*	*1*	9×10^{-11}	4×10^{-21}

This tabular shows that when there are not too many faulty non-suspected components (let n *be the number of such components), performing the analysis on the less severe mode gives very good coverage, since if* $\mathcal{I}$ *is not a necessary cause, we get full coverage. If it is, we get* 0.9^n*. It gives us a coverage of* 0.9*,* 0.81 *and* 0.72 *for* n *equal to* 1*,* 2*, and* 3*. For* $n = 1$*, it is a good coverage. For higher values of* n*, the coverage may be insufficient.*

Therefore, it could be a good idea to perform more than one analysis, with different choices of fault model. If you do so, you can sum the coverage with the previous one.

For instance, let us suppose that $n = 2$*. A first causality analysis is performed. Since* $\mathbb{M}^0$ *gives the maximum coverage if we do not have any information about the outcome (*0.81 *and* 1*), those are the fault modes that are chosen. If* $\mathcal{I}$ *is not a necessary cause, the coverage is* 1*, so the analysis can stop. If* $\mathcal{I}$ *is a necessary cause, we know that for* 81% *of the possible fault mode combination, pondered by the probability of occurrence,* $\mathcal{I}$ *is a cause. There is still* 19% *to explore. If we take* $\mathbb{M}^0$ *for one of the faulty non-suspected component, and* $\mathbb{M}^1$ *for the other, the coverage become* $p(\mathbb{M}^0) \times set\_p(\{\mathbb{M}^0, \mathbb{M}^1)\}) = 89\%$ *if* $\mathrm{nec}(\underline{tr}, \mathcal{I})$ *and* $set\_co\_p(\emptyset) \times set\_co\_p(\{\mathbb{M}^0)\}) = 8.9\%$*. It then increases the coverage by* $89\% - 81\% = 8\%$ *if* $\mathrm{nec}(\underline{tr}, \mathcal{I})$*, and* 10% *otherwise, which ends up being a total coverage of* 89% *or* 91%*. If we do the analysis, but swap the modes for the two components, we achieve a new increase by* 8% *(*nec *for the new fault mode setup),* 9% *(*$\neg\,\mathrm{nec}$ *for both setups) or* 10% *(*$\neg\,\mathrm{nec}$ *for this setup and* nec *for the other), with a total coverage between* 97% *and* 100%*, which is almost total, to total coverage.*

If we take the same example with a factor 5 *between the probabilities of the fault modes, instead of* 10*, we get a coverage of from* 91% *(with nec for all fault models combination, which give* $66\% + 13\% + 13\%$*) to* 100%*. We can enhance the* 91% *to* 94% *with an extra analysis.*

We can achieve almost full to full coverage, performing three analyses, instead of nine ($|\mathbb{FM}|^2$*).*

If we consider an example with a factor 10 *between the fault modes, and three non-suspected faulty components, we get a coverage between* 97% *to* 100% *in* 7 *analyses. Note that* 7 *is the worst case,* 97% *can be achieved in* 4

analyses, and 100% *after one.*

Here, we almost get full coverage with seven analyses instead of twenty seven.

This example shows that using a choice of fault mode driven by maximising the coverage, we can achieve almost full coverage, without performing all the possible fault modes combination.

What's more, even when performing an exhaustive analysis, the fault mode probabilities can be used to ponder the different results, giving a more representative result for the causality analysis. A very improbable fault combination that leads to $\mathcal{I}$ being a cause has then less weight than a very probable fault mode combination for which $\mathcal{I}$ is not a cause.

Note that given the theorem 4 (sufficient causality results are not transposable to tighter constraints) of Subsection 7.1, this result cannot be transposed to the sufficient causality, because of the *sup* in the sufficient cause definition.

7.2.5 Enhancing the precision using extra information

The specification is an abstraction of the actual behaviour of the component. Thus, having more information logged can help building counter-factuals that are closer to what would actually have happened.

Having access to extra information can also help knowing in which fault mode a component is. For instance, having access to the inner state of the component may help determining the fault mode, as well as whether the component is faulty or not. If not giving a deterministic assessment of the fault mode, it may modify the probability of the different modes, thus helping better optimising the coverage approach.

Another effect of extra information would be to constraint the counter-factuals more. Indeed, by logging more information, it may reduce the possible values a component can output, with the constraint of the extra information. It means that the counter-factuals would be closer to what would have happened.

Conclusion

This section presented several new results. The first one is that, excluding the case where a suspect is a necessary cause on a partial trace, we do not have any information about the causality analysis result on the full trace from an analysis on a partial trace. The grey-box component approach proposes a way of having the same result of the causality analysis on the full trace

and on critical logs. Though a way of assessing the fact that component are faulty or not, at runtime, is needed, this approach can reduce significantly the amount of data to log and to process, especially on system runs where the failure occurs shortly after the first fault.

The second subsection presented a way to use fault models explicitly in causality analysis. Those fault models can be used to refine the causes, by checking the relation between component faults. Those relations can be used to perform a reduction of the cause sets, thus giving a more accurate causality analysis. Whats more, a way of assessing the coverage of multiple analyses is proposed, in case the components have multiple fault modes. The coverage helps both checking if all the possible fault mode combinations have been covered, and choosing the next analysis to perform.

Lastly, the previous subsection showed some intuition on the benefit that can bring the use of extra information, to both reduce the needed logs and helping enhancing the precision of the causality analysis.

Section 8

Conclusion

8.1 Summary

This thesis expands the causality analysis framework (from [Gössler and Le Métayer, 2013, Gössler and Métayer, 2015]) in several ways.

The first one is to put causality analysis into perspective, thus drawing requirements that are the basis of the approach.

An instantiation of approach for the synchronous data-flow systems (namely Lustre) has been implemented in the existing Loca tool.

The first main contribution of this document is to move causality analysis from a purely black-box setting to a mixed one. Two different frameworks are proposed. The mixed-one is a straight up extension of the causality analysis framework, as it uses the counter-factuals to generate all the possible behaviours for a set of white-box components (protagonists), while fixing the faults of a set of black-box ones. A controller synthesis is then performed on those counter-factuals, to check whether a global strategy was accessible to the protagonists, would the black-box ones being fixed, that would avoid the failure. From this, we can build several causality definitions, that have a practical interpretation in a design perspective. Having such a tool for mixed framework systems is desirable, as systems mixing black and white-box components are very common, given the widespread use of off-the-shelf components. A way of performing the synthesis is also proposed, using the techniques developed for controller synthesis.

However, this extension is not enough to be able to use the mixed framework as a responsibility assignment tool. This problem arises from the fact that the synthesis is made at a system level, and thus does not take into account the information the components have access to, while computing their outputs. Therefore, a more expressive framework has been developed in this

document: the game framework. The first difference with the mixed one is to refine the system description from a global tick granularity to a granularity that is able to assess how the flows are valued during a time-step. The second enhancement is to use a game to build the counter-factuals. This allow to add another player, namely the antagonists, that uses strategies to make the system fail. What's more, the counter-factuals are built using a more advanced approach, similar to the one proposed in [Gössler and Stefani, 2016], that gives rise to counter-factuals that are more accurate. With those enhancements, the game-framework is a full-fledged responsibility ascription framework for mixed systems. The causality definitions from the previous frameworks can be adapted to this new one, and some new definitions are possible, thanks to the addition of the antagonists. A refinement of the synthesis proposed for the mixed framework is proposed to compute winning strategies and spoiling strategies for the game framework.

The effect of the amount of information accessible to build the counter-factuals is also explored. Usually, performing causality analysis on a partial trace does not give any insight on the results for the full trace. The grey-box components concept provides a way of being able to make sure that the result of the causality analysis is the same with the critical logs and with the full trace. This is an important result, as traces may be very big, and this technique helps reducing the size of the logs needed to be able to perform the causality analysis, without loosing in accuracy.
The impact on causality analysis of having more information on the system is also studied. Namely the introduction of a fault model for the components. This helps having better counter-factuals, as the faulty behaviours of the components are better reconstructed. Although the precision is enhanced, this is at the cost of a computational blowup. The coverage can be used to mitigate this extra cost, while giving a confidence value on the results obtained. The fault models can also be used with horizontal causality (first introduced in [Wang et al., 2015]), thus refining the responsibility ascription, and reducing the set of "responsible" components to minimal ones.

8.2 Future prospect

The results obtained in this thesis open up some new research directions.

Applying the new frameworks to real life systems This thesis develops the mixed and the game framework. Their capabilities are assessed in a theoretical way. It would be interesting to implement those frameworks and apply them to real life system failures. This would gives some insight on how

the results of the different causality definitions can be used, as well as the scalability. It would help establish the mixed and the game frameworks as design/responsibility ascription tools.

Extending the approaches to other system models The systems considered in this manuscript are synchronous data-flow ones. They have been chosen because they are simple to understand, and the communications model is straight-forward. Besides, this system class is often used to develop safety critical systems, which benefit from having access to responsibility assignment tools. Nevertheless, a great care has been given to making the frameworks as general as possible, having a minimum of constructs relying on the system models. What's more, experience has shown that causality analysis is easy to instantiate to different models. Therefore, the frameworks developed in this thesis are a sound basis to be able to expand them to more complex system models, with more complicated communication models. This would allow to be able to have access to a responsibility ascription tool for more classes of systems.

Developing conservative heuristics Synthesis usually resorts to model-checking which generally scales poorly. Therefore, heuristics will be needed in order to treat big systems. The idea of conservative (i.e. that a cause generated by the heuristic is also a cause when applying the normal approach) heuristics is closely tied to the work in Section 7. Indeed the grey-box components, for instance give a heuristic to keep the exact same result as the vanilla causality analysis on critical logs. The different results from this section are a basis to create those heuristics.

On the fly causality analysis The grey-box components framework assesses the minimum amount of information needed to compute counter-factuals that are the same as the one from the full trace. Would they be combined with fault detection mechanism, it would be possible to start building counter-factuals as soon as faulty components are detected. Monitored systems are particularly suited for such an approach, as they have build in fault detection and logging mechanisms. Being able to assess responsibility in real time can be very useful when a critical failure occurs, as it gives set of components that are responsible for the failure. Those causes can be used to choose which components should be fixed first, in order to get the system back to a non-failing state. Having optimal fixes (by weighting the causes, e.g. a reparation cost or time) means that you can fix the system in most efficient and fast way.

Exploring further the impact of additional information Section 7 sketches some ideas on the impact of having access to extra-information on causality analysis. Be it to reduce the needed logs, or enhancing the accuracy of causality analysis, having access to the inner state of components, or other information can be helpful. This is an interesting direction for future research.

Accountability by design The idea is to consider accountability during the design phase, thus making it easier to perform the causality analysis on the finished system. This ties well with the impact of information on causality analysis aspect of this thesis, as for instance, by design, logging only certain flows from the components might be enough to perform a complete causality analysis, if the system is designed for it. Another interesting area to explore is the externalisation of inner variables to ease causality analysis, and it is better to address those concerns during the design phase. An initial work was produced during this thesis on a contract class that would ease causality analysis. However, it was hard to logically link this work to the two main axes (mixed framework and information impact) and was not mature enough to be a meaningful contribution. Nevertheless, it is a solid basis to start exploring accountability by design.

Bibliography

[Agrawal and Horgan, 1990] Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256.

[Agrawal et al., 1995] Agrawal, H., Horgan, J. R., London, S., and Wong, W. E. (1995). Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151.

[Alipour, 2012] Alipour, A. (2012). Automated fault localization techniques; a survey. *Technical Report Oregon State University*.

[Arumuga Nainar et al., 2007] Arumuga Nainar, P., Chen, T., Rosin, J., and Liblit, B. (2007). Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 5–15, New York, NY, USA. ACM.

[Asarin et al., 1995] Asarin, E., Maler, O., and Pnueli, A. (1995). *Symbolic controller synthesis for discrete and timed systems*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Avizienis et al., 2004] Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Ball and Larus, 1996] Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA. IEEE Computer Society.

[Ball et al., 2003] Ball, T., Naik, M., and Rajamani, S. K. (2003). From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105.

[Barrett et al., 2015] Barrett, C., Fontaine, P., and Tinelli, C. (2015). The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[Beer et al., 2012] Beer, I., Ben-David, S., Chockler, H., Orni, A., and Trefler, R. J. (2012). Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40.

[Benveniste et al., 2003] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Le Guernic, P., and De Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83.

[Berry, 1989] Berry, G. (1989). Real time programming : special purpose or general purpose languages. Research Report RR-1065, INRIA.

[Biere et al., 2009] Biere, A., Biere, A., Heule, M., van Maaren, H., and Walsh, T. (2009). *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.

[Cassandras and Lafortune, 1999] Cassandras, C. and Lafortune, S. (1999). *Introduction to Discrete Event Systems*. Kluwer.

[Chilimbi et al., 2009] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. (2009). Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA. IEEE Computer Society.

[Chockler et al., 2015] Chockler, H., Fenton, N., Keppens, J., and Lagnado, D. A. (2015). Causal analysis for attributing responsibility in legal cases. In *Proceedings of the 15th International Conference on Artificial Intelligence and Law*, ICAIL '15, pages 33–42, New York, NY, USA. ACM.

[Chockler et al., 2008] Chockler, H., Grumberg, O., and Yadgar, A. (2008). *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, chapter Efficient Automatic STE Refinement Using Responsibility, pages 233–248. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Chockler and Halpern, 2004] Chockler, H. and Halpern, J. Y. (2004). Responsibility and blame: A structural-model approach. *J. Artif. Intell. Res. (JAIR)*, 22:93–115.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA. ACM.

[Console et al., 1993] Console, L., Friedrich, G., and Dupré, D. T. (1993). Model-based diagnosis meets error diagnosis in logic programs (extended abstract). In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 85–87, London, UK, UK. Springer-Verlag.

[Csallner et al., 2008] Csallner, C., Tillmann, N., and Smaragdakis, Y. (2008). Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 281–290, New York, NY, USA. ACM.

[Datta et al., 2015] Datta, A., Garg, D., Kaynar, D. K., Sharma, D., and Sinha, A. (2015). Program actions as actual causes: A building block for accountability. *CoRR*, abs/1505.01131.

[de Kleer and Williams, 1987] de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Debbi and Bourahla, 2013] Debbi, H. and Bourahla, M. (2013). Generating diagnoses for probabilistic model checking using causality. *CIT*, 21(1):13–23.

[Dupin de Saint Cyr Bannay, 2008] Dupin de Saint Cyr Bannay, F. (2008). Scenario Update Applied to Causal Reasoning (regular paper). In Brewka, G. and Lang, J., editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR), Sydney, Australia, 16/09/2008-19/09/2008*, pages 188–197, http://www.aaai.org/Press/press.php. AAAI Press.

[Ehlers et al., 2016] Ehlers, R., Lafortune, S., Tripakis, S., and Vardi, M. Y. (2016). Supervisory control and reactive synthesis: a comparative introduction. *Discrete Event Dynamic Systems*, pages 1–52.

[Ernst et al., 2007] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45.

[Erwan Jahier, 2016] Erwan Jahier, Pascal Raymond, N. H. (2016). The lustre v6 reference manual.

[Goessler and Astefanoaei, 2014] Goessler, G. and Astefanoaei, L. (2014). Blaming in component-based real-time systems. In Mitra, T. and Reineke, J., editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 7:1–7:10. ACM.

[Gössler and Le Métayer, 2013] Gössler, G. and Le Métayer, D. (2013). A general trace-based framework of logical causality. In *International Workshop on Formal Aspects of Component Software*, pages 157–173. Springer.

[Gössler and Métayer, 2015] Gössler, G. and Métayer, D. L. (2015). A general framework for blaming in component-based systems. *Sci. Comput. Program.*, 113:223–235.

[Gössler and Stefani, 2016] Gössler, G. and Stefani, J.-B. (2016). *Fault Ascription in Concurrent Systems*, pages 79–94. Springer International Publishing, Cham.

[Groce et al., 2006] Groce, A., Chaki, S., Kroening, D., and Strichman, O. (2006). Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247.

[Groce and Visser, 2003] Groce, A. and Visser, W. (2003). *Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings*, chapter What Went Wrong: Explaining Counterexamples, pages 121–136. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hagen, 2008] Hagen, G. (2008). *VERIFYING SAFETY PROPERTIES OF LUSTRE PROGRAMS: AN SMT-BASED APPROACH*. PhD thesis, The University of Iowa.

[Hagen and Tinelli, 2008] Hagen, G. and Tinelli, C. (2008). Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–9.

[Halbwachs et al., 1991a] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991a). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320.

[Halbwachs et al., 1991b] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991b). The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320.

[Halbwachs et al., 1989] Halbwachs, N., Pilaud, D., Ouabdesselam, F., and Glory, A. (1989). Specifying, programming and verifying real-time systems using a synchronous declarative language. In Sifakis, J., editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 213–231. Springer.

[Halpern and Pearl, 2001a] Halpern, J. Y. and Pearl, J. (2001a). Causes and explanations: A structural-model approach - part II: explanations. In Nebel, B., editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 27–34. Morgan Kaufmann.

[Halpern and Pearl, 2001b] Halpern, J. Y. and Pearl, J. (2001b). Causes and explanations: A structural-model approach: Part 1: Causes. In Breese, J. S. and Koller, D., editors, *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*, pages 194–202. Morgan Kaufmann.

[Heh-Tyan et al., 1990] Heh-Tyan, L., Jia-Horng, T., and Chen-Shang, L. (1990). Efficient automatic diagnosis of digital circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 464–467.

[Hitchcock, 2001] Hitchcock, C. (2001). The intransitivity of causation revealed in equations and graphs. *The Journal of Philosophy*, 98(6):273–299.

[Hu et al., 2008] Hu, P., Zhang, Z., Chan, W. K., and Tse, T. H. (2008). Fault localization with non-parametric program behavior model. In *2008 The Eighth International Conference on Quality Software*, pages 385–395.

[Hume, 2004] Hume, D. (2004). *An Enquiry Concerning Human Understanding*. Dover philosophical classics. Dover Publications.

[Jalbert and Sen, 2010] Jalbert, N. and Sen, K. (2010). A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of*

the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 57–66, New York, NY, USA. ACM.

[Jiang and Su, 2007] Jiang, L. and Su, Z. (2007). Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 184–193, New York, NY, USA. ACM.

[Jin et al., 2004] Jin, H., Ravi, K., and Somenzi, F. (2004). Fate and free will in error traces. *International Journal on Software Tools for Technology Transfer*, 6(2):102–116.

[Jobstmann et al., 2012] Jobstmann, B., Staber, S., Griesmayer, A., and Bloem, R. (2012). Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460.

[Jones and Harrold, 2005] Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA. ACM.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA. ACM.

[Jose and Majumdar, 2010] Jose, M. and Majumdar, R. (2010). Cause clue clauses: Error localization using maximum satisfiability. *CoRR*, abs/1011.1589.

[Kayser and Nouioua, 2005] Kayser, D. and Nouioua, F. (2005). About Norms and Causes. *International Journal on Artificial Intelligence Tools*, 14(1-2):7–23.

[Kuntz et al., 2011] Kuntz, M., Leitner-Fischer, F., and Leue, S. (2011). *Computer Safety, Reliability, and Security: 30th International Conference,SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, chapter From Probabilistic Counterexamples via Causality to Fault Trees, pages 71–84. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Laprie et al., 1990] Laprie, J.-C., Béounes, C., and Kanoun, K. (1990). Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51.

[Lewis, 1973] Lewis, D. (1973). Causation. *Journal of Philosophy*, 70(17):556–567.

[Lewis, 2000] Lewis, D. (2000). Causation as influence. *Journal of Philosophy*, 97(4):182–197.

[Liao and Cohen, 1992] Liao, Y. and Cohen, D. (1992). A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978.

[Liblit et al., 2005] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26.

[Liu et al., 2006] Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. P. (2006). Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Software Eng.*, 32(10):831–848.

[Liu et al., 2005] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). Sober: statistical model-based bug localization. In Wermelinger, M. and Gall, H. C., editors, *ESEC/SIGSOFT FSE*, pages 286–295. ACM.

[Lyle and Weiser, 1987] Lyle, J. R. and Weiser, M. (1987). Automatic Program Bug Location by Program Slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking. IEEE Computer Society Press, Los Alamitos, California, USA.

[Madre et al., 1989] Madre, J. C., Coudert, O., and Billon, J. P. (1989). Automating the diagnosis and the rectification of design errors with priam. In *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pages 30–33.

[Menzies, 2014] Menzies, P. (2014). Counterfactual theories of causation. `http://plato.stanford.edu/archives/spr2014/entries/causation-counterfactual/`.

[Misherghi and Su, 2006] Misherghi, G. and Su, Z. (2006). Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151, New York, NY, USA. ACM.

[Odersky, 2004] Odersky, M. (2004). The scala programming language.

[Pal and Mohiuddin, 2013] Pal, D. and Mohiuddin, R. (2013). Automated bug localization of software programs : A survey report. Technical report, Univerisity of Illinois at Urbana-Champaign.

[Pan and Spafford, 1992] Pan, H. and Spafford, E. H. (1992). Heuristics for automatic localization of software faults. Technical report, Purdue University.

[Pons et al., 2015] Pons, R., Subias, A., and Travé-Massuyès, L. (2015). Iterative hybrid causal model based diagnosis: Application to automotive embedded functions. *Engineering Applications of Artificial Intelligence*, 37:319–335.

[Poole et al., 1987] Poole, D., Goebel, R., and Aleliunas, R. (1987). Theorist: A logical reasoning system for defaults and diagnosis. In Cercone, N. and McCalla, G., editors, *The Knowledge Frontier*, pages 331–352. Springer, New York.

[Powell, 1992] Powell, D. (1992). Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 386–395.

[Ramadge and Wonham, 1987] Ramadge, P. and Wonham, W. (1987). Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1).

[Ramadge and Wonham, 1989] Ramadge, P. and Wonham, W. (1989). The control of discrete event systems. *Proceedings of IEEE*, 77(1):81–98.

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57 – 95.

[Renieris and Reiss, 2003] Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *ASE*, pages 30–39. IEEE Computer Society.

[Sagdeo et al., 2011] Sagdeo, P., Athavale, V., Kowshik, S., and Vasudevan, S. (2011). Precis: Inferring invariants using program path guided clustering. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 532–535.

[Steinder and Sethi, 2004] Steinder, M. and Sethi, A. S. (2004). A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165 – 194. Topics in System Administration.

[US Department of Defense, 1949] US Department of Defense (1949). Procedures for performing a failure mode, effects and criticality analysis.

[Wang et al., 2006] Wang, C., Yang, Z., Ivančić, F., and Gupta, A. (2006). *Automated Technology for Verification and Analysis: 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006. Proceedings*, chapter Whodunit? Causal Analysis for Counterexamples, pages 82–95. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Wang et al., 2015] Wang, S., Geoffroy, Y., Gössler, G., Sokolsky, O., and Lee, I. (2015). A Hybrid Approach to Causality Analysis. In *RV 2015 - 6th International Conference on Runtime Verification*, volume 9333 of *LNCS*, Vienna, Austria.

[Weiser, 1981] Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press.

[Weiser, 1982] Weiser, M. (1982). Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452.

[Wong et al., 2008] Wong, E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 42–51, Washington, DC, USA. IEEE Computer Society.

[Wong and Qi, 2004] Wong, W. E. and Qi, Y. (2004). An execution slice and inter-block data dependency-based approach for fault localization. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 366–373, Washington, DC, USA. IEEE Computer Society.

[Woodward and Hitchcock, 2003] Woodward, J. and Hitchcock, C. (2003). Explanatory generalizations, part i: A counterfactual account. *Noûs*, 37(1):1–24.

[Xuan et al., 2016] Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Berre, D. L., and Monperrus, M. (2016). Nopol:

Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, PP(99):1–1.

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA. ACM.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200.

[Zhang et al., 2005] Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 33–42, New York, NY, USA. ACM.

[Zhang et al., 2007] Zhang, X., Tallam, S., Gupta, N., and Gupta, R. (2007). Towards locating execution omission errors. *SIGPLAN Not.*, 42(6):415–424.

[Zhang et al., 2009] Zhang, Z., Chan, W. K., Tse, T. H., Hu, P., and Wang, X. (2009). Is non-parametric hypothesis testing model robust for statistical fault localization? *Information & Software Technology*, 51(11):1573–1585.

[Zhang et al., 2010] Zhang, Z., Jiang, B., Chan, W., Tse, T., and Wang, X. (2010). Fault localization through evaluation sequences. *Journal of Systems and Software*, 83(2):174 – 187. Computer Software and Applications.