# Analysis of Cyclic Fault Propagation via ASP

Marco Bozzano[1 [0000−0002−4135−103X]],
Alessandro Cimatti[1 [0000−0002−1315−6990]], Alberto Griggio[1 [0000−0002−3311−0893]],
Martin Jonáš[1 [0000−0003−4703−0795]], and Greg Kimberly[2]

[1] Fondazione Bruno Kessler, Trento, Italy
{bozzano,cimatti,griggio,mjonas}@fbk.eu
[2] The Boeing Company, Seattle, USA
greg.kimberly@boeing.com

**Abstract.** Analyzing the propagation of faults is part of the preliminary safety assessment for complex safety-critical systems. A recent work proposes an SMT-based approach to deal with propagation of faults in presence of circular dependencies. The set of all the fault configurations that cause the violation of a property, also referred to as the set of minimal cut sets, is computed by means of repeated calls to the SMT solver, hence enumerating all minimal models of an SMT formula. Circularity is dealt with by imposing a strict temporal order, using the theory of difference logic.
In this paper, we explore the use of Answer-Set Programming to tackle the same problem. We propose two encodings, leveraging the notion of stable model. The first approach deals with cycles in the encoding, while the second relies on ASP Modulo Acyclicity (ASPMA).
We experimentally evaluate the three approaches on a comprehensive set of benchmarks. The first ASP-based encoding significantly outperforms the SMT-based approach; the ASPMA-based encoding, on the other hand, does not yield the expected performance gains.

**Keywords:** Fault propagation; SMT; ASP modulo acyclicity; Minimal models

## 1 Introduction

Analyzing the propagation of faults is an important step of the preliminary safety assessment for complex safety-critical systems. When a physical component fails, its faults can propagate to the other components and compromise their behaviour. Fault propagation is often mitigated by adopting suitable architectures based on redundancy and voting. In order to analyze such architectures, the challenge is to compute the set of all minimal cut sets (MCS), i.e., minimal fault configurations that can compromise a given function under investigation. Since the behavior of the systems in question is usually monotone, i.e., adding more faults does not fix the compromised function, the *minimal* cut sets are sufficient to succinctly represent the set of *all* cut sets, which might be exponentially larger. From the set of all minimal cut sets it is thus possible to extract

important artifacts such as fault trees and reliability measures (e.g., overall system failure probability). For this reason, the main focus of this paper is on the task of enumerating all minimal cut sets of the given system.

MCS enumeration is particularly challenging when dealing with cyclic dependencies. Consider, for example, the case of an electrically-controlled hydraulic system. Its fault may compromise power generation; on the other hand, the failure of power generation may compromise the hydraulic operation. This circularity makes it difficult to model fault propagation in form of simple logical implications because self-supporting, unjustified models arise. A recent work [4] shows how the inherent sequential nature of the problem can be reduced to an approach based on Satisfiability Modulo Theories (SMT). The set of minimal cut sets is computed by means of repeated calls to the SMT solver, hence enumerating all minimal models of an SMT formula. The key idea in dealing with circularity is to impose a strict temporal ordering on the propagation of events, using the theory of difference logic. The results presented in [3] and in [4] show that the SMT approach is able to deal with realistically-sized redundancy architectures.

In this paper, we explore an alternative approach to minimal cut set enumeration, based on the use of Answer-Set Programming (ASP). The intuition is to leverage the fact that in ASP clauses are interpreted as (directed) rules rather than implications, thus limiting the search based on the notion of stable model.

We propose two approaches. The first one is a direct encoding into ASP. It deals with cycles in the encoding by requiring that the failure of a component must be justified either by a local fault or by the justified failure of neighboring components (or their combination). Default negation is used to model the justifications of the propagation.

The second encoding relies on the idea of ASP Modulo Acyclicity (ASPMA) [2], where models can be required to be acyclic with directives to an extended solver. Acyclicity is then enforced at run-time by means of a dedicated, graph-based data-structure preventing circular dependencies. Although not all ASP solvers deal with a built-in "modulo acyclicity" feature, we expected that this could lead to additional performance boost.

We carried out an extensive experimental evaluation, on all Boolean (real-world and random scalable) fault propagation benchmarks from [4] and [3]. The benchmark suite includes both acyclic and cyclic problems. We contrasted the SMT approach with the ASP and ASPMA approaches proposed in this paper. On acyclic benchmarks, the ASP encodings demonstrate better scalability than the SMT-based cut set enumeration. On the cyclic benchmarks, the ASP encoding dominates over the SMT-based encoding. Quite surprisingly, the ASPMA encoding does not scale as well, and it is outperformed, especially on the hardest benchmarks, both by the ASP-based and SMT-based encodings.

The paper is structured as follows. Section 2 presents the logical preliminaries. Section 3 describes fault propagation graphs and the SMT-based encoding. Section 4 presents the ASP-based and ASPMA-based encodings. Section 5 discusses the issue of minimality. Section 6 presents the experimental evaluation. Section 7 draws conclusions and outlines directions for future works.

## 2 Preliminaries

### 2.1 Logic and Notation

We assume that the reader is familiar with standard first-order logic and the basic ideas of Satisfiability Modulo Theories (SMT), as presented, e.g., in [1]. We use the standard notions of interpretation, theory, assignment, model, and satisfiability.

Given a quantifier-free formula $\varphi(B, R)$ in real arithmetic defined over a set of Boolean variables $B$ and of real variables $R$, a *model* of $\varphi$ is an assignment $\mu$ that maps each $b \in B$ to a truth value $\mu(b) \in \mathbb{B}$ ($\top$ for true and $\bot$ for false) and each $x \in R$ to a real number $\mu(x) \in \mathbb{R}$, such that $\varphi$ evaluates to true on $\mu$. We denote this with $\mu \models \varphi$. If $\varphi$ is a formula and $\mu$ is an assignment that maps each variable of $\varphi$ to a value of the corresponding sort, $[\![\varphi]\!]_\mu$ denotes the result of the evaluation of $\varphi$ under this assignment. If $B' \subseteq B$ is a subset of the Boolean variables of $\varphi$, $\mu$ is called its *minimal model with respect to* $B'$ if $\mu$ is a model of $\varphi$ and there is no model $\mu' \models \varphi$ such that $\{b \in B' \mid \mu'(b) = \top\} \subsetneq \{b \in B' \mid \mu(b) = \top\}$.

### 2.2 Answer Set Programming

This subsection briefly introduces the syntax and semantics of disjunctive Answer-Set Programs and ASP modulo acyclicity, based on [8] and [2], respectively.

*Rules.* A disjunctive rule $r$ is an expression of the form

$$p_1 \vee \ldots \vee p_l \;\leftarrow\; p_{l+1}, \ldots, p_m, {\sim}p_{m+1}, \ldots, {\sim}p_n,$$

where $0 \leq l \leq m \leq n$ and $p_1, \ldots, p_n$ are propositional atoms. The *head* of $r$ is defined as $hd(r) = \{p_1, \ldots p_l\}$ and the *body* of $r$ is defined as $bd(r) = \{p_{l+1}, \ldots, p_m, {\sim}p_{m+1}, \ldots, {\sim}p_n\}$. For any set $L = \{p_{l+1}, \ldots, p_m, {\sim}p_{m+1}, \ldots, {\sim}p_n\}$, let $L^+ = \{p_{l+1}, \ldots, p_m\}$ and $L^- = \{p_{m+1}, \ldots, p_n\}$. A rule $r$ is said to be *applicable* with respect to a set of propositional atoms $X$ if the set $X$ contains all the positive atoms from $bd(r)$ and no negative atoms from $bd(r)$, i.e., $bd(r)^+ \subseteq X$ and $bd(r)^- \cap X = \emptyset$. The rule $r$ is said to be *satisfied* with respect to $X$ if its body implies its head, i.e., the rule is not applicable or $hd(r) \cap X \neq \emptyset$. The rules with $bd(r) = \emptyset$ are called *facts* and are written as $p_1 \vee \ldots \vee p_l$. The rules with $hd(r) = \emptyset$ are called *integrity constraints*, are written as $\leftarrow p_{l+1}, \ldots, p_m, {\sim}p_{m+1}, \ldots, {\sim}p_n$, and are satisfied only if they are not applicable, i.e., if at least one of the atoms in the body is not satisfied.

*Answer Set Programs.* A disjunctive answer set program $P$ is a set of rules. A set of atoms that occur in the program $P$ is denoted as $\mathrm{At}(P)$. A set of atoms $X$ is called a *model* of $P$ if all rules $r \in P$ are satisfied with respect to $X$. The *reduct of $P$ with respect to the set of atoms $X$* is defined as $P^X = \{hd(r) \leftarrow bd(r)^+ \mid r \in P, bd(r)^- \cap X = \emptyset\}$. A model $X$ of $P$ is called an *answer set of $P$* or a *stable model of $P$* if $X$ is a minimal model of $P^X$, i.e., there is no $Y \subsetneq X$ such that $Y$ is a model of $P^X$.

*ASP modulo acyclicity.* An *acyclicity extension* of a program $P$ is a pair $(V, e)$ where $V$ is a set of nodes and $e\colon \mathrm{At}(P) \to V \times V$ is a partial function that assigns edges between vertices of $V$ to atoms of $P$. A program together with its acyclicity extension is called an *acyclicity program*.

A set of atoms $X$ is called a *stable model of the acyclicity program $P$ subject to the acyclicity extension $(V, e)$* if $X$ is a stable model of $P$ and the graph $(V, \{e(p) \mid p \in X\})$ induced by the set of atoms $X$ is acyclic.

## 3 Fault Propagation Graphs

In this section we briefly introduce the formalism of *(symbolic) fault propagation graphs* (FPGs). Intuitively, FPGs describe how failures of some components of a given system can cause the failure of other components of the system. In an explicit graph representation, nodes correspond to components, and edges model their dependencies, with the meaning that an edge from $c_1$ to $c_2$ states that the failure of $c_1$ can cause the failure (propagation) of $c_2$. Here, we adopt a symbolic representation, in which components are modeled as Boolean variables (where $\bot$ means "not failed" and $\top$ means "failed"), and the failure dependencies are encoded as formulae $\mathsf{canFail}(c)$, which describe the conditions that may cause a failure of $c$. The basic concepts are formalized in the following definitions, which are simplified definitions from [3] and [4]. The original paper [4] also defines FPGs with multiple failure modes with arbitrary orderings. We do not treat these features here to simplify the presentation, but we note that the approach of this paper can be extended to accommodate them in the same way as in [4].

**Definition 1 (Fault propagation graph [3]).** *A (symbolic) fault propagation graph (*FPG*) is a pair $(C, \mathsf{canFail})$, where $C$ is a finite set of* system components *and* $\mathsf{canFail}$ *is a function that assigns to each component $c$ a Boolean formula $\mathsf{canFail}(c)$ over the set of variables $C$.*

We assume that all the $\mathsf{canFail}(c)$ formulas are *positive*, i.e., they can contain only conjunctions, disjunctions, and variables. Moreover, without loss of generality, we assume that all the $\mathsf{canFail}(c)$ formulas are in disjunctive normal form, i.e., they are of the form $\bigvee_{D \in F} \bigwedge_{d \in D} d$ for some set $F$ of *cubes* of dependencies.

**Definition 2 (Trace of FPG [3]).** *Let $G$ be an* FPG *$(C, \mathsf{canFail})$. A* state *of $G$ is a function from $C$ to $\mathbb{B}$. A trace of $G$ is a (potentially infinite) sequence of states $\pi = \pi_0\pi_1\ldots$ such that all $i > 0$ and $c \in C$ satisfy (i) $\pi_i(c) = \pi_{i-1}(c)$ or (ii) $\pi_{i-1}(c) = \bot$ and $\pi_i(c) = [\![\mathsf{canFail}(c)]\!]_{\pi_{i-1}}$.*

*Example 1 ([3]).* Consider a system with components *control on ground* (G), *hydraulic control* (H), and *electric control* (E) such that G can fail if both H and E have failed, H can fail if E has failed, and E can fail if H has failed. This system can be modeled by a fault propagation graph $(\{\mathrm{G}, \mathrm{H}, \mathrm{E}\}, \mathsf{canFail})$, where $\mathsf{canFail}(\mathrm{G}) = \mathrm{H} \wedge \mathrm{E}$, $\mathsf{canFail}(\mathrm{H}) = \mathrm{E}$, and $\mathsf{canFail}(\mathrm{E}) = \mathrm{H}$.

One of the traces of this system is $\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \bot\}\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}\{\mathrm{G} \mapsto \top, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}$, where H is failed initially, which causes

failure of E in the second step, and the failures of H and E together cause a failure of G in the third step.

**Definition 3 (Cut set [3]).** *Let $G$ be an* FPG $G = (C, \mathsf{canFail})$ *and $\varphi$ a positive Boolean formula, called* top level event*. The assignment $cs \colon C \to \mathbb{B}$ is called a* cut set *of $G$ for $\varphi$ if there is a trace $\pi$ of $G$ that starts in the state $cs$ and there is some $k \geq 0$ such that $\pi_k \models \varphi$. A cut set $cs$ is called* minimal *if there is no other cut set $cs'$ such that $\{c \in C \mid cs'(c) = \top\} \subsetneq \{c \in C \mid cs(c) = \top\}$.*

Without loss of generality, we assume in the rest of the paper that the top level event $\varphi$ consists only of one variable, i.e., $\varphi = c$ for some $c \in C$. For brevity, when talking about cut sets, we often mention only the components that are set to $\top$ by the cut set.

*Example 2 ([3]).* The minimal cut sets of the FPG from Example 1 for the top level event $\varphi = \mathrm{G}$ are $\{\mathrm{G}\}$, $\{\mathrm{H}\}$, and $\{\mathrm{E}\}$. These three cut sets are witnessed by the following traces:

1. $\{\mathrm{G} \mapsto \top, \mathrm{H} \mapsto \bot, \mathrm{E} \mapsto \bot\}$,
2. $\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \bot\}\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}\{\mathrm{G} \mapsto \top, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}$,
3. $\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \bot, \mathrm{E} \mapsto \top\}\{\mathrm{G} \mapsto \bot, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}\{\mathrm{G} \mapsto \top, \mathrm{H} \mapsto \top, \mathrm{E} \mapsto \top\}$.

Note that the FPG has also other cut sets, such as $\{\mathrm{G}, \mathrm{E}\}$, $\{\mathrm{H}, \mathrm{E}\}$, and $\{\mathrm{G}, \mathrm{H}, \mathrm{E}\}$, which are not minimal.

### 3.1 SMT-based Encoding of Fault Propagation

In our previous work [4], we have shown that MCS enumeration of cyclic FPGs can be reduced to enumeration of projected minimal models of a certain SMT formula over the difference logic fragment of linear real arithmetic. The arithmetic is used to enforce causality ordering between the propagated failures, which would otherwise cause spurious self-supported propagations.

In particular, for each FPG $G = (C, \mathsf{canFail})$, the paper defines a formula that contains two Boolean variables $I_c$ and $F_c$ and one real variable $o_c$ for each component $c$. The variables have the following intuitive meaning: $I_c$ denotes that the component $c$ is failed in the initial state, $F_c$ denotes that the component $c$ is failed at some point during the propagation, and $o_c$ is a so called *time stamp* variable, which intuitively denotes the time when the component $c$ failed.

These variables are then used to construct a formula $\varphi_{prop}$, which describes fault propagations. The formula contains the following constraints for each $c \in C$ with $\mathsf{canFail}(c) = \bigvee_{D \in F} \bigwedge_{d \in D} d$:

- $I_c \to F_c$, i.e., if the component is failed initially, it is failed at some point during the propagation,
- $F_c \to (I_c \vee \bigvee_{D \in F} \bigwedge_{d \in D} (F_d \wedge o_d < o_c))$, i.e., if component $c$ fails at some point during the propagation, it is failed either initially or as a result of a propagation from its failed dependencies *that failed before* $c$.

Insisting that a failure of a variable can be caused only by failures that occurred before it is a crucial point to preserve causality and prohibit self-supporting cyclic propagations where a component causes its own failure.

## 4 Encoding in Disjunctive ASP

In this section we present our novel encodings of fault propagation in ASP. In the rest of the section, let $G = (C, \mathsf{canFail})$ be a fixed FPG and $c_{tle} \in C$ a top level event.

### 4.1 Encoding Propagations

The encoding uses the following variables for each component $c \in C$ with $\mathsf{canFail}(c) = \bigvee_{D \in F} \bigwedge_{d \in D} d$ and $F = \{D_1, \ldots, D_n\}$:

- $\mathtt{fail}(c)$, which will denote that $\pi_i(c) = \top$ for some $i \geq 0$,
- $\mathtt{fail\_local}(c)$, which will denote that $\pi_0(c) = \top$, i.e., the component $c$ is initially failed,
- $\mathtt{fail\_ext}(c)$, which will denote that $\pi_0(c) \neq \top$ and $\pi_i(c) = \top$ for some $i > 0$, i.e., the component $c$ is failed as a result of fault propagation.
- $\mathtt{fail\_dep}(c, j)$ for each $1 \leq j \leq n$, which will denote that $\pi_i \models \bigwedge_{d \in D_j} d$ for some $i \geq 0$, i.e., the conditions of a propagated failure of $c$ are satisfied thanks to the $j$-th disjunct of $\mathsf{canFail}(c)$.

Using these variables, we construct an answer set program that contains the fact

$$\mathtt{fail}(c_{tle}) \tag{1}$$

and the following rules for each component $c \in C$:

$$\mathtt{fail\_local}(c) \vee \mathtt{fail\_ext}(c) \leftarrow \mathtt{fail}(c), \tag{2}$$

$$\mathtt{fail\_dep}(c, 1) \vee \ldots \vee \mathtt{fail\_dep}(c, n) \leftarrow \mathtt{fail\_ext}(c), \tag{3}$$

$$\mathtt{fail}(d) \leftarrow \mathtt{fail\_dep}(c, j) \text{ for each } 1 \leq j \leq n, d \in D_j. \tag{4}$$

The rules have the following meaning: (1) states that the TLE must be satisfied; (2) that if a component is failed, it has to be failed either initially or as a result of a propagation; (3) that if a component is failed as a result of a propagation, one of the disjuncts in $\mathsf{canFail}(c)$ must be satisfied; and (4) that if the $j$-th disjunct of $\mathsf{canFail}(c)$ is satisfied, all the components that it depends on must be failed.

However, similarly to a naive SMT encoding, this encoding allows spurious propagations in presence of cycles. Given the FPG from Example 1, the encoding has a stable model $\{\mathtt{fail}(c), \mathtt{fail\_ext}(c), \mathtt{fail\_dep}(c, 1) \mid c \in \{\text{G}, \text{H}, \text{E}\}\}$, where none of the components is failed initially, yet all are failed in the end. This model does not correspond to any real fault propagation and relies on the impossible propagation where H fails because of E, which in turn fails because of H. We now show two possible extensions of the encoding that solve this problem.

### 4.2 Enforcing Causality by ASP

To solve the problem with self-supporting circular propagations, we introduce new variables that will encode that a failure is *justified*, i.e., it is supported by

sufficient initial faults. Intuitively, a failure of component $c$ is justified if it is due to an initial fault of component $c$. Moreover, a failure of component $c$ is justified if it is due to a propagation from a dependency $D_j$ (the $j$-th disjunct of $\mathsf{canFail}(c)$) such that all $d \in D_j$ are in turn failed and justified.

Therefore, we introduce the following additional variables for each component $c \in C$ with $\mathsf{canFail}(c) = \bigvee_{D \in F} \bigwedge_{d \in D} d$ and $F = \{D_1, \ldots, D_n\}$:

- $\texttt{justified}(c)$, which will denote that the failure of $c$ is justified,
- $\texttt{justified\_dep}(c, j)$ for each $1 \leq j \leq n$, which will denote that the failure of $c$ is justified by the $j$-th disjunct of $\mathsf{canFail}(c)$.

We then define the program $P_{asp}$ as a union of the rules from the previous subsection and the additional *causality rules*:

$$\leftarrow \texttt{fail}(c), {\sim}\texttt{justified}(c) \tag{5}$$

$$\texttt{justified}(c) \leftarrow \texttt{fail\_local}(c) \tag{6}$$

$$\texttt{justified}(c) \leftarrow \texttt{justified\_dep}(c, j), \texttt{fail\_dep}(c, j) \text{ for all } 1 \leq j \leq n \tag{7}$$

$$\leftarrow \texttt{fail\_dep}(c, j), {\sim}\texttt{justified\_dep}(c, j) \text{ for all } 1 \leq j \leq n \tag{8}$$

$$\texttt{justified\_dep}(c, j) \leftarrow \texttt{justified}(d_1), \ldots, \texttt{justified}(d_m) \text{ where } d_i \in D_j \tag{9}$$

The rules have the following intuitive meaning: (5) states that it is not possible that the component $c$ is failed without a justification for the failure; (6) that the local failure is enough to justify the failure of the component; (7) that if the $j$-th disjunct is justified and satisfied, the failure of $c$ is justified; (8) that it is not possible that the $j$-th disjunct of the component $c$ is satisfied without a justification; and (9) that if all dependencies of the $j$-th disjunct are justified, the $j$-th disjunct itself is justified.

Observe how we use integrity constraints and default negation to impose that failed components/dependencies must be justified. This prohibits the spurious cyclic propagations and gives the following correctness result:

**Theorem 1.** *Let $X \subseteq \mathrm{At}(P_{asp})$ be a set of atoms. If $X$ is a stable model of $P_{asp}$ then $\{c \in C \mid \texttt{fail\_local}(c) \in X\}$ is a cut set of $G$ for $c_{tle}$. Conversely, if $\{c \in C \mid \texttt{fail\_local}(c) \in X\}$ is a* minimal *cut set of $G$ then $X$ is a stable model of $P_{asp}$.*

Note that due to the stable model semantics, the program $P_{asp}$ does not represent all cut sets of $G$, because some non-minimal cut sets are prohibited. Nevertheless, it represents all *minimal* cut sets, in which we are mainly interested.

### 4.3 Enforcing Causality by ASP Modulo Acyclicity

In this subsection, we present a second encoding of fault propagation. In contrast to the encoding from the previous subsection, which uses justification rules to break self-supporting cyclic propagations, this encoding relies on ASP modulo acyclicity. This makes the encoding simpler, easier to implement, and might offer

better performance due to dedicated implementation of acyclicity propagation in ASP solvers. On the other hand, it restricts the set of usable ASP solvers as not all ASP solvers support acyclicity constraints.

The encoding uses the variables $\texttt{fail}(c)$, $\texttt{fail\_local}(c)$, $\texttt{fail\_ext}(c)$, and $\texttt{fail\_dep}(c, j)$ with the same intuitive meaning as in the previous encoding. Moreover, for every pair of components $c, d \in C$ it uses a variable $\texttt{caused\_by}(c, d)$ with the intuitive meaning that the failure of $c$ was directly caused by the failure (initial or propagated) of $d$.

Using these variables, we construct the program $P_{aspma}$ that contains rule (1), for each component $c \in C$ contains the rules (2),(3),(4), and for each $c \in C$, $1 \leq j \leq n$ and $d \in D_j$ also the rule $\texttt{caused\_by}(c, d) \leftarrow \texttt{fail\_dep}(c, j)$. The rules state that if the $j$-th disjunct of $\mathsf{canFail}(c)$ is satisfied, the failure of $c$ is caused by failures of all the components in the disjunct. We then define the acyclicity extension $(C, e)$, where $e(\texttt{caused\_by}(c, d)) = (c, d)$ and $e$ is undefined for the remaining variables. This ensures that there are no causal cycles among the propagated failures and therefore no component can cause its own failure. As a result, an analogue of Theorem 1 holds also for $P_{aspma}$.

## 5 Minimality of the Cut Sets

As was shown in the previous section, both the introduced ASP encodings contain stable models for all MCSs of the given FPG. Although the stable-model semantics is able to rule out some non-minimal cut sets thanks to the condition that the model $X$ must be a *minimal* model of $P^X$, the programs still admit some stable models that correspond to *non-minimal* cut sets. This can be seen in the following example. Note that the FPG in question is acyclic, and therefore there is no need of encoding the causality constraints.

*Example 3.* Consider an FPG $(C, \mathsf{canFail})$ with $C = \{c_1, c_2, c_3\}$ and $\mathsf{canFail}(c_1) = c_2 \wedge c_3$, $\mathsf{canFail}(c_2) = c_3$, $\mathsf{canFail}(c_3) = \bot$ with the top level event $c_1$. The ASP encodings from the previous section produce the following program $P$:

$\texttt{fail}(c_1).$
$\texttt{fail\_local}(c_1) \vee \texttt{fail\_ext}(c_1) \leftarrow \texttt{fail}(c_1).$
$\texttt{fail\_local}(c_2) \vee \texttt{fail\_ext}(c_2) \leftarrow \texttt{fail}(c_2).$
$\texttt{fail\_local}(c_3) \vee \texttt{fail\_ext}(c_3) \leftarrow \texttt{fail}(c_3).$
$\texttt{fail\_dep}(c_1, 1) \leftarrow \texttt{fail\_ext}(c_1).$
$\texttt{fail\_dep}(c_2, 1) \leftarrow \texttt{fail\_ext}(c_2).$
$\leftarrow \texttt{fail\_ext}(c_3).$
$\texttt{fail}(c_2) \leftarrow \texttt{fail\_dep}(c_1, 1).$
$\texttt{fail}(c_3) \leftarrow \texttt{fail\_dep}(c_1, 1).$
$\texttt{fail}(c_3) \leftarrow \texttt{fail\_dep}(c_2, 1).$

This program has a stable model $M = \{\texttt{fail}(c_1), \texttt{fail}(c_2), \texttt{fail}(c_3), \texttt{fail\_ext}(c_1), \texttt{fail\_local}(c_2), \texttt{fail\_local}(c_3), \texttt{fail\_dep}(c_1, 1)\}$, which corresponds to a non-minimal cut set $\{c_2, c_3\}$. The reason for this is that in order to obtain a model for the minimal cut set $\{c_3\}$, the model $M$ would have to be extended with $\texttt{fail\_ext}(c_2)$ and $\texttt{fail\_dep}(c_2, 1)$ before removing $\texttt{fail\_local}(c_2)$. □

Non-minimal cut sets arise because the minimality of the cut set is defined with respect to the local faults, while the minimality of the model is defined with respect to all atoms, which also include the atoms used for propagation. Fortunately, ASP solvers offer *optimization* facilities for enumerating stable models that are minimal according to a given criterion. In particular, it is possible to enumerate minimal stable models with respect to a given subset of atoms, either by using subset preference [5] or modified branching heuristics [9]. Our preliminary experiments shown that the latter option provides vastly superior performance. Moreover, as each minimal cut set is identified only by values of atoms $\mathit{FailLocal} = \{\texttt{fail\_local}(c) \mid c \in C\}$, it is sufficient to enumerate the minimal models w.r.t *FailLocal*, projected to the set of atoms *FailLocal*.[3]

Note that the enumeration of *minimal* models is more expensive. It prevents the solver to perform enumeration based only on backtracking: it either forces the solver to minimize each model and possibly enumerate a single minimal model multiple times, or it forces the solver to remember *all* already enumerated models, which can increase the space complexity of the search. However, the technique based on branching heuristics is also successfully used for MCS enumeration in the original SMT-based approach.

## 6 Experimental Evaluation

### 6.1 Implementation and Setup

We implemented the encodings proposed in Section 4 in a simple Python script. In the following experiments, these two encodings are denoted as `asp` and `aspma`. To enumerate their minimal stable models, we have used the state-of-the-art ASP solver Clingo [7] in version 5.5.1, which supports both ASP modulo acyclicity and also Boolean model minimization by modified branching heuristics.

For comparison, we used the SMT-based MCS enumerator SMT-PGFDS [4], which is implemented as a Python tool that produces the SMT encoding of the FPG and uses the SMT solver MathSAT 5 [6] to enumerate its minimal models. In the experiments below, the approach is denoted as `smt`.

We used several families of FPG benchmarks, which are described in §6.2. We ran all the experiments on a cluster of 13 nodes with Intel Xeon CPU E5520 @ 2.27GHz CPUs. We used a 30 minutes timeout and 8 GiB of RAM. All the measured times are wall-times. Additional data for the experiments are available from `https://es-static.fbk.eu/people/griggio/papers/lpnmr2022.html`.

### 6.2 Benchmarks

We compared the three encodings (`asp`, `aspma`, `smt`) on various FPG benchmarks from the literature. We do not restrict the evaluation only to cyclic FPGs, where the three encodings differ, but also use benchmarks without cycles, which do not

[3] This can be achieved, e.g, by adding a directive `#show fail_local/1.` and calling `clingo --project --heuristic=Domain --dom-mod=5,16 --enum-mod=domRec 0`.

require the causality constraints and can thus be encoded in a purely Boolean way. This allows us to compare the performance of the underlying solvers (i.e., Clingo, MathSAT 5) for purely Boolean search.

**Acyclic** We used *acyclic* benchmarks that result from encoding acyclic redundancy architectures extended by triple modular redundancy (TMR) with voters [3]. In particular, these benchmarks consist of families `linear`, `rectangular`, and `redarch-random-acycl`. The `linear` benchmark family consists of linear-shaped architectures of sizes between 1 and 200, extended by one to three voters; each architecture of size $n$ corresponds to a FPG with $3n + (\#voters \cdot n)$ components. Similarly, `rectangular` benchmarks come from encoding of rectangular-shaped redundancy architectures of sizes between 1 and 200 and one to three voters and yield FPGs with $6n+2\cdot(\#voters\cdot n)$ components. Family `redarch-random-acycl` consists of FPG encodings of randomly generated acyclic TMR architectures.

**Cyclic** As *cyclic* benchmarks, we first used the benchmark family `cav21`, which is an extension of a benchmark set used to evaluate the performance of the SMT-based FPG analysis. It is generated exactly the same way as in the original paper [4] and consists of randomly generated FPGs of size between 500 and 1500, which have similar distribution of degrees as our proprietary industrial systems.

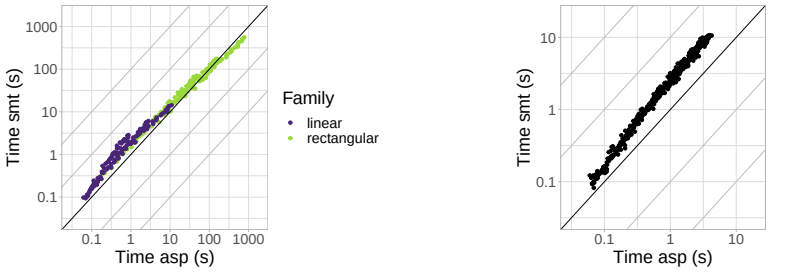
Second, we also use cyclic benchmarks that result from encoding cyclic redundancy architectures [3]. In particular, these benchmarks consist of three families: `ladder`, `radiator`, and `redarch-random-cycl`. Ladder-shaped benchmarks come from architectures of size between 1 and 200 and radiator-shaped benchmarks come from architectures of size between 1 and 50. Both these architecture shapes of size $n$ yield an FPG with $6n+2\cdot(\#voters\cdot n)$ components. However, the redundancy architectures differ in the shape of the dependency graph; whereas `ladder` benchmarks contain a linear number of cycles, `radiator` benchmarks contain an exponential number of cycles. Finally, `redarch-random-cycl` family consists of FPG encodings of randomly generated cyclic TMR architectures.

### 6.3 Results

This subsection presents MCS enumeration times for the compared encodings on the benchmarks. Note that all plots show times on a logarithmic scale.

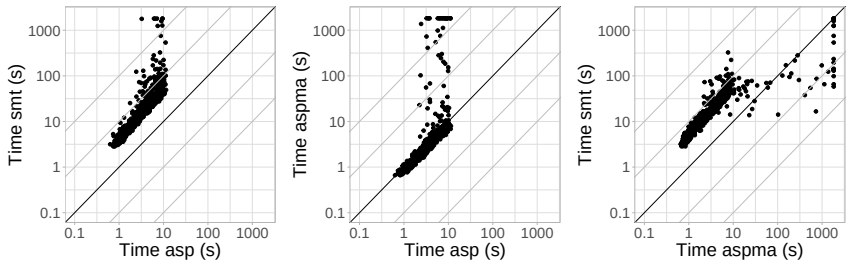
**Acyclic** The MCS enumeration times for the `smt` and `asp` encodings for `linear` and `rectangular` benchmarks are shown in Figure 1a. The plot does not show the runtimes of `aspma` encoding, because for benchmarks without cycles, it is identical to `asp`. For the simple benchmarks, ASP-based enumeration performs slightly better, but the difference vanishes with the increasing hardness of the benchmarks, i.e., going to rectangular structure or adding voters.

The comparison of MCS enumeration times of `smt` and `asp` encodings for `redarch-random-acycl` benchmarks is shown in Figure 1b. For FPGs coming




(a) `linear` and `rectangular` (b) `redarch-random-acycl`

Fig. 1: Scatter plots of solving time on acyclic benchmarks.




(a) `smt` vs `asp` (b) `asp` vs `aspma` (c) `smt` vs `aspma`

Fig. 2: Scatter plots of solving time on `cav21` cyclic benchmarks.

from random redundancy architectures, `asp` provides 2 to 3-times better performance.

**Cyclic** The comparison of MCS enumeration times of all three encodings for `cav21` benchmarks is shown in Figure 2. The ASP-based breaking of self-supporting propagation cycles is beneficial in comparison to the previously proposed SMT-based encoding; for some benchmarks, the ASP-based techniques provide 10-times and even better performance. The performance of `aspma` is significantly worse than the purely ASP-based one on a non-trivial number of the benchmarks.

Figures 3 and 4 show the performance of the three approaches on the `ladder` and `radiator` benchmarks. On the ladder-shaped benchmarks, the ASP-based approach provides substantial speedup with respect to the SMT-based approach. Interestingly, the ASPMA approach performs worse than the purely ASP-based approach and even comparable to the SMT-based one for the more complicated systems with more voters.

The situation is more interesting on `radiator` benchmarks, which are substantially harder as they contain a larger number of cycles. While the SMT-based approach provides a better performance for architectures with two voters, the ASP-based approach provides a better performance for even harder architectures

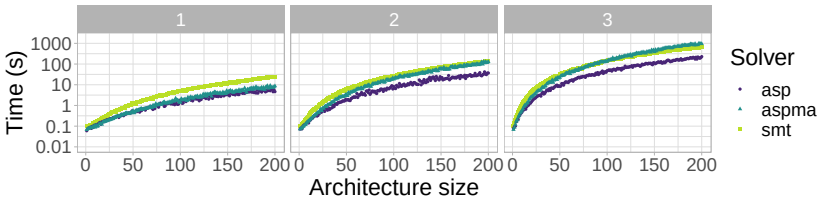



Fig. 3: Solving time on ladder-shaped cyclic benchmarks. Divided according to the number of voters per one reference module.

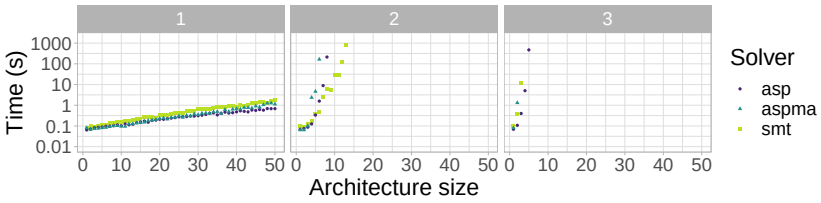
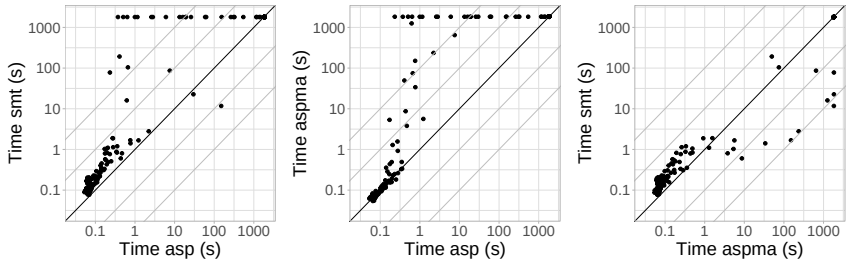



Fig. 4: Solving time on radiator-shaped cyclic benchmarks. Divided according to the number of voters per one reference module.

with three voters. Nevertheless, the benchmarks with two and three voters are difficult for all of the approaches and pose a good target for future research.

Finally, Figure 5 compares the three approaches on the family of benchmarks `redarch-random-cycl`. The purely ASP-based encoding provides significantly better performance both than `smt` and `aspma`. The difference can be even in several orders of magnitude. Nevertheless, there are a few benchmarks where the SMT-based encoding provides better performance.

In total, the approach based on ASP, which we introduced in this paper, provides better performance for most of the benchmarks used; the difference is sometimes even in several orders of magnitude. Interestingly, the approach based




(a) `smt` vs `asp` (b) `asp` vs `aspma` (c) `smt` vs `aspma`

Fig. 5: Scatter plots of solving time on `redarch-random-cycl` cyclic benchmarks.

on ASPMA, which uses a dedicated acyclicity solver, does not bring a significant benefit in comparison to the previously introduced solver based on SMT and is mostly inferior to our purely ASP-based encoding.

## 7 Conclusions and Future Work

We investigated the effectiveness of Answer Set Programming in the analysis of fault propagation with cyclic dependencies, an important problem in the design of critical systems. We propose two ASP approaches: in the first one, acyclicity is enforced by means of encoding constraints, while in the second we rely on ASP modulo acyclicity. The experimental evaluation shows that the ASP encoding has significant advantages over the state-of-the-art SMT encoding. We also see that, quite surprisingly, ASP modulo acyclicity does not yield the expected results.

In the future we will investigate in detail why the ASP-based encoding is superior to the SMT-based one and whether the observation can be leveraged to improve performance of SMT solvers. We will also investigate precise computational complexity of decision problems related to fault propagation analysis. Finally, we will explore extensions of the ASP approach to deal with fault propagation under timing constrains, partial observability, and with dynamic fault degradation structures with recovery.

## References


1. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.
2. Jori Bomanson, Martin Gebser, Tomi Janhunen, Benjamin Kaufmann, and Torsten Schaub. Answer set programming modulo acyclicity. *Fundam. Inform.*, 147(1):63–91, 2016.
3. Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Martin Jonáš. Efficient analysis of cyclic redundancy architectures via boolean fault propagation. In *TACAS*, volume 13244 of *LNCS*. Springer, 2022.
4. Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, Alberto Griggio, Martin Jonáš, and Greg Kimberly. Efficient SMT-Based Analysis of Failure Propagation. In *CAV*, volume 12760 of *LNCS*, pages 209–230. Springer, 2021.
5. Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In *AAAI*, pages 1467–1474. AAAI Press, 2015.
6. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.
8. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Advanced conflict-driven disjunctive answer set solving. In *IJCAI*, 2013.
9. Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. Solving satisfiability problems with preferences. *Constraints An Int. J.*, 15(4):485–515, 2010.