Resilience of Systems under Maximum Component Deviations

Abigail Hammer[1]✉, Changjian Zhang[1], Vick Dini[2], Ryan Wagner[1], Bradley Schmerl[1], Eunsuk Kang[1], and David Garlan[1]

[1] Carnegie Mellon University Pittsburgh PA 15213, USA
[2] Politecnico di Milano, Milano 20133, Italy
✉ arhammer@andrew.cmu.edu

Abstract. Software systems are composed of interacting processes that share data between each other in order to satisfy system properties. When these processes deviate and are unavailable to transmit data—due to events such as software bugs, hardware failures, or security attacks—a *resilient* system will continue delivering safety-critical services. Identifying the processes required to satisfy system properties is not a trivial task as there may exist multiple alternative dataflow paths that each satisfy the property. In this work, we propose a formal modeling and analysis technique to compute the sets of minimal processes required to satisfy dataflow properties. The computation of these sets is reduced to a *maximum satisfiability* problem via modeling in Alloy$^{\text{Max}}$, a formal modeling language that performs bounded model checking. We then present a method to formally define the resilience criteria for a system by constraining the minimal dataflow required under maximum system deviations. The efficacy of this work is then evaluated with four case studies motivated from real-world systems with promising results.

Keywords: Resilience · Software Architecture · MaxSAT · Alloy$^{\text{Max}}$

1 Introduction

A software system consists of a set of interacting components that share data with each other in order to satisfy various system requirements. During the lifetime of the system, one or more of these components may become unavailable and fail to fulfill its intended functionality due to abnormal events such as a hardware failure, software bugs, or a security attack. Ideally, a system that is *resilient* is capable of preserving the critical data services, even if non-critical services fail. For example, consider a Magnetic Resonance Imaging (MRI) system that is exposed to a ransomware attack: a resilient design would ensure that the system provides the most essential patient services (e.g., integrity and security of patient information), even if other services (e.g., patient billing) become temporarily unavailable or degraded. In a brittle design, a single component unable to transmit data might undermine all critical properties.

In this work, we propose a formal framework to support modeling and analysis tasks for the systematic, rigorous design of resilient software systems. The

proposed framework consists of (1) a formal definition of *resilience* that characterizes the ability of a system to tolerate process *deviations* – changes to the data sent and received by a process – while ensuring satisfaction of critical properties, and (2) an automated model-based analysis to evaluate the resilience of a given system. To be more specific, in our approach, a system is modeled as a set of processes (each corresponding to a software or hardware component) that interact with each other by transmitting various types of data. A process is considered *unavailable* when it can no longer send or receive any transmission.

Building on this definition, we propose an analysis method that takes (1) a formal model of the system (including its processes and associated dataflow connections) and (2) a desired property to be satisfied, and automatically evaluates the resilience of the system with respect to that property. In particular, we show how this analysis can be formulated as an instance of the *maximum satisfiability* (MaxSAT) problem [10], where evaluating resilience amounts to computing the *property core*—a minimal set of processes that are sufficient to imply a property—and verifying that the system properties are satisfied.

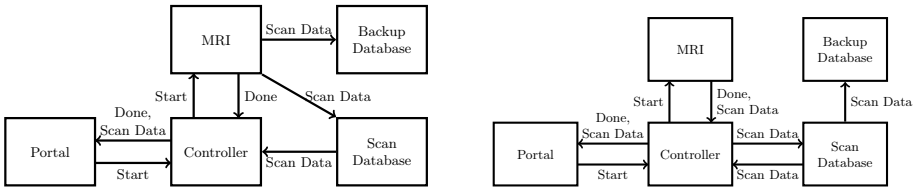
We also describe how this basic analysis for computing resilience can be further utilized for rigorously answering various design questions—for instance, what are the components that are the weakest links in the system, in that their deviation could undermine a critical property? Given two alternative architectural designs of a system (both satisfying a common property under no deviations), which one of them is more resilient under possible deviations?

To demonstrate the applicability of the proposed resilience definition and analysis method, we present a set of realistic case studies from multiple safety-critical domains: (1) a smart hospital bed, (2) a distributed electric vehicle charging network, (3) a smart electric grid, and (4) a smart factory system. The outcome of the case studies suggest that our notion can capture resilience of system designs across multiple application domains, and that our MaxSAT-based analysis method scales to models of complex software systems.

The contributions of the paper are as follows: (1) a formal definition of resilience for software systems that characterizes the ability of a system to tolerate component deviations while ensuring a critical property (Section 3); (2) an automated, model-based method for computing the resilience of a system by formulation as a MaxSAT problem (Section 4); and (3) case studies demonstrating the applicability and utility of the proposed definition and analysis method (Section 5).

1.1 Running Example

To motivate our approach, we present the following example system with two alternative designs. A hospital needs to perform MRI scans to diagnose ailments in patients and observe changes in conditions over time. A medical professional uses a third party's `Portal` as an interface to send a command to an `MRI` to `Start` performing a scan on a patient. A `Controller` handles the request, and forwards it to the `MRI`. Upon completion of the scan, the `MRI` sends a notification back to the `Controller` that the scan is `Done`, which is forwarded to the



(a) $\texttt{MCS}_\texttt{A}$ design, where the `MRI` uploads `Scan Data` directly to both of the databases.

(b) $\texttt{MCS}_\texttt{B}$ design, where the `MRI` sends the `Scan Data` to the `Controller` which uploads to the `Main Database`.

Fig. 1: Two alternative dataflow diagrams for an MRI Control System (`MCS`).

`Portal`. Medical professionals can then use the `Portal` to obtain the `Scan Data` via the `Controller` which obtains the data from the `Scan Database`. For $\texttt{MCS}_\texttt{A}$ (Figure 1a), the `MRI` uploads the `Scan Data` directly to both the `Scan Database` and the `Backup Database`, whereas $\texttt{MCS}_\texttt{B}$ (Figure 1b) uploads the `Scan Data` to the `Controller` where it is forwarded to the `Scan Database`, and then forwarded again to the `Backup Database`.

The hospital requires that the `MCS` is resilient against data loss. If a process becomes unavailable (due to a software bug or a ransomware attack), the hospital requires that `Scan Data` is still securely stored in at least one `Database`. In $\texttt{MCS}_\texttt{A}$, not all processes are used to store the `Scan Data`; as long as the `MRI` and one of the two `Database`s are available, the `Scan Data` will be stored. In $\texttt{MCS}_\texttt{B}$, the `MRI`, `Controller`, and `Scan Database` are all needed to store the `Scan Data` to at least one `Database`.

In the event that the `Scan Database` is unavailable, $\texttt{MCS}_\texttt{A}$ will still be able to save scans to the `Backup Database`; however, in $\texttt{MCS}_\texttt{B}$, the scans will not be saved at all. Even though both `MCS`s are designed to have backups of `Scan Data`, $\texttt{MCS}_\texttt{A}$ is more *resilient* than $\texttt{MCS}_\texttt{B}$ with respect to the storage of all `Scan Data`. By identifying the minimal set(s) of processes to save the `Scan Data`, we can evaluate the resilience of the `MCS` to maximal component deviation.

2 Background

In our approach, systems are modeled as *dataflow diagrams*, as we are interested in reasoning about properties that describe how different pieces of data may flow throughout a system.

Definition 1 (Dataflow Diagram). Derived from the definition by Tao and Kung [21], a *dataflow diagram* (DFD) is a quintuple $A = (C, D, O, Q, K)$ where

- $C = P \cup S \cup E$ is a non-empty, finite set of components consisting of disjoint sets P, the set of processes; S, the set of data stores; and E, the set of external entities.
- D is a non-empty, finite set of data that flows between components.

- $O \subseteq C \times D$ is the relation that maps each component to the set of data it produces.
- $Q \subseteq C \times C \times D$ is the set of dataflows from a source component to a destination component.
- $K \subseteq P \cup S$ is the set of internal processes, and $C \setminus K$ are external components.

For the remainder of the paper, we refer to the components of a dataflow diagram A as C_A, D_A, O_A, Q_A, and K_A. The *size* of a diagram is the number of components; i.e., $size(A) \equiv |C_A|$.

Example 1 (DFD for $\texttt{MCS}_\texttt{A}$). The design $\texttt{MCS}_\texttt{A}$ from Section 1.1 contains the following set of processes and data:

$$\texttt{C}_{\texttt{MCS}_\texttt{A}} = \{\texttt{Portal}, \texttt{MRI}, \texttt{Controller}, \texttt{Database}, \texttt{BackupDatabase}\}$$
$$\texttt{D}_{\texttt{MCS}_\texttt{A}} = \{\texttt{Start}, \texttt{Done}, \texttt{Scan Data}\}$$

The size of $\texttt{MCS}_\texttt{A}$ is 5, and the only external component is the `Portal` (i.e., $\texttt{K}_{\texttt{MCS}_\texttt{A}} = \texttt{C}_{\texttt{MCS}_\texttt{A}} \setminus \{\texttt{Portal}\}$). The `MRI` produces the `Scan Data` and `Done`, so $\{(\texttt{MRI}, \texttt{Scan Data}), (\texttt{MRI}, \texttt{Done})\} \subseteq \texttt{O}_{\texttt{MCS}_\texttt{A}}$, but the `Controller` only forwards data, so there is no data `d` such that $(\texttt{Controller}, \texttt{d})$ is in $\texttt{O}_{\texttt{MCS}_\texttt{A}}$. The dataflow connections for the portal are $(\texttt{Controller}, \texttt{Portal}, \{\texttt{Done}, \texttt{Scan Data}\})$ and $(\texttt{Portal}, \texttt{Controller}, \{\texttt{Start}\})$, both in $\texttt{Q}_{\texttt{MCS}_\texttt{A}}$. For brevity, we do not detail all tuples in $\texttt{O}_{\texttt{MCS}_\texttt{A}}$ and $\texttt{Q}_{\texttt{MCS}_\texttt{A}}$.

Definition 2 (Process Deviation). For a pair of DFDs A, A', and a process p such that $p \in C_A$ and $p \in C_{A'}$, p *deviates* between A and A' when there is a process p' and data d such that (1) $(p, p', d) \in Q_A \iff (p, p', d) \notin Q_{A'}$, (2) $(p', p, d) \in Q_A \iff (p', p, d) \notin Q_{A'}$, or (3) $(p, d) \in O_A \iff (p, d) \notin O_{A'}$.

Definition 3 (Typed Transitive Set of a DFD). The *Typed Transitive Set* (TTS) over a relation $R : C \times C \times D$ is the set of processes p, p' and data element d such that p sends d to another process, which then sends it to p'. Formally, $TTS(R : C \times C \times D) \equiv \{(p, p', d) \in C \times C \times D \mid \exists p'' \in C \mid (p, p'', d) \in R \wedge (p'', p', d) \in R\}$.

Definition 4 (Typed Transitive Closure of a DFD). For a given DFD A, the *Typed Transitive Closure* (denoted $\tilde{Q}_A$) is the closure of the transitive set that includes Q_A. Given that $Q_{A_0} \equiv Q_A$, and $Q_{A_{n+1}} \equiv TTS(Q_{A_n})$, we define $\tilde{Q}_A \equiv \bigcup_{n=0}^{\infty} Q_{A_n}$.

Example 2 (Typed Transitive Closure for $\texttt{MCS}_\texttt{A}$). In $\texttt{MCS}_\texttt{A}$, the `Scan Data` reaches the `Controller` from the `MRI` by being sent to the `Scan Database`, which sends it to the `Controller`. Therefore, $(\texttt{MRI}, \texttt{Controller}, \texttt{Scan Data}) \in \tilde{\texttt{Q}}_{\texttt{MCS}_\texttt{A}}$.

Definition 5 (Reachability Predicate). For a pair of processes p, p', data elements of type d reaches p' from p if there is a path of processes from p to p' that all send d. A data always reaches the process that sends it, so if p produces d, d reaches p. Formally, a DFD A satisfies *canReach* for any p, p', d by the following: $canReach(p, p', d) \equiv (p, p', d) \in \tilde{Q}_A \vee (p = p' \wedge (p, d) \in O_A)$.

Definition 6 (Dataflow Property). A *dataflow property* φ over a DFD A is a first-order logic formula that utilizes the predicate *canReach* to express the presence of desired dataflow in a system. In particular, we write $A \models \varphi$ iff the evaluation of φ over the set of tuples in *canReach* is true.

Example 3 (MCS Properties). There are two important MCS properties: (1) $\varphi_{\texttt{op}}$: Medical professionals can access the MRI scans through the portal and (2) $\varphi_{\texttt{db}}$: At least one database always receives a scan in order to maintain data integrity.

$$\varphi_{\texttt{op}} = \texttt{canReach}(\texttt{MRI}, \texttt{Portal}, \texttt{Scan Data})$$
$$\varphi_{\texttt{db}} = \exists \texttt{db} : \texttt{Database} \mid \texttt{canReach}(\texttt{MRI}, \texttt{db}, \texttt{Scan Data})$$

Threat Model We assume an adversary attacking a system, represented as DFD A, can cause a set of processes, denoted $\hat{C}_A \subseteq C_A$, to deviate such that for all processes $p \in \hat{C}_A$, there is no data sent or received by p.

3 Resilience of Systems

A resilient system is capable of delivering critical services even when some system processes have deviated. Often, these critical services are dependent on the data sent to and received from other processes; in this section we detail how the resilience of these services can be formally defined with dataflow properties.

3.1 Property Core

When a process p in a diagram A deviates and no longer sends or receives any data, processes that interact with p will deviate as they will not transmit any data with p. If a set of processes ceases data transmission, the remaining processes should operate with minimal deviations; these remaining processes form a *subdiagram* of A.

Definition 7 (Subdiagram). A *subdiagram*, A', contains a subset of the processes of a diagram A such that each process deviates minimally. In the subdiagram, the processes, data, and dataflows of A' are a subset of the ones present in A. If a dataflow relation (p, p', d) is in Q_A and both p and p' are processes in the subdiagram, then (p, p', d) is also in $Q_{A'}$. For any process in the diagram, the process is internal in the subdiagram iff the process is internal in the diagram. Similarly, for any process p and data d in the subdiagram, p produces d in the subdiagram iff p produces d in the diagram. For a diagram $A = (C, D, O, Q, K)$, a diagram $A' = (C', D', O', Q', K')$ is a subdiagram (denoted $A' \subseteq A$) such that:

$$C' \subseteq C \quad \forall p, p' \in C, \forall d \in D \mid (\{p, p'\} \subseteq C' \land (p, p', d) \in Q) \implies (p, p', d) \in Q'$$
$$D' \subseteq D \quad \forall p \in C' \mid p \in K \iff p \in K'$$
$$Q' \subseteq Q \quad \forall p \in C', d \in D' \mid (p, d) \in O \iff (p, d) \in O'$$

Example 4 (Subdiagram of $\texttt{MCS}_\texttt{A}$). During an event when an adversary attacks the Portal and Controller in the MCS, neither process transmits or receives data, and $\hat{\texttt{C}}_{\texttt{MCS}_\texttt{A}} = \{\texttt{Portal}, \texttt{Controller}\}$. The processes MRI, Scan Database, and Backup Database then form a subdiagram of the MCS when the only deviations of the processes are to cease communication with $\hat{\texttt{C}}_{\texttt{MCS}_\texttt{A}}$.

Definition 8 (All Subdiagrams of a DFD). The set of all subdiagrams of a DFD A is denoted as SD_A; trivially, $A \in SD_A$. Formally, $SD_A \equiv \{A' | A' \subseteq A\}$.

We note that the subdiagram in Example 4 does not require all of the processes in order to satisfy $\varphi_{\texttt{db}}$ from Example 3. The `Backup Database` can be removed, and $\varphi_{\texttt{db}}$ will still be satisfied. To identify the maximum unavailable processes a system can withstand, we compute the minimal subdiagrams with respect to the number of processes; these minimal subdiagrams are denoted as *property cores*.

Definition 9 (Property Core). For a given dataflow property φ, a *property core* of the diagram A is a subdiagram A' with a minimal number of processes that satisfies φ; if any process in A' can be removed and still satisfy φ, it is not a property core. We define the predicate $core(A, A', \varphi)$ to be true iff A' is a property core of A w.r.t. φ. Formally, $core(A, A', \varphi) \equiv (A' \in SD_A \wedge A' \models \varphi \wedge \forall A'' \in SD_{A'} \setminus \{A'\} \mid A'' \not\models \varphi)$. If $A \not\models \varphi$, then there does not exist a subdiagram that is a property core of A.

Definition 10 (All Property Cores of a DFD). For a given DFD A and dataflow property φ, the set of all property cores A that satisfies φ is denoted PC_A^{φ}; formally, $PC_A^{\varphi} \equiv \{A' \in SD_A \mid core(A, A', \varphi)\}$.

Because there may exist multiple dataflow paths in a DFD that independently satisfy the property φ, it is possible to have multiple property cores of a DFD w.r.t. φ. These cores may overlap and share processes, but cannot be a subdiagram of another core by virtue of the minimality requirement.

Example 5 (`MCS` property cores). $\texttt{PC}_{\texttt{MCS}_\texttt{A}}^{\varphi_{\texttt{db}}}$ contains two cores: the subdiagram with the `MRI` and `Scan Database`, and the subdiagram with the `MRI` and `Backup Database`. The `MRI` can send `Scan Data` to the `Backup Database` even when the `Scan Database`, `Controller`, and `Portal` processes are unavailable.

In order to send `Scan Data` to the `Backup Database`, $\texttt{MCS}_\texttt{B}$ must first send the `Scan Data` to the `Scan Database`; this means the only core in $\texttt{PC}_{\texttt{MCS}_\texttt{B}}^{\varphi_{\texttt{db}}}$ is the core containing the `MRI`, `Controller`, and `Scan Database`.

$$\begin{aligned}\texttt{PC}_{\texttt{MCS}_\texttt{A}}^{\varphi_{\texttt{db}}} &= \{\texttt{X} \in \texttt{SD}_{\texttt{MCS}_\texttt{A}} \mid (\texttt{C}_\texttt{X} = \{\texttt{MRI}, \texttt{Scan Database}\} \\ &\qquad\qquad \vee \texttt{C}_\texttt{X} = \{\texttt{MRI}, \texttt{Backup Database}\})\} \\ \texttt{PC}_{\texttt{MCS}_\texttt{B}}^{\varphi_{\texttt{db}}} &= \{\texttt{X} \in \texttt{SD}_{\texttt{MCS}_\texttt{B}} \mid \texttt{C}_\texttt{X} = \{\texttt{MRI}, \texttt{Scan Database}, \texttt{Controller}\}\}\end{aligned}$$

If a security attack targets the `Scan Database`, $\texttt{MCS}_\texttt{A}$ can store the `Scan Data` in the `Backup Database` and satisfy $\varphi_{\texttt{db}}$. However, without the `Scan Database`, $\texttt{MCS}_\texttt{B}$ cannot store `Scan Data` in either `Database`, leaving the system unable to satisfy $\varphi_{\texttt{db}}$.

3.2 Evaluation of Resilience Criteria with Property Cores

Because $\texttt{MCS}_\texttt{A}$ has multiple property cores for $\varphi_{\texttt{db}}$, there are redundancies in the system. Conversely, $\texttt{MCS}_\texttt{B}$ has a single core, with limited redundancies – if the `MRI`, `Scan Database`, or `Controller` are unavailable, the `Backup Database` may

be unable to store `Scan Data` and limit its redundant capabilities. We can then say that $\texttt{MCS}_\texttt{A}$ is more *resilient* than $\texttt{MCS}_\texttt{B}$.

Safety-critical systems often have *resilience criteria* to ensure the systems can withstand deviations of system processes and the system's environment.

Definition 11 (Resilience Criterion). A *resilience criterion* ψ is a formula over the cores of one or more dataflow properties of a DFD A. For a resilience criterion ψ, the system modeled as diagram A is resilient iff $A \models \psi$.

We propose that property cores can be utilized to define resilience criteria; different attributes of property cores can express different constraints on the required dataflow, even as the processes in the system are subjected to major deviations. A subset of these attributes and how they can express resilience are presented below.

Size of Cores The size of a property core reflects the number of processes required to transmit the associated data; if the size of a core is equal to the size of the diagram, then every process in the diagram is required to satisfy the given property. Should any process become unavailable, the property can no longer be satisfied. A resilient system allows for some fraction of processes to be unavailable while satisfying the property.

Example 6 (Criterion 1 for `MCS`). A desirable resilience criterion states that not every process is required for $\varphi_{\texttt{op}}$ to be satisfied. Formally, $\psi_{\texttt{op}} \equiv \exists \texttt{X} \in \texttt{PC}_{\texttt{MCS}}^{\varphi_{\texttt{op}}} \mid \texttt{size}(\texttt{X}) < \texttt{size}(\texttt{MCS})$. For both $\texttt{MCS}_\texttt{A}$ and $\texttt{MCS}_\texttt{B}$, the `Backup Database` is not required to be available to send `Scan Data` from the `MRI` to the `Portal`, and thus $\texttt{MCS}_\texttt{A} \models \psi_{op}$ and $\texttt{MCS}_\texttt{B} \models \psi_{op}$.

Number of Cores Having multiple cores in a diagram indicates a level of redundancy within the associated system. If one or more processes in one core becomes unavailable, the system can still satisfy the dataflow property by relying on processes that belong to another core. Having more cores in a system allows for more independent dataflow paths that satisfy a desired property.

Example 7 (Criterion 2 for `MCS`). To ensure data integrity, it is desirable that there exist alternate means to save `Scan Data`; therefore, a resilience criteria states that there are at least two property cores w.r.t. $\varphi_{\texttt{db}}$. Formally, $\psi_{\texttt{db}} \equiv |\texttt{PC}_{\texttt{MCS}}^{\varphi_{\texttt{db}}}| \geq 2$. Using the cores computed in Example 5, we find $\texttt{PC}_{\texttt{MCS}_\texttt{A}}^{\varphi_{\texttt{db}}} \models \psi_{\texttt{db}}$ and $\texttt{PC}_{\texttt{MCS}_\texttt{B}}^{\varphi_{\texttt{db}}} \not\models \psi_{\texttt{db}}$.

Intersection of Cores A diagram may not be considered resilient solely because it has multiple cores. For a diagram A and property φ, if there exists a process p such that p is in every core of PC_A^φ, then p is a weak link in the system. If there is an event where p is unavailable, then the system is unable to satisfy property φ. Conversely, for a different system A', if there exist two cores, c, c' in $PC_{A'}^\varphi$ such that $C_c \cap C_{c'} = \emptyset$, then A' may be considered more resilient than A. A diagram where cores share few processes is considered more resilient than a diagram where all cores share many processes.

Example 8 (Criterion 3 for MCS). A desirable resilience criterion states that there are two independent dataflow paths to save Scan Data to the databases. This requires that there exists two cores in $\mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}}$ where the intersection of the processes contain at most the MRI. Formally, $\psi_{\mathtt{ind}} \equiv \exists \mathtt{X}, \mathtt{Y} \in \mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}} \mid \mathtt{X} \neq \mathtt{Y} \wedge (\mathtt{C}_{\mathtt{X}} \cap \mathtt{C}_{\mathtt{Y}}) \setminus \{\mathtt{MRI}\} = \emptyset$. Since $\mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}_{\mathtt{A}}}$ has no overlap between processes besides the MRI, and there is only one core for $\mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}_{\mathtt{B}}}$, $\mathtt{MCS}_{\mathtt{A}} \models \psi_{\mathtt{ind}}$ and $\mathtt{MCS}_{\mathtt{B}} \not\models \psi_{\mathtt{ind}}$.

Boundary of Cores In some domains, a system may only be considered resilient if it is not dependent on external systems or third party entities. If at least one core of a diagram does not rely on external processes, then satisfaction of the associated property is dependent only on the availability of the internal processes.

Example 9 (Criterion 4 for MCS). Saving the Scan Data should only require local processes to ensure data integrity: $\psi_{\mathtt{loc}} \equiv \exists \mathtt{X} \in \mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}} \mid \mathtt{X} \subseteq \mathtt{K}_{\mathtt{MCS}}$. Neither MCS requires the Portal to save Scan Data, so $\mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}_{\mathtt{A}}} \models \psi_{\mathtt{loc}}$ and $\mathtt{PC}^{\varphi_{\mathtt{db}}}_{\mathtt{MCS}_{\mathtt{B}}} \models \psi_{\mathtt{loc}}$.

4 Property Core Computation as MaxSAT

We formulate the problem of computing a property core as a MaxSAT problem. Definitions 1–7 are expressed using relations and first-order logic; additionally, the notion of a property core can be modeled via minimizing the set of processes in the property core while satisfying a given dataflow property. Any solver capable of translating a problem to a MaxSAT problem can be utilized to compute property cores; we select Alloy$^{\mathrm{Max}}$ [26].

Alloy [8] is a formal modeling language based on first-order predicate logic and relational algebra that performs constraint solving over bounded domains. Alloy$^{\mathrm{Max}}$ is an extension of Alloy that can express optimization problems to a MaxSAT problem [26]. The logic of Alloy$^{\mathrm{Max}}$ is expressive enough to specify DFDs, dataflow properties, and resilience criteria, and the MaxSAT-based analysis can be used to compute property cores. Because the systems modeled have finite processes and data, Alloy solving over bounded domains does not affect the soundness or completeness of the results.

Abstract DFDs Encoding The encoding of an **abstract** DFD (Definition 1) in Alloy is shown in Listing 1. The data of a DFD are represented by the Data **sig**nature (line 2), and the processes, data stores, and external entities are represented as a Process with a relation, owns, to describe the set of data it produces (line 3). A DFD (line 4) contains relations to represent the different components of the quintuple, with the **set** of processes, the subset of internal processes, and data (line 5); the dataflow connections are represented as a **set** of relations and $\tilde{Q}$ is modeled as a relation of the DFD (line 6). We impose additional constraints to disallow unowned data (lines 8–9), unused processes (lines 10–12), unused data (lines 13–14), or two identical DFDs of the same name (lines 15–17).

Listing 1: Abstract DFD and Constraints in Alloy$^{\mathrm{Max}}$

```
1  // a 'sig' nature defines a type of atom in Alloy
2  abstract sig Data {} // datatype
3  abstract sig Process {owns: set Data} // process
```

```
4  abstract sig DFD { // DFD, with field/relations: processes,
5    C : set Process, K : set C, D : set Data, // internal and datatypes.
6    Q : set C -> C -> D, QTilde : set C -> C -> D } // dataflow & TTC
7  // 'pred'icates allow for boolean evaluations of given sigs
8  pred DataOwned[ dfd : DFD ] { // all data is owned
9    }//all dat : Data | some proc : Process | dat in proc.owns }
10 pred Connected[ dfd : DFD ] { // processes have some connection
11   all c : dfd.C | some c2 : Process, d : Data |
12   (c -> c2 -> d in dfd.Q) or (c2 -> c -> d in dfd.Q)}
13 pred IncludedDatatypes[ dfd : DFD ] { // dfd datatypes are used
14    all d : dfd.D | some c1, c2 : Process | (c1 -> c2 -> d) in dfd.Q}
15 pred Unique [ dfd : DFD ] { // each DFD is unique
16   all other : DFD | (other.C = dfd.C
17   and other.D = dfd.D and other.Q = dfd.Q) implies (other=dfd) }
18 pred WellFormed[ dfd : DFD ] { // well formed is connected w/ uniqueness
19   DataOwned[dfd] and Connected[ dfd ] and // and owned data
20   IncludedDatatypes[ dfd ] and Unique[ dfd ] }
```

Typed Transitive Closure Constraints To express the dataflow reachability, we encode constraints for the Typed Transitive Closure (Definition 4). In Listing 2, we use relational transitive closure to define $\tilde{Q}$ as it is native to Alloy. On lines 28–30, we use an Alloy **fun**ction to compute and return the set of process tuples (p_1, p_2) such that for a given data d, the tuple (p_1, p_2, d) is a dataflow in the given DFD. This function is used on lines 25–27 to compute the reflexive transitive closure for the given data and DFD. The $\tilde{Q}$ of a DFD is then defined on lines 21–24 by stating that for any data d and processes p_1 and p_2, the tuple (p_1, p_2, d) is in $\tilde{Q}$ iff (p_1, p_2) is in the reflexive transitive closure for d.

Listing 2: Constraints for Typed Transitive Closure of a DFD

```
21 // QTilde is correctly formatted
22 pred QTildeValid[ dfd : DFD ] {// all p1->p2 is in typed
23   all p1, p2 : DFD.C, d : DFD.D | // transitive closure of d
24     p1 -> p2 -> d in dfd.QTilde iff p1 -> p2 in TTS[d, dfd]}
25 // reflexive transitive set for the given datatype
26 fun TTS[ dat : Data, dfd : DFD ] : set(Process -> Process) {{
27   let qdat = QForDat[ dat, dfd ] {qdat.*qdat} }}
28 // processes in the dataflow for the given datatype
29 fun QForDat[ dat : Data, dfd : DFD ] : set(Process -> Process) {
30   {p1: Process, p2 : Process | (p1 -> p2 -> dat) in dfd.Q }}
31 // alloy 'fact's hold true for every instance of an alloy model
32 fact WellFormedDFD { // validate the QTilde
33   all a : DFD | WellFormed[a] and QTildeValid[a]}
```

Subdiagram Comparison In Listing 3, two diagrams are compared to check whether the first is a subdiagram of the latter (Definition 7). On line 35, for subdiagram A_1 and diagram A_2, we evaluate if $C_{A_1} \subseteq C_{A_2}$, $D_{A_1} \subseteq D_{A_2}$, and $Q_{A_1} \subseteq Q_{A_2}$. To ensure that dataflow between processes in the subdiagram are preserved, lines 36–38 check that for all pairs of processes (p, p') in the subdiagram, if a dataflow (p, p', d) is in Q_{A_2}, then (p, p', d) is in Q_{A_1} as well. Line 39 constrains any external component in A_2 to be an external component in A_1 as well. Since data production is defined as a relation of processes via owns in Listing 1, we do not constrain it here.

Listing 3: Predicate to Compare Subdiagram and Diagram

```
34 pred Subdfd[A1, A2 : DFD] { // A1 is subdiagram of A2
35   A1.C in A2.C and A1.D in A2.D and A1.Q in A2.Q
36   all phi, psi : Process, q : Data |
```

```
37    ( (phi -> psi -> q in A2.Q) and (phi in A1.C) and (psi in A1.C) )
38    implies (phi -> psi -> q in A1.Q)
39    all lambda : A1.C | lambda in A2.K implies lambda in A1.K}
```

Dataflow Property Definition Dataflow properties (Definition 6) are first-order logic sentences that utilize the predicate *canReach*. Since Alloy is based in first-order logic, we define the *canReach* predicate in Listing 4 to be used for property definitions (lines 40–41).

Listing 4: Predicate for Dataflow Properties

```
40  pred canReach[ c1, c2 : Process, d : Data, dfd : DFD ] { //canReach pred
41    (c1 -> c2 -> d) in dfd.QTilde or ((c1=c2) and d in c1.owns) }
```

Property Core Computation To compute one property core of a DFD, in Listing 5, we identify a singleton DFD Core via the keyword **one** (line 42). On line 43, we use the **minsome** keyword to minimize the number of processes that are in the core. The remainder of the constraints to compute a property core are modeled with a concrete DFD model.

Listing 5: Constraints for Computation of Property Core

```
42  one sig Core extends DFD {} // One sig to describe generated sub DFD
43  fact{ minsome Core.C } // minimize components of Core
```

Concrete DFD Encoding Listings 1–5 provide the constraints for a generic DFD where the processes, data, and dataflow connections are constrained, but not defined. For a concrete DFD such as the `MCS`, the specific processes, data, and dataflow connections must be modeled explicitly. We refer to Listing 6 for the computation of property cores for a concrete DFD, utilizing $\texttt{MCS}_\texttt{A}$ as an example.

On line 1, the definitions from Listings 1–5 are included. Lines 2–3 are used to define the `Scan Database` and `Backup Database` as a type of Database, and constrain both to own no data (in Alloy, the second pair of braces that follow a signature definition can be used to define *signature constraints*, which apply to every element of that signature type). The other processes are modeled on lines 4–6. The data types in `MCS` are defined on line 7, and the DFD for $\texttt{MCS}_\texttt{A}$ is defined on line 8; we ensure that the property core is a subdiagram of $\texttt{MCS}_\texttt{A}$ on line 9. Since the only external process is the `Portal`, line 10 ensures that the internal processes exclude only the `Portal`. The dataflow between processes is modeled on lines 11–15 and the predicates from Example 3 are modeled on lines 16–19.

The constraints for the property cores are encoded on lines 20–21, and the core(s) of the DFDs are computed by executing the **run** command for Run_OP or Rub_DB. This execution uses the backend MaxSAT solver to compute a satisfying instance of the encoded constraints; the result can then be viewed via Alloy$^{\text{Max}}$'s built-in visualizer and evaluator.

Listing 6: $\texttt{MCS}_\texttt{A}$ DFD in Alloy$^{\text{Max}}$

```
1  open DFDs // Include DFD definitions
2  abstract sig Database extends Process {} // abstract database
3  one sig Main, Backup extends Database {}{no owns} // MCS databases
4  one sig MRI extends Process {}{owns = ScanData+Done} // MRI process
5  one sig Controller extends Process {}{no owns} // Controller process
```

```
one sig Portal extends Process {}{owns=Start} // Portal process
one sig Start, Done, ScanData extends Data {} // datatypes
one sig MCS_A extends DFD {} // MCS A DFD definition
fact { Subdfd[Core, MCS_A] } // core is a subdfd of the MCS
fact MCS_A_Internal { MCS_A.K = MCS_A.C - Portal } // internal
fact MCS_A_Dataflow { // MCS Dataflow connections
  MCS_A.Q = MRI -> Controller -> Done + MRI -> Main -> ScanData +
  MRI -> Backup -> ScanData + Controller -> MRI -> Done +
  Main -> Controller -> ScanData + Portal -> Controller -> Start +
  Controller -> Portal -> {Done + ScanData} }
// property op: data from MRI to portal
pred phi_op[ dfd : DFD ] { canReach[ MRI, Portal, ScanData, dfd ] }
pred phi_backup[ dfd : DFD ] { // data reaches a database
  some db : Database | canReach[ MRI, db, ScanData, dfd ]}
run Run_OP { phi_op[ MCS_A ] and phi_op[ Core ] } // OP cores
run Run_DB { phi_backup[ MCS_A ] and phi_backup[ Core ] } // db cores
```

5 Case Study Analysis

To investigate the utility of property cores in evaluation of resilience criteria, we provide four case studies motivated by real-world systems, as well as the `MCS` example. Each case study system is modeled as a concrete DFD in Alloy$^{\text{Max}}$, and analyzed with the integrated OpenWBO [11] SAT solver. The case studies include an electric vehicle charging station [6], a smart hospital bed [18], a smart factory [9], and a smart electric grid [7,20]; these were chosen by examining publications from the last decade that describe realistic software architectures. Each software architecture is modeled with a Component and Connector (C&C) architecture view; this depicts the software processes, hardware components, external interfaces, dataflow connections, and connector types of a software system [4]. Each system had informal resilience criteria described in their corresponding publications that were formalized before modeling each system in Alloy$^{\text{Max}}$.

5.1 Electric Vehicle Charging Stations

Electric vehicle charging stations (EVCS) use complex, interconnected systems that require resilient design principles to ensure scalability, availability, and security. As electric vehicles continue to grow in popularity, ensuring the safety of the physical infrastructure as well as the software ecosystem—payment processing, energy management, power source integration—becomes paramount for the EVCS infrastructure. Dini et. al. propose an architecture for the EVCS [6], designed to provide charging capabilities at all times, and be resilient against network connectivity issues. If a station is in a remote area, it can still charge vehicles even when connection to processes in the cloud is precarious.

Dataflow Diagram As depicted in Figure 2, an `EVCS` is composed of a set of global network processes, local processes, external components, and vehicle processes. For brevity, we do not explicitly differentiate between processes and datastores as it has no impact on this model. The vehicle processes are the `battery management system` (`BMS`), which regulates a vehicle's battery, and the manager of the `EV Supply Equipment` (`EVSE`), which receives charging instructions and provides the charging station with vehicle information.

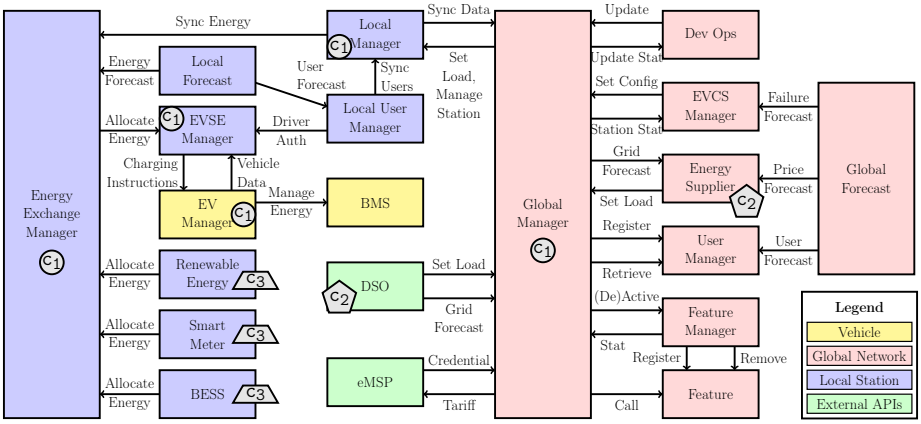


Fig. 2: Dataflow diagram for an Electric Vehicle Charging Station. There can exist multiple local stations that connect to the global network, but only one is shown here for brevity. Similarly, each local station can have multiple electric vehicles, but only one is shown.

A station controls the energy flow to one or more vehicles and charges customers for energy consumption. The `energy exchange manager` (`EEM`) receives information about available energy from `renewable energy`, the `smart meter` connected to an electric grid, and the `Battery Energy Storage System` (`BESS`). The `EEM` additionally receives the energy forecast for the area, and syncs current energy usage with the `local manager` to provide the `EVSE` with the allocated energy. The `EVSE` receives authorization to charge a customer based on the `local user manager`'s authentication after syncing user information with the `local manager`. The `local manager` of the stations syncs data with the `global manager` to provide updates required for operation.

The global network is responsible for tracking all local stations and managing grid loads. The `global forecast` process provides future forecasts for energy prices, grid failures, and user frequency. The `feature manager` registers and removes features at the request of the `global manager`. The `user manager` keeps track of the current and active users, and the `energy supplier` sets the grid load for all local stations. The `EVCS manager` configures each local station for locally stored users and protocols. External components include the `Distribution System Operator` (`DSO`) for power management and `eMobility Service Provider` (`eMSP`) for payment processing.

Resilience Evaluation A safety property that the `EVCS` must satisfy states that a vehicle can charge without overloading the electric grid. This requires at least one source of energy being provided to the `EVSE manager` and charging instructions being provided to the `EV manager`. Formally, we state $\varphi_{\texttt{evcs}} \equiv \texttt{gridIntact} \wedge \texttt{canCharge}$ where

$$\texttt{gridIntact} \equiv \exists \texttt{s} \in \{\texttt{DSO}, \texttt{Energy Supplier}\} \mid \texttt{canReach}(\texttt{s}, \texttt{Local Manager}, \texttt{Set Load})$$

$$\begin{aligned}\texttt{canCharge} \equiv \exists \texttt{s} \in \{&\texttt{Renewable Energy},\ \texttt{Smart Meter},\ \texttt{BESS}\} \mid \\ &\texttt{canReach}(\texttt{s}, \texttt{EVSE Manager}, \texttt{Allocate Energy}) \\ \wedge &\texttt{canReach}(\texttt{EVSE Manager}, \texttt{EV Manager}, \\ &\texttt{Charging Instructions})\end{aligned}$$

This architecture was designed to be resilient against loss of internet connection, so a corresponding resilience criterion states that there exists at least one property core that does not include any processes that require an internet connection - i.e., the processes in the global network. $\psi_{\texttt{evcs}} \equiv \exists \texttt{b} \in \texttt{PC}_{\texttt{evcs}}^{\varphi_{\texttt{evcs}}} \mid \forall \texttt{p} \in \texttt{C}_\texttt{b} \mid \texttt{p} \notin \texttt{Global}$ where `Global` returns the set of processes comprising the global network of the `EVCS` (see Figure 2). Utilizing the Alloy$^{\text{Max}}$ solver, we identified six cores:

$$\begin{aligned}\texttt{PC}_{\texttt{EVCS}}^{\varphi_{\texttt{evcs}}} &\equiv \{\texttt{X} \in \texttt{SD}_{\texttt{evcs}} \mid \exists \texttt{a} \in \texttt{c}_2 \mid \exists \texttt{b} \in \texttt{c}_3 \mid \texttt{C}_\texttt{X} = \texttt{c}_1 \cup \{\texttt{a}, \texttt{b}\}\} \\ \texttt{c}_1 &\equiv \{\texttt{Energy Exchange Manager}, \texttt{EV Manager}, \texttt{EVSE Manager}, \\ &\quad\ \texttt{Local Manager}, \texttt{Global Manager}\} \\ \texttt{c}_2 &\equiv \{\texttt{DSO}, \texttt{Energy Supplier}\} \\ \texttt{c}_3 &\equiv \{\texttt{Renewable Energy}, \texttt{Smart Meter}, \texttt{BESS}\}\end{aligned}$$

Informally, every property core contains all processes in $\texttt{c}_1$, as well as one process within $\texttt{c}_2$, and one process within $\texttt{c}_3$. A visual depiction is shown in Figure 2. If we consider the core with the `DSO` and the `BESS`, to satisfy `canCharge`, the `BESS` sends energy allocation data to the `EEM`, which then sends it to the `EVSE Manager`. To satisfy `gridIntact`, the `DSO` sends the load data to the `global manager`, which then sends it to the `local manager`.

However, this set of property cores does not meet the resilience criterion, as there are at least two global processes in each property core (the global manager and either the DSO or the energy supplier): i.e., $\texttt{EVCS} \not\models \psi_{\texttt{evcs}}$. Even though the architecture was originally designed with the goal of achieving resilience against network failures, we can see that it fails to be resilient against certain failures; i.e., those that involve these global processes. Our analysis suggests that the architecture can be further improved to meet the intended resilience criterion.

5.2 Case Studies Results

For brevity, we present the results of the remaining case studies at a high level[3]. Table 1 provides the summary of all of the case study results. We note the large disparity in computation time shown in Table 1, which is due to the large differences in model sizes and complexity.

Smart Bed A smart bed is utilized in hospitals where monitors are attached to a patient, and biomedical data is sent to interfaces for medical professionals to monitor and analyze. The smart bed [18] connects electronic patches and

[3] Artifacts are available at `https://doi.org/10.5281/zenodo.17013599`

						Time (minutes)		
Case Study	LoC	$\#\varphi$	Cores	#RC	Sat	Core comp	RC comp	Total
MCS_A	80	2	1, 2	4	4	< 0.01	< 0.01	0.01
MCS_B	70	2	1, 1	4	2	< 0.01	< 0.01	< 0.01
EVCS	212	1	6	1	0	188.87	4.52	193.39
Smart Bed	144	2	2, 3	2	2	82.41	45.78	128.19
Smart Factory	148	2	6, 6	2	2	11.17	2.97	14.14
Smart Grid	52	1	1	1	0	0.08	0.04	0.012

Table 1: Results of each case study, including the lines of Alloy^{Max} code (LoC), no. of dataflow properties ($\#\varphi$), no. of cores for each property (cores), no. of resilience criteria (#RC) and no. of criteria satisfied (SAT). The computation time (comp) of the Alloy^{Max} models for the cores and RC are shown in minutes.

oximeters to a patient and transmits data via Bluetooth and WiFi to a gateway, where data is converted to an interoperability standard connection. The biodata is then displayed on a GUI where a medical professional monitors the patient.

The smart bed requires that data be available to medical professionals, and the data's integrity be maintained [18].We derive two resilience criteria to analyze for the smart bed: (1) medical professionals are able to view data without requiring internet access, and (2) biodata is saved to several databases even without access to a GUI. After implementing the smart bed architecture in Alloy^{Max}, we find that both criteria are satisfied, and the smart bed architecture in [18] is resilient w.r.t. data availability and integrity.

Smart Factory The Industry 4.0 effort seeks to modernize current factory production to utilize cloud technologies, cyber-physical systems, and data analytic engines [9]. An architecture [9] for one of these modern facilities—a smart factory—is designed to support decision making and execution when some processes, particularly external processes, are offline. This architecture uses a cloud controller to send commands to the cyber-physical system manager, which then executes these decisions with the smart machines. Data from the factory floor is then collected, sent to an analytic engine, and then uploaded to the cloud for the managers of the factory to analyze and decide on new actions.

We modeled the architecture in Alloy^{Max}, and encoded two criteria for analysis: (1) actions are sent to the smart machines from the factory managers without relying on external processes, and (2) data from the factory floor is sent to the analytics engine for performance analysis without relying on external processes. Both of these criteria are satisfied in the architecture, and the design is resilient against failure of external processes.

Smart Grid Modern electric grids are highly interconnected systems that are crucial to modern infrastructure; many current electric grids contain software processes that control voltages and the flow of electricity to maintain safe deployment. The National Institute of Standards and Technology (NIST) has com-

piled criteria [7] for modern smart grids for safe execution during cybersecurity attacks and voltage surges.

We analyzed an architecture for a smart grid connected to a printer and engineering workstation [20]. This architecture was designed to be resilient against process unavailability and cybersecurity attacks. In a smart grid, this requires end components (i.e., the printer and engineering workstation) to receive certain data in order to regulate the electricity used and prevent a blackout or brownout of the grid. A resilient smart grid requires at least two paths to deliver this data to maintain grid integrity [7]. After analyzing the grid in Alloy$^{\text{Max}}$, we found that this is not the case; the electric grid architecture [20] is not resilient against process unavailability.

5.3 Discussion of Case Study Results

For the case study systems, we defined DFDs from C&C architectures and specified the existing resilience criteria using property cores. We formally verified that the Smart Bed and Smart Factory architectures (as modeled) are resilient, and identified resiliency flaws in the EVCS and Smart Grid systems. These results demonstrate that property cores are a promising approach to formalizing safety-critical resilience criteria and verifying the resiliency of a DFD, and suggest that resilience evaluation via property cores may be feasible for practical applications.

6 Related Work

The work by Becker and Voss has evaluated architecture models where communication is lost with processes and require alternate processes to deliver services [2]. This work allows for partial service fulfillment and degraded functionality in order to satisfy a property, whereas our work requires complete service delivery for a property to be satisfied in a subdiagram. Another prior work has used dataflow properties to reconfigure architectures in order to ensure service delivery when components fail [19]. This work identifies all possible configurations of an architecture where at least one dataflow path is present for service fulfillment, whereas our work identifies all subsets of processes that ensure fulfillment in the given configuration.

Tarrach et. al proposes an architecture-level method to identify when service failure occurs when a given threat model is successful in an attack. Their work encodes the dataflow of an architecture using Satisfiability Modulo Theory (SMT) formulas [22] and evaluates the repair of a dataflow connection after corruption by an adversary. Our approach does not attempt to repair a dataflow connection when an attack is successful, but rather evaluates if service delivery can still be satisfied without the connection to the corrupted process. The work by Wagner evaluates the ability of architectures to cease communication with processes at run-time that have been successfully attacked by an adversary, while maintaining safety-critical services [25]. Our work instead identifies the subsets of processes where dataflow cannot be disrupted in order to maintain these services.

There are prior works that formally verify the resilience of a system using Petri Nets [1,12], Behavior Interaction Priority systems [3], and Markov Decision Processes [5]. These works assign each system component a probability of failure and verify that the system satisfies a resilience requirement that states a threshold for the total probability of service failure. Similarly, our work verifies the resilience of a system, but identifies the minimal subsets of processes required to ensure service instead of computing probability of service failure.

Prior works on trusted computing bases identify a minimal set of hardware and software processes that ensure that a given adversary cannot access secure data in a system [15,16,23]. These works identify a singular set of components necessary for secure communication or service delivery, whereas our work identifies all minimal subsets of processes required for service delivery.

Previous efforts in architecture slicing [17,24,27] propose methods to identify processes from each layer of an architecture to ensure delivery of services. A layer of an architecture is a set of processes with related functionality, where data is sent between layers. These approaches in architecture slicing assume the transmission of data between layers, and select processes from each layer to ensure services can be fulfilled; our approach explicitly defines the dataflow connections to identify the processes required for property satisfaction. Topological proofs [13,14] identify the minimal core of a system that is required to satisfy a property. These works evaluate stateful systems with properties reasoning over actions in a system, and identify a minimal core that implies a property is satisfied, unsatisfied, or possibly satisfied. Our work evaluates systems with properties that reason over the dataflow in a system, and does not reason about possible satisfaction of a property.

To the best of our knowledge, this work is the first to formally model and analyze a system to identify the processes needed for dataflow completion against environmental and software deviations.

7 Conclusion

In this work, we have proposed *property cores* as a method for evaluating the resilience of a system. The key idea is to identify *minimal subdiagrams* of a DFD that satisfies a given dataflow property. Our work leverages a MaxSAT solver to compute the minimal process set for property satisfaction, and we have demonstrated how these property cores can be utilized to define and analyze formal resilience criteria. Analyzing these criteria can identify weaknesses in the system architecture and suggest ways to improve its resilience.

Future work includes modeling the propagation of process unavailability throughout a DFD. The data output from a process may depend on the input from another process for computation of data, and it is likely that unavailability of one process may cause other, dependent processes to become unavailable as well. Additional future work will examine automatic reconfiguration of systems to prevent the propagation of failure and ensure service delivery in an event all property cores contain an unavailable process. Lastly, the threat model in this work can be extended to encompass a wider range of attacks and consequences, besides the removal of data connections from an unavailable component.

References

1. Alzahrani, N.A.M., Petriu, D.C.: Modeling fault tolerance tactics with reusable aspects. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures. p. 43–52. QoSA '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737182.2737189
2. Becker, K., Voss, S.: A formal model and analysis of feature degradation in fault-tolerant systems. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems. pp. 139–154. Springer International Publishing, Cham (2016)
3. Ben Hafaiedh, I., Elaoud, A., Maddouri, A.: A formal model-based approach to design failure-aware internet of things architectures. Journal of Reliable Intelligent Environments **10**(4), 413–430 (Dec 2024). https://doi.org/10.1007/s40860-024-00225-z
4. Bertram, V., Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Component and connector views in practice: An experience report. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 167–177 (2017). https://doi.org/10.1109/MODELS.2017.29
5. Chen, X., Zhou, G., Yang, Y., Huang, H.: A newly developed safety-critical computer system for china metro. IEEE Transactions on Intelligent Transportation Systems **14**(2), 709–719 (2013). https://doi.org/10.1109/TITS.2012.2230258
6. Dini, V., Tamburri, D.A., Di Nitto, E.: "Electric Vehicle Fast-Charging Software: Architectural Considerations Towards Trustworthiness". In: Galster, M., Scandurra, P., Mikkonen, T., Oliveira Antonino, P., Nakagawa, E.Y., Navarro, E. (eds.) Software Architecture. pp. 288–304. Springer Nature Switzerland, Cham (2024)
7. Hastings, N., Marron, J., Bartock, M.: Cybersecurity for smart grid systems (2023), https://www.nist.gov/programs-projects/cybersecurity-smart-grid-systems
8. Jackson, D.: Software Abstractions: Logic, Language, & Analysis. MIT Press (2012)
9. Kavakli, E., Buenabad-Chavez, J., Tountopoulos, V., Loucopoulos, P., Sakellariou, R.: Specification of a software architecture for an industry 4.0 environment. In: 2018 Sixth International Conference on Enterprise Systems (ES). pp. 36–43 (2018). https://doi.org/10.1109/ES.2018.00013
10. Krentel, M.W.: The complexity of optimization problems. Journal of Computer and System Sciences **36**(3), 490–509 (1988). https://doi.org/https://doi.org/10.1016/0022-0000(88)90039-6
11. Martins, R., Manquinho, V., Lynce, I.: Open-wbo: A modular maxsat solver,. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 438–445. Springer International Publishing, Cham (2014)
12. Marussy, K., Majzik, I.: Constructing dependability analysis models of reconfigurable production systems. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE). pp. 1158–1163 (2018). https://doi.org/10.1109/COASE.2018.8560551
13. Menghi, C., Rizzi, A.M., Bernasconi, A.: Integrating topological proofs with model checking to instrument iterative design. In: Wehrheim, H., Cabot, J. (eds.) Fundamental Approaches to Software Engineering. pp. 53–74. Springer International Publishing, Cham (2020)
14. Menghi, C., Rizzi, A.M., Bernasconi, A., Spoletini, P.: Torpedo: witnessing model correctness with topological proofs. Formal Aspects of Computing **33**(6), 1039–1066 (Dec 2021). https://doi.org/10.1007/s00165-021-00564-1

15. Mishra, P., Yadav, S.K., Arora, S.: Tcb minimization towards secured and lightweight iot end device architecture using virtualization at fog node. In: 2020 Sixth International Conference on Parallel, Distributed and Grid Computing (PDGC). pp. 16–21 (2020). https://doi.org/10.1109/PDGC50313.2020.9315850
16. Mohanty, S., Ramkumar, M.: Securing file storage in an untrusted server - using a minimal trusted computing base. In: CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science. pp. 460–470. Noordwijkerhout, Netherlands (01 2011)
17. Nouruzi, A., Mokari, N., Azmi, P., Jorswieck, E.A., Erol-Kantarci, M.: Ai-based e2e resilient and proactive resource management in slice-enabled 6g networks. IEEE Transactions on Network Science and Engineering **12**(2), 1311–1328 (2025). https://doi.org/10.1109/TNSE.2025.3528190
18. Nunes, T., Gaspar, L., Faria, J.N., Portugal, D., Lopes, T., Fernandes, P., Tavakoli, M.: Deployment and validation of a smart bed architecture for untethered patients with wireless biomonitoring stickers. Medical & Biological Engineering & Computing **62**(12), 3815–3840 (Dec 2024). https://doi.org/10.1007/s11517-024-03155-3
19. Shelton, C., Koopman, P.: Using architectural properties to model and measure graceful degradation. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. pp. 267–289. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
20. Stouffer, K., Pease, M., CheeYee, T., Zimmerman, T., Pillitteri, V., Lightman, S., Hahn, A., Saravia, S., Sherule, A., Thompson, M.: Guide to operational technology (ot) security. Tech. Rep. NIST SP 800-82 Rev. 3, National Institue of Standards and Technology, Gaithersburg, Maryland, USA (Sep 2023)
21. Tao, Y., Kung, C.: Formal definition and verification of data flow diagrams. Journal of Systems and Software **16**(1), 29–36 (1991). https://doi.org/https://doi.org/10.1016/0164-1212(91)90029-6
22. Tarrach, T., Ebrahimi, M., König, S., Schmittner, C., Bloem, R., Nickovic, D.: Threat repair with optimization modulo theories (2022), https://arxiv.org/abs/2210.03207
23. Thotakura, V., Ramkumar, M.: Minimal trusted computing base for manet nodes. In: 2010 IEEE 6th International Conference on Wireless and Mobile Computing, Networking and Communications. pp. 91–99 (2010). https://doi.org/10.1109/WIMOB.2010.5644867
24. Vatten, T.: Investigating 5g network slicing resilience through survivability modeling. In: 2023 IEEE 9th International Conference on Network Softwarization (NetSoft). pp. 370–373 (2023). https://doi.org/10.1109/NetSoft57336.2023.10175399
25. Wagner, R.R.: Architecture-Based Graceful Degradation for Cybersecurity. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (May 2025)
26. Zhang, C., Wagner, R., Orvalho, P., Garlan, D., Manquinho, V., Martins, R., Kang, E.: Alloymax: bringing maximum satisfaction to relational specifications. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 155–167. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3468264.3468587
27. Zhong, K., Yang, Z., Yu, S., Li, K.: Deep reinforcement learning-based multi-layer cascaded resilient recovery for cyber-physical systems. IEEE Transactions on Services Computing **17**(6), 3330–3344 (2024). https://doi.org/10.1109/TSC.2024.3478729