Automating Pattern Selection for Assurance Case Development for Cyber-Physical Systems

Shreyas Ramakrishna[1(✉)], Hyunjee Jin[2], Abhishek Dubey[1], and Arun Ramamurthy[2]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, USA
{shreyas.ramakrishna,abhishek.dubey}@vanderbilt.edu
[2] Siemens Corporation, Technology, New Jersey, USA
{hyunjee.jin,arun.ramamurthy}@siemens.com

Abstract. Assurance Cases are increasingly being required for regulatory acceptance of Cyber-Physical Systems. However, the ever-increasing complexity of these systems has made the assurance cases development complex, labor-intensive and time-consuming. Assurance case fragments called patterns are used to handle the complexity. The state-of-the-art approach has been to manually select generic patterns from online catalogs, instantiate them with system-specific information, and assemble them into an assurance case. While there has been some work in automating the instantiation and assembly, a less researched area is the automation of the pattern selection process, which takes a considerable amount of the assurance case development time. To close this automation gap, we have developed an automated pattern selection workflow that handles the selection problem as a coverage problem, intending to find the smallest set of patterns that can cover the available system artifacts. For this, we utilize the ontology graphs of the system artifacts and the patterns and perform graph analytics. The selected patterns are fed into an external instantiation function to develop an assurance case. Then, they are evaluated for coverage using two coverage metrics. An illustrative autonomous vehicle example is provided, demonstrating the utility of the proposed workflow in developing an assurance case with reduced efforts and time compared to the manual development alternative.

Keywords: Cyber Physical Systems · Assurance Case · Patterns · GSN · Optimization · Ontology · Graph Isomorphism · Coverage Metrics

1 Introduction

Assurance Cases (ACs) are increasingly being required for regulatory acceptance of Cyber-Physical Systems (CPSs) in several safety-critical applications, such as automotive [19], aviation [8] and medical devices [9]. For example, the development of a safety case (AC with a focus on safety) is a requirement for compliance with the ISO 26262 safety standard in the automotive domain [19]. An AC is a structured argument, supported by evidence, intended to demonstrate that the system satisfies its assurance guarantees under a particular context and under

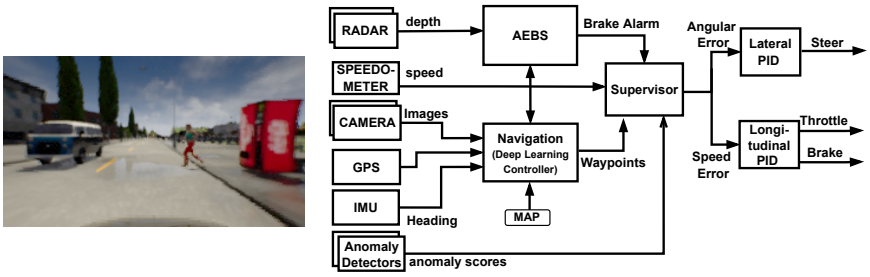

Fig. 1: (left) Image from the forward-looking camera of an autonomous vehicle in CARLA simulation. (right) The system model of the autonomous vehicle.

a given set of assumptions about the behavior of the system's components and its operating environment [1]. Goal Structuring Notation (GSN) [15] has been a widely used graphical modeling language used to represent an AC.

However, the increasing complexity of CPS has made the assurance process complex, labor-intensive, and time-consuming because of the activities that involve managing numerous requirements, curating a large number of artifacts and evidence, developing and managing huge ACs, among others [18]. These problems can be alleviated with an adequate tool that can partially automate some of these activities. In this regard, several tools like Advocate [6], Resolute [11], Isabelle [10], AMASS [23], among others [17] have been developed in the recent years. In addition to managing the requirements and artifacts, these tools utilize modular assurance arguments called assurance case patterns[3] [16] to handle the size and complexity of AC being developed. Patterns are argument fragments that provide a partial solution for one aspect of the overall AC. They provide a reusable structure through parameter placeholders that can be instantiated with system-specific artifacts and assembled with other patterns into an AC.

While these tools specialize in data management and automation of the instantiation and assembly algorithm, an activity that has not been researched is the automation of the pattern selection process. To contextualize the selection process, consider the Autonomous Vehicle (AV) example in Fig. 1. Assume we want to develop an AC with the goal that the "Automatic Emergency Braking System (AEBS) will function satisfactorily in applying the emergency brake", given that the operating context is a clear day. For this, we are given an artifact database with system architecture, component decomposition, component testing results (from different contexts like a clear day, rainy day, night, etc.), and a pattern database with patterns related to requirement decomposition, component decomposition, and failures, functional decomposition, hazard decomposition, etc. The problem is to select patterns that (a) support the goal and (b) have all the artifacts in the given context required for instantiation. Typically, a designer manually compares each pattern against the system artifacts to check if all the artifacts required for instantiation are available [22]. It is assumed the designer has complete knowledge of the system artifacts and is familiar with the

[3] In the rest of this paper, we will refer to "AC patterns" as "patterns"

content of the patterns and the context to which they are applicable. However, this comparison gets complicated and tedious for complex systems with more goals and diverse heterogeneous system artifacts [23]. For example, in one of the recent studies, Yamamoto, Shuichiro *et al.* [24] have shown that manual pattern selection took one designer 30 hours (14% of the development time) and required significant understanding about the available artifacts and patterns. Therefore, automating the selection process can aid the assurance process.

Contributions: To close this automation gap, we have developed a workflow that handles the selection problem as a coverage problem, intending to find the smallest set of patterns that can cover the system artifacts. For this, we leverage the ontology graph of the system artifacts and patterns and perform graph analytics. We address the coverage problem using an optimization problem setup, which is assisted by a data preparation function that utilizes a weaving model[4] [3] to generate data files, a mapping file, and an ontology graph of the artifacts. A selection function uses the processed files and a database of patterns to select a set of patterns, which are then plugged into an instantiation function to develop an AC. Finally, the AC is evaluated for coverage, and a report with information about unused artifacts and patterns is generated to aid the developer with future refinement. To evaluate the workflow, we have integrated it with a newly developed tool called ACCELERATE to automatically construct an AC for an Autonomous Vehicle [5] example within a CARLA simulation [7].

Outline: The rest of this paper is organized as follows. In Section 2, we formalize assurance case patterns. In Section 3, we present the proposed workflow that includes the data preparation, pattern selection, and evaluation functions to automate the development and evaluation of an AC. In Section 4, we demonstrate the utility of our workflow by an AV example in a CARLA simulation. Finally, we present the related research in Section 5 followed by our conclusion in Section 6.

2 Assurance Case Patterns

Goal Structuring Notation uses an iterative decomposition strategy to decompose the top-level system goal(s) to be proved to lower-level component goals, often supported by evidence. Although this notation has simplified the documentation of ACs, it is challenging to design monolithic GSNs for complex systems. Patterns [16] are argument fragments that provide a partial solution for one aspect of the overall AC. They capture common repeated argument structures and abstract system-specific argument details as placeholders with free parameters to be instantiated. Patterns typically include information like name, intent, motivation, structure, applicability, related patterns, description of the decomposition strategy, and implementation details. In addition, Kelly has introduced several structural and entity abstractions to the GSN modeling language for describing a pattern [16]. Currently, there are several online catalogs [21, 16, 22] with patterns in GSN format that can be readily used to design an AC.

[4] Captures the fine-grained relationships between different system artifacts

[5] For the CARLA AV setup, visit https://github.com/scope-lab-vu/AV-Assurance

2.1 Pattern Formalization

We adapt the formal definition of patterns as presented by Denney *et al.* [4] with slight modifications, including a metadata field that holds additional information about the pattern and a modifier function that specifies the operations that need to be performed on the nodes. We provide a formal definition below:

Definition 1 (Pattern). *A pattern $\mathcal{P}$ is a tuple $\langle \mathcal{M}, \mathcal{N}, l, t, i, mul, c, mod, \rightarrow \rangle$, where $\langle \mathcal{N}, \rightarrow \rangle$ is a finite, directed hypergraph in which each edge has a single source and possibly several destination targets. $\rightarrow$: $\langle \mathcal{N}, \mathcal{N} \rangle$ is the connector relation between nodes, $\mathcal{M}$ is the pattern metadata, and the functions l, t, p, mul, and mod are defined below:*

- $\mathcal{M}$ is the pattern metadata tuple $\langle N, R, pl \rangle$, where N is the name of the pattern, R is a set of relevant patterns that share the same intent as this pattern or patterns that can be composed with this pattern for further growing the assurance case, and pl is a dictionary that maps a system artifact label (key) to a placeholder variable (value) that requires instantiation. The artifact label is of the type string. An illustrative placeholder dictionary is of the form {"system" : SM, "top-level-goals : TG", "requirements" : SR}.
- l and t are labeling functions, where $l : \mathcal{N} \rightarrow \{g, s, c, a, j, e\}$ maps each node onto its type, namely on g (goal), s (decomposition strategy), c (context), a (assumption), j (justification), or e (evidence).
- i is the id label of each node, i: $\mathcal{N} \rightarrow id \times class$, which returns the identifier and the type of each node, i.e. class = $\{g, s, c, a, j, e\}$.
- mul provides a multiplicity label for each outgoing connector. For the example shown in Fig. 2, $mul = n : RC$ represents a one-to-many relationship, where n is an integer value determined by placeholder RC, such that each instance of node $S2$ is related to n instances of node $G4$. The relationship is one-to-one if not explicitly stated otherwise.
- mod indicates the modifying operation to be performed on a given node: no-operation, instantiate, or develop.

In addition to the pattern entities, there are several structural rules required:

- The root node of a pattern is always a goal.
- The connectors can only go out of the goal and the strategy nodes: $n_a \rightarrow n_b \Rightarrow \mathrm{l}(n_a) \in \{g, s\}$.
- A strategy node cannot directly connect to another strategy node or an evidence node: $(n_a \rightarrow n_b) \wedge [\mathrm{l}(n_a){=}\mathrm{s}] \Rightarrow \mathrm{l}(n_b) \in \{g, a, c, j\}$.

Fig. 2 illustrates an example requirements decomposition pattern based on the formalization in Definition 1. This pattern argues for the satisfaction of the system's high-level requirements through the requirements decomposition of all the associated components. A node in this pattern is represented by its labels (e.g., G1, G2) and content with placeholders (e.g., SM, C) that can be replaced by system-specific information. The node multiplicity (mul) is marked on the graph edges, representing how one node is related to another. Further, "instantiate" and "develop" are the two modifier (mod) operations that can be performed on the nodes.

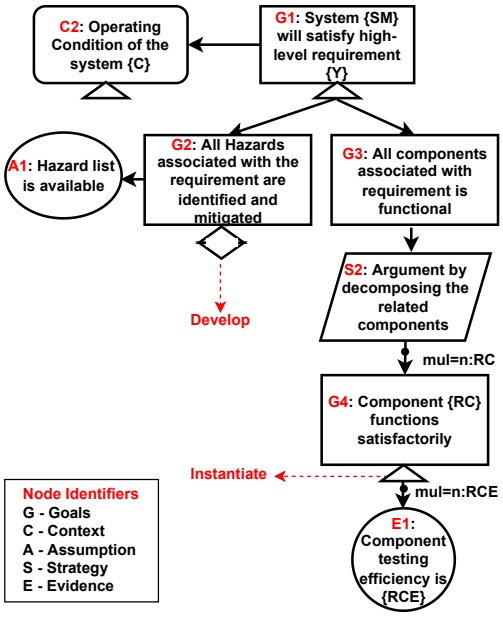

Fig. 2: An example pattern based on requirements decomposition arguments.

3 Pattern Selection Workflow

We present the proposed workflow that leverages an ontology graph of the system artifacts and patterns to automatically select patterns that can be instantiated to construct an AC. The workflow is composed of several functions that work as follows: First, the *prepare*$(\mathcal{AD}, WM)$ function uses a weaving model (WM) to map artifact files from the artifact database $(\mathcal{AD})$ onto several data files (F_D) and a mapping file F_M. Then, the function *select*$(\mathcal{AD}, \mathcal{PD})$ selects a set of patterns $\mathcal{P}_S$ from the pattern database $(\mathcal{PD})$. The selected patterns are instantiated and assembled into an AC using an external *instantiate* function. Finally, the *evaluate*(AC, F_D, F_M) function generates a report with the coverage score (CS) and additional information to aid the evaluation and further refinement of the AC. We discuss these functions in the rest of this section.

3.1 Data Preparation

The artifacts (e.g., goals, requirements, system models) for the assurance process are typically curated using several engineering activities and stored in a database $\mathcal{AD}$. These activities include requirements engineering, system analysis, hazard analysis and risk assessment, and evidence generation. The artifacts generated from these activities are usually in heterogeneous file formats like PDF, Text, Architecture Analysis and Design Language (AADL) and System Modeling Language (SysML). The *prepare* function takes these raw artifact files to prepare the

Algorithm 1 Data Preparation

```
1: function PREPARE(𝒜𝒟:Artifact Database,WM:weaving model)
2:     F_D ← {}, F_M ← {}, temp ← {}
3:     for each file in 𝒜𝒟 do
4:         processed file ← process(file)
5:         temp ← temp ⋃ {processed file}
6:     end for
7:     accepted_files ← manual_check(temp)
8:     for each file in accepted_files do
9:         data file ← arrange(file,WM)
10:        F_D ← F_D ⋃ {data file}
11:    end for
12:    place ← {}, depend ← {}, source ← {}
13:    for each file in F_D do
14:        source ← get_source_query(file)
15:        place ← extract(header)
16:        for each entry in header do
17:            result ← search(entry, F_D)
18:            depend ← result
19:        end for
20:        F_M ← {place, depend, source}
21:        𝒢_A ← make_graph(place,depend)
22:    end for
23:    𝒜𝒟 ← F_M, F_D, 𝒢_A
24:    return F_M, F_D
25: end function
```

processed files required for the pattern selection discussed in the next section. The function performs two operations as shown in Algorithm 1.

The first operation processes relevant artifacts required for the AC into processed data files stored in tabular format (CSV file). The function can currently process AADL files. We are working towards automatically processing other file formats. Then, the processed files are checked for completeness, correctness, and relevance. The check is to ensure that only complete and essential artifacts necessary for the development of the AC are retained while discarding the non-essential artifacts. Non-essential artifacts bloat the $\mathcal{AD}$, which slows the selection process and impacts the evaluation metrics (discussed later in Section 4). In the current implementation, the checking is manually performed by a designer. We assume the designer has complete knowledge of the system for which the AC is being developed. The accepted files are passed through an arrange function to generate a set of data files F_D. To generate the file, we use a weaving model WM that weaves the different artifacts and transforms them into a single model file. The model is developed based on our domain knowledge and previous experience with CPSs. Each data file is a table where the column headers represent the name of the artifacts, and the rows capture the content of these artifacts. Also, each column in the data file is related to the other columns, with the relation-

ship derived using the weaving model. Finally, these accepted files are manually tagged with labels required for the selection process. For example, the context in which particular artifacts and evidence are applicable is one label that we currently include.

The second operation generates a mapping file F_M, which is a lookup table of system artifacts and their ontology required to bridge the data files for the pattern selection algorithm discussed in Section 3. F_M holds the physical link to the data file location obtained using a simple query to the database. It also holds the placeholder and dependency mapping derived from an *extract* function, which reads the header of each data file to create an intra-file dependency mapping between them. For example, one entry capturing the relationship between a cause and a hazard in the dependency mapping file looks like $[cause, cause\_table, hazard]$. If there are multiple causes for the same hazard, they will be stored as separate entries. Next, to capture the inter-file dependencies, each header (e.g., cause) is searched across every data file using a *search* function. The search result is used for placeholder mapping, required for pattern selection. For example, the search result for the cause header is $\{[mitigation\_table, cause], [cause\_table, cause], [risk\_table, cause]\}$, which shows all the other files in which the entry is present. Finally, the ontology captured in F_M is also stored as an artifact graph $\mathcal{G}_A$ as shown in Fig. 3.

Then, we curate $\mathcal{PD}$ for which we gather patterns from online catalogs [16, 22] and manually re-design them using the formalization and rules discussed in Section 2. While re-designing, a designer checks the language consistency across data in the nodes. These patterns are stored in textual format (JSON) and as a graph $\mathcal{G}_P$ with placeholders as nodes (See Fig. 3).

3.2 Pattern Selection

As discussed earlier, the goal of the selection algorithm is to select a smallest set of patterns $\mathcal{P}_S$ from the database $\mathcal{PD} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n\}$ that maximizes the artifact coverage. We formulate the selection as a two-objective optimization problem: (a) maximizing the coverage such that the placeholders of every selected pattern have the corresponding artifact for instantiation and (b) minimizing the number of patterns selected by iteratively comparing the pattern graph to the artifact graph (See Fig. 3).

The optimization is realized using the *select*$(\mathcal{PD}, F_M, F_D)$ function shown in Algorithm 2. It takes the patterns from $\mathcal{PD}$, the mapping file and the data files as inputs to select $\mathcal{P}_S$. The selection is performed using the *findmatch*, the *findconflict* and the *findsubgraph* functions. The selected patterns are then instantiated and assembled into an AC using an external *instantiate* function. An existing algorithm [13, 12] can be used for instantiation and assembly.

Next, the $findcomplete(\mathcal{P}, F_M)$ function in Definition 2 checks if the placeholders in the patterns have a matching entry in F_M. If all the placeholders have corresponding entries, the pattern is said to be complete, and it is added to $\mathcal{P}_S$. Otherwise, the pattern is discarded from the selection process.

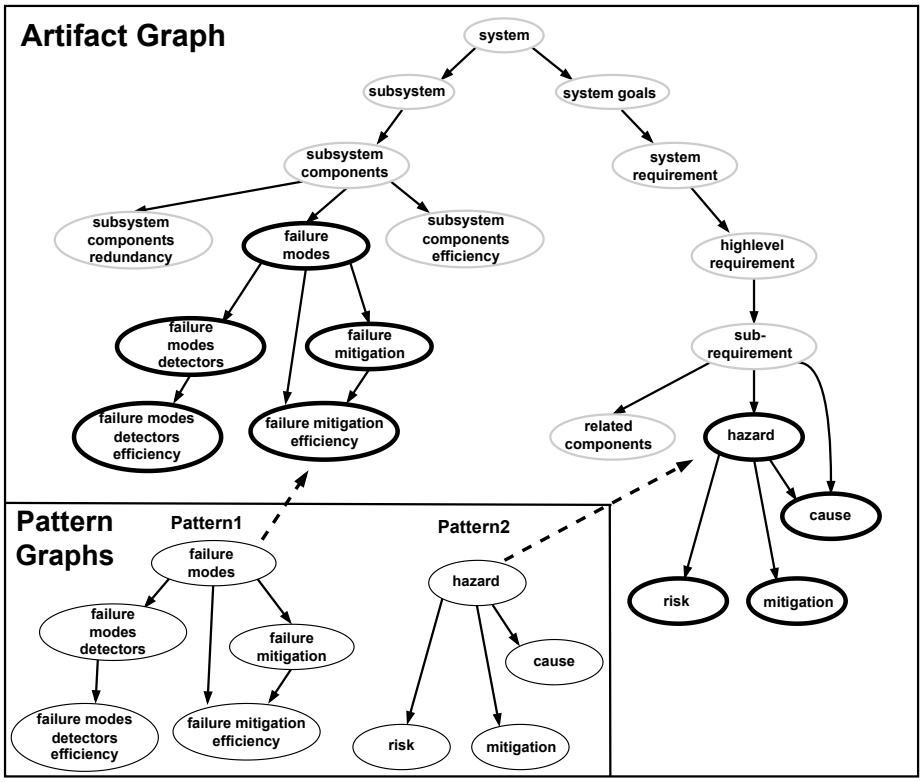

Fig. 3: The artifact and pattern ontology extracted for pattern selection. The selected patterns are highlighted by thick outlines in the artifact graph.

Definition 2 (Pattern Completeness). *We say a pattern is complete if each placeholder has a corresponding entry in F_M. We define a function $findmatch(pl)$ that determines if a given placeholder has a corresponding entry.*

Once $\mathcal{P}_S$ has been selected, the *findsubgraph* and the *findconflict* functions are used to minimize the cardinality of $\mathcal{P}_S$ and remove duplicate patterns. First, the $findsubgraph(\mathcal{G}_A, \mathcal{G}_Pi)$ function checks whether the artifact graph $\mathcal{G}_A$ contains a subgraph that is isomorphic to $\mathcal{G}_Pi$, the graph of the i^{th} pattern. Two graphs are isomorphic (or equivalent) if their structures preserve a one-to-one correspondence between their vertices and between their edges. For example, in Fig. 3, pattern1 and pattern2 are isomorphic to subgraphs of $\mathcal{G}_A$. The non-isomorphic patterns are removed from $\mathcal{P}_S$.

Next, the $findconflict(\mathcal{P}_1, \mathcal{P}_2)$ function checks $\mathcal{P}_S$ for redundant patterns. For this, it performs the following steps: (a) duplication checking checks if the patterns have the same set of placeholders requiring instantiation (see Definition 3). (b) graph checking checks if the graphs of the two patterns are isomorphic. While performing this, we also require data on corresponding nodes of the patterns to be equivalent. Only if the duplication checking fails, the function

Algorithm 2 Pattern Selection

```
1: function SELECT(PD:Pattern Database,F_M:Mapping File,F_D:Data Files)
2:     P_S ← {}, dups ← {}
3:     for each pattern P ∈ PD do
4:         temp ← True
5:         for each placeholder p ∈ P do
6:             temp ← temp ∧ findmatch(p)
7:         end for
8:         if temp is True then
9:             P_S ← P_S ⋃ {P}
10:        end if
11:    end for
12:    for each pattern P ∈ P_S do
13:        if findsubgraph(G_A, G_P) is False then
14:            P_S.remove(G_P)
15:        end if
16:    end for
17:    for i ← 1 to len(P_S) do
18:        for j ← i + 1 to len(P_S) do
19:            match ← findconflict(P_i, P_j)
20:            if match is True then
21:                dups ← dups ⋃ {P_j}
22:            end if
23:        end for
24:    end for
25:    for each entry E in dups do
26:        P_S.remove(E)
27:    end for
28:    AC ← instantiate(P_S,F_M,F_D)
29:    return AC
30: end function
```

performs graph checking. On the whole, the function $findconflict$ returns true if steps (a) or (b) return true, i.e., if the patterns are redundant.

Definition 3 (Duplication Checking). *We say two patterns* $\mathcal{P}_1 = \langle \mathcal{M}_1, \mathcal{N}_1, l_1, t_1, i_1, m_1, s_1, \rightarrow_1 \rangle$ *and* $\mathcal{P}_2 = \langle \mathcal{M}_2, \mathcal{N}_2, l_2, t_2, i_2, m_2, s_2, \rightarrow_2 \rangle$ *are duplicates if they contain exactly the same placeholders.*

3.3 Coverage Evaluation

As discussed previously, automating different activities of the assurance process reduces the development time and manual efforts. However, this gain is at the expense of increased effort and time required to review and evaluate the quality and correctness of the generated AC. To aid the evaluation process, the $evaluate(AC, F_D, F_M)$ function takes the AC and generates a report to provide qualitative insights which is not available in the generated AC graphical structure. We believe this information can aid the designer in further refinement. The

report includes the coverage score, the selected and unused patterns, and the unused artifacts. The coverage score is a tuple $\langle \mathcal{A}, \mathcal{S} \rangle$ of the artifact coverage ($\mathcal{A}$) and the problem coverage ($\mathcal{S}$).

1) Artifact Coverage ($\mathcal{A}$): The artifact coverage metric measures the proportion of the artifacts available in $\mathcal{AD}$ that have been included in the AC. Also, a relevance check (discussed in Section 3.1) on the artifact is essential for this metric to be accurate. Besides the score itself, this metric can also be used to derive a list of unused artifacts.

$$\mathcal{A} = \frac{\text{\# Artifacts used in the AC}}{\text{\# Artifacts available in } \mathcal{AD}} \tag{1}$$

2) Problem Coverage ($\mathcal{S}$): The problem coverage metric quantifies the coverage of all the known problems affecting a system's property (e.g., safety, availability). Problem coverage is a tuple $\mathcal{S} = (C_{\mathcal{P}_1}, C_{\mathcal{P}_2}, \cdots, C_{\mathcal{P}_n})$ consisting of coverage measures $C_{\mathcal{P}_i}$ related to different problem classes $\mathcal{P}_i$. The coverage measure is shown in Eq. (2), and it is computed as the percentage of problems within a given problem class which are addressed by an AC.

$$C_{\mathcal{P}_i} = \frac{\text{\# Problems from } \mathcal{P}_i \text{ addressed by the AC}}{\text{\# Problems in } \mathcal{P}_i \text{ identified during analysis}} \tag{2}$$

While coverage metrics and the report can aid the refinement process by providing insights into the missing patterns or artifacts, they do not fully quantify the quality of artifacts (e.g., evidence) required for AC selection. So, the coverage metrics cannot solely rely on measuring the quality of the AC. A combination of coverage and confidence metrics is needed for robust quantitative assessment. We are therefore working towards integrating a confidence metric.

4 Illustrative Example

In this section, we provide an illustrative example by applying the proposed workflow to develop an AC for an AV in the CARLA simulator [7]. In this example, the AV is required to navigate a town while avoiding collisions with obstacles in its travel path. We integrate our workflow with the ACCELERATE tool [6] for pattern instantiation and assembly as shown in Fig. 4.

Artifacts and Patterns Preparation: We performed the analysis steps listed in Section 3.1 to curate $\mathcal{AD}$. We first performed a requirement and system analysis using the given requirements document. The vehicle has three goals associated with two system requirements and three high-level requirements, each associated with several sub-requirements. We then designed the AV system model shown in Fig. 1. It has a navigation component that uses three cameras, a global positioning system (GPS), an inertial measurement unit (IMU), a speedometer, and a route planner to compute the vehicle's next position. Then a velocity

[6] Tool is being built as part of the DARPA ARCOS program. Check our GitHub for release information.

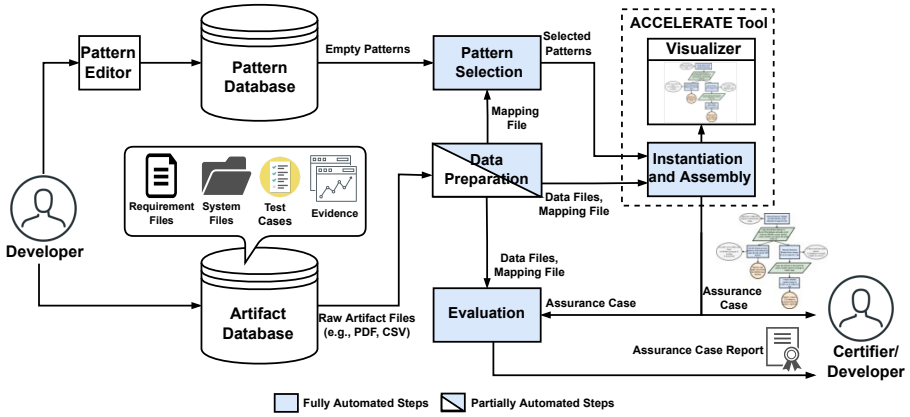

Fig. 4: The proposed pattern selection workflow integrated with the ACCELERATE tool for AC development.

planner calculates the average velocity needed to traverse from the current position to the next position. The velocity and the camera images are fed to a deep-learning controller to predict the waypoints, which are passed to a motion estimator to compute throttle, brake, and steer errors. In addition, it has an AEBS controller that uses two radars to raise a brake alarm on detecting obstacles. We then performed fault analysis of the system model to identify 14 component faults and analyzed the possible mitigation strategies. Further, we performed hazard analysis to identify eight operational and functional hazards associated with the different system components. Finally, we curated $\mathcal{PD}$ for which we gathered several patterns from online catalogs [16, 22] and re-designed them using the formalization discussed in Section 3.2.

Results: We applied the integrated tool to develop an AC for the vehicle. We summarize the key results in terms of the coverage metrics and the size of the AC (computed in terms of GSN nodes) in two revisions. We used the AC report[7] from "revision1" to refine the AC in "revision2". The pattern database had seven patterns for the selection process to choose from in these revisions. The analysis of the revisions is: In "revision1", four patterns were selected to develop an AC[7] with 805 nodes. The evaluation function returned a coverage score with artifact coverage of 76%, a problem coverage of $[C_H : 60\%, C_F : 100\%]$ with five unused artifacts. Here, C_H represents the percentage of known hazards that are covered, and C_F represents the percentage of known system faults that are covered. From the report, we analyzed the artifacts relating to failure decomposition that was unused. So, we designed a new failure decomposition pattern that was added to $\mathcal{PD}$. Further, some of the sub-requirements associated with the hazards were missing, which we included. In "revision2", the selection mechanism selected five patterns, including the new pattern to develop an AC with 909 nodes. The

[7] For a bird's eye view of the "revision1" assurance case and the report, visit https://github.com/scope-lab-vu/AV-Assurance

refined AC had a higher coverage with an artifact coverage of 90%, a problem coverage of $[C_H : 85\%, C_F : 100\%]$ with unused artifacts reduced by two. We performed several iterations until all the artifacts were included in the AC.

To estimate the time saved by the workflow, the data preparation and selection steps were first performed manually by a developer who performed the following tasks: re-design of patterns into the defined formalization, which took approximately one hour, processing the artifact files, extracting the artifact dependencies to generate an ontology file, and instantiation and assembly of the patterns using the ACCELERATE tool. While instantiation and assembly were performed in less than a minute, the manual selection and data curation process took approximately three hours. Next, for comparison, we fed the manually redesigned patterns and the artifacts to the integrated tool (See Fig. 4), which only took close to one minute for data preparation, pattern selection, instantiation, and assembly. Finally, to stress test the integration, we increased the artifacts and patterns in the database. Our workflow took less than five minutes, even for large ACs with 1500 to 3000 nodes. Significantly less manual processing was needed when the artifact files were changed or updated. We expect the time saving to get even more significant as the size of the artifact database grows. However, the manual steps involved in the data preparation step are a bottleneck for scaling the workflow, which we want to address in the future.

5 Related Work

The last decade has seen several tools with automation capabilities to support different activities of the AC development process. A comprehensive survey on these tools is available in [17]. We discuss a few of these tools that support automation. Advocate [6] is one such tool that provides an editor for designing system architectures, patterns, and automated development of ACs from patterns. A pattern formalization and the instantiation algorithm is built into the tool for automating pattern instantiation [4]. Here, a pattern dataset and a parameter table are manually created to assist the instantiation algorithm. Resolute [11] is another tool that automatically synthesizes an AC from AADL models. Isabelle [10] is a recently developed tool with integrated formal methods for evidence generation. Assurance language and automated document processing are a few tool features that support the development process. AMASS tool [23] provides a partially automated heterogeneous collaborative environment that supports activities such as requirement management, artifacts and evidence generation, pattern composition, and AC construction.

There are several independent efforts. For example, Ramakrishna *et al.* [20] have presented a methodology to partially automate AC construction directly from system models and graphs. Hawkins *et al.* [13] utilize the concept of model weaving to automatically learn the artifact files from system models and use them for instantiating patterns. The authors of [12] provide an automated mechanism for instantiation and composition of patterns, where the artifacts are heterogeneous system models that are linked to represent the cross-domain relationship.

While these tools and approaches automate the instantiation and assembly of patterns, their selection largely remains manual.

Further, evaluation is key to automating AC development. Confidence metrics are often used to represent the assurance deficit [14]. However, there has been minimal work in coverage evaluation. Denney *et al.* [5] have presented several coverage metrics for different system artifacts like hazards and requirements. These metrics measure the proportion of the system artifacts used in the AC to those available in the database. Chindamaikul *et al.* [2] have presented two coverage metrics: a claim coverage that is similar to those in [5], and an argument coverage metric that measures the arguments and evidence covered in the AC. We build on prior work to provide additional coverage metrics.

6 Conclusion and Future Work

In this paper, we have presented a workflow that can automate the pattern selection process. We formulate the selection problem as a coverage problem that selects the smallest set of patterns that can maximally cover the available system artifacts. The coverage problem is realized using an optimization problem that leverages the ontology graphs of the artifacts and patterns and performs graph analytics. The optimization is aided by an array of functions that perform data preparation, pattern selection, and AC evaluation. These functions collectively reduce the manual effort and time required in selecting the necessary patterns.

We plan to move this research in several directions. First, fully automating the data processing function using natural language processing (NLP). Second, design a translator to convert textual patterns into our format. Third, automate the language check using NLP and relevance check using topic modeling [2]. Finally, include confidence metrics for AC evaluation.

Acknowledgement. The authors would like to thank Sarah C. Helble and Dennis M. Volpano for helpful discussions and feedback. This work was supported by the DARPA ARCOS project under Contract FA8750-20-C-0515 (ACCELERATE) and the DARPA Assured Autonomy project. The views, opinions, and/or findings expressed are those of the author(s) and do not necessarily reflect the views of DARPA. We would like to thank the reviewers and editors for taking the time and effort necessary to review the manuscript. We appreciate the valuable feedback, which helped us to improve the quality of the manuscript.

References

1. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Safety and Reliability. vol. 20, pp. 34–42. Taylor & Francis (2000)
2. Chindamaikul, K., Toshinori, T., Port, D., Hajimu, I.: Automatic approach to prepare information for constructing an assurance case. In: International Conference of Product Focused Software Development and Process Improvement (2014)
3. Del Fabro, M.D., Bézivin, J., Jouault, F., Valduriez, P., et al.: Applying generic model management to data mapping. In: BDA (2005)

4. Denney, E., Pai, G.: A formal basis for safety case patterns. In: International Conference on Computer Safety, Reliability, and Security. pp. 21–32. Springer (2013)
5. Denney, E., Pai, G.: Automating the assembly of aviation safety cases. IEEE Transactions on Reliability **63**(4), 830–849 (2014)
6. Denney, E., Pai, G., Pohl, J.: Advocate: An assurance case automation toolset. In: International Conference on Computer Safety, Reliability, and Security. pp. 8–21. Springer (2012)
7. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: Carla: An open urban driving simulator. arXiv:1711.03938 (2017)
8. European Organisation for the Safety of Air Navigation: Safety case development manual, ver 2.2 (2006)
9. FDA, [Online]: Introduction of assurance case method and its application in regulatory science (2019), `https://www.fda.gov/media/125182/download`
10. Foster, S., Nemouchi, Y., O'Halloran, C., Stephenson, K., Tudor, N.: Formal model-based assurance cases in isabelle/sacm (2020)
11. Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M.: Resolute: an assurance case language for architecture models. ACM SIGAda Ada Letters **34**(3) (2014)
12. Hartsell, C., Mahadevan, N., Dubey, A., Karsai, G.: Automated method for assurance case construction from system design models. In: 2021 5th International Conference on System Reliability and Safety (ICSRS). pp. 230–239 (2021)
13. Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T.: Weaving an assurance case from design: a model-based approach. In: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering. pp. 110–117. IEEE (2015)
14. Hawkins, R., Kelly, T., Knight, J., Graydon, P.: A new approach to creating clear safety arguments. In: Advances in systems safety, pp. 3–23. Springer (2011)
15. Kelly, T., Weaver, R.: The goal structuring notation–a safety argument notation. In: Proceedings of the dependable systems and networks workshop on assurance cases. p. 6. Citeseer (2004)
16. Kelly, T.P.: Arguing safety: a systematic approach to managing safety cases. Ph.D. thesis, University of York York, UK (1999)
17. Maksimov, M., Fung, N.L., Kokaly, S., Chechik, M.: Two decades of assurance case tools: a survey. In: International Conference on Computer Safety, Reliability, and Security. pp. 49–59. Springer (2018)
18. Nair, S., de la Vara, J.L., Sabetzadeh, M., Falessi, D.: Evidence management for compliance of critical systems with safety standards: A survey on the state of practice. Information and Software Technology **60**, 1–15 (2015)
19. Palin, R., Ward, D., Habli, I., Rivett, R.: Iso 26262 safety cases: Compliance and assurance (2011)
20. Ramakrishna, S., Hartsell, C., Dubey, A., Pal, P., Karsai, G.: A methodology for automating assurance case generation. arXiv preprint arXiv:2003.05388 (2020)
21. Safety-Critical Systems Club, [Online]: Tiered pattern catalogue (2022), `https://scsc.uk/gsn?page=gsn%205Library%20Patterns`
22. Szczygielska, M., Jarzkebowicz, A.: Assurance case patterns on-line catalogue. In: Advances in Dependability Engineering of Complex Systems, pp. 407–417 (2017)
23. de la Vara, J.L., Parra, E., Ruiz, A., Gallina, B.: The amass tool platform: An innovative solution for assurance and certication of cyber-physical systems. In: REFSQ Workshops (2020)
24. Yamamoto, S., Matsuno, Y.: An evaluation of argument patterns to reduce pitfalls of applying assurance case. In: 2013 1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE). pp. 12–17. IEEE (2013)