A Formal Basis for Safety Case Patterns

Ewen Denney and Ganesh Pai

SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
{ewen.denney,ganesh.pai}@nasa.gov

Abstract. By capturing common structures of successful arguments, safety case patterns provide an approach for reusing strategies for reasoning about safety. In the current state of the practice, patterns exist as descriptive specifications with informal semantics, which not only offer little opportunity for more sophisticated usage such as automated instantiation, composition and manipulation, but also impede standardization efforts and tool interoperability. To address these concerns, this paper gives (i) a formal definition for safety case patterns, clarifying both restrictions on the usage of multiplicity and well-founded recursion in structural abstraction, (ii) formal semantics to patterns, and (iii) a generic data model and algorithm for pattern instantiation. We illustrate our contributions by application to a new pattern, the requirements breakdown pattern, which builds upon our previous work.

Keywords: Safety cases, Safety case patterns, Formal methods, Automation.

1 Introduction

Safety case patterns are intended to capture repeatedly used structures of successful (i.e., correct, comprehensive and convincing) arguments, within a safety case [11]. In effect, they provide a re-usable approach to safety argumentation by serving as a means to capture expertise, so-called *tricks of the trade*, i.e., known best practices, successful certification approaches, and solutions that have evolved over time. The existing notion of a pattern[1] is an extended argument structure, often specified graphically using the Goal Structuring Notation (GSN) [8], which abstractly captures the reasoning linking certain (types of) claims to the available (types of) evidence, and is accompanied by a clear prescription and proscription of its usage.

In current practice, patterns have informal semantics and, in general, they are given as descriptive non-executable specifications. Specifically, in existing tools, pattern-based reuse does not go beyond simple replication of a pattern argument structure, and manual replacement of its abstract elements with their concrete instances. Such usage is not only effort intensive but also unlikely to scale well. Algorithmically instantiating patterns is a natural solution to address this deficiency. However, to our knowledge, existing tools provide little to no such functionality, in part, because of the lack of a formal basis. The latter additionally impedes standardization and tool interoperability.

[1] In the rest of the paper we will simply use "pattern" instead of "safety case pattern".

F. Bitsch, J. Guiochet, and M. Kaniche (Eds.): SAFECOMP 2013, LNCS 8153, pp. 21–32, 2013.

This paper extends the state of the art in safety case research though the following contributions: we give a formalization for argument structures elaborating on the nuances and ambiguities that arise when using the available GSN abstractions [8] for pattern specification. In particular, we clarify restrictions on the usage of multiplicity and extend the basic concepts to include a notion of well-founded recursion. Next, we give a formal semantics to patterns in terms of their (set of) concrete instantiations. We specify a generic data model and pattern instantiation algorithm, and illustrate their application to a new pattern: the requirements breakdown pattern, which builds upon our previous work [3], [6]. Specifically, both generalize and replace their previous incarnations [3] that mainly operated on requirements and hazard tables.

2 Background

Currently [10], [11], a pattern specification mainly contains:

- *Name*: the identifying label of the pattern giving the key principle of its argument.
- *Intent*: that which the pattern is trying to achieve.
- *Motivation*: the reasons that gave rise to the pattern.
- *Structure*: the abstract structure of the argument given graphically in GSN.
- *Participants*: each element in the pattern and its description.
- *Collaborations*: how the interactions of the pattern elements achieve the desired effect of the pattern.
- *Applicability*: the circumstances under which the pattern could be applied, i.e., the necessary context.
- *Consequences*: that which remains to be completed after pattern application.
- *Implementation:* how the pattern should be applied.

In addition, previously known usages, examples of pattern application, and related patterns are also given to assist in properly deploying a particular pattern. A variety of such pattern specifications can be found in [1], [10], and [15].

We assume that the reader is familiar with the basic syntax of GSN and do not repeat it here. The GSN standard [8] provides two types of abstractions for pattern specification: *structural* and *entity*.

Structural abstraction, which applies to the *is-solved-by* and the *in-context-of* GSN relations, is supported by the concepts of *multiplicity* and *optionality*. The former generalizes n-ary relations between GSN elements, while the latter captures alternatives in the relations, to represent a k-of-m choice, where $k \geq 1$. There are, further, two types of multiplicity: *optional*, implying zero or one, and *many*, implying zero or more. Multiplicity can be combined with optionality: placing a multiplicity symbol prior to the option describes a multiplicity over all the options. This is equivalent to placing that multiplicity symbol on all the alternatives after the option [8].

For entity abstraction, GSN provides the notions “Uninstantiated (UI)”, and “Uninstantiated and Undeveloped (UU)”. The former refers to abstract elements whose parameters are replaced with concrete values upon instantiation. The latter refers to UI

entities that are also undeveloped[2]. Thus, upon instantiation, an abstract UU entity is replaced with a concrete, but undeveloped, instance.

In addition to these, there are (limited) examples of the use of a *recursion* abstraction in the literature [12], although it is not formally part of the GSN standard. Recursion, in the context of patterns, expresses the notion that a pattern (or a part of it) can itself be repeated and unrolled, e.g., as part of an optional relation or a larger pattern. Recursion abstractions may or may not be labeled with an expression giving the number of iterations to be applied in a concrete instance.

3 Formalization

In this section, first we modify an earlier definition of a partial safety case argument structure [3], [7], adding a labeling function for node contents. Then, we give a formal definition of a pattern, clarifying conditions on multiplicity and recursion. Subsequently, we give a formal semantics to patterns as the set of their concrete instances, via a notion of pattern refinement.

Definition 1 (Partial Safety Case Argument Structure). *Let* $\{s, g, e, a, j, c\}$ *be the node types* strategy, goal, evidence, assumption, justification*, and* context *respectively. A* partial safety case argument structure $\mathcal{S}$ *is a tuple* $\langle N, l, t, \rightarrow \rangle$*, comprising the set of nodes,* N*, the labeling functions* $l : N \rightarrow \{s, g, e, a, j, c\}$ *that gives the node type,* $t : N \rightarrow E$ *giving the node contents, where* E *is a set of expressions, and the connector relation,* $\rightarrow$*:* $\langle N, N \rangle$*, which is defined on nodes. We define the transitive closure,* $\rightarrow^*$*:* $\langle N, N \rangle$*, in the usual way. We require the connector relation to form a finite* directed acyclic graph *(DAG) with the operation* $isroot_N(r)$ *checking if the node* r *is a root in the DAG*[3]*. Furthermore, the following structural conditions must be met:*

(1) Each root of the partial safety case is a goal: $isroot_N(r) \Rightarrow l(r) = g$
(2) Connectors only leave strategies or goals: $n \rightarrow m \Rightarrow l(n) \in \{s, g\}$
(3) Goals cannot connect to other goals: $(n \rightarrow m) \wedge [l(n) = g] \Rightarrow l(m) \in \{s, e, a, j, c\}$
(4) Strategies cannot connect to other strategies or evidence: $(n \rightarrow m) \wedge [l(n) = s] \Rightarrow l(m) \in \{g, a, j, c\}$

For this paper, note that in Definition 1 we have excluded the concept of an undeveloped node; consequently our definition of a pattern (Definition 2) excludes the notions of UI or UU nodes. Extending both definitions to include these is straightforward.

Definition 2 (Pattern). *A* pattern $\mathcal{P}$ *is a tuple* $\langle N, l, t, p, m, c, \rightarrow \rangle$*, where* $\langle N, \rightarrow \rangle$ *is a directed hypergraph*[4] *in which each hyperedge has a single source and possibly multiple targets, the structural conditions from Definition 1 hold, and* l*,* t*,* p*,* m*, and* c *are labeling functions, given as follows:*

[2] Annotating an entity as "undeveloped" is part of the main GSN syntax to indicate incompleteness, i.e., that an entity requires further support.
[3] A safety case argument structure has a single root.
[4] A graph where edges connect multiple vertices.

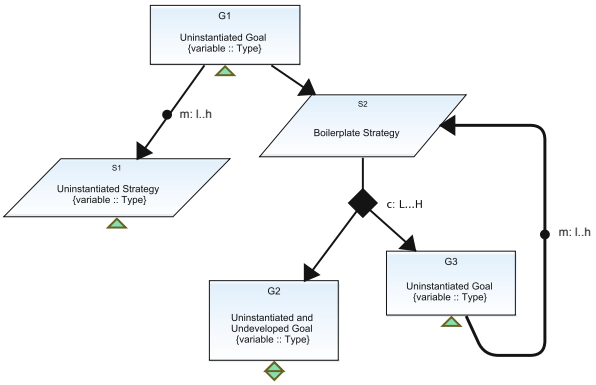

Fig. 1. Abstractions in GSN for patterns specification and our proposed modifications

(1) l *and* t *are as in Definition 1 above*
(2) p *is a parameter label on nodes,* $p : N \rightharpoonup Id \times T$*, giving the parameter identifier and type. Without loss of generality, we assume that nodes have at most a single parameter*
(3) $m : (\rightarrow) \times N \rightarrow (N \times N)$ *gives the label on the* i^{th} *outgoing connector*[5]*. Without loss of generality, we assume that multiplicity only applies to outgoing connectors. If it is* $\langle l, h\rangle$ *then multiplicity has the range* $l..h$*, where* $l \leq h$*. An optional connector has range* $0..1$*.*
(4) $c : (\rightarrow) \rightarrow N \times N$*, gives the "*$l..h$ *of* n*" choice range. We give ranges and omit the* n*.*

Fig. 1 illustrates the GSN abstractions for pattern specification formalized in Definition 2. We now give some notational conventions and auxiliary definitions that we will make use of:

(a) As shown in Fig. 1, pattern nodes take parameters, which reference a set of values V, partitioned into types, and T ranges over types. We write $v :: T$, when v is a value of type T.
(b) A pattern node N is a *data node*, written as $data(N)$, if it has a parameter, i.e., $N \in dom(p)$ (nodes G1, S1, G2 and G3 in Fig. 1). Otherwise, a node is *boilerplate* (node S2 in Fig. 1). We will write $bp(N)$ when N is a boilerplate node. For certain nodes, e.g., so-called *evidence assertions* [14], data may not be available until *after* instantiation. Although, strictly speaking, they are data nodes, we consider them to be boilerplate here (see Section 5 for an example).
(c) The links of the hypergraph, $A \rightarrow \mathbf{B}$, where A is a single node and $\mathbf{B}$ is a set of nodes, represent choices. We write $A \rightarrow B$ when $A \rightarrow \mathbf{B}$ and $B \in \mathbf{B}$.
(d) The bounds on multiplicity and optionality are represented as ranges. To define the labeling functions m and c, we treat $\rightarrow$ as a set with members $\langle A, \mathbf{B}\rangle$, where $A \rightarrow \mathbf{B}$. Then,
   – If $c(\langle A, \mathbf{B}\rangle) = \langle l, h\rangle$ we write $A \rightarrow^{l..h} \mathbf{B}$ (range on choice).

[5] Although siblings are unordered in GSN, it is convenient to assume an ordering.

- If $m(\langle A, B\rangle, \_) = \langle l, h\rangle$, we write $A \rightarrow^{l..h} B$ (range on multiplicity).

(e) We write $sub(\mathcal{P}, A)$ for the *sub-pattern* of $\mathcal{P}$ at A, i.e., the restriction of $\mathcal{P}$ to $N' = \{X \mid A \rightarrow^* X\}$, and $sub(\mathcal{P}, A, B)$ for the restriction of $\mathcal{P}$ to $\{X \mid A \rightarrow^* X$ and $B \not\rightarrow^* X\}$. Roughly, this is the fragment of $\mathcal{P}$ between A and B (including A, but excluding B and everything below it).

(f) Write $multi(\mathcal{P}, B)$ if there exists an $A \in \mathcal{P}$ such that $A \rightarrow^{l..h} B$ and $h > 1$, that is, pattern node B can be repeated in instances of $\mathcal{P}$. We will write $multi(B)$ when $\mathcal{P}$ is obvious, and often consider $multi(G, B)$, where G is a subgraph of $\mathcal{P}$.

(g) A path, s, in the pattern is a sequence of connected nodes. If s connects A and B, we write this as $s : A \rightarrow^* B$.

(h) Write $A < B$ if for all paths from the root $s : R \rightarrow^* B$, we have $A \in s$.

(i) Write $A \rightarrow^n B$ when there is a path of length n in the pattern between nodes A and B. Then we define $A \rightarrow^{must} \mathbf{B}$, when every path from A that is sufficiently long must eventually pass through some $B \in \mathbf{B}$, i.e., $\exists n. \forall s : A \rightarrow^n X. \exists B \in \mathbf{B}. B \in s$.

We now introduce a restriction on the combination of multiplicities and boilerplate nodes. The intuition is that multiplicities should be resolved by data, and not arbitrarily duplicated: it is only meaningful to repeat those boilerplate nodes associated with distinctly instantiated data nodes.

Definition 3 (Multiplicity Condition). *We say that a pattern satisfies the* multiplicity condition *when for all nodes B, if $multi(B)$, and not $data(B)$, then there exists a C such that $B \rightarrow^* C$, $data(C)$, and for all X such that $B \rightarrow^+ X \rightarrow^* C$, not both $multi(X)$ and $bp(X)$.*

In other words, a multiplicity that is followed by boilerplate must eventually be followed by a data node, with no other multiplicity in between. This has two consequences: (i) we cannot have multiplicities that do not end in data, and (ii) two multiplicities must have intervening data.

In contrast to concrete argument structures, we allow cyclic structures and multiple parents in patterns. However, we need a restriction to rule out 'inescapable' loops, so that recursion is well-founded.

Definition 4 (Well-foundedness). *We say that an argument pattern is* well-founded *when, for all pattern nodes A, and sets of nodes $\mathbf{B}$, such that $A \notin \mathbf{B}$, if $A \rightarrow^{must} \mathbf{B}$ then it is not the case that for all $B \in \mathbf{B}$, $B \rightarrow^{must} A$.*

We give semantics to patterns in the style of a single-step refinement relation $\sqsubseteq_1$. Intuitively, the idea is to define the various ways in which indeterminism can be resolved in a pattern. As before (Definition 2), pattern $\mathcal{P} = \langle N, l, t, p, m, c, \rightarrow\rangle$ and we describe the components of $\mathcal{P}$ which are replaced in $\mathcal{P}'$.

Definition 5 (Pattern Refinement). *For patterns $\mathcal{P}$, $\mathcal{P}'$, we say that $\mathcal{P} \sqsubseteq_1 \mathcal{P}'$ iff any of the following cases hold:*

(1) Instantiate parameters*: If $p(n) = \langle id, T\rangle$ and $v :: T$, then replace node contents, t, with $t' = t \oplus \{n \mapsto t(n)[v/id]\}$.*

(2) Resolve choices*: If $A \rightarrow^{l..h} \mathbf{B}$, $\mathbf{B}' \subseteq \mathbf{B}$ and $l \leq |\mathbf{B}'| \leq h$, then replace $A \rightarrow \mathbf{B}$ with $A \rightarrow B$ for each $B \in \mathbf{B}'$.*

(3) Resolve multiplicities*: If $A \rightarrow^{l..h} B$ then replace the link $A \rightarrow B$ with n copies (that is, disjoint nodes, with the same connections and labels), where $l \leq n \leq h$.*

(4) Unfold loops*: If $A \rightarrow^{*} B$, $B \rightarrow A$, and $A < B$, then let S be the sub-pattern of $\mathcal{P}$ at A, $sub(\mathcal{P}, A)$. We create a copy of S and replace the link from B to A with a link from B to the copy of S (i.e., we sequentially compose the two fragments).*

Then, $\mathcal{P} \sqsubseteq \mathcal{P}'$ iff $\mathcal{P} \sqsubseteq_1^* \mathcal{P}'$.

We will define pattern semantics in terms of refinement to arguments. Formally, however, a pattern refines to another pattern, so we need to set up a correspondence between concrete patterns and arguments structures. We define this as an embedding from the set of argument structures into patterns.

Definition 6 (Pattern Embedding). *An embedding $\mathcal{E}$ of an argument structure into a pattern is given as $\mathcal{E}(\langle N, l, t, \rightarrow\rangle) = \langle N, l, t, p, m, c, \rightarrow'\rangle$ where $p = \varnothing$, the labeling functions m and c always return $1..1$, and hyperedges have a single target, i.e., for all nodes $A \in N$, $\rightarrow'(a) = \{\rightarrow(a)\}$.*

We can now define the semantics of a pattern as the set of arguments equivalent to the refinements of the pattern.

Definition 7 (Pattern Semantics). *Let $\mathcal{P}$ be a pattern, let C and A range over patterns, and safety case argument structures, respectively. Then*[6]*, we give the semantics of $\mathcal{P}$ as $[\![\mathcal{P}]\!] = \{A \mid \mathcal{P} \sqsubseteq C, \mathcal{E}(A) = C\}$.*

4 Instantiation

Now, we formalize the concept of a pattern dataset, define a notion of *compliance* between data and a pattern, and specify a generic instantiation algorithm.

4.1 Datasets and Tables

We use sets of values to instantiate parameters in patterns to create instance arguments. Roughly speaking, data can be given as a mapping from the identifiers of data nodes to lists of values. However, since a pattern is a graph there can be multiple ways to navigate through it (due to recursion and nodes with multiple parents) and, therefore, connect the instance nodes. To make clear where an instantiated node should be connected, we need to associate each 'instantiation path' through the pattern with a *join point* (or simply *join*), indicating where a "pass" through the pattern begins. A join uniquely indicates the location at which an instantiated branch of the argument structure is to be appended. In practice, join points can be omitted if the location can be unambiguously determined.

We adopt a liberal notion of pattern instance and do not require all node parameters to be instantiated. Moreover, uninstantiated nodes do not appear in the resulting instance[7].

[6] Strictly speaking, this should be defined as a set of equivalence classes of arguments, where we abstract over node identifiers, but we can safely gloss over that here.

[7] Except for special cases where they have been considered as boilerplate (see item (b) on p. 24).

Definition 8 (Pattern Dataset). *Given a pattern, $\mathcal{P}$, define a* $\mathcal{P}$-dataset *as a partial function $\tau : (D \times V) \times D \rightharpoonup (\mathbb{N}^* \rightharpoonup V)$, where D is the set of data nodes in $\mathcal{P}$, V is the set of values, and $\mathbb{N}^*$ is the set of indices. We write $v \in^{r,c} \tau$ when for some i, $\tau(r,c)(i) = v$, and require that values be well-typed, i.e., if $v \in^{r,c} \tau$ and $p(c) = \langle id, T \rangle$ then $v :: T$.*

Data will typically be represented in tabular form where we label columns by data nodes, D, and rows by $D \times V$ pairs, i.e., joins. Entries in the table are represented as indexed lists of values. The order in which a dataset is tabulated does not actually provide any additional information, but in order to be processed by the instantiation algorithm, must be *consistent* with the pattern, in the following sense: the order of columns must respect node order[8] $<$, i.e., if $A < B$ then the corresponding columns are in that order; and for each row $\langle D, v \rangle$, we require that v appears in column D in a preceding row. In the following, we will assume that a consistent order has been chosen for a dataset, and refer to it as a $\mathcal{P}$-table (see Table 1 for an example).

Definition 9 (Data Compliance). *For pattern $\mathcal{P}$ and $\mathcal{P}$-table τ, we say that the table complies with the pattern, $\tau \vDash \mathcal{P}$, if the following two conditions hold:*

(i) τ meets the cardinality constraints of $\mathcal{P}$, i.e., $\forall c \,.\, l \leq |\tau((\_,\_), c)| \leq h$, where $\langle l, h \rangle = m(i, c')$, where $c' \rightarrow^i c$.

(ii) τ is upwards-closed*, i.e., for each r labeled (D, v) and column c, if $v \in^{r,c} \tau$ and $c \leq c' \leq D$, then there exists v' such that $v' \in^{r,c'} \tau$.*

Note that the ordering used in upwards-closure is with respect to the pattern and not the column order. The intuition behind upwards closure is that, in line with our notion of partial instantiation and although not all nodes need be instantiated, we do require that parameters can be instantiated in order from the root. A row, therefore, consists of the data that instantiates an *upward-closed* fragment of the pattern, following the paths of the fragment up until the join (see Fig. 4 for an example).

4.2 Algorithm

Fig. 2 specifies $instantiate(\mathcal{P}, \tau)$, our generic algorithm for pattern instantiation. We write $new(D.v)$, to create a new instance node, given by instantiating data node D with value v. When a boilerplate node B is instantiated, then we reference its instance simply as B. Let $\mathcal{F}$ be the set of argument structure fragments. To connect an instance node $D.v$ to a fragment $f \in \mathcal{F}$, we use a function $connect(D.v, f)$, which sequentially composes f with the current instance fragment at node $D.v$.

To instantiate a pattern $\mathcal{P}$, given its $\mathcal{P}$-table τ, we process each row to create a *row instance* fragment, which is effectively the assignment of parameter values in the table to the corresponding data nodes in the pattern. We construct the row instance based on the ordering of the data nodes in the columns. For each value we add not just the instantiation of the appropriate data node, but also any boilerplate between that node and the preceding data node. We give a row instance as $RI \in N \times \mathbf{N} \times \mathbb{N}^*$, where N, $\mathbf{N}$, and $\mathbb{N}^*$ are the sets of pattern nodes, instance nodes and natural number indices respectively.

[8] See item (h) on p. 25.

```
1  Instantiate(P: Pattern, τ: P-table)
2  begin
3      foreach row r ∈ table τ do
4          initialize row instance RI ← ∅
5          if row label = ⟨root, v⟩ then
6              create instance node j ← new(root.v) and assign pattern node current ← root
7              update row instance RI ← RI ∪ ⟨current, j, [ ]⟩
8          else if row label = ⟨C, v⟩ then
9              create join instance node j ← new(C.v) and assign current ← C
10         foreach column E ∈ table τ not including root do
11             assign pattern node N ← current
12             foreach (v, index i) ∈ table τ(r, E) do
13                 assign fragment f ← boilerplate B ∈ sub(P, current, E) such that multi(B) ∨ ⟨B, B, [ ]⟩ ∉ RI
14                 if E is first column in row r with data then assign instance node n ← j
15                 else find parent node n with index k such that ∃⟨N, n, k⟩ ∈ RI
16                 connect(n, f)
17                 foreach boilerplate B ∈ f do update row instance RI ← RI ∪ ⟨B, B, i⟩
18                 if ∃P ∈ sub(P, current, E) | multi(P) then assign pattern node N ← parent(P)
19                 assign pattern node M ← parent(E)
20                 assign instance node p ← instance node m ∈ f such that m = M.v
21                 connect(p, E.v)
22                 update row instance RI ← RI ∪ ⟨E, E.v, i⟩
23                 assign current ← E
```

Fig. 2. Generic algorithm for pattern instantiation

Multiplicities, especially, require careful consideration: multiple values in the $\mathcal{P}$-table lead to multiple instances of a data node, but we only repeat those boilerplate nodes which appear after a multiplicity (see Fig. 4 for an example). We use instance indices to connect nodes to the correct parent when there are such multiples. At any point in the algorithm we identify the "current node" as $current$, and the pattern root as $root$.

We now state, without proof, the correctness property of the instantiation algorithm.

Correctness: If $\mathcal{P}$ is a well-founded pattern that satisfies the multiplicity condition, and $\tau \vDash \mathcal{P}$, then $instantiate(\mathcal{P}, \tau) \in [\![\mathcal{P}]\!]$. A consequence is that the algorithm produces well-formed instances.

5 Illustrative Example

To illustrate pattern instantiation, we use the *requirements breakdown pattern* (Fig. 3), which we have derived from our ongoing experience with safety case development for an unmanned aircraft system [2], [4], [6]. It also extends our previous work[9] on algorithmically deriving argument structure fragments from requirements/hazards tables [3].

The requirements breakdown pattern (Fig. 3) provides a framework to abstractly represent the argument implicit in a requirements table[10]. Specifically, it shows how the claims entailed by requirements can be hierarchically developed and linked to the supporting evidence produced from the specified verification methods. Due to space limitations, we do not provide a complete pattern specification.

[9] In fact, a $\mathcal{P}$-table similar to Table 1 can be extracted from the tables in [3].

[10] See [3] for an example of a requirements table.

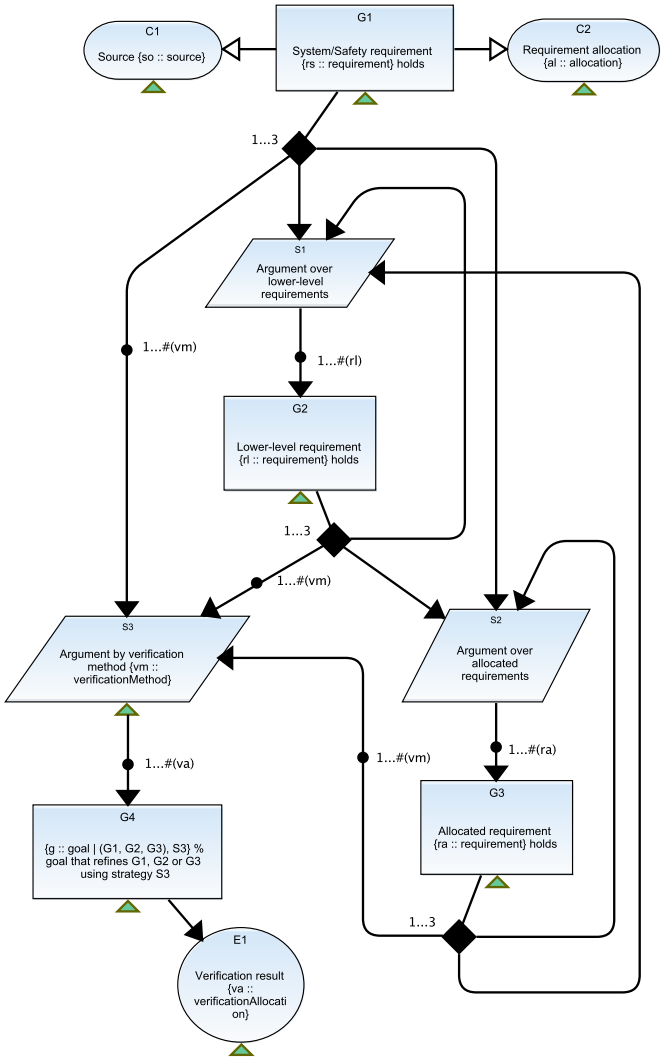

Fig. 3. Requirements breakdown pattern, abstracting the structure of the argument implicit in a requirements table

Table 1. Example of a populated $\mathcal{P}$-table to instantiate the requirements breakdown pattern

Parameter Type	*Requirement*	*Lower-level requirement*	*Allocated Requirement*	*Source*	*Requirement Allocation*	*Verification Method*	*Verification Allocation*
Data node / Join	**G1**	**G2**	**G3**	**C1**	**C2**	**S3**	**E1**
	R1	R1.1, R1.2	AR1	S	A	VM11, VM12	VA11, VA12
(S3, VM12)							VA22
(G2, R1.1)						VM1.11, VM1.12	VA1.11, VA1.12
(G2, R1.2)		R1.2.1, R1.2.2	AR1.2				
(G2, R1.2.1)						VM1.2.1	VA1.2.1
(G3, AR1.2)			AR1.21			VM1.2	VA1.2

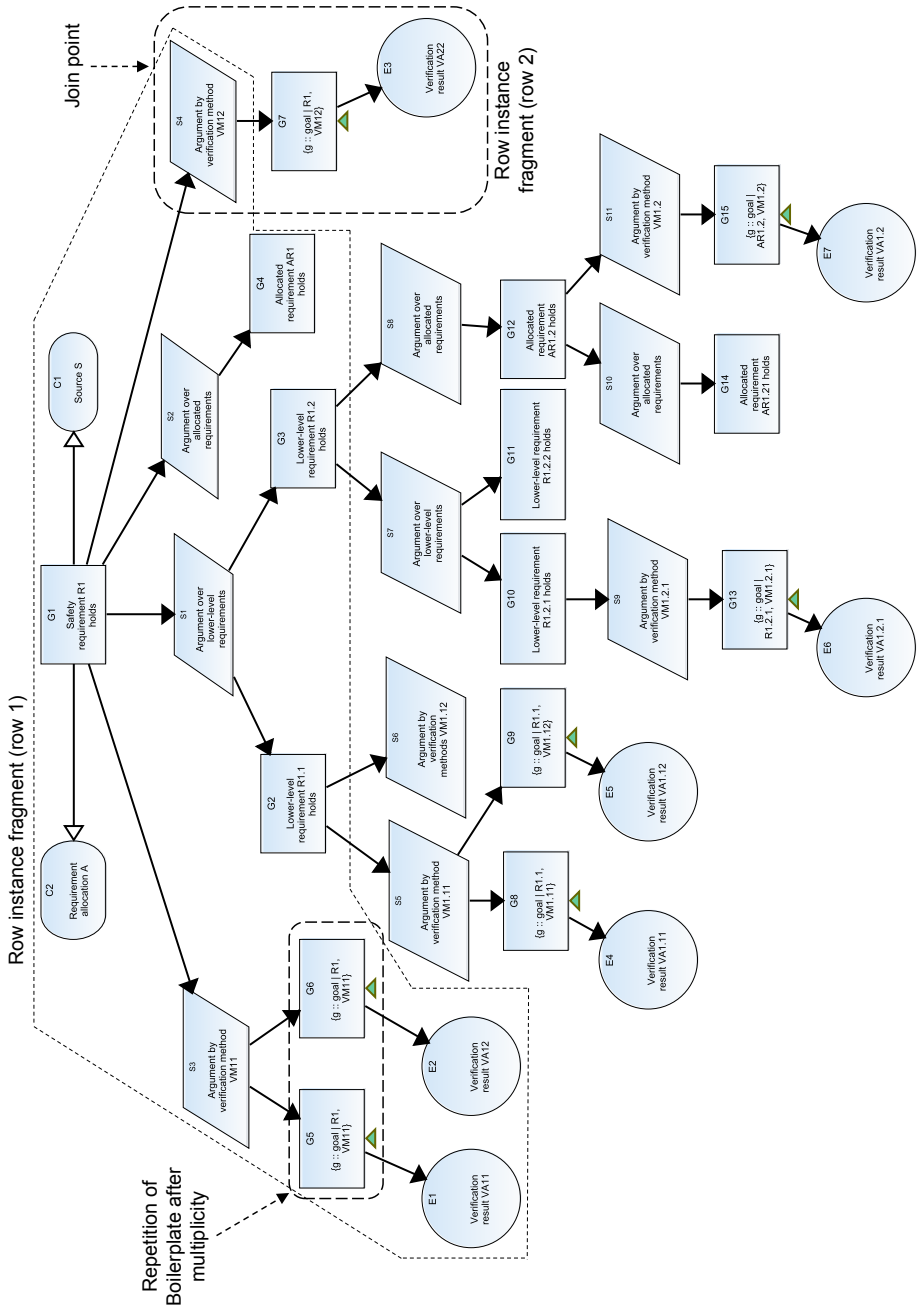

Fig. 4. Application of the generic pattern instantiation procedure (Algorithm 2): Concrete instance of the requirements breakdown pattern (Fig. 3) using the values from the $\mathcal{P}$-table (Table 1), highlighting row instance fragments, join points and repetition of boilerplate nodes

In brief, the claim in the root goal (G1) of the pattern is that a safety/system requirement, which is usually made in the contexts of some source (C1), or system, i.e., requirement allocation (C2), holds. A choice of three strategies is available to develop G1: hierarchical decomposition (S1, S2) and appeal to one or more verification methods (S3). The sub-claims (G2, G3) resulting from applying hierarchical decomposition are semantically similar to the root claim that they refine. Consequently, we can apply the same strategies to develop them further. Eventually, we support all claims by verification evidence (E1). The evidence is preceded by an *evidence assertion* (G4), i.e., a minimal proposition directly concerning the source data of the evidence [14].

Table 1 shows a populated $\mathcal{P}$-table for the requirements breakdown pattern with the columns, labeled by the pattern data nodes, containing example data entries entered corresponding to the root node and the join points. We have listed the data node parameter type for clarification purposes and it is not formally part of the data model.

Fig. 4 shows an instance of the pattern derived by applying our generic pattern instantiation procedure (Algorithm 2) and using the $\mathcal{P}$-table (Table 1). It highlights the repetition of boilerplate nodes[11] after multiplicity, and illustrates how a join point connects two row instance fragments.

6 Conclusion

We have presented the foundational steps towards, we believe, a rich theory of safety case patterns that will enable more sophistication in their usage than is currently available, e.g., automated instantiation, composition, and transformation-based manipulation. The main benefit of our work from a practitioner's perspective, we anticipate, is a reduction in the effort involved in safety case creation/management due to the raised level of abstraction at which arguments can be formulated, together with improved assurance. Specifically, given the assurance afforded by automated instantiation that a pattern instance is well-formed and meets its specification, practitioners, i.e., safety engineers who create safety arguments, and certification/qualification authorities who evaluate them, can divert efforts to domain-specific issues, e.g., selecting the appropriate patterns for assurance, evaluating a smaller, abstract argument structure for fallacies/deficits instead of its larger concrete instantiation, determining the evidence required to support the claims made, etc.

However, more can be done: as mentioned earlier, the formal definitions and the algorithm can be extended to include the notions of undeveloped, UI and UU. One design choice in the algorithm was to instantiate only those nodes for which parameters have values in the data table. An alternative choice could be to use the whole pattern so that those data nodes that do not take values in the table are also reproduced in the instance but left as UI or UU, as appropriate. The relationship between modular abstractions, hierarchies [7], and patterns is, as yet, unclear although there are a few examples of applying patterns within a modular organization [9]. The goal of formalization, here, would be to raise the level of abstraction and to increase automation. We use a notion of sequential composition of patterns. We have also defined a notion of parallel composition (not given in this paper) to create patterns, such as for requirements breakdown

[11] Recall that we consider evidence assertion nodes as boilerplate (see item (b) on p. 24).

(Fig. 3), from simpler patterns. Future work will involve, in part, extending the formal basis given in this paper to the topics mentioned above.

We have already implemented the GSN abstractions and our notational extensions for patterns in our toolset, AdvoCATE [5]; we plan to extend the tool with the algorithm described here. Clarifying concepts such as patterns and the data for their instantiation will be necessary to support tool interoperability, which is one of the goals [13] of emerging safety/assurance case standards.

Acknowledgement. This work has been funded by the AFCS element of the SSAT project in the Aviation Safety Program of the NASA Aeronautics Mission Directorate.

References

1. Alexander, R., Kelly, T., Kurd, Z., McDermid, J.: Safety Cases for Advanced Control Software: Safety Case Patterns. Final Report, NASA Contract FA8655-07-1-3025, Univ. of York (October 2007)
2. Denney, E., Habli, I., Pai, G.: Perspectives on Software Safety Case Development for Unmanned Aircraft. In: Proc. 42nd IEEE/IFIP Intl. Conf. Dep. Sys. and Networks (June 2012)
3. Denney, E., Pai, G.: A lightweight methodology for safety case assembly. In: Ortmeier, F., Lipaczewski, M. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 1–12. Springer, Heidelberg (2012)
4. Denney, E., Pai, G., Pohl, J.: Automating the generation of heterogeneous aviation safety cases. Tech. Rep. NASA/CR-2011-215983, NASA Ames Research Center (August 2011)
5. Denney, E., Pai, G., Pohl, J.: AdvoCATE: An Assurance Case Automation Toolset. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP Workshops 2012. LNCS, vol. 7613, pp. 8–21. Springer, Heidelberg (2012)
6. Denney, E., Pai, G., Pohl, J.: Heterogeneous aviation safety cases: Integrating the formal and the non-formal. In: 17th IEEE Intl. Conf. Engineering of Complex Computer Systems pp. 199–208 (July 2012)
7. Denney, E., Pai, G., Whiteside, I.: Hierarchical safety cases. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 478–483. Springer, Heidelberg (2013)
8. Goal Structuring Notation Working Group: GSN Community Standard Version 1 (November 2011), `http://www.goalstructuringnotation.info/`
9. Industrial Avionics Working Group: Modular Software Safety Case Process, Parts A and B: Process and Guidance. Tech. Rep. IAWG-AJT-301, Issue 2 (October 2007)
10. Kelly, T.: Arguing Safety: A Systematic Approach to Managing Safety Cases. Ph.D. thesis, Univ. of York (1998)
11. Kelly, T., McDermid, J.: Safety case construction and reuse using patterns. In: Daniel, P. (ed.) Safe Comp 1997, pp. 55–69 (1997)
12. Menon, C., Hawkins, R., McDermid, J.: Interim standard of best practice on software in the context of DS 00-56 Issue 4. SSEI Standard of Best Practice (Issue 1). Univ. of York (2009)
13. Object Management Group: Structured Assurance Case Metamodel (SACM) version 1.0. Formal/2013-02-01 (February 2013), `http://www.omg.org/spec/SACM/`
14. Sun, L., Kelly, T.: Elaborating the concept of evidence in Safety Cases. In: Proc. 21st Safety Critical Sys. Symp. (February 2013)
15. Weaver, R.: The Safety of Software – Constructing and Assuring Arguments. Ph.D. thesis, Dept. of Comp. Sci., Univ. of York (2003)